Application Note

Common Software Use Case Examples with TI Smart Fuse High-Side Switches

Timothy Logan

ABSTRACT

The smart fuse high-side switch portfolio (HCS device family) from Texas Instruments (TI) provides a versatile and powerful set of devices that allow for seamless replacement of physical melting fuse devices with a configurable semiconductor design in automotive applications. The HCS high-side switch device family uses SPI as communication to configure various parameters such as capacitive charging and I2T, and to read out diagnostics such as current sense (through an integrated ADC) and fault detection. To simplify software development, TI provides a full suite of software drivers and code examples across multiple processor or microcontroller platforms (both from TI and otherwise). This application note describes all the aspects of the smart fuse software ecosystem from TI, including the top level drivers, configuration or evaluation tool, and a full set of code examples that show common use cases of the underlying smart fuse devices.

Table of Contents

Trademarks

1 Software Ecosystem

The smart fuse ecosystem from TI consists of the following software components.

Table 1-1. Smart Fuse Collateral Offering

Collateral Name	Description
Smart Fuse Configurator	Host GUI tool used to configure TPSxHCxx-Q1 device and export C configuration file for software development. This software is also used to control the HSS-HCSMOTHERBRDEVM and corresponding daughter cards.
Device Specific C Headers Files	Header file representation of the HCS device's register map. This file contains all register definitions and enumerations of the device.
HCS Platform Drivers	Generic set of drivers with a lower level SPI driver porting layer. Implementation examples are provided for a variety of processors/microcontrollers.
Application Code Examples	Common set of code examples that show case some common functionality and differentiation of using the HCS family of high-side switches from Texas Instruments.

A software package containing the HCS Platform Drivers and Application code examples can be found on the corresponding software page. The drivers and code examples are BSD licensed open source allowing for flexible porting/re-use. The software package can be found at HCS-SMARTFUSE-DRIVERS.

Note that the functionality of the Smart Fuse Configurator software with relation to the HSS-HCSMOTHERBRDEVM are detailed in the *Smart Fuse Evaluation Module*, user's guide.

All pieces of the software collateral are designed to work with each other, simplify software development, and provides easy instruction to get started with the HCS family of high-side switches. A standard development flow can include the following steps:

1. Use the Smart Fuse Configurator to generate the initial configuration. These settings are loaded to the high-side switch over SPI by the microcontroller during boot-up. The settings include current limit configuration, capacitive charge modes, and any specific I2T turning required based on the specific wire gauge curve.
2. Once configured, the software exports a generic C files that contain a generic C structure that is used for initial configuration or programming.
3. A pointer of this configuration structure is passed into the HCS_initializeDevice function (Section 2.4.9). The driver then programs all of the relevant registers of the device. This API is typically called once on the microcontroller boot or initialization.

Each one of these steps and individual components are described in the following sections.

2 Platform Drivers

2.1 Driver Concept

The suite of drivers that Texas Instruments provides for the HCS family of smart fuse high-side switches were designed to be generic and allow for complete configuration and utilization of the underlying device. Features of these drivers include:

- Initial configuration through exported file from the Smart Fuse Configurator software.
- Generic register reads or writes with support for returning individual transaction headers.
- Convenience functions that convert raw ADC results from the high-side switch's ADC registers to a human readable float value.
- Functions that provide both device and channel level diagnostic status.

The drivers and relation to the provided code examples are shown in Figure 2-1.

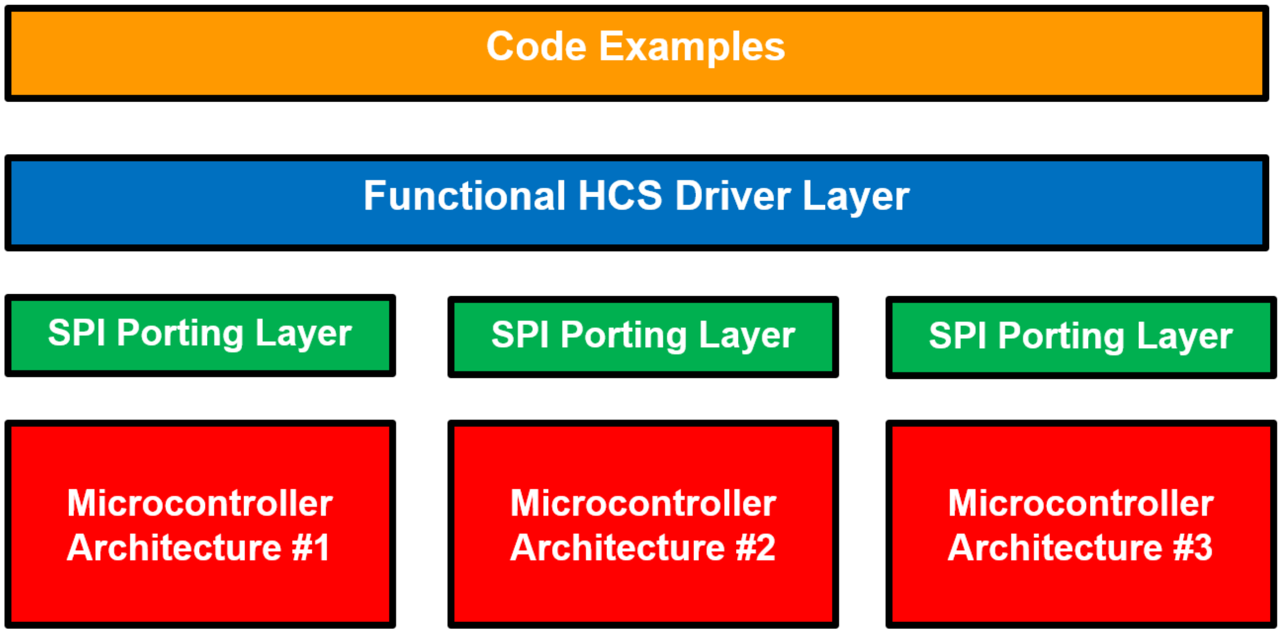

Code Examples
Functional HCS Driver Layer
SPI Porting Layer
SPI Porting Layer
SPI Porting Layer
Microcontroller Architecture #1
Microcontroller Architecture #2
Microcontroller Architecture #3

Figure 2-1. Driver Architecture

The top level driver APIs have the HCS_ prefix to signify the HCS family of smart fuse devices and are provided in ***hcs_control_driver.h*** and ***hcs_control_driver.c*** from the software package. The top level code examples use these APIs to provide a generic set of functionality to control and configure the high-side switch. For the physical SPI communication, a set of external functions are declared in ***hcs_control_driver.h*** :

```
/* --------------------------- Porting Functions ---------------------------- */
/*
 * These functions need to be implemented by each individual device port. The functions
 * handle the low-level hardware specific implementation with the respective
 * architecture's specific hardware peripherals (SPI and GPIO)
 */
bool HCS_port_spiSendData(uint8_t *data, uint8_t len, uint8_t* respData);
void HCS_port_assertSPI(void);
void HCS_port_deassertSPI(void);
```

These functions are defined in each individual architecture implementation and handle the hardware SPI interaction of each platform. More details of porting these functions to additional architectures can be found in Porting to Other Platforms (Section 2.3). A full list of APIs including functionality, parameters, and return values can be found in the API guide (Section 2.4).

2.2 Supported Platforms

The smart fuse driver and configuration package contains example implementations for a variety of different microcontrollers or processors. Additional implementations are routinely added, however a list of current architectures and the relevant development kits are shown in Table 2-1:

Table 2-1. Supported Platforms

Architecture	Development Board	Ecosystem Notes
Texas Instruments MSPM0G3507-Q1	LP-MSPM0G3507	Code Composer Studio Theia
STMicroelectronics STM32H723ZGT6	NUCLEO-H723ZG	STM32CubeIDE

Additional device support can be added by implementing the relevant device level functions described in Section 2.3.

The HSS-HCMOTHERBRDEVM has generic SPI headers that can be used to *plug in* external SPI signals to the attached high-side switch daughter card. During development of the code examples and HCS platform drivers, the development boards listed below were used in tandem with the HSS-HCMOTHERBRDEVM for validation. An example of the LP-MSPM0G3507 evaluation module used in this configuration can be found below in Figure 2-2:

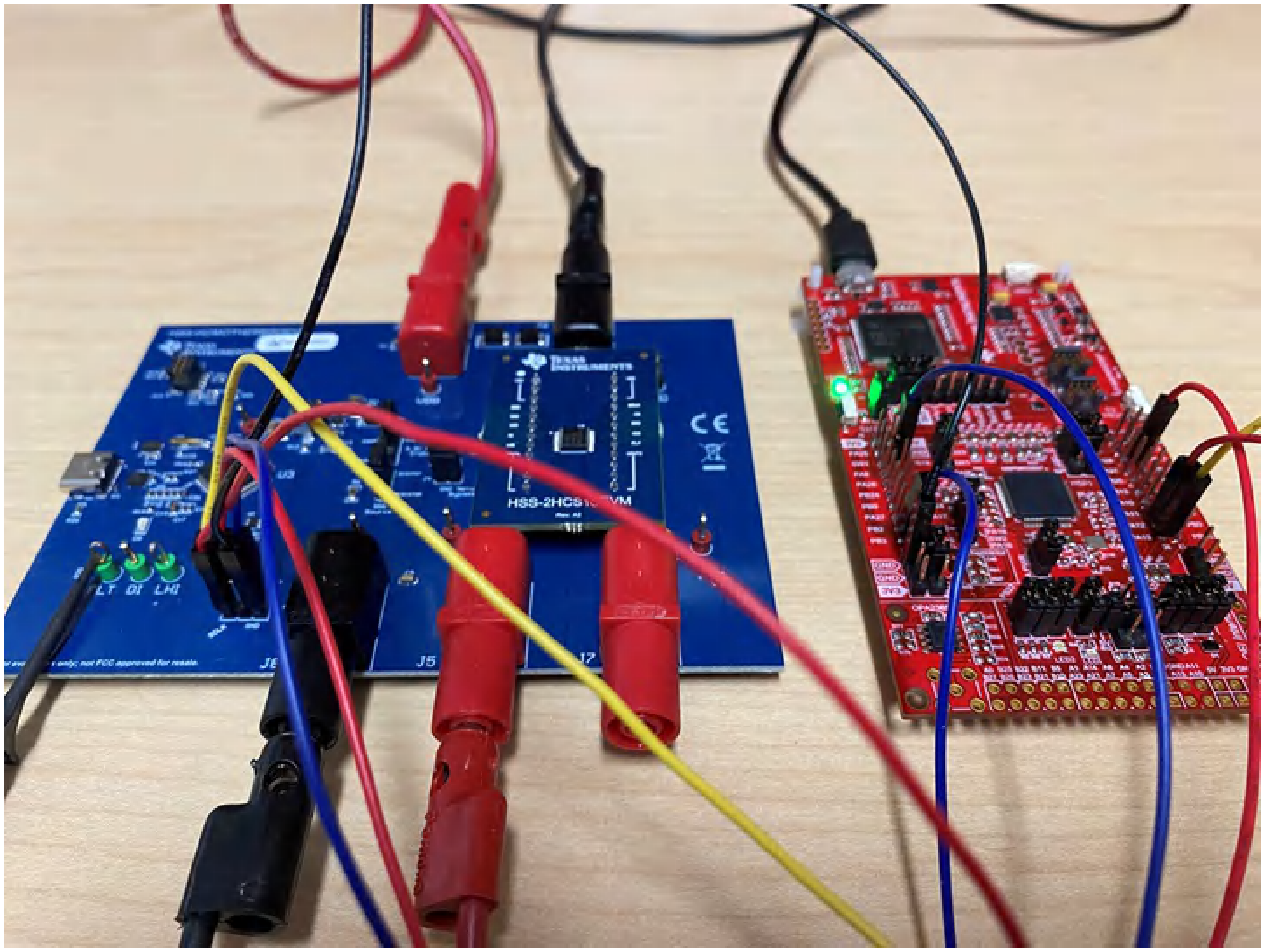


Figure 2-2. Connected MSPM0+

The following shows a connection using the NUCLEO-H723ZG board:

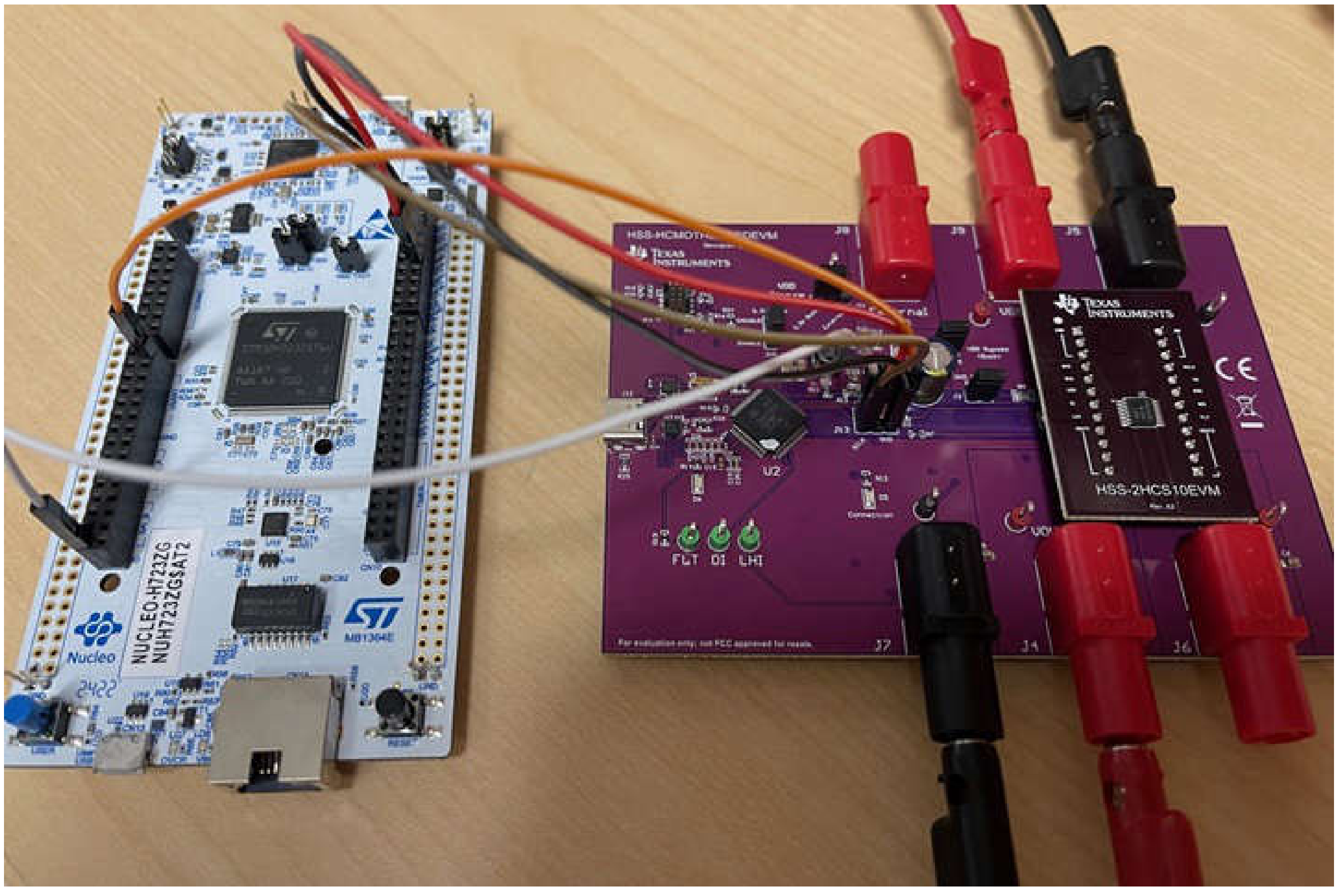


Figure 2-3. NUCLEO-H723ZG Connection

2.3 Porting to Other Platforms

While a variety of example architecture implementations are provided, the HCS platform drivers are architected to be easily ported to any microprocessor or architecture that supports SPI and the C programming language. In the header file for the drivers (***hcs_control_driver.h***), the following two functions are declared with an external reference:

```
/* --------------------------- Porting Functions ---------------------------- */
/*
 * These functions need to be implemented by each individual device port. The functions
 * handle the low-level hardware specific implementation with the respective
 * architecture's specific hardware peripherals (SPI and GPIO)
 */
bool HCS_port_spiSendData(uint8_t *data, uint8_t len, uint8_t* respData);
void HCS_port_assertSPI(void);
void HCS_port_deassertSPI(void);
```

Porting is meant to be straight forward, however a description of each function and guidance for porting is shown in Table 2-2.

Table 2-2. Architecture Porting Functions

Function	Description	Return Value
HCS_port_spiSendData	Performs a full-duplex SPI transaction. ***data*** represents the data to be sent and ***respData*** is the data that is received. ***len*** is the length of the transaction. The function needs to be blocking (either by sleeping or polling) until the entirety of the SPI transaction is complete.	stdbool representation of result of transaction. ***true*** if transaction completed without issue, ***false*** otherwise.
HCS_port_assertSPI	Sets the chip select (CS) line of the SPI bus low. This is used for both single and daisy chained transactions as well as used by the HCS_port_spiSendData function above.	None
HCS_port_deassertSPI	Sets the chip select (CS) line of the SPI bus high. This is used for both single and daisy chained transactions as well as used by the HCS_port_spiSendData function above.	None

2.4 API Guide

References

- tHCSResponseCode Union (Section 2.4.1)
- HCS_convertCurrent (Section 2.4.2)
- HCS_convertTemperature (Section 2.4.3)
- HCS_convertVoltage (Section 2.4.4)
- HCS_getChannelFaultStatus (Section 2.4.5)
- HCS_getDeviceFaultSatus (Section 2.4.6)
- HCS_gotoLPM (Section 2.4.7)
- HCS_gotoSleep (Section 2.4.8)
- HCS_initializeDevice (Section 2.4.9)
- HCS_readRegister (Section 2.4.10)
- HCS_setSwitchState (Section 2.4.11)
- HCS_updateConfig (Section 2.4.12)
- HCS_wakeupDevice (Section 2.4.13)
- HCS_writeRegister (Section 2.4.14)

2.4.1 tHCSResponseCode Union Reference

Data Fields

```
typedef union
{
    uint8_t byte;
    struct
    {
        unsigned I2T_FLT : 1;
        unsigned LPM_FLT : 1;
        unsigned CHAN_TSD : 1;
        unsigned ILIMIT_FLT : 1;
        unsigned SHRT_VBB_FLT : 1;
        unsigned OL_FLT : 1;
        unsigned SUPPLY_FLT : 1;
        unsigned GLOBAL_ERR_WRN : 1;
    } bits;
} tHCSResponseCode;
```

2.4.2 float_t HCS_convertCurrent (uint16_t rawValue, uint16_t ksnsVal, uint16_t snsRes)

Converts a raw ADC current value to a readable float value.

This is a convenience function that can take a raw value read from one of the ADC result registers and convert the register into a human readable float value.

Table 2-3. Parameters

in	*rawValue*	Raw value to convert
in	*ksnsVal*	KSNS constant from the data sheet
in	*snsRes*	Value of resistor on the SNS pin

Table 2-4. Return Values

returnCode	Floating point representation of current

2.4.3 float_t HCS_convertTemperature (uint16_t rawValue)

Converts a raw ADC temperature value to a readable float value.

This is a convenience function that can take a raw value read from one of the ADC result registers and convert into a human readable float value.

Table 2-5. Parameters

in	*rawValue*	Raw value to convert

Table 2-6. Return Values

returnCode	Floating point representation of temperature

2.4.4 float_t HCS_convertVoltage (uint16_t rawValue)

Converts a raw ADC voltage value to a read float value.

This is a convenience function that can take a raw value read from one of the ADC result registers and convert the value into a human readable float value.

Table 2-7. Parameters

in	*rawValue*	Raw value to convert

Table 2-8. Return Values

returnCode	Floating point representation of voltage

Table 2-9. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.5 tHCSResponseCode HCS_getChannelFaultStatus (uint8_t chanNum, uint16_t * fltStatus)

Reads an individual channel's fault status.

Reads the individual channel's fault status and stores the status into fltStatus

Table 2-10. Parameters

in	*chanNum*	Channel number to read
out	*fltStatus*	Fault status can be stored here

2.4.6 tHCSResponseCode HCS_getDeviceFaultSatus (uint16_t * fltStatus)

Reads the device's fault status.

Reads the device's fault status and stores the status into fltStatus

Table 2-11. Parameters

out	*fltStatus*	Fault status can be stored here

Table 2-12. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.7 tHCSResponseCode HCS_gotoLPM (tps2hcsxx_man_lpm_exit_curr_ch1_mask_t ch1ExitCurrent, tps2hcsxx_man_lpm_exit_curr_ch2_mask_t ch2ExitCurrent, uint16_t existingValue)

Puts the device into LPM mode.

This function can send the command to go into LPM mode. The MCU is responsible to monitor the FLT pin for a falling edge if the exit current if the device exceeds the exit current level. Alternatively, the HCS_wakeupDevice function can be used to wakeup the device.

Table 2-13. Parameters

in	*ch1ExitCurrent*	Exit current for channel 1
in	*ch2ExitCurrent*	Exit current for channel 2
in	*existingValue*	Exisiting value of LPM register

2.4.8 tHCSResponseCode HCS_gotoSleep (void)

Puts the device into SLEEP mode.

This function can send the command to go into SLEEP mode. Note that all values in the device's registers can be reset after exiting sleep mode. The device needs to be woken up by the HCS_wakeupDevice function.

Table 2-14. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.9 tHCSResponseCode HCS_initializeDevice (TPS2HCSXXQ1_CONFIG * config)

Initializes the device with device configuration structure.

This function can take a configuration structure that is generated from the Smart Fuse Configurator GUI software and send all of the values to the attached high-side switch. The common practice here is to call this function on initial boot of the MCU or when the device wakes up from sleep mode with all of the register configurations lost.

Table 2-15. Parameters

in	*config*	Pointer to configuration structure

Table 2-16. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.10 tHCSResponseCode HCS_readRegister (uint8_t addr, uint16_t * readValue)

Performs a raw register read for specified register address.

This function performs a simple register read from the given address and populates the provided pointer with the register contents.

Table 2-17. Parameters

in	*addr*	Register address to read
out	*payload*	Value of register populated to this parameter

Table 2-18. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.11 tHCSResponseCode HCS_setSwitchState (uint8_t swState)

Sets the on and off switch state status per channel.

This can turn on or off a channel of the high-side switch. The parameter is a bitwise representation of the channels to be enabled.

Table 2-19. Parameters

in	*swState*	Pointer to configuration structure

Table 2-20. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.12 tHCSResponseCode HCS_updateConfig (TPS2HCS10Q1_CONFIG * config)

Updates the provided configuration with smart fuse settings.

This API can populate the provided configuration structure with the values from the high-side switch. The can be used when the host software changes settings within the register and wants to update the initial configuration structure.

Table 2-21. Parameters

out	*config*	Pointer to configuration structure

Table 2-22. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.13 tHCSResponseCode HCS_wakeupDevice (void)

Wakes up the device from sleep mode.

This API simply issues a dummy write to register address 0xFF (which does not exist) so that the device wakes from sleep mode. Since the device wakes from sleep mode via a CS pin transition, the dummy write causes the device to wake up.

Table 2-23. Return Values

returnCode	An instance of **tHCSResponseCode**

2.4.14 tHCSResponseCode HCS_writeRegister (uint8_t addr, uint16_t payload)

Performs a raw register write for specified register address.

This function performs a simple register write to the given address with the provided payload

Table 2-24. Parameters

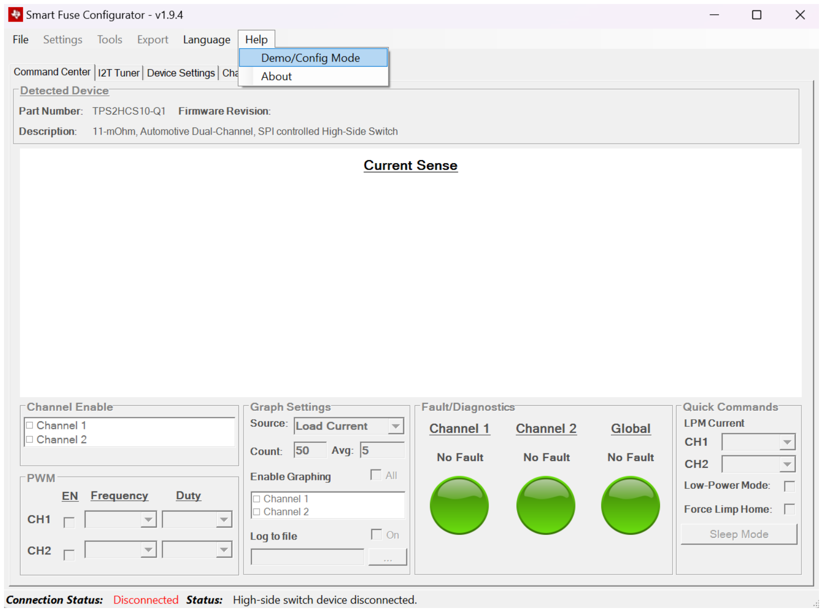
in	*addr*	Register address to write
in	*payload*	Value to write to addr

Table 2-25. Return Values

returnCode	An instance of **tHCSResponseCode**

3 Configuration or Evaluation Tool

The Smart Fuse Configurator tool is a software host tool that can be used along side of the HSS-HCMOTHERBRDEVM to live configure an HCS high-side switch as well as read out diagnostics such as current sense and fault conditions. Additionally, starting from version 1.9.4 the software has the ability to enter *configuration mode* without the need for a physical EVM board. In configuration mode, the user is able to change all different aspects of the device's settings such as current limit, capacitive charging modes, diagnostic reporting, and so on. The use can also use the I2T tuner to configure the device to match the wire profile and capabilities of the melting fuse being replaced. To enter configuration mode, select *Help->Demo/Config Mode* as shown in Figure 3-1:



Part Number: TPS2HCS10-Q1 Firmware Revision:
Description: 11-mOhm, Automotive Dual-Channel, SPI controlled High-Side Switch
Source: Load Current
Count: 50 Avg: 5
Connection Status: Disconnected Status: High-side switch device disconnected.

Figure 3-1. Configuration Mode

Once in demo mode, the user can use the software as described in the Smart Fuse Evaluation Module user's guide. Note that actual communication with the EVM are not performed in this mode and any diagnostics reported on the GUI are also not reflected. Demo mode can be exited by either selecting *Help->Demo/Config Mode* or by plugging in an EVM to the device.

Once the device has been configured to meet the application needs, the configuration can be exported by selecting *Export->Configuration Files*:

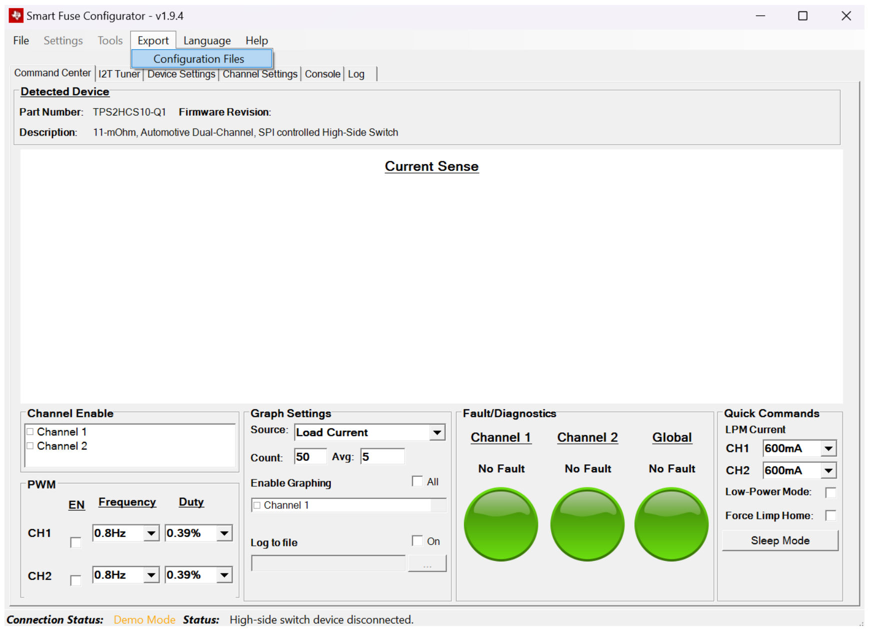

Smart Fuse Configurator - v1.9.4
File Settings Tools Export Language Help
Configuration Files
Command Center | I2T Tuner | Device Settings | Channel Settings | Console | Log
Detected Device
Part Number: TPS2HCS10-Q1 Firmware Revision:
Description: 11-mOhm, Automotive Dual-Channel, SPI controlled High-Side Switch
Current Sense
Channel Enable
Channel 1
Channel 2
PWM
EN Frequency Duty
CH1 0.8Hz 0.39%
CH2 0.8Hz 0.39%
Graph Settings
Source: Load Current
Count: 50 Avg: 5
Enable Graphing All
Channel 1
Log to file On
Fault/Diagnostics
Channel 1 Channel 2 Global
No Fault No Fault No Fault
Quick Commands
LPM Current
CH1 600mA
CH2 600mA
Low-Power Mode:
Force Limp Home:
Sleep Mode
Connection Status: Demo Mode Status: High-side switch device disconnected.

Figure 3-2. Export Configuration

The files that are exported from this dialog are the same configuration files that come packaged with the provided code examples (default titled ***tps2hcsxx_config.h*** and ***tps2hcsxx_config.c***). As an easy point-of-entry, the configuration files can be exported to the Empty Example (Section 4.1) and override the defaults file for a blank program used to start smart fuse development.

Note that the exported configuration files depend on the device specific header file that is provided on the project page. This header file contains all off the register definitions and enumerations relevant to the specific high-side switch part number.

The exported files contains a register definition that corresponds to every relevant configuration register of the device:

```
typedef struct TPS2HCSXXQ1_CONFIG
{
    TPS2HCSXX_CRC_CONFIG_OBJ        crcConfig;
    TPS2HCSXX_LPM_OBJ               lpmConfig;
    TPS2HCSXX_FAULT_MASK_OBJ        faultMaskConfig;
    TPS2HCSXX_SW_STATE_OBJ          swState;
    TPS2HCSXX_DEV_CONFIG_OBJ        devConfig;
    TPS2HCSXX_ADC_CONFIG_OBJ        adcConfig;
    TPS2HCSXX_PWM_CH1_OBJ           pwmCh1Config;
    TPS2HCSXX_ILIM_CONFIG_CH1_OBJ   ilimCh1Config;
    TPS2HCSXX_CH1_CONFIG_OBJ        diagConfigCh1;
    TPS2HCSXX_I2T_CONFIG_CH1_OBJ    i2tConfigCh1;
    TPS2HCSXX_PWM_CH2_OBJ           pwmCh2Config;
    TPS2HCSXX_ILIM_CONFIG_CH2_OBJ   ilimCh2Config;
    TPS2HCSXX_CH2_CONFIG_OBJ        diagConfigCh2;
    TPS2HCSXX_I2T_CONFIG_CH2_OBJ    i2tConfigCh2;
 } TPS2HCSXXQ1_CONFIG;
```

A pointer to this function is passed into the HCS_initializeDevice function of the platform driver (typically at microcontroller boot-up) to initially configure the high-side switch. After the structure definition, an instantiation of the structure is declared with values that represent all of the configured value from the Smart Fuse Configurator tool. The user can user this instantiation as a starting point and update manually through the code if a change is required, or regenerate the configuration file from the Smart Fuse Configurator program.

4 Code Examples

As part of the smart fuse software package, as set of code examples are provided that highlight both the functionality or differentiation of the smart fuse device family as well as the ease-of-use of the HCS platform drivers. A summary of the available code examples are shown in Table 4-1.

Table 4-1. Smart Fuse Code Examples

Code Example Name	Description
Empty	Simple code example that initializes the high-side switch from a Smart Fuse Configurator exported structure.
I2T Trip	Showcases I2T functionality and how to detect/recover from an I2T fault.
Low-Power Mode	Puts the device into low-power mode and waits for a wake-up event
Current Sense	Demonstrates how to use the HCS family's high-accuracy current sense and optimize readings for both lower and higher currents

4.1 Empty Example

The empty code example is a simple code example that is used to serve as a starting point for smart fuse applications. All this code example does is boot up, configure the underlying SPI peripheral, and then pass control off to the user. The main part of the initial configuration are shown in the following:

```
    /* Configuring the device initially */
    HCS_wakeupDevice();
    HCS_initializeDevice(&exportConfig);
```

The HCS_wakeupDevice function simply issues a dummy write to the device. Out of reset, the HCS family is in sleep mode. The wakeup function issues a write to register 0xFF (which doesn't exist) to make sure that the device is out of sleep.

The HCS_initializeDevice function takes the configuration file exported from the Smart Fuse Configurator application and loads into the high-side switch. This function is typically called every time on MCU boot-up to initialize the high-side switch.

4.2 I2T Trip Example

The I2T trip code examples shows how to handle an event on the high-side switch when an I2T event happens in the system. This code example assumes that the device is set up to LATCH mode for I2T faults and shows the correct sequence of events to *reset* the device after the I2T event occurs. In auto-retry mode (where TCLDN_CHx of the I2T_CONFIG_CHx register are set to a timeout), the device automatically waits the appropriate duration of time before re-enabling the channel.

Note, that in this code example the FAULT pin is set up to be used as a falling edge interrupt. When the I2T trip occurs, the FAULT pin (which is open drain) is pulled low and the microcontroller is interrupted. The software on the microcontroller can then wake up device execution and take the necessary mitigations to reenable the device. In LATCH mode, the following are appropriate steps to take after the I2T trip event occurs is to:

1. Disable the affected channel
2. Set the TCLDN_CHx of the I2T_CONFIG_CHx register to an appropriate cool-down time
3. Wait the specified duration by sleeping on the microcontroller
4. Set the TCLDN_CHx of the I2T_CONFIG_CHx register back to LATCH mode
5. Reenable the channel

The following is a relevant snippet of code:

```
        if(currentValue & TPS2HCSXX_FLT_STAT_CH1_I2T_FLT_CH1_MASK)
        {
            /* Disabling the channel */
            HCS_setSwitchState(0);

            /* Setting the device to 2s retry state */
            exportConfig.i2tConfigCh1.value.bits.TCLDN_CH1 =
                (tcldn_ch1_en_4p0s_mask);
            HCS_writeRegister(TPS2HCSXX_I2T_CONFIG_CH1_REG,
                        exportConfig.i2tConfigCh1.value.word);

            /* Waiting for two milliseconds. At this point, normally
                yield the tasks if in an RTOS, but wait for
                an interrupt here.  */

            DL_TimerA_startCounter(TIMER_0_INST);
            while(timerTriggered == false)
            {
                __WFI();
            }
            timerTriggered = false;

            /* Setting back to latch mode */
            exportConfig.i2tConfigCh1.value.bits.TCLDN_CH1 = 0;
            HCS_writeRegister(TPS2HCSXX_I2T_CONFIG_CH1_REG,
                        exportConfig.i2tConfigCh1.value.word);

            /* Re-enabling the channel */
            HCS_setSwitchState(1);
        }
```

4.3 Low-Power Mode Example

The low-power mode example shows how to set the HCS high-side switch into low-power mode and then wake up on a fault event. To use this example, before downloading the code to the microcontroller set a load on channel 1 that consumes less that 800mA of current (what the LPM exit current is set to using HCS_gotoSleep). Once the code example has been downloaded, increase the load current to greater than 800mA. This can cause the LPM to exit, the FAULT pin to trigger, and for the microcontroller to interrupt or handle the event. The relevant application code can be seen below:

```
    while(1)
    {
        /* Putting the device into LPM. 800mA exit current on CH1 */
        HCS_gotoLPM(man_lpm_exit_curr_ch1_en_800mA_mask,
                    man_lpm_exit_curr_ch2_en_800mA_mask,
                    0);

        /* Wait for the fault line to trigger low on PB3 */
        __WFI();

        /* If waking up from the interrupt, then check to make sure a signal was
            for an LPM wakeup. The idea here is that the user increases the
            load current somehow to "force the device" from LPM. */
        resCode = HCS_readRegister(TPS2HCSXX_FLT_STAT_CH1_REG,
                                    &currentValue);
        resCode.byte |=  HCS_readRegister(TPS2HCSXX_FLT_STAT_CH1_REG,
                                    &currentValue).byte;

        if(currentValue & TPS2HCSXX_FLT_STAT_CH1_LPM_WAKE_CH1_MASK)
        {
            /* Set a breakpoint here for demonstration */
            asm ("nop");
        }
    }
```

4.4 Current Sense Example

The current sense code example shows how the HCS family of smart fuse high-side switches has a scalable and extremely flexible current sense. This code example sets up a 1ms time that periodically wakes up and samples the load current of channel 1 of the high-side switch. If the load current is below 500mA, then the device enables the following settings of the high-side switch:

- Input voltage scaling (ISNS_SCALE_CHx of CHx_CONFIG register) is enabled allowing for the ADC input voltage to be scaled by a factor of 8x
- Open load detection enabled (OL_ON_EN_CHx of CHx_CONFIG) which changes KSNS ratio to lower value (see electrical specs in the data sheet)

A breakpoint line is set at each event to allow the user to break and understand the behavior of the current sense. When the low current sense mode is entered in the software, a state variable (inLowCurrent) is set to signify the current status and the device continues to periodically sample the current. If the current level saturates the ADC reading, the low current mode is exited and the normal scaling or KSNS modes are used. A snippet of the relevant code are shown in the following code:

```
 while(1)
{
    __WFI();

    /* Reading the load current value. Read the register twice
        as the */
    resCode = HCS_readRegister(TPS2HCSXX_ADC_RESULT_CH1_I_REG,
                            &currentValue);
    resCode.byte |=  HCS_readRegister(TPS2HCSXX_ADC_RESULT_CH1_I_REG,
                            &currentValue).byte;

    /* For each transaction, the high-side switch returns an error
        status code that reports common faults of the device. */
    if(resCode.byte != 0)
    {
        handleError(resCode);
    }

    /* Masking out the relevant bits */
    currentValue &= TPS2HCSXX_ADC_RESULT_CH1_I_ADC_RESULT_CH1_I_MASK;

    /* Check to see if the threshold is below where to turn
        on current sense scaling and change the KSNS ratio and enable
        scaling by 8x. This allows for  */
    if((currentValue < LOW_CURRENT_SNS_THRESHOLD) &&
            (inLowCurrent == false))
    {
        exportConfig.diagConfigCh1.value.bits.OL_ON_EN_CH1 = 1;
        exportConfig.diagConfigCh1.value.bits.ISNS_SCALE_CH1 = 1;
        inLowCurrent = true;
        HCS_writeRegister(TPS2HCSXX_CH1_CONFIG_REG,
                        exportConfig.diagConfigCh1.value.word);

        /* Adding place to set a breakpoint for the sake of demonstration */
        asm ("nop");
    }
    else if((currentValue == LOW_CURRENT_SNS_SATURATION) &&
                (inLowCurrent == true))
    {
        exportConfig.diagConfigCh1.value.bits.OL_ON_EN_CH1 = 0;
        exportConfig.diagConfigCh1.value.bits.ISNS_SCALE_CH1 = 0;
        HCS_writeRegister(TPS2HCSXX_CH1_CONFIG_REG,
                        exportConfig.diagConfigCh1.value.word);

        /* Adding place to set a breakpoint for the sake of demonstration */
        asm ("nop");
    }
}
```

By being able to detect low current and enable the scaling and KSNS modes, the high-side sense can adapt the current sense resolution and meet applications requiring high current sense accuracy.

5 Summary

The software drivers and code examples for the HCS family of smart fuse devices provide a versatile range of functions for developing software for automotive smart fuse applications. These drivers can be easily ported to support a variety of different host architectures and the simplistic design of the drivers allows a developer to get up-to-speed with software development for the HCS devices with little backend overhead. Additionally, with the use of the Smart Fuse Configurator software tool, the user has the ability to seamlessly configure the smart fuse device and tune the drivers to the exact requirement of the end application. These software tools remove barriers for adapting the HCS device family into a smart fuse or zonal application and provide a flexible approach for combining software or hardware development.

6 References

- Texas Instruments, *TPS2HCS10-Q1 Automotive, dual-channel 10mΩ smart high-side switch with I^2T wire protection, low IQ mode and SPI*
- Texas Instruments, *HCS-SMARTFUSE-DRIVERS Simple C drivers and code examples for HCS smart fuse devices*
- Texas Instruments, *HSS-HCMOTHERBRDEVM Smart fuse evaluation module*
- Texas Instruments, *HSS-2HCS10EVM TPS2HCS10-Q1 daughter card for smart fuse high-side switch*
- Texas Instruments, *HSS-SMART-CONFIGURATOR Configuration tool for the HSS-HCMOTHERBRDEVM and TI's smart fuse high-side switches*
- Texas Instruments, *HCS-HEADER-FILES C Header files for smart fuse high-side switches with register definitions*

7 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision A (July 2024) to Revision B (April 2025) **Page**

- Updated the part numbers and code examples throughout the document..1