QUICKSENSE™ FIRMWARE API

1. Introduction

This document describes the QuickSense™ Firmware API v2.xx and outlines the process of using the API to develop capacitive sensing firmware systems.

This document covers the following:

- Firmware API description
- Guide to configuring the API for use in a firmware system
- Complete firmware details for the QuickSense Firmware API

For information about the QuickSense Firmware API's method for baselining, see Application Note AN418. For a detailed discussion on setting thresholds and SNR measurement, please see Application Note AN367. For the QuickSense Communications Interface specification, refer to Application note AN494. For a list of changes to the QuickSense API firmware code base, the QuickSense interface firmware, or QuickSense firmware examples, please refer to the change logs found in QuickSense kit documentation directories.

2. Terminology

In this application note, the following definitions apply:

- Channel—Port pin to be measured by the MCU.
- Threshold—A system configuration set point that allows a change in threshold 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. Defines if a channel is active or inactive.
- 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.
- Pad—Grouping of capacitive sensing channels or IR channels being measured by firmware to provide 1-dimensional, 2-dimensional, or 3-dimensional position of an object.
- *n*D—Represents 1-dimensional, 2-dimensional, and 3-dimensional in this document.
- Pad point—Coordinate that describes a position on a pad where firmware has detected an object.
- Pad array—Array of pad points.
- Data type—An identifier for values measured by hardware peripherals or interpreted by functions in the QuickSense API.
- Process—A system mechanism that allows the measurement and interpretation of data type values.
- Class—An identifier for a process.

3. QuickSense Firmware API Overview

The QuickSense Firmware API performs the following tasks:

- Captures and stores capacitive sensing and proximity/ambient light sensing measurements from multiple input channels.
- Computes threshold state information on enabled input channels.
- Applies an exponential averaging filter to reduce the effects of noise on input channels.
- Compensates for changes to environmental conditions using a method of baselining.
- Compatibility with many Silicon Laboratories MCUs and fixed function sensing devices such as the Si1102 and the Si1120.

- Allows users to bind input channels into groups such as control wheels, sliders, and pads that describe position in an *n*D measurement region.
- 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 sensing methods.
- Creates a serial interface that abides to the QuickSense Communications Interface specification which enables users to output channel values, thresholds, and group values and adjust threshold levels without recompiling code using the QuickSense Studio's Performance Analysis Tool.

Figure 1 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 Application Note AN494.

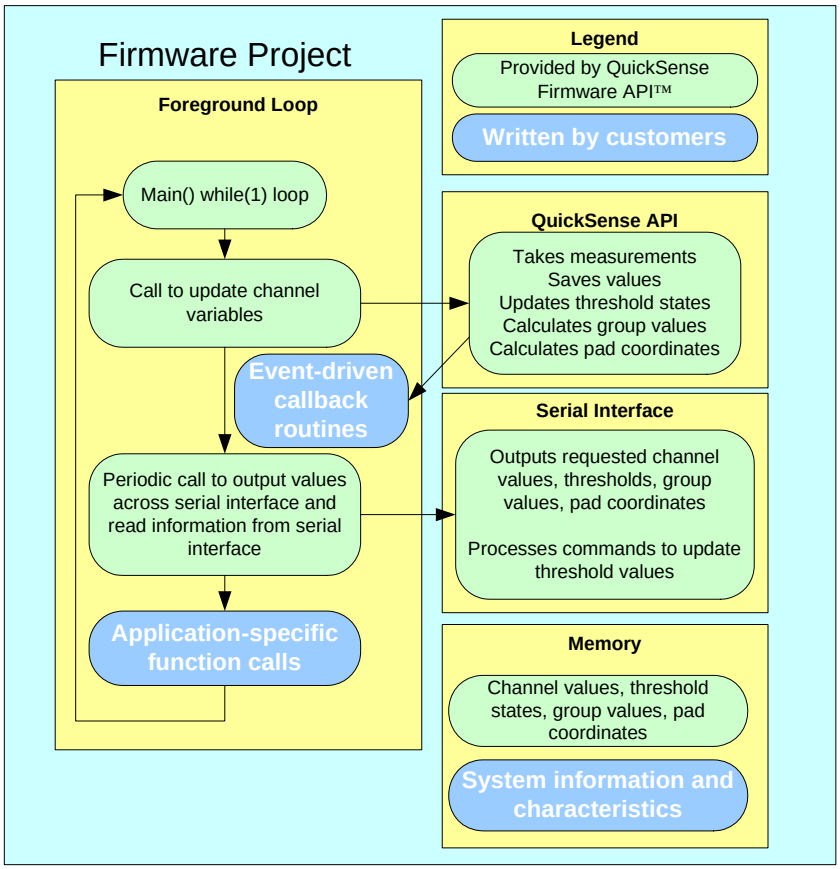

Figure 1. Basic Data Flow Diagram

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 including 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 size 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 "4. Designing with the API" on page 9.

The QuickSense Configuration Wizard, which is included with the QuickSense Studio, provides a graphical interface that generates a pre-configured QuickSense firmware project. For detailed information on the

QuickSense Studio software, see the QuickSense Studio User's Guide available on the Silicon Labs QuickSense webpage (www.silabs.com/quicksense) in the QuickSense Studio section.

3.2. Capturing Data

The QuickSense Firmware API measures capacitance and ambient/proximity light data using Silicon Laboratories MCU hardware such as the C8051F70x family's capacitance sensing peripheral and fixed function devices such as the Silicon Laboratories Si1120 and Si1102 light sensors. Abstraction methods separate higher-level data collection and storage from the low-level physical interface. Once firmware captures channel data, routines perform calculations including threshold state updates, group position updates, pad points, and gestures.

Out of reset, the QuickSense API is not initialized to perform measurements nor calculations on captured data. To enable the API to process data, the QS_SetProcessEnable() function must be called to enable a particular class (channels, thresholds, groups, nD pads). The function sets a bit in a bit array which is checked whenever measurement and processing functions are called. To set all indices of a class, use the QS_SetAllProcessEnable() function.

If processing is enabled for a class that calculates data which needs sampled data, then classes that sample data will be enabled as well. The following table shows how enabling a class affects another class.

Table 1. Additional effects when enabling processing of a class

Class	Effects when enabled
Channel	Does not affect another class
Threshold	Enables the channel index used to calculate the threshold.
Groups	Enables the channel indices that are in the group channel list.
1D pads	Enables the channel indices that are in the 1D pad channel list.
2D pads	Enables the channel indices that are in the 2D pad channel list.
3D pads	Enables the channel indices that are in the 3D pad channel list.

If processing for a channel is disabled, then any class index using that channel is disabled as well.

3.3. Accessing Data

The API stores data 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, though this method is not recommended.

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 Performance Analysis Tool uses this serial interface to read the data it displays onscreen.

In addition to outputting values, the serial interface can also be used to configure device threshold levels and baseline-related calibration constants.

For a detailed description of the serial interface command set, see Application Note AN494.

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 two states. 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 two defined threshold states and the points where the QuickSense Firmware API calls rising and falling edge event functions.

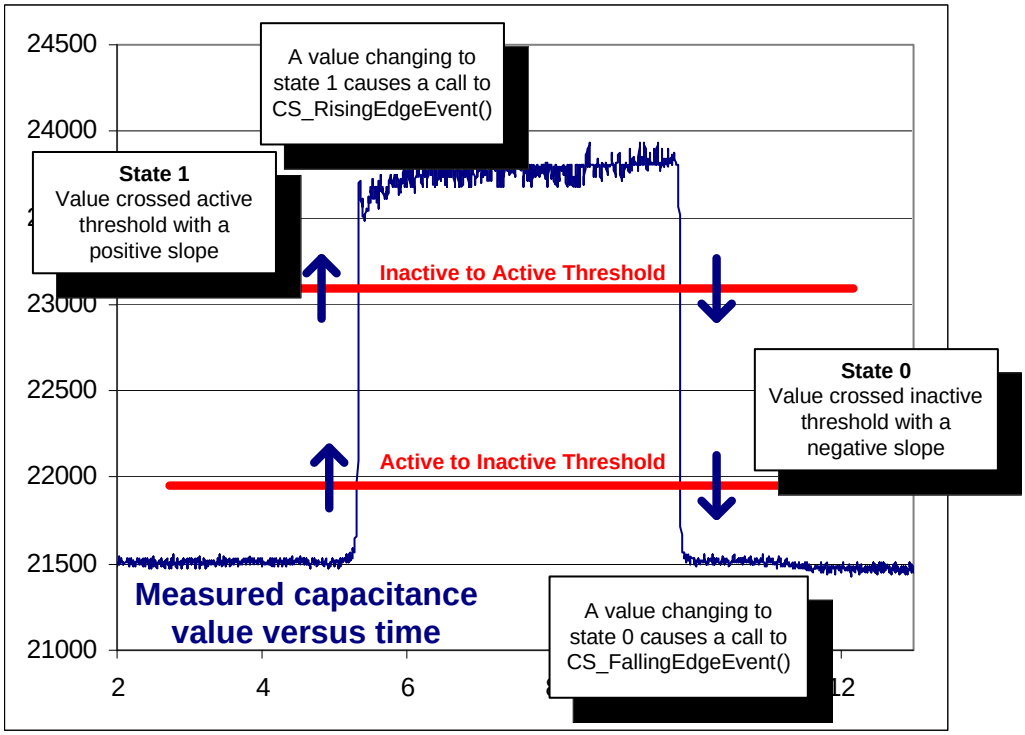

Figure 2. Threshold States and Rising and Falling Edge Events

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 "4. Designing with the API" on page 9.

3.6. Groups

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

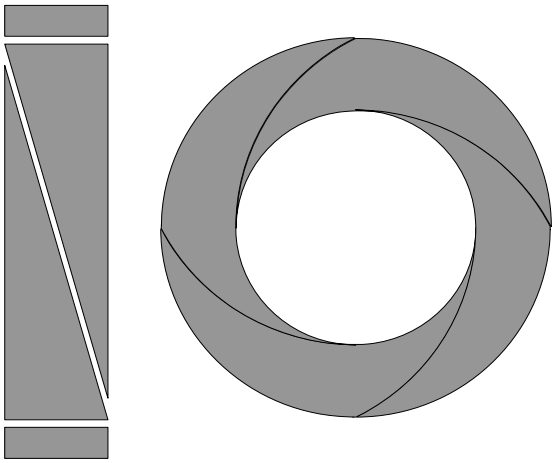

Figure 3. Sliders and Control Wheels Compatible with the API

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.

Gesture and ambient light sensor outputs are also stored in QS_GroupValue[]. *n*D pad algorithms with gesture support write algorithm-specific values to group array elements. If ambient light sensing is enabled and an ambient light type group is created, the ambient light sensing algorithm can identify the light source type.

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:

1. A finger presses down on a group, and the algorithm calls the QS_RisingGroupEvent().
2. As a finger remains on the group, producing updated group position values, the API calls QS_GroupEvent().
3. When the finger is lifted from the group, the API calls QS_FallingGroupEvent().

3.7. *n*D Pads

The QuickSense Firmware API provides support for algorithms that interpret channel values to find the position of a conductive object such as a finger relative to an *n*-dimensional physical space. For example, the QuickSense Configuration Wizard provides an algorithm that takes values measured from two LEDs using an Si1120 to form an “IR slider.” Hand position above the slider is defined by a 2-dimensional pad. The algorithm calculates a 2-D pad point describing the hand position, and the algorithm also detects a number of different gestures, including “right swipe,” “left swipe,” and “pause.” When the algorithm detects one of these gestures, it writes the corresponding gesture value to an allocated slot in the group value array.

3.8. Ambient light sensing algorithm

The ambient light sensing algorithm takes samples from a light sensing device such as the Si1120, calculates luminescence in units of lux, and outputs that value in a channel array element. The algorithm also outputs a symbol in the QS_GroupValue[] array that corresponds to the type of light detected.

The luxcalc() function calculates a lux value and the light type given the Si1120 VAMB, VIRL and VIRH measurements. The luxcalc() function expects the VAMB, VIRL and VIRH values taken the PCA counter values latched when CEX*n* is toggled by the Si1120 PRX pin. These values, when multiplied by the pre-compiler directive PCA_SAMPLE_RATE, results in the time (in microseconds) the Si1120 asserts PRX. Therefore, PCA_SAMPLE_RATE should be customized at compile time for proper operation. By default, PCA_SAMPLE_RATE it is set at 6, assuming a 6 MHz PCA clock.

The luxcalc() function, once completed, updates the pointers to memory so that the group storing light type and the channel index storing lux value can be updated. *retLux is 16-bit word representing the lux value. When the MSB (bit 15) is zero, bits 14-0 represent the lux value. When the MSB (bit 15) is one, bits 14-0 represent the lux value multiplied by four. This encoding allows the lux from 1 to 120 klux without needing to use a 32-bit integer. Unless the Si1120 is exposed to direct sunlight, the MSB should normally be zero.

*retype is a byte holding three bits. Each bit represents a light type. The available light types are set in *retype based on the user-definable pre-compiler directives INCANDESCENT, FLUORESCENT and SUNLIGHT. INCANDESCENT represents blackbody sources such as halogen or incandescent; FLUORESENT for CFL or fluorescent; SUNLIGHT for direct sunlight. As there can be light-mixing, it is possible for multiple bits to be set at the same time.

The VAMB measurement is proportional to the current reported by the visible light photodiode. The VIRL and VIRH measurements are proportional to the current reported by the IR photodiode. VIRH and VIRL measurements differ in its sensitivity. For lower light conditions, VIRL is used generally. However, for higher levels of ambient light, VIRH is used when VIRL saturates. The ratio of the measured light between the visible photodiode (VAMB) and IR photodiode (VIRL or VIRH) provides an indication of the color ratio of the light source. With the color ratio of the light, the light source can be estimated and reported. The color ratio information is then used as a correction factor applied to measurements to arrive at a lux value.

Since the algorithm relies on examining the ratio between various measurements, it is important to ensure that the pulse widths taken under dark conditions for each of the three measurements are subtracted from the current reading. The reason is that the PRX pulse widths reported by the Si1120 are at a minimum, 4 us and the actual value depends on semiconductor process and temperature. If the dark condition pulse width is not removed, this will result in an error in the color ratio, leading to improper correction factors. To avoid imposing a calibration step in the end application, luxcalc() uses three static variables (minVamb, minVirh and minVirl). Each representing the dark condition pulse width for each mode. These static variables are initialized with a default average value, end are continuously updated with the minimum pulse width taken. Whenever ambient conditions ever become dark, these static variables will reach the dark condition pulse width, thereby making luxcalc() more accurate over time.

3.9. Generic Data

The QuickSense Firmware API provides an array of 16-bit values to store application specific data that can be transferred to a host. It provides a place for device and host to transfer and modify application specific data that resides in the QS_GenericData[] array. It is suited for data that does not apply to the other data types defined in QuickSense. If values larger than 16-bits need to be passed to the host, multiple generic data elements can be allocated.

3.10. Non-Volatile Calibration and Configuration Area

The Non-Volatile Calibration and Configuration Area (NVCCA) stores calibration information for the device such as threshold percentages, reference baseline magnitudes and the baseline update rate. In previous version of the QuickSense Firmware API, the start location of this page could be configured. Starting with QuickSense API version 2.30, the NVCCA page is now located on the second page of flash. This corresponds to address 0x0200 for devices with flash pages of 512 bytes or address 0x0400 for devices with flash pages of 1024 bytes. A variable called NVCCA_PADDING[] makes sure the rest of the page is allocated so execution code does not reside in NVCCA. If FLASH_MODE is READ_ONLY then NVCCA_PADDING[] is removed so code can reside on the NVCCA page to save code space.

3.11. Linker Options

With the introduction of QSCI Lite in QuickSense API Version 2.30, there is a need to locate QuickSense variables. Most QuickSense XDATA arrays are located at specific addresses; the exact addresses are defined in QS_Definitions.h and shift depending on system configurations. There is also a need to locate the configuration arrays found in QS_Config.c. Keil places a restriction that a variable cannot be located using the “at” keyword and initialized at the same time. Since these arrays must be initialized since they are user configurable variables, linker options must be used to locate these arrays.

We link the configuration arrays by linking the entire QS_Config.c module. The QS_Config.c module must be located at 0x0100. The linker commands for each compiler are:

- Keil: CODE(0, ?CO?QS_CONFIG(0x100))
- Raisonance: CODE(0, ?PR?QS_CONFIG(0x100))
- SDCC: -Wl-bCONFIG=0x100,-bCSEG=0xYYYY,-bXSEG=0xZZZZ

For SDCC, there are addresses labeled 0xYYYY and 0xZZZZ. This is because the SDCC compiler does not reserve memory for located variables. It is up to the user to make sure that memory addresses are not overlayed. The CSEG module contains the execution code which must be contiguous. Therefore, the user must locate CSEG after the NVCCA page which would be the third page of flash, address 0x400 for devices with flash pages of 512 bytes or address 0x800 for devices with flash pages of 1024 bytes. If FLASH_MODE is set to READ_ONLY then it is okay for CSEG to be placed on the NVCCA page of flash. CSEG should be placed after QS_BaselineUpdateRate.

The user must do the same with the XSEG module, which contains all of the variables located in XDATA. The length of each located QuickSense XDATA variable is defined below. For definitions with ROUND_UP( ), the user should round up to the nearest byte.

- QS_ChannelValue[] NUM_CHANNELS * 2 bytes
- QS_RuntimeBaseline[] NUM_CHANNELS * 4 bytes
- QS_GroupValue[] NUM_GROUPS * 2 bytes
- QS_1DPadPoints[] NUM_1D_PADS * 2 bytes * MAX_1D_POINTS
- QS_2DPadPoints[] NUM_2D_PADS * 4 bytes * MAX_2D_POINTS
- QS_3DPadPoints[] NUM_3D_PADS * 6 bytes * MAX_3D_POINTS
- QS_GenericData[] NUM_GDATA_ELEMENTS * 2 bytes
- QS_ThresholdState[] ROUND_UP(NUM_CHANNELS / 8)
- QS_ChannelCalibration[] ROUND_UP(NUM_CHANNELS / 8)
- QS_ChannelProcessEnable[] ROUND_UP(NUM_CHANNELS / 8)
- QS_ThresholdProcessEnable[] ROUND_UP(NUM_CHANNELS / 8)
- QS_GroupProcessEnable[] ROUND_UP(NUM_GROUPS / 8)
- QS_1DProcessEnable[] ROUND_UP(NUM_1D_PADS / 8)
- QS_2DProcessEnable[] ROUND_UP(NUM_2D_PADS / 8)
- QS_3DProcessEnable[] ROUND_UP(NUM_3D_PADS / 8)
- QS_FlashKey[] 2 bytes

3.12. Low Power

In version 2.40, the low power library and SmaRTClock library are introduced into QuickSense for the C8051F9xx devices. These libraries provide API routines that place the MCUs into lower power modes such as sleep and suspend.

Timing for a low power system is provided through the SmaRTClock peripheral instead of the Timer 0 peripheral. To ensure that Timer 0 is not used, the configuration EXTERNAL_TIME_MANAGEMENT must be set to ENABLED. Using the SmaRTClock peripheral also requires a different main application loop due to the use of alarms and sleep mode. The new main application file is named QS_Main_LowPower.c in the firmware template directory. If a project is generated through the QuickSense Configuration Wizard, the QS_Main.c file is replaced with the low power main file if LOW_POWER is ENABLED.

If LOW_POWER is ENABLED, the serial interface cannot be used. There is currently no mechanism in QSCI that allows the host to know whenever the device is awake to receive a command. This is a problem since the serial interface is necessary for debugging systems with QuickSense applications. Therefore if LOW_POWER is ENABLED and the SERIAL_INTERFACE is not DISABLED, the MCU is not placed into any low power modes. This allows the user to debug the system using the low power framework. Any problems that occur from leaving and entering low power modes must be done without the help of QuickSense applications with the current code base.

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.

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 implemented.

1. Which MCU will be used in the system? Will the MCU be interfacing with external sensing devices?
2. How many input channels does the system have?
3. What is the sensing method (capacitive sensing or proximity/ambient light sensing) and hardware multiplexer setting for each channel to be measured?

Note: See the relevant MCU data sheet for multiplexer settings.

4. Do any of the channels require threshold detection? If so, which channels?
5. Is the serial interface needed by the application? Will the application be communicating with the Performance Analysis Tool or a host system?
6. If “yes” to #5: Does enumeration information (description of the device’s capabilities) need to be provided to a host system, or does the host already know the device’s capabilities?
7. If “yes” to #5: What transfer modes and types need to be supported?
8. If “yes” to #5: Is there a need to save code space and therefore use a simpler serial interface?
9. How many groups are defined in the system and what is each group’s type (control wheel, slider, etc.)?
10. Does the system need to respond to threshold detection rising edge or falling edge events?
11. Does the system need to respond to group events?
12. Does the system need to respond to pad events generated by different pad calculation algorithms?
13. What is the frequency that each channel’s value will be updated?
14. Where is the calibration information located in Flash?
15. Should writing/erasing on-chip calibration information be enabled or disabled?
16. Is there any need for application specific data to be transferred to a host?

The information that follows guides 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 designer, and the QS_Config.c file contains the non-volatile, code space-based arrays that must be defined. The subsections below describe each definition and array, and point out dependencies between different items. The designer must also configure the QS_DeviceInit.c file with the hardware configuration for their system. If the designer is using an ambient/proximity light sensing device, the appropriate device file must be included in the build, and QS_ExtDeviceConfig.h must be configured to define the hardware interface between the MCU and the device.

4.2.1. QS_Config.h

This file contains definitions that control the size of memory arrays and determine which code segments of the firmware system are included in the build.

4.2.1.1. BOARDID

Description: If the firmware will be using the serial interface to communicate with the Performance Analysis Tool, then the system needs to have a defined board identification value. The Performance Analysis Tool 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 Performance Analysis Tool to use the generic display method.

4.2.1.2. 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 device which have a CS0 module.

4.2.1.3. NUM_IR_CHANNELS

Can be set to: Numeric value

Description: This definition describes the number of ambient/proximity light sensor channels.

Notes: This method is currently supported in low-level API routines for the Si1102 and Si1120.

4.2.1.4. 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[].

4.2.1.5. 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[].

4.2.1.6. NUM_GESTURES

Can be set to: Numeric positive value

Description: When this number is not set to 0, ambient light information must be included QS_GroupType[] and QS_GroupChannelAssignmentTable[].

4.2.1.7. NUM_AMBLIGHT

Can be set to: Numeric positive value

Description: This defines the number of groups that describe ambient light information.

4.2.1.8. NUM_1D_PADS

Can be set to: Numeric positive value

Description: This definition describes the number of 1D pads that are measured by the API.

Notes: When this number is not set to 0, pad configuration information should be included in QS_1DPadCapabilities[] and QS_1DPadChannels[].

4.2.1.9. NUM_2D_PADS

Can be set to: Numeric positive value

Description: This definition describes the number of 2D pads that are measured by the API.

Notes: When this number is not set to 0, pad configuration information should be included in QS_2DPadCapabilities[] and QS_2DPadChannels[].

4.2.1.10. NUM_3D_PADS

Can be set to: Numeric positive value

Description: This definition describes the number of 3D pads that are measured by the API.

Notes: When this number is not set to 0, pad configuration information should be included in QS_3DPadCapabilities[] and QS_3DPadChannels[].

4.2.1.11. NUM_GDATA_ELEMENTS

Can be set to: Numeric positive value

Description: This definitions describes the number of generic data elements in the system.

4.2.1.12. MCU

Can be set to: F70X, F80X, F91X, F93X, or F99X

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

Notes: The definition F70x includes the C8051F700-C8051F715. The definition F80X includes the C8051F800-C8051F835. The definition F91X includes the C8051F901-C8051F902 and the C8051F9110-C8051F912. The definition F93X includes the C8051F920-C8051F921 and the C8051F930-C8051F931.

4.2.1.13. IR_DEVICE

Can be set to: DISABLED, SI1102, SI1120

Description: This defines which low-level IR measurement routines should be included in the build.

4.2.1.14. FLASH_MODE

Can be set to: READ_WRITE, READ_ONLY

Description: This defines if flash modification routines are included in the build. If FLASH_MODE is READ_ONLY, NVCCA_PADDING[] is removed and executable code memory can reside on the NVCCA page.

4.2.1.15. 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.

4.2.1.16. CHANNEL_THRESHOLD

Can be set to: ENABLED or DISABLED

Description: This definition adds or removes the threshold class from the system. Thresholds cannot be processed, or transferred if CHANNEL_THRESHOLD is DISABLED.

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

4.2.1.17. 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, see AN418. If this definition is set to 100 and ACTIVE_BASELINE_MAX_PERCENT is set to 100, code space can be saved.

4.2.1.18. ACTIVE_BASELINE_MAX_PERCENT

Can be set to: Numeric value

Description: This value controls the maximum level that the runtime active baseline can rise to. If the active baseline is pushed higher than the size defined by this definition, the runtime inactive baseline will be pulled up with it. For more detailed information on this definition, see AN418. If this definition is set to 100 and ACTIVE_BASELINE_DECAY_PERCENT is set to 100, code space can be saved.

4.2.1.19. HISM

Can be set to: Numeric value

Description: The High Inactive Sensitivity Margin (HISM) sets a threshold above which the baselining algorithm assumes that a channel could be active. Values greater than the HISM threshold will not be used to update that channel's runtime inactive baseline. For more information about this definition, see AN418.

4.2.1.20. LISM

Can be set to: Numeric value

Description: The Low Inactive Sensitivity Margin (LISM) sets a threshold below which the baselining algorithm forces a runtime baseline update. For more information about this definition, see AN418.

4.2.1.21. MAX_1D_POINTS

Can be set to: Numeric positive value

Description: Defines the maximum number of simultaneously active points for 1D pads. Allocates memory to accommodate all points being active at once.

4.2.1.22. MAX_2D_POINTS

Can be set to: Numeric positive value

Description: Defines the maximum number of simultaneously active points for 2D pads. Allocates memory to accommodate all points being active at once.

4.2.1.23. MAX_3D_POINTS

Can be set to: Numeric positive value

Description: Defines the maximum number of simultaneously active points for 3D pads. Allocates memory to accommodate all points being active at once.

4.2.1.24. 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.

4.2.1.25. GROUP_EVENTS

Can be set to: RISING, FALLING, RISING_AND_FALLING, ACTIVE_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.

4.2.1.26. PAD_1D_EVENTS

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

Description: Defines what types of 1D pad callback routines can be called by the developer's algorithm whenever 1D pad events are detected.

4.2.1.27. PAD_2D_EVENTS

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

Description: Defines what types of 2D pad callback routines can be called by the developer's algorithm whenever 2D pad events are detected.

4.2.1.28. PAD_3D_EVENTS

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

Description: Defines what types of 3D pad callback routines can be called by the developer's algorithm whenever 3D pad events are detected.

4.2.1.29. EXTERNAL_TIME_MANAGEMENT

Can be set to: DISABLED, ENABLED

Description: If ENABLED, QuickSense system timing through Timer 0 is not included in the build. The application layer is responsible for providing system timing by setting the following flags at the appropriate intervals:

QS_CS0UpdateFlag

S_IRUpdateFlag

QS_BaselineUpdateFlag

If ENABLED, the IR_SAMPLE_RATE and CS0_SAMPLE_RATE definitions are not valid.

EXTERNAL_TIME_MANAGEMENT must be set to ENABLED if LOW_POWER is ENABLED.

4.2.1.30. IR_SAMPLE_RATE

Can be set to: Numeric positive integer within the range of 1–6553.

Description: The rate at which IR channels are sampled in milliseconds.

Note: STX_MAX_HIGH_TIME x number of irLEDs x 4 is the recommended IR_SAMPLING_RATE when using the Si1120.

4.2.1.31. CS0_SAMPLE_RATE

Can be set to: Numeric positive integer within the range of 1–6553.

Description: The rate at which CS0 channels are sampled in milliseconds.

4.2.1.32. CHANNEL_AVERAGING

Can be set to: DISABLED, ENABLED

Description: Controls whether channel samples are run through the exponential averager before being stored in QS_ChannelValue[].

4.2.1.33. AVERAGING_EXPONENT

Can be set to: Numeric positive integer within the range of 1–7.

Description: This controls how aggressively the exponential averaging function filters noise from data channels. Please see "5.2.4. Exponential Averaging Routines" on page 26 for details.

4.2.1.34. LOW_POWER

Can be set to: DISABLED, ENABLED

Description: Defines if the low power library and the SmaRTClock library are added to the build. Also enables the use of the low power main routine. If LOW_POWER is ENABLED, t hen EXTERNAL_TIME_MANAGEMENT must be ENABLED.

4.2.1.35. WAKEUP_INTERVAL

Can be set to: Numeric positive integer within the range of 1–65535

Description: Defines, in milliseconds, how often the device wakes up from sleep mode. With the default low power main routine, this interval also defines how often samples are taken and how often the system is updated.

4.2.1.36. RTC_CLKSRC

Can be set to: SELFOSC, CRYSTAL

Description: Determines the clock source for the SmaRTClock peripheral when it is being used for low power. SELFOSC uses one of the MCU's internal sources to clock the RTC. CRYSTAL mode allows an external crystal to drive the RTC.

4.2.1.37. LOADCAP_VALUE

Can be set to: Numeric positive integer within the range of 0–15

Description: Defines the RTC load capacitance value. A smaller load capacitance will increase the RTC frequency. A larger load capacitance will decrease the RTC frequency

0= 4.0 pF,	1 = 4.5 pF,	2 = 5.0 pF,	3 = 5.5 pF
4 = 6.0 pF,	5 = 6.5 pF,	6 = 7.0 pF,	7 = 7.5 pF
8 = 8.0 pF,	9 = 8.5 pF,	10 = 9.0 pF,	11 = 9.5 pF
12 = 10.5 pF,	13 = 11.5 pF,	14 = 12.5 pF,	15 = 13.5 pF

4.2.1.38. SERIAL_INTERFACE

Can be set to: DISABLED, QSCI_LITE, QSCI

Description: This definition determines if a serial interface is added to the project. If so, it decides which serial interface implementation to use. Please refer to Application Note AN494 for more information about the QuickSense Communications Interface.

4.2.1.39. QSCI_HARDWARE

Can be set to: UART

Description: This definition defines the hardware peripheral used for QSCI.

4.2.1.40. RX_BUFFER_SIZE

Can be set to: Numeric positive integer within the range of 8–255

Description: This definition defines the length of the receive buffer for QSCI.

4.2.1.41. ENUMERATION_INFO

Can be set to: DISABLED, ENABLED

Description: Controls if the enumeration command is included in QSCI. This option is usually used to lower code size whenever the host already knows the system configuration.

4.2.1.42. TRANSFER_MODE_SUPPORTED

Can be set to: A combination (using “+”) of PERIODIC, ON_UPDATE, ON_DEMAND; DISABLED

Description: Controls which transfer modes are included in the firmware build. For example, assigning TRANSFER_MODE_SUPPORTED to PERIODIC+ON_UPDATE enables periodic and on-update transfer modes and disables the on-demand transfer mode.

Note: Should be set to DISABLED when using QSCI_LITE

4.2.1.43. TRANSFER_TYPE_SUPPORTED

Can be set to: A combination (using “+”) of SELECTED, SELECTED_AND_UPDATED; DISABLED

Description: Controls which transfer types are included in the firmware build.

Note: Should be set to DISABLED when using QSCI_LITE

4.2.2. QS_Config.c

The declarations listed below are included in QS_Config.c.

4.2.2.1. CODE_CONTENTS[ ]

Description: This array contains the 16-bit addresses of configuration variables stored in flash memory. This array is used in conjunction with the QSCI Lite implementation to tell the host the location of configuration variables. Each entry in the array corresponds to a specific variable, and if that variable does not exist due to system configurations, the value for that entry is 0x0000. This array starts at address 0x0100.

CODE_CONTENTS[0]—QS_GroupChannelAssignmentTable[]

CODE_CONTENTS[1]—QS_1DPadCapabilities[]

CODE_CONTENTS[2]—QS_1DPadChannels[]

CODE_CONTENTS[3]—QS_2DPadCapabilities[]

CODE_CONTENTS[4]—QS_2DPadChannels[]

CODE_CONTENTS[5]—QS_3DPadCapabilities[]

CODE_CONTENTS[6]—QS_3DPadChannels[]

CODE_CONTENTS[7]—QS_ChannelInfo[]

CODE_CONTENTS[8]—QS_IRConfig[]

CODE_CONTENTS[9]—QS_GroupType[]

CODE_CONTENTS[10]—End address of configuration variables

4.2.2.2. NVCCA_CONTENTS[ ]

Description: This array contains the 16-bit addresses of NVCCA variables stored in flash memory. This array is used in conjunction with the QSCI Lite implementation to tell the host the locations of NVCCA variables. Each entry in the array corresponds to a specific NVCCA variable and if that variable does not exist due to system configurations, the value for the entry is 0x0000. The array starts at address 0x0116.

NVCCA_CONTENTS[0]—QS_Threshold[]

NVCCA_CONTENTS[1]—QS_ReferenceBaselineMagnitude[]

NVCCA_CONTENTS[2]—QS_BaselineUpdateRate

NVCCA_CONTENTS[3]—End address of NVCCA variables

4.2.2.3. XDATA_CONTENTS

Description: This array contains the 16-bit addresses of XDATA variables stored in on-chip RAM. This array is used in conjunction with the QSCI Lite implementation to tell the host the locations of runtime variables. Each entry in the array corresponds to a specific runtime variable and if the variable does not exist due to system configurations, the value for the entry is the same as the next entry. The array starts at address 0x011E.

XDATA_CONTENTS[0]—QS_ChannelValue[]

XDATA_CONTENTS[1]—QS_RuntimeBaseline[]

XDATA_CONTENTS[2]—QS_GroupValue[]

XDATA_CONTENTS[3]—QS_1DPadPoints[]

XDATA_CONTENTS[4]—QS_2DPadPoints[]

XDATA_CONTENTS[5]—QS_3DPadPoints[]

XDATA_CONTENTS[6]—QS_GenericData[]

XDATA_CONTENTS[7]—QS_ThresholdState[]

XDATA_CONTENTS[8]—QS_ChannelCalibration[]

XDATA_CONTENTS[9]—QS_ChannelProcessEnable[]

XDATA_CONTENTS[10]—QS_ThresholdProcessEnable[]

XDATA_CONTENTS[11]—QS_GroupProcessEnable[]

XDATA_CONTENTS[12]—QS_1DProcessEnable[]

XDATA_CONTENTS[13]—QS_2DProcessEnable[]

XDATA_CONTENTS[14]—QS_3DProcessEnable[]

XDATA_CONTENTS[15]—QS_FlashKey[]

XDATA_CONTENTS[16]—End address for XDATA variables

4.2.2.4. FIRMWARE_REV_CAP

Description: This array contains the QuickSense Firmware API version information that is communicated to a host. It also contains bytes which detail the capabilities of a particular build. The serial interface revision number and the QuickSense API revision number should not be modified by the user because these values are used by QuickSense software tools to determine how to communicate with the device. The application revision value can be edited by the user to designate the version of the application layer of their firmware system.

Byte 0—QSCI major revision in BCD

Byte 1—QSCI minor revision in BCD

Byte 2—Baseline capability

Byte 3—QSCI configuration capabilities

Byte 4—QSCI data transfer capabilities

Byte 5—QuickSense API major revision in BCD

Byte 6—QuickSense API minor revision in BCD

Byte 7—Application major revision in BCD

Byte 8—Application minor revision in BCD

Byte 9—RX_BUFFER_SIZE

4.2.2.5. 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, IR_PROX, and IR_AMBL. The defined threshold types are NO_THRESHOLD_DETECT, THRESHOLD_NOT_MODIFIABLE, THRESHOLD_MODIFIABLE, and GROUP_CALIBRATION. The defined sensor sizes are QS_SENSOR_FINGER and QS_SENSOR_LARGE.

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 |QS_SENSOR_FINGER | NO_THRESHOLD_DETECT,

CS0 |QS_SENSOR_LARGE | THRESHOLD_MODIFIABLE,

};
```

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

Channels that will be bound to control wheels and sliders must have a threshold detection type defined as "GROUP_CALIBRATION".

Sensor size determines behavior of the exponential averaging filter that reduces noise in the sample data.

4.2.2.6. 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.

For infrared channels using the Si1120, the IR LEDs are usually controlled by transistors which allow the IR LED to be measured by the Si1120. Therefore the mux input for IR channels should be the pin that controls the transistor that connects each LED with the Si1120's LED driver. For single IR LED systems, the mux input for the IR channel should be 0xFF, denoting that there is not a transistor connecting the LED to the driver.

For an infrared channel using the Si1102, the mux input for the channel should be 0xFF.

Notes: The number of channels defined in QS_MuxInput[] must equal NUM_CHANNELS.

For ambient light channels, if the system is using an ambient light type group, the mux input must be the ambient light type group index. Otherwise, the mux input must be 0xFF.

4.2.2.7. 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 Performance Analysis Tool. This allows designers to prevent the firmware performance from being affected by too many resources being devoted to data transfer.

Note: Removed from build is SERIAL_INTERFACE is DISABLED or QSCI_LITE

4.2.2.8. QS_IRConfig

Can be set to: For the Si1120:

MODE_PRX400, MODE_OFC, MODE_PRX50, MODE_SLEEP

Please see the Si1120 datasheet for mode settings details.

For the Si1102:

0xFF

Description: This array defines the mode the ambient/proximity light sensing device should be configured to before taking a channel measurement. This array is transferred to the host at enumeration. Index 0 in this array refers to the first IR channel. The indices in this array are not equivalent to the indices of the QS_ChannelInfo array if there are any CS0 channels in the system.

4.2.2.9. QS_GroupType

Can be set to: One array element for each defined group, with values of SLIDER_TYPE, CONTROL_WHEEL_TYPE, GESTURE1D_TYPE, GESTURE2D_TYPE, GESTURE3D_TYPE, AMBLIGHT_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: This array will be included in the project's build only if at least one group is defined in QS_Config.h.

4.2.2.10. 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:

```
SEGMENT_VARIABLE(ControlWheel1[], U8, SEG_CODE) =
{
 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:

```
SEGMENT_POINTER(QS_GroupChannelAssignementTable[], U8, SEG_CODE) =
{
 ControlWheel1
};
```

For gesture groups, the array should contain the index of the pad(s) that generates the gesture.

For ambient light type groups, the array should contain the channel that is measuring the ambient light intensity and type.

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

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.

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[].

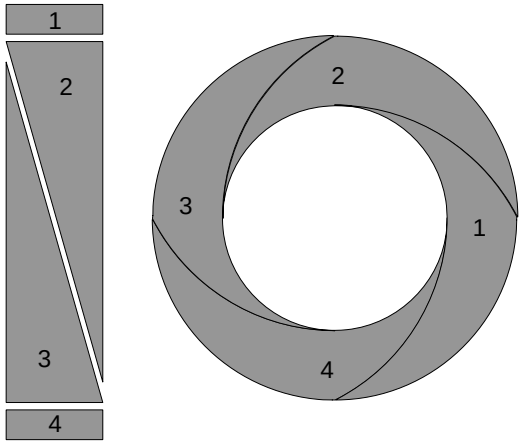

Figure 4. Slider and Control Wheel Channel Assignment Order

4.2.2.11. QS_*n*DPadCapabilities

Can be set to: One element per defined *n*D pad. Contains pointers to QS_*n*DPadCapStruct structures.

Description: This array links *n*D pad capability structures to *n*D pad indices.

4.2.2.12. QS_*n*DPadChannels

Can be set to: One element per defined *n*D pad, where each element is a pointer to a channel list array which contains the channels that are attached to the pad.

Description: This array defines the channels that make up a particular *n*D pad. Each pad index should have an array which lists the channels that make up the particular pad. For example, a system might have a 2D pad that uses channels 2, 3, and 5. The channel list array for this pad might look like the following:

```
SEGMENT_VARIABLE (Pad2D_0[], U8, SEG_CODE) = {
3, // Number of channels
2, 4, 5 // Channel indices};
```

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_*n*DPadChannels as follows:

```
SEGMENT_VARIABLE (QS_2DPadChannels[], U8, SEG_CODE) = {
{
Pad2D_0,
}
```

4.2.3. QS_DeviceInit.c

This file contains the hardware initialization routines for all peripherals used in the system. For the correct hardware settings, refer to the MCU's data sheet. If the developer is using the QuickSense Studio, hardware configuration can be done through the Config2 graphical interface and copied into this file.

The functions that must be populated are as follows:.

PORT_Init() - Initialize the ports on the MCU

SYSCLK_Init() - Initialize the oscillator to run at 24.5 MHz.

UART0_Init() - Initialize the UART to run at 230400 baud, 8-N-1.

Timer0_Init() - Initialize Timer 0 to interrupt every 100 µs (10 kHz).

Other MCU peripheral initializations should be done in this file. All of these initialization functions are referenced in the Init_Device() function which should be called in the application layer. Any other initializations made to the project should be included in Init_Device().

4.2.4. QS_ExtDeviceConfig.h

This file contains external device configurations. The QuickSense Firmware API needs to know where the mode and output pins on the external device are connected to the MCU. The current external devices supported by the API are the Si1120 and the Si1102.

The following must be defined for each ambient/proximity light sensing device:

Si1102:

Si1102_PRX - The MCU pin where the Si1102's PRX output pin is connected.

Si1120:

Si1120_PRX - The MCU pin where the output of the Si1120 is connected.

Si1120_MD - The MCU pin which controls the MD line on the Si1120.

Si1120_STX - The MCU pin which drives the STX line for sampling.

Si1120_SC - The MCU pin which drives the SC line for mode changes.

4.2.4.1. STX_MAX_HIGH_TIME

Can be set to: Numeric positive integer within the range of 500-2000 rounded to the nearest 100.

Description: The longest the STX pin on the Si1102 is asserted during IR measurements. De-asserting this pin causes the Si1120 to stop IR measurement for the specific irLED. This can reduce the power used by the Si1102. Note: STX_MAX_HIGH_TIME x number of irLEDs x 4 is the recommended IR_SAMPLING_RATE.

Note: This only applies to the Si1102 IR device.

4.2.4.2. AMBLIGHT_ENABLE

Can be set to: ENABLED or DISABLED.

Description: Controls whether the ambient light calculation algorithm is included in the firmware build.

Note: If the ambient light calculation algorithm is included in code, the firmware should also create an AMBLIGHT channel and an AMBLIGHT group to store the data output from the algorithm.

5. QuickSense Firmware API Specification

5.1. System Block Diagram

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

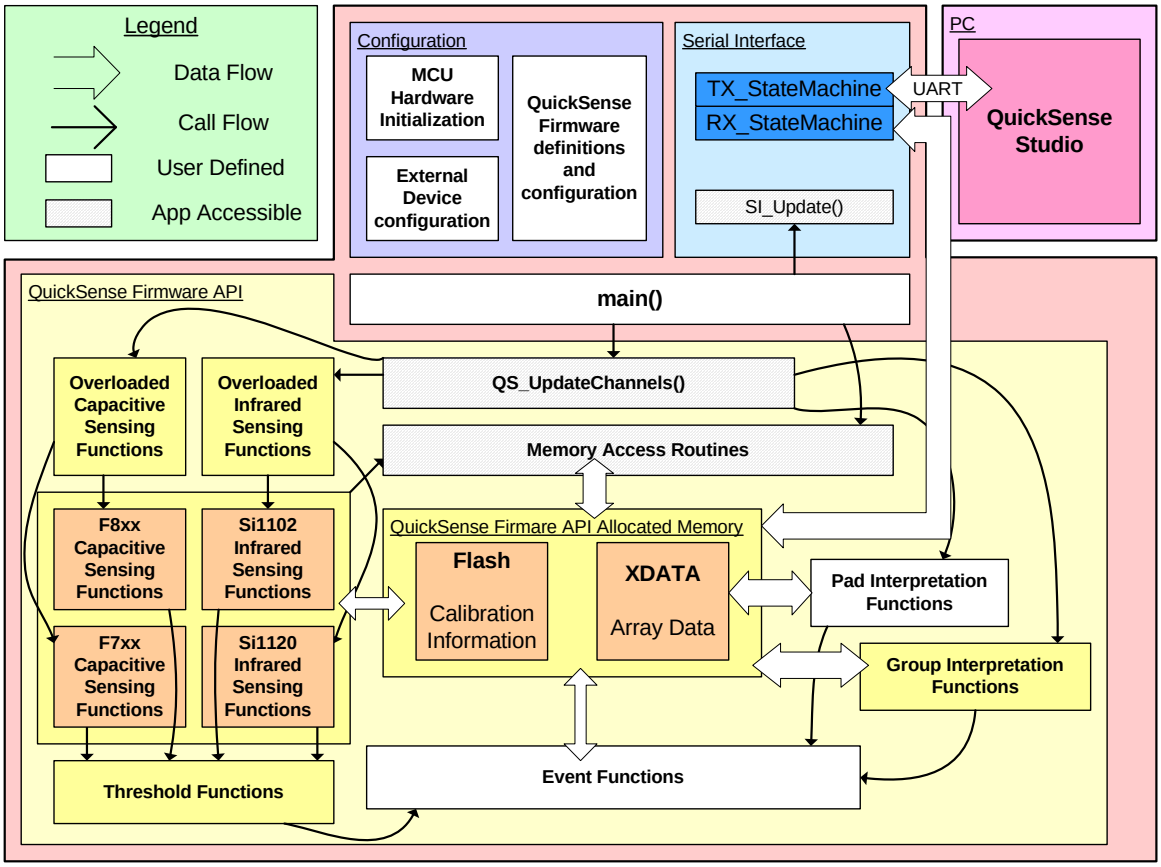

Figure 5. Block Diagram of Application using QuickSense Firmware API

The routines take measurements on channels and use interpretive functions to process the data and make updates to threshold states, group values, and nD pad 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 major functions interact with each other.

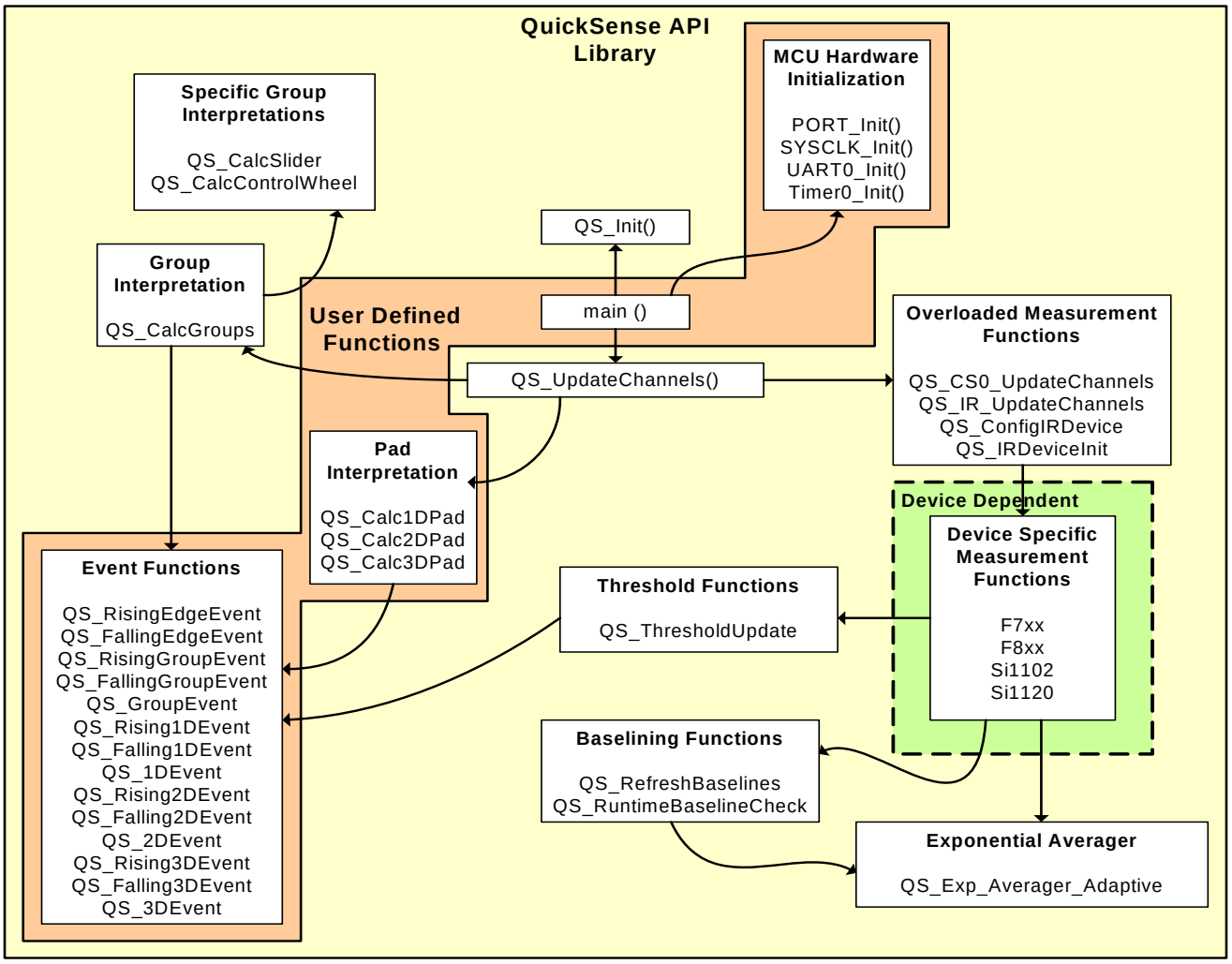

Figure 6. Function Call Graph

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_GetSensorSize`	- Returns configured size of channel's capacitive sensing pad.
`QS_GetChannelValue`	- Returns the value measured at the channel.
`QS_SetChannelValue`	- Writes value to channel's memory.
`QS_GetMuxConfig`	- Returns the mux configuration for the channel.
`QS_GetThrCapabilities`	- Returns threshold detection information.
`QS_GetToActiveThr`	- Returns inactive to active threshold percentage for a channel.
`QS_GetToInactiveThr`	- Returns active to inactive threshold percentage for a channel.
`QS_SetToActiveThr`	- Set inactive to active threshold percentage for a channel.
`QS_SetToInactiveThr`	- Set active to inactive threshold percentage for a channel.
`QS_EraseFlash`	- 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_GetGroupValue`	- Returns a group's interpreted position value.
`QS_SetGroupValue`	– Writes 2-byte position value 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_GetReferenceBaselineMagnitude`	- Writes reference magnitude to memory.
`QS_SetReferenceBaselineMagnitude`	- Returns reference magnitude to memory.
`QS_SetBaselineUpdateRate`	- Updates the runtime baseline update rate.
`QS_GetBaselineUpdateRate`	- Returns the runtime baseline update rate.
`QS_SetChannelCalState`	- Sets array value describing whether channel has been calibrated.
`QS_GetChannelCalState`	- Returns array value describing whether channel has been calibrated.
`QS_SetProcessEnable`	- Enables/disables defined QuickSense process for a data type index.
`QS_SetAllProcessEnable`	- Enables/disables defined QuickSense process for all of a data type.
`QS_GetProcessEnable`	- Returns whether a process is enabled or disabled
`QS_GetIRConfig`	- Returns element of IR Config array describing an IR channel.
`QS_Get`*n*`DPoint`	- Gets a *n*D point element in the *n*D point array.
`QS_Set`*n*`DPoint`	- Sets an *n*D point element in the *n*D point array to a value.
`QS_Clear`*n*`DPoint`	- Clears a *n*D point element in the *n*D point array.
`QS_Clear`*n*`DPadPoints`	- Clears all points in *n*D array.
`QS_NumValid`*n*`DPoints`	- Returns number of active points in *n*D array.
`QS_SetGenericData`	- Sets a 16-bit value to a generic data element.
`QS_GetGenericData`	- Returns a generic data element value.

5.2.2. Core 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 and IR channels.
- `QS_ThresholdUpdate` - Update the threshold state of a channel.
- `QS_ChannelCalCheck` - Return whether channel has stored calibration values.
- `QS_GetScaledChannel` - Converts a channel value to a percentage of the runtime baselines.

5.2.3. Group Routines

These routines interpret channel values bound to groups and output group positions. Note that, although gesture values are stored in the group position array, they do not get calculated by these functions. Instead, gestures are calculated within *n*D pad routines.

- `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_ScaleChannelInGroup` - Finds the linear 0–127 scaling of a channel's value between its inactive and active threshold.

5.2.4. Exponential Averaging Routines

These routines perform an exponential average using a new sample, the exponential average of samples, and an averaging constant that controls how much weight to place on the new sample. The equation used by the functions is shown in Equation 1.

$$y(n) = (x(n) - y(n-1)/2^m + y(n-1)$$

Equation 1.

where m equals the averaging constant, x(n) equals the newest sample, y(n-1) is the previously exponentially averaged sample set, and y(n) is the new exponentially averaged value.

- `Exp_Averager_Adaptive` - Performs exponential average.

5.2.5. 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_UpdateChannels` - Update all CS0 measured channels.
- `QS_IR_UpdateChannels` - Update all IR measured channels.
- `FLASH_ByteWrite` - Write a byte into flash.
- `FLASH_PageErase` - Erase a page of flash.
- `Assert_Pin` - Asserts an MCU pin to a logical value of 1.
- `Clear_Pin` - De-asserts an MCU pin to a logical value of 0.

5.2.6. 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, see AN418.

`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.

5.2.7. 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.
`CheckChannelList`	- Checks if an index is in a channel list.
`EnableChannelList`	- Enables all channels in a channel list.

5.2.8. Event 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.
`QS_`*n*`DPadEvent`	- Callback function for when an nD pad is active.
`QS_Rising`*n*`DPadEvent`	- Callback function for when an nD pad was inactive and is now active.
`QS_Falling`*n*`DPadEvent`	- Callback function for when an nD pad was active and is now inactive.

5.2.9. Timing Controls

These routines control sampling rates for capacitive sensing and ambient/proximity light sensing. They also provide a global timer that can be read by application layer functions.

`QS_GlobalCounter` - Global counter variable that increments at a rate of 100 µs.

`QS_GlobalCounterOverflow` - Global counter variable that increments at a rate of 100 µs x 128 or 12.8 ms.

5.2.10. *n*D Pad Functions

These routines are only stub functions in the QuickSense Firmware API template file. They are filled in with algorithm-specific code, either by the QuickSense Configuration Wizard, or by the user. These routines are called after measurements have completed for all defined channels. Similar to group calculation routines, *n*D pad routines take captured channel values and interpret them to determine the position of a conductive material such as a finger in a defined *n*D physical space.

`QS_Calc`*n*`DPads` - Routines that can calculate position on a defined nD pad space.

5.2.11. External Device Routines

These routines provide the communication interface between the MCU and external sensing devices. At this time, supported devices are as follows:

- Si1102
- Si1120

`Config_IRDevice` - Configures external device hardware to modes defined by IR_Config array elements.

`QS_IRDevice_Init` - Resets external device to a known, default state

`QS_IR_UpdateChannels` - Updates all IR defined channels.

5.2.12. Low Power Routines

These routines provides the application layer functions to modify wake up sources and to place the low power device in a certain low power mode.

`LPM_Init` - Initializes low power functionality.

`LPM_Enable_Wakeup` - Enables the specified wakeup sources.

`LPM_Disable_Wakeup` - Disables the specified wakeup sources.

`LPM` - Places the MCU in a certain low power mode.

5.2.13. SmaRTClock Routines

These routines provide abstracted access to the SmaRTClock peripheral; registers in the RTC peripheral are indirectly addressed.

`RTC_Init` - Initializes the SmaRTClock peripheral.

`RTC_Read` - Reads from a SmaRTClock register.

`RTC_Write` - Writes to a SmaRTClock register.

`RTC_WriteAlarm` - Write a specific alarm value to wake up on.

`RTC_GetCurrentTime` - Read the 32-bit current time from the RTC counter.

`RTC_SetCurrentTime` - Write a 32-bit time to the RTC counter.

`RTC_ZeroCurrentTime` - Zero out the 32-bit RTC counter.

`RTC0CN_SetBits()` - Set certain bits in RTC0CN.

`RTC0CN_ClearBits()` - Clear certain bits from RTC0CN.

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 threshold percentages, reference baseline magnitude, baseline update rate
  - 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

The subsections that follow cover different memory types and describe each variable and definition individually.

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* CODE_CONTENTS[]                     - Addresses for configuration variables.
U8* NVCCA_CONTENTS[]                    - Addresses for NVCCA variables.
U8* XDATA_CONTENTS[]                    - Addresses for runtime variables.
U8 FIRMWARE_REV_CAP[]                   - Firmware revision and capabilities for the system.
U8 QS_MuxInput[]                        - Mux configuration for each channel.
U8 QS_GroupType[]                       - Defines the types of groups being use.
U8* QS_GroupChannelAssignmentTable      - Maps channels to each group.
UU16 QS_Threshold[]                     - Threshold percentages for each channel.
U8 QS_ChannelInfo[]                     - Threshold Capabilities and other information.
UU16 QS_ReferenceBaseline[]             - Stores reference baseline magnitudes for all channels.
U8 QS_BaselineUpdateRate                - Stores rate at which runtime baselines are refreshed.
U8 NVCCA_PADDING[]                      - Allocates the rest of the NVCCA page.
QS_nDPadCapStruct QS_nDPadCapabilities  - Stores the capabilities for each nD pad.
U8 QS_nDPadChannels[]                   - Maps channels to an nD pad.
U8 QS_IRConfig[]                        - Stores the IR configuration for each IR channel.
```

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:

- `UU16 QS_ChannelValue[]` - Last measured value of the channel.
- `U8 QS_ThresholdState[]` - Current threshold state of the channel.
- `UU16 QS_GroupValue[]` - Current group position.
- `UU32 QS_RuntimeBaseline[]` - Saves runtime inactive and active baselines.
- `U8 QS_ChannelCalibration[]` - Bit array which signifies if a channel index is calibrated.
- `U8 QS_ChannelProcessEnable[]` - Bit array which signifies if a channel index is enabled for sampling.
- `U8 QS_ThresholdProcessEnable[]` - Bit array which signifies if a channel index is enabled for threshold processing.
- `U8 QS_GroupProcessEnable[]` - Bit array which signifies if a group index is enabled for processing.
- `U8 QS_nDProcessEnable[]` - Bit array which signifies if an nD pad is enabled for processing.
- `QS_nDPadPointStruct QS_nDPadPoints[]` - Array which contains nD pad coordinate data.
- `U16 QS_GenericData[]` - Array containing application specific data.

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 "5.2.1. Memory Access Routines" on page 25.

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 "4. Designing with the API" on page 9.

5.4.1. QS_Config.h Definitions that should be Modified

- NUM_CS0_CHANNELS - Number of channels measured by CS0 module.
- NUM_IR_CHANNELS - Number of channels measured by light sensing device.
- NUM_CONTROLWHEELS - Defines the number of control wheels being used.
- NUM_SLIDERS - Defines the number of sliders being used.
- NUM_GESTURES - Defines the number of gestures being used.
- NUM_*n*D_PADS - Sets number of pads in *n*D pad array.
- NUM_GDATA_ELEMENTS - Defines the number of generic data elements.
- MCU - Define which MCU is being used.
- IR_DEVICE - Includes/removes prox/ambient light sensing code.
- FLASH_MODE - Includes/removes Flash write/erase routines.
- GROUP_VALUE_SAVE - Enable/Disable the saving of intermediate group values
- CHANNEL_THRESHOLD - Enables/Disables thresholds comparison and allocation.
- ACTIVE_BASELINE_DECAY_PERCENT - Defines minimum range active baseline can drop to.
- ACTIVE_BASELINE_MAX_PERCENT - Defines maximum range active baseline can rise to.
- HISM - High Inactive Sensitivity Margin for baselining.
- LISM - Low Inactive Sensitivity Margin for baselining.
- MAX_*n*D_POINTS - Sets max number of simultaneous points in pad array.
- EXTENDED_*n*D_ENABLE - Not implemented.
- SWITCH_EVENTS - Enable/Disable rising/falling edge switch events.
- GROUP_EVENTS - Enable/Disable group events.
- PAD_*n*D_EVENTS - Enables/Disables pad callback functions.
- IR_SAMPLE_RATE - Configures sample period for prox/ambient sensor.
- CS0_SAMPLE_RATE - Configures sample period for capacitive sensing.
- AVERAGING_EXPONENT - Controls exponential averaging function.
- SERIAL_INTERFACE - Enables/disables serial interface in code.
- QSCI_HARDWARE - Controls which hardware is used for QSCI.
- RX_BUFFER_SIZE - Controls the size of the receive buffer for QSCI.
- ENUMERATION_INFO - Enables/disables transfer of enumeration structures.
- TRANSFER_MODE_SUPPORTED - Enables/disables transfer modes.
- TRANSFER_TYPE_SUPPORTED - Enables/disables transfer types.
- EXTERNAL_TIME_MANAGEMENET - Enables/disables QuickSense timing.
- CHANNEL_AVERAGING - Enables/disables exponential averaging for channels.
- LOW_POWER - Enables/disables low power functionality.
- WAKEUP_INTERVAL - Defines the wake up period in a low power system.
- RTC_CLKSRC - Defines the clock source for the SmaRTClock.
- LOADCAP_VALUE - Defines the load capacitance for the RTC clock source.

5.4.2. Definitions that should not be Modified

QS_BYTES_PER_PAGE	-	Defines the number of bytes per flash page.
QS_NVCCA_LOCATION	-	Defines the location of the NVCCA page.
CHANNEL_ELEMENT_SIZE	-	Defines the number of bytes per channel element.
GROUP_ELEMENT_SIZE	-	Defines the number of bytes per group element.
RUNBASE_ELEMENT_SIZE	-	Defines the number of bytes per runtime baseline element.
GENERIC_ELEMENT_SIZE	-	Defines the number of bytes per generic data element.
THRESHOLD_ELEMENT_SIZE	-	Defines the size of threshold percentage elements.
REFBASE_ELEMENT_SIZE	-	Defines the size of reference baseline magnitude elements.
BASE_UPRATE_SIZE	-	Defines the size of the Baseline Update Rate.
COORDINATE_SIZE	-	Defines the number of bytes per coordinate value.
PAD_*n*D_ELEMENTS	-	Defines the number of coordinates per pad.
PAD_nD_CAP_SIZE	-	Defines the number of bytes for nD pad capabilities.
CS0_CHANNEL_INDEX	-	Defines the starting index for CS0 channels.
IR_CHANNEL_INDEX	-	Defines the starting index for IR channels.
NUM_CHANNELS	-	Total number of channels in the system.
NUM_GROUPS	-	Defines the total number of grouped channels.
QS_NUM_FLASH_PAGES	-	Number of flash pages NVCCA variables occupy.
THRESHOLD_STATE_SIZE	-	Defines the number of bytes reserved for threshold states.
NUM_THRESHOLDS_SIZE	-	Defines the size of the threshold buffer in bytes.
NUM_REFBASE_SIZE	-	Defines the size of the runtime baseline magnitude array.
CHANNEL_VALUE_SIZE	-	Defines the size of QS_ChannelValue[].
RUNTIME_BASELINE_SIZE	-	Defines the size of QS_RuntimeBaseline[].
GROUP_VALUE_SIZE	-	Defines the size of QS_GroupValue[].
PAD_nD_POINT_SIZE	-	Defines the size of QS_*n*DPadPoints[].
GENERIC_DATA_SIZE	-	Defines the size of QS_GenericData[].
THRESHOLD_STATE_SIZE	-	Sizes threshold state bit array.
CHANNEL_CAL_SIZE	-	Sizes channel calibration bit array.
CHANNEL_BITARRAY_SIZE	-	Sizes bit arrays for the channels.
BASELINE_BITARRAY_SIZE	-	Sizes bit arrays for the baselines.
THRESHOLD_BITARRAY_SIZE	-	Sizes bit arrays for thresholds.
GROUP_BITARRAY_SIZE	-	Sizes bit array for groups..
PAD_nD_BITARRAY_SIZE	-	Sizes bit arrays for nD pads.
QS_CHANNEL_VALUE_LOCATION	-	Defines the start address for QS_ChannelValue[].
QS_RUNTIME_BASELINE_LOCATION	-	Defines the start address of QS_RuntimeBaseline[].
QS_GROUP_VALUE_LOCATION	-	Defines the start address of QS_GroupValue[].
QS_nD_POINT_LOCATION	-	Defines the start address of QS_*n*DPadPoints.
QS_GENERIC_DATA_LOCATION	-	Defines the start address of QS_GenericData[].
QS_THRESHOLD_STATE_LOCATION	-	Defines the start address of QS_ThresholdState[].
QS_CHANNEL_CAL_LOCATION	-	Defines the start address of QS_ChannelCalibration[].
QS_CHANNEL_PROCESS_LOCATION	-	Defines the start address of QS_ChannelProcessEnable[].
QS_THRESHOLD_PROCESS_LOCATION	-	Defines the start address of QS_ThresholdProcessEnable[].
QS_GROUP_PROCESS_LOCATION	-	Defines the start address of QS_GroupProcessEnable[].
QS_nD_PROCESS_LOCATION	-	Defines the start address of QS_*n*DProcessEnable[].
QS_FLASH_KEY_LOCATION	-	Defines the start address of QS_FlashKey[].
QS_REFERENCE_BASELINE_LOCATION	-	Defines the start address of QS_ReferenceBaseline[].
QS_BASELINE_UPDATE_RATE_LOCATION	-	Defines the start address of QS_BaselineUpdateRate.

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.c/.h	- Get/Set routines, memory allocation.
QS_Baselining.c/.h	- Baselining routines.
QS_Core.c/.h	- System level routines.
QS_Groups.c/.h	- Group interpretation routines.
QS_MCURoutines.h	- Overloaded functions.
QS_F70xRoutines.c	- F70x-F71x implementations and overloaded functions.
QS_F80xRoutines.c	- F80x-F83x implementations and overloaded functions.
QS_F91xRoutines.c	- F90x-F91x implementations and overloaded functions.
QS_F93xRoutines.c	- F92x-F93x implementations and overloaded functions.
QS_F99xRoutines.c	- F99x implementation and overloaded functions.
QS_SI1102.c	- Device implementations of overloaded functions.
QS_SI1120.c/.h	- Device implementations of overloaded functions.
QS_Timing.c/.h	- Controls timers and sampling rates.
QS_Exp_Averager.c/.h	- Exponential averaging routines.
QS_DeviceInit.h	- MCU initialization routine prototypes.
QS_ExtDevice.h	- Overloaded functions for external devices.
QS_LowPower.c/.h	- QuickSense version of low power library.
QS_SmaRTClock.c/.h	- QuickSense version of SmaRTClock library.
QSCI_Definitions.h	- General QSCI/QSCI Lite definitions.
QSCI_Core_Lite.c	- Communications routines for QSCI Lite.
QSCI_UARTRoutines.c	- Defines UART ISR for QSCI.
QSCI_UARTRoutines_Lite.c	- Defines UART ISR for QSCI Lite.
QSCI_Memory_Lite.c	- Defines variables for QSCI Lite.
QSCI_Core.c/.h	- Communication routines for QSCI.
QSCI_MCURoutines.h	- Header for communication protocols.
QSCI_Memory.c/.h	- Declares variables used by QSCI.

The following files need to be modified:

QS_ExtDeviceConfig.h	- Configuration of external device.
QS_Config.h	- Contains control panel.
QS_Config.c	- Code variables need to be filled.
QS_Events.c	- Includes all callback functions.
QS_nDPad.c	- Includes stub functions for nD pad algorithms.
QS_DeviceInit.c	- MCU initialization routines.

For details on how the two configuration files need to be modified, see "3. QuickSense Firmware API Overview" on page 1.

5.5.1. File Structure

Figure 7 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.

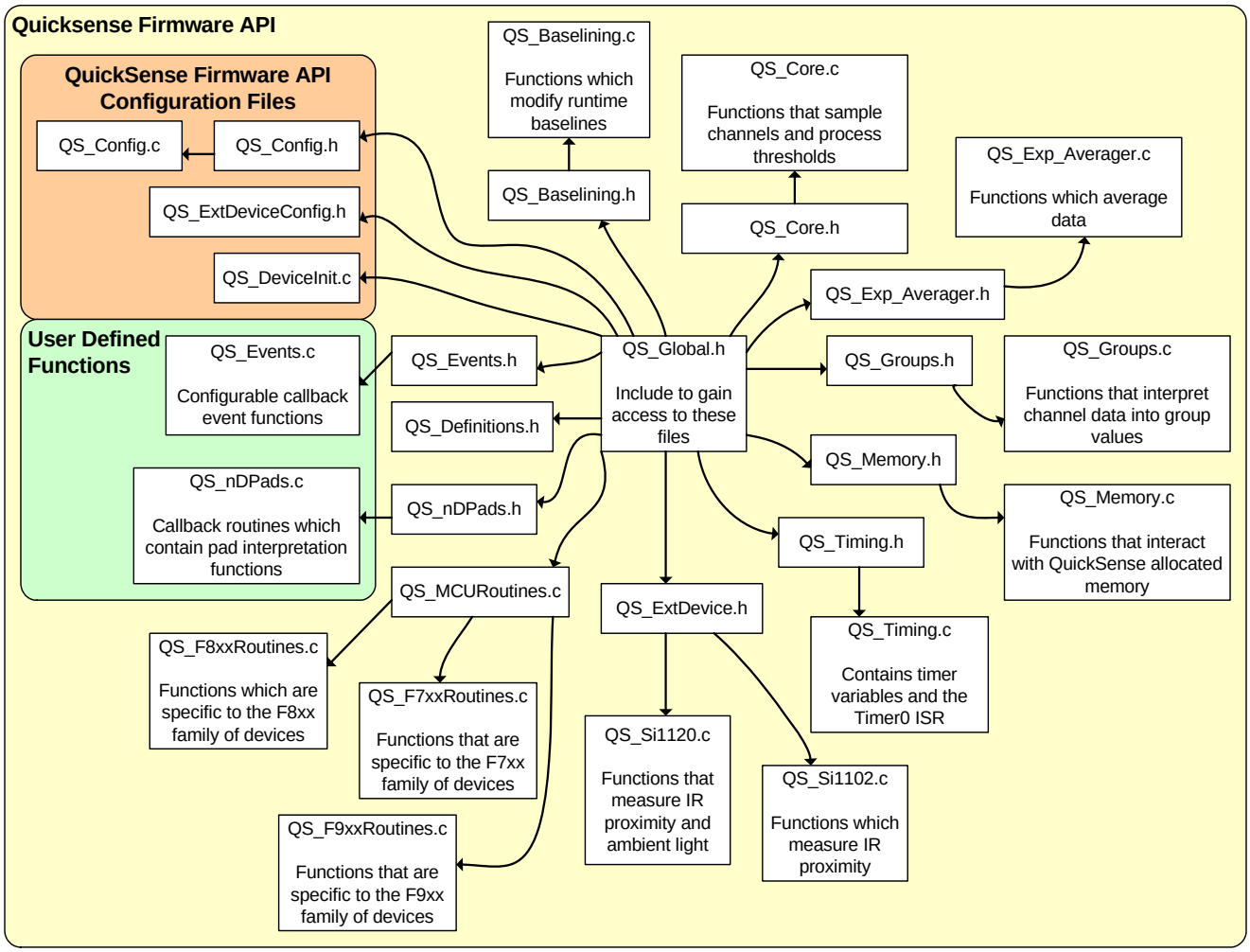

Figure 7. File and Functioning Mapping

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 Performance Analysis Tool. Add: Timer 0 is free when EXTERNAL_TIME_MANAGEMENT is ENABLED.
- CS0—Converts capacitance found on input channels.
- PCA0—Used to interface with the Si1120.
- RTC0—Provides system timing for F9xx devices when LOW_POWER is ENABLED.

NOTES:

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.