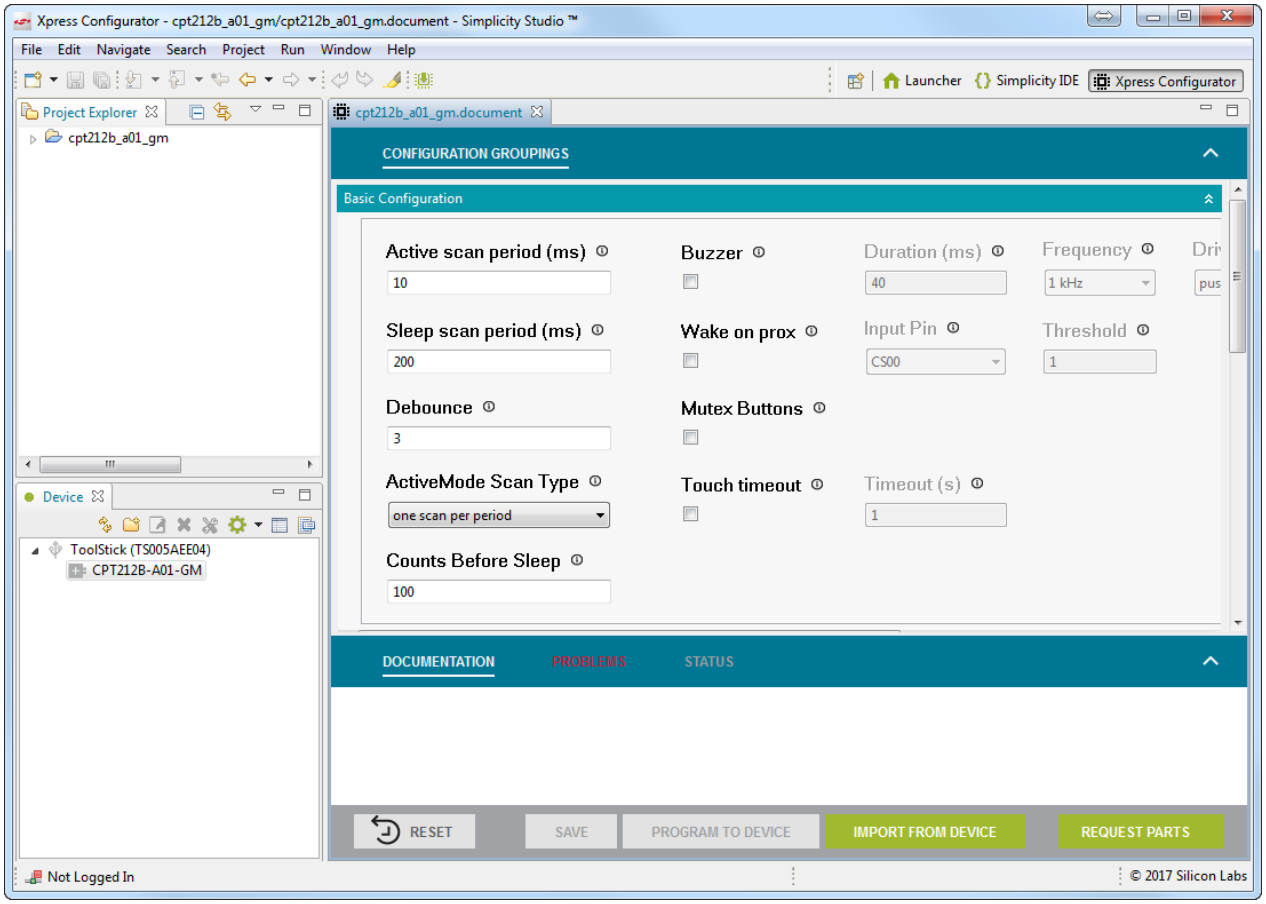
AN1092: TouchXpress™ Custom Configuration Profile Programming Guide

This application note gives an overview of the configuration profile data structure, and how a configuration profile can be modified and generated without the use of Simplicity Studio's Xpress Configurator.

The configuration profile defines all configurable performance characteristics in a CPTxxxx TouchXpress device. All TouchXpress devices support configuration profile programming through the configuration pin interface, and devices can be pre-programmed at the factory. Some TouchXpress devices can also be programmed through the I2C interface, and in these cases, developers may want to modify configuration profile settings without re-generating the profile through Simplicity Studio. This document steps developers through that process.

KEY POINTS

- TouchXpress performance is governed by a set of inter-dependent parameters stored in a configuration profile.
- This configuration profile can be generated in Xpress Configurator or through a manual process.



1. Introduction

This document discusses configuration profile programming of TouchXpress devices (CPTxxxx) that support configuration profile loading over the I2C interface (CPT212B and CPT213B). The configuration profile is a data structure stored in non-volatile memory on a TouchXpress device that controls runtime sensing and performance characteristics of the device, including sensor settings like threshold levels.

Simplicity Studio (www.silabs.com/simplicity) includes a tool called Xpress Configurator, which enables configuration profile creation for a design through a graphical interface. Xpress Configurator can output a configuration profile image in three different formats:

1. A config profile intel hex image that can be downloaded to TouchXpress devices through the 2-pin config interface on the part.
2. An XML file that Silicon Labs can use to provide pre-programmed devices.
3. Finally, Xpress Configurator can provide an ANSI C .c/.h file set that includes a struct defining the configuration profile and its contents. This struct can be used by a host to program a TouchXpress device's configuration profile through its configuration loader-enabled I2C inteface.

Generating the product's configuration profile through Xpress Configurator is preferable to manually editing a configuration profile byte array, because Xpress Configurator:

- Manages any dependencies between settings and alerts the user of invalid settings, preventing a configuration profile with errors from being generated and distributed.
- Provides context for each configuration profile setting in the documentation window.
- Automatically generates a CRC that must be provided by the host device during a configuration profile update session.

For use cases that do require configuration profiles generated or modified outside of Xpress Configurator, the developer should follow the guidelines in the following sections to ensure that the profile is valid and keeps the device running within specified limits.

1.1 Configuration Tips

Whenever possible, users should always generate configuration profiles within Xpress Configurator. If profile images must be generated by hand, Xpress Configurator can still be used to check manual configuration choices. If a manually-generated profile causes unpredictable behavior in a device, a user can try to replicate the manual configuration within Xpress Configurator and compare the byte array output of that tool to the manually-generated array. If the issue persists in the Xpress Configurator-generated image, the Xpress Configurator project can be included in a request for technical support.

The documentation that follows focuses on sensor configuration and scan periodicity. The document does not go into detail regarding the configuration of peripherals such as the haptic output, mutually exclusive button settings, and other features. If the product requires configuration of those features, a starter configuration profile with the appropriate features configured should be generated and used as a starting point.

Refer to the data sheet for the TouchXpress device being programmed for a complete description of all available settings.

2. Configuration Profile Overview

The configuration profile is a data structure where elements can be arrays or discrete values. Elements of those arrays and discrete values can be one of three types:

- `uint8_t`, big endian
- `uint16_t`, big endian
- Bit array (X bits padded out to a specified number of bytes, as defined in the following sections)

The tables in the following sections show the configuration profile structures with each array expanded into individual elements.

2.1 CPT212B Configuration Profile Structure

The configuration profile for the CPT212B can be defined as a `struct` as shown in the following definition.

```
typedef struct
{
  uint8_t reserved_0[3];
  uint8_t active_threshold[12];
  uint8_t inactive_threshold[12];
  uint8_t average_touch_delta[12];
  uint8_t accumulation[12];
  uint8_t gain[12];
  uint16_t active_mode_period;
  uint16_t sleep_mode_period;
  uint8_t button_debounce;
  uint8_t active_mode_mask[2];
  uint8_t sleep_mode_mask[2];
  uint8_t counts_before_sleep;
  uint8_t active_mode_scan_type;
  uint8_t reserved_1[4];
  uint8_t buzzer_output_duration;
  uint8_t buzzer_output_frequency;
  uint8_t buzzer_drive_strength;
  uint8_t touch_time_duration;
  uint8_t mutex_buttons;
  uint8_t prox_settings;
  uint8_t i2c_slave_address;
  uint8_t reserved_2;
  uint8_t i2c_timeout_duration;
  uint8_t sensor_string[12][10];
} config_profile_t;
```

All reserved areas in the profile should be set to 0x00. Values for other settings should follow the guidance in the following table.

Table 2.1. CPT212B Configuration Profile

Type	Name	Bounds	Description
uint8_t	reserved_0[3]	—	placeholder
uint8_t	active_threshold[12]	0-100	Sets the inactive-to-active threshold, configured relative to the touch delta. Must be greater than inactive_threshold.
uint8_t	inactive_threshold[12]	0-100	Sets the active-to-inactive threshold, configured relative to the touch delta. Must be less than active_threshold.
uint8_t	average_touch_delta[12]	0-255	Sets the expected difference between the touched and untouched state of the sensor, in units of output codes/16.
uint8_t	accumulation[12]	0 = 1x 1 = 4x 2 = 8x 3 = 16x 4 = 32x 5 = 64x	Sets the number of times sensor CS[x] performs a conversion per output word. Conversions are accumulated and then right-shifted to the output raw value.
uint8_t	gain[12]	0 = 1x 1 = 2x 2 = 3x 3 = 4x 4 = 5x 5 = 6x 6 = 7x 7 = 8x	Sets the dynamic range of sensor CS[x], where 1x is a maximum range of ~600 µF and 8x is ~60 µF.
uint16_t	active_mode_period	1-5000	Sets active mode scan period in ms
uint16_t	sleep_mode_period	1-5000	Sets sleep mode scan period in ms
uint8_t	button_debounce	0-15	Sets the number of consecutive values from an inactive sensor that must fall above the active threshold before sensor is qualified as active. Sets the number of consecutive values from an active sensor that must fall below the inactive threshold before sensor is qualified as inactive.

Type	Name	Bounds	Description
uint8_t	active_mode_mask[2]	ac...sk[0].b0=CS00 ac...sk[0].b1=CS01 ... ac...sk[0].b7=CS07 ac...sk[1].b0=CS08 ... ac...sk[1].b3=CS11	Bit array where a bit set to 1 = sensor is enabled, will be scanned, and will qualify touch events. Note that each byte is most-significant-bit oriented.
uint8_t	sleep_mode_mask[2]	sl...sk[0].b0=CS00 sl...sk[0].b1=CS01 ... sl...sk[0].b7=CS07 sl...sk[1].b0=CS08 ... sl...sk[1].b3=CS11	Bit array where a bit set to 1 = sensor is enabled as a wake-up source when in sleep mode. Note that each byte is most-significant-bit oriented.
uint8_t	counts_before_sleep	0-255	Sets the number of active mode scan periods that occur without any touches being qualified before the device switches from active mode scanning to sleep mode scanning
uint8_t	active_mode_scan_type	0 or 1	0 = scan each sensor once per active mode scan period 1 = scan each sensor as many times as possible within a active mode scan period
uint8_t	reserved_1[4]	—	—
uint8_t	buzzer_output_duration	0-255	Duration of buzzer activity during an inactive-to-active event, in 10s of ms. Note that setting this value to 0 disables the buzzer.
uint8_t	buzzer_output_frequency	25 = 1 kHz ... 100 = 4 kHz	Configures the frequency of the square wave output of the buzzer.
uint8_t	buzzer_drive_strength	0 = open drain output 1 = push-pull output	Configures whether the buzzer output pin drives the pin high or if the output requires a pull-up resistor.
uint8_t	touch_time_duration	0-14	Touch timeout in units of 500 ms. Note that setting this value to 0 disables the touch time-out feature.
uint8_t	mutex_buttons	0 = disabled 1 = enabled	Controls whether only one sensor will be qualified as active at a time.

Type	Name	Bounds	Description
uint8_t	prox_settings	upper nibble = prox threshold, where a higher threshold requires closer/larger object to trigger lower nibble = CSxx sensor to be used as wake on proximity input	Enables wake on proximity feature and sets the proximity threshold. Setting this byte to 0 disables wake on prox.
uint8_t	i2c_slave_address	0-0xFE	7-bit I2C address, left-shifted
uint8_t	reserved_2	—	—
uint8_t	i2c_timeout_duration	0-15	Configures the maximum duration at which packets will be stored in the I2C buffer without being read from the host. If the interrupt pin is allowed to remain active for the duration defined by the timeout register, the I2C buffer will be flushed, and the interrupt pin will deactivate. Setting this value to 0 disables the timeout feature.
char	sensor_string[12][10]	ASCII characters	Null-terminated strings used to store the name of each sensor. These strings are used by Capacitive Sense Profiler in data logging and display.

2.2 CPT213B Configuration Profile Structure

The configuration profile for the CPT213B can be defined as a struct as shown in the following definition.

```
typedef struct
{
  uint8_t reserved_0[3];
  uint8_t active_threshold[13];
  uint8_t inactive_threshold[13];
  uint8_t average_touch_delta[13];
  uint8_t accumulation[13];
  uint8_t gain[13];
  uint8_t reserved_1[4];
  uint8_t button_debounce;
  uint8_t active_mode_mask[2];
  uint8_t reserved_2[14];
  uint8_t i2c_slave_address;
  uint8_t reserved_3;
  uint8_t i2c_timeout_duration;
  uint8_t sensor_string[13][10];
} config_profile_t;
```

All reserved areas in the profile should be set to 0x00. Values for other settings should follow the guidance in the following table.

Table 2.2. CPT213B Configuration Profile

Type	Name	Bounds	Description
uint8_t	reserved_0[3]	—	placeholder
uint8_t	active_threshold[13]	0-100	Sets the inactive-to-active threshold, configured relative to the touch delta. Must be greater than inactive_threshold.
uint8_t	inactive_threshold[13]	0-100	Sets the active-to-inactive threshold, configured relative to the touch delta. Must be greater than inactive_threshold.
uint8_t	average_touch_delta[13]	0-255	Sets the expected difference between the touched and untouched state of the sensor, in units of output codes/16.
uint8_t	accumulation[13]	0 = 1x 1 = 4x 2 = 8x 3 = 16x 4 = 32x 5 = 64x	Sets the number of times sensor CS[x] performs a conversion per output word. Conversions are accumulated and then right-shifted to the output raw value.

Type	Name	Bounds	Description
uint8_t	gain[13]	0 = 1x 1 = 2x 2 = 3x 3 = 4x 4 = 5x 5 = 6x 6 = 7x 7 = 8x	Sets the dynamic range of sensor CS[x], where 1x is a maximum range of ~600 µF and 8x is ~60 µF.
uint8_t	reserved_1[4]	—	—
uint8_t	button_debounce	0-15	Sets the number of consecutive values from an inactive sensor that must fall above the active threshold before sensor is qualified as active. Sets the number of consecutive values from an active sensor that must fall below the inactive threshold before sensor is qualified as inactive.
uint8_t	active_mode_mask[2]	ac...sk[0].b0=CS00 ac...sk[0].b1=CS01 ... ac...sk[0].b7=CS07 ac...sk[1].b0=CS08 ... ac...sk[1].b4=CS12	Bit array where a bit set to 1 = sensor is enabled, will be scanned, and will qualify touch events. Note that each byte is most-significant-bit oriented.
uint8_t	reserved_2[14]	—	—
uint8_t	i2c_slave_address	0-0xFE	7-bit I2C address, left-shifted
uint8_t	reserved_3	—	—
uint8_t	i2c_timeout_duration	0-15	Configures the maximum duration at which packets will be stored in the I2C buffer without being read from the host. If the interrupt pin is allowed to remain active for the duration defined by the timeout register, the I2C buffer will be flushed, and the interrupt pin will deactivate. Setting this value to 0 disables the timeout feature.
char	sensor_string[13][10]	ASCII characters	Null-terminated strings used to store the name of each sensor. These strings are used by Capacitive Sense Profiler in data logging and display.

3. Configuration Loading Procedure

Refer to the configuration loading-enabled TouchXpress device data sheet for a detailed description of each command used by an embedded I2C host during the configuration loading procedure. Failure to follow the procedure outlined in the data sheet will prevent any further configuration loading operation and will prevent the device from entering sensing mode. The device will remain in this state until the next power cycle or until the next hardware reset.

In addition to the data sheet and the description in the following sections, see the TouchXpress code examples included within Simplicity Studio for the supported EFM8 and EFM32 devices.

3.1 Step 1 — Unlocking the Interface

The configuration loading process begins with a command that unlocks configuration profile programming. The unlock command uses two key codes. Failure to send these two key codes before erasing and writing bytes will result in a programming lock-out until a power cycle or hardware reset.

3.2 Step 2 — Erasing the Old Profile

After unlocking the interface, the host erases the configuration profile currently stored in non-volatile memory. Note that this command erases the config profile and the CRC stored on the device. After erasing the old configuration profile, the device cannot enter sensing mode again until a new configuration profile with a valid CRC has been loaded into the device.

3.3 Step 3 — Programming the New Profile

After erasing the old configuration profile, a new profile can be programmed. The embedded host must write this new profile starting at byte 0 of the byte array. The configuration profile bytes are written to the device using a command that accepts 8 bytes per command execution. The host should pad the last [**write bytes**] command up to a payload of 8 bytes, with 0xFF used as padding. All unwritten byte addresses after the last [**write bytes**] command execution will be stored as 0xFF in the TouchXpress device.

3.4 Step 4 — Generating and Sending the CRC

After all configuration profile bytes have been written to the TouchXpress device, the embedded host must send a command that writes a CRC generated using the configuration profile data. This CRC should include all configuration profile bytes and should be padded with '0xFF' up to and including byte 509. Refer to the device data sheet for detailed information on how to properly generate the CRC, including the polynomial used by the device. The TouchXpress device compares the CRC written by the embedded host to a CRC the TouchXpress device generates internally. If those two CRCs match, then the configuration profile is considered to be valid, and the device is allowed to enter sensing mode.

3.5 Step 5 — Resetting the Device

After the configuration process is complete, the device must be reset for all of the new settings take effect. To do this, the host must toggle the device's reset pin. See the device data sheet for more information about the low time requirements of this signal and the necessary wait time after reset before the device is ready.

4. Example — Enabling and Configuring a Sensor

The example outlined in the following sections shows how an embedded I2C master host can modify values in a config profile data structure for a CPT212B to enable sensor CS08 and configure that sensor with the following characteristics:

- Sensor (CS08) enabled for active mode scanning and sleep mode scanning
- Expected touch delta is 700 codes
- Active threshold is 80%
- Inactive threshold is 20%
- Gain is set to 8x
- Accumulation set to 4x

4.1 Step 1 — Generating the Profile

Using Xpress Configurator as a starting point, generate a configuration profile structure. For the purposes of this demonstration, sensors 0-3 will be enabled and configured, and all other sensors will be disabled. Additional settings will also be populated in the .c file generated by Xpress Configurator.

The following example shows the contents of the header file generated by Xpress Configurator, including the config profile struct definition and a default profile:

```
typedef struct
{
  uint8_t reserved_0[3];
  uint8_t active_threshold[12];
  uint8_t inactive_threshold[12];
  uint8_t average_touch_delta[12];
  uint8_t accumulation[12];
  uint8_t gain[12];
  uint16_t active_mode_period;
  uint16_t sleep_mode_period;
  uint8_t button_debounce;
  uint8_t active_mode_mask[2];
  uint8_t sleep_mode_mask[2];
  uint8_t counts_before_sleep;
  uint8_t active_mode_scan_type;
  uint8_t reserved_1[4];
  uint8_t buzzer_output_duration;
  uint8_t buzzer_output_frequency;
  uint8_t buzzer_drive_strength;
  uint8_t touch_time_duration;
  uint8_t mutex_buttons;
  uint8_t prox_settings;
  uint8_t i2c_slave_address;
  uint8_t reserved_2;
  uint8_t i2c_timeout_duration;
  uint8_t cs00_sensor_string[10];
  uint8_t cs01_sensor_string[10];
  uint8_t cs02_sensor_string[10];
  uint8_t cs03_sensor_string[10];
  uint8_t cs04_sensor_string[10];
  uint8_t cs05_sensor_string[10];
  uint8_t cs06_sensor_string[10];
  uint8_t cs07_sensor_string[10];
  uint8_t cs08_sensor_string[10];
  uint8_t cs09_sensor_string[10];
  uint8_t cs10_sensor_string[10];
  uint8_t cs11_sensor_string[10];
} config_profile_t;
```

```
#define CPT212B_A01_GM_DEFAULT_CONFIG \
{ \
  {0x00, 0x00, 0x00},                                                                   /* reserved_0 */ \
  {0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C, 0x3C},         /* Active Threshold */ \
  {0x28, 0x28, 0x28, 0x28, 0x28, 0x28, 0x28, 0x28, 0x28, 0x28, 0x28, 0x28},         /* Inactive Threshold */ \
  {0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B, 0x2B},         /* Average Touch Delta */ \
  {0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02},         /* Accumulation */ \
  {0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06},         /* Gain */ \
  htobe16(0x0014),                                                                      /* active mode period */ \
  htobe16(0x00C8),                                                                      /* sleep mode period */ \
  0x02,                                                                                 /* button debounce */ \
  {0xFF, 0x0F},                                                                         /* active mode mask */ \
  {0x01, 0x00},                                                                         /* sleep mode mask */ \
  0x64,                                                                                 /* counts before sleep */ \
  0x01,                                                                                 /* active mode scan type */
\
  {0x00, 0x00, 0x00, 0x00},                                                             /* reserved_1 */ \
  0x04,                                                                                  /* buzzer output duration *
/ \
  0x64,                                                                                 /* buzzer output frequency *
/ \
  0x01,                                                                                 /* buzzer drive strength */
\
  0x00,                                                                                 /* touch time duration */ \
  0x00,                                                                                 /* mutex buttons */ \
  0x00,                                                                                 /* prox settings */ \
  0xF0,                                                                                 /* I2C slave address */ \
  0x00,                                                                                 /* reserved_2 */ \
  0x01,                                                                                 /* I2C timeout duration */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS00 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS01 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS02 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS03 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS04 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS05 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS06 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS07 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS08 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS09 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS10 sensor string */ \
  {'\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00'},  /* CS11 sensor string */ \
}
```

The .c file generated by Xpress Configurator includes this initialization of the configuration profile, stored in volatile memory:

```
    config_profile_t configProfile = CPT212B_A01_GM_DEFAULT_CONFIG;
```

4.2 Step 2 — Modifying the Config Profile

With the configuration profile stored in the global variable in volatile memory, an embedded host can edit contents and then load the configuration profile into the CPT212B.

The following code demonstrates how the `struct`'s contents can be changed to enabled CS08 with the settings defined in the previous section:

```
config_profile.active_mode_mask[1] |= 0x01;
config_profile.sleep_mode_mask[1] |= 0x01;
config_profile.active_threshold[8] = 80;
config_profile.inactive_threshold[8] = 20;
config_profile.average_touch_delta[8] = 700/16;
config_profile.accumulation[8] = 2;
config_profile.gain[8] = 7;
```

4.3 Step 3 — Generating the CRC

The host can generate a CRC on the new configuration profile using the software-based CRC routine provided in the Xpress Configurator output. The function generates a two-byte CRC based on the configuration profile included as an argument, and then finishes CRC generation by padding the remainder of the array with 0xFF. The function is as follows:

```
uint16_t generateConfigProfileCRC(config_profile_t* profile)
{
  uint16_t byte_index;
  uint8_t i, CRC_input;
  uint16_t CRC_acc = 0xFFFF;
  uint8_t * bytePtr = (uint8_t*)profile;

 #define POLY 0x1021

  for(byte_index = 0; byte_index < 510; byte_index++)
  {
    // If not all bytes of config profile have been used to generate the CRC,
    // read another byte
    if(byte_index < sizeof(config_profile_t))
    {
      CRC_input = *bytePtr;                     // Read byte from config profile
      bytePtr++;                                // Increment pointer by one byte
    }
    // else if all bytes have been used, use padding byte of 0xFF
    else
    {
      CRC_input = 0xFF;
    }
    // Create the CRC "dividend" for polynomial arithmetic (binary arithmetic
    // with no carries)
    CRC_acc = CRC_acc ^ (CRC_input << 8);
    // "Divide" the poly into the dividend using CRC XOR subtraction
    // CRC_acc holds the "remainder" of each divide
    //
    // Only complete this division for 8 bits since input is 1 byte
    for (i = 0; i < 8; i++)
    {
      // Check if the MSB is set (if MSB is 1, then the POLY can "divide"
      // into the "dividend")
      if ((CRC_acc & 0x8000) == 0x8000)
      {
        // if so, shift the CRC value, and XOR "subtract" the poly
        CRC_acc = CRC_acc << 1;
        CRC_acc ^= POLY;
      }
      else
      {
        // if not, just shift the CRC value
        CRC_acc = CRC_acc << 1;
      }
    }
  }
  return CRC_acc;
}
```

5. Revision History

5.1 Revision 0.3

July 12th, 2017

Added 3.5 Step 5 — Resetting the Device.

5.2 Revision 0.2

June 16th, 2017

Updated the Xpress Configurator screenshot on the front page.

Changed the `reserved` elements in the CPT212B structure to start at index '0' instead of '1'.

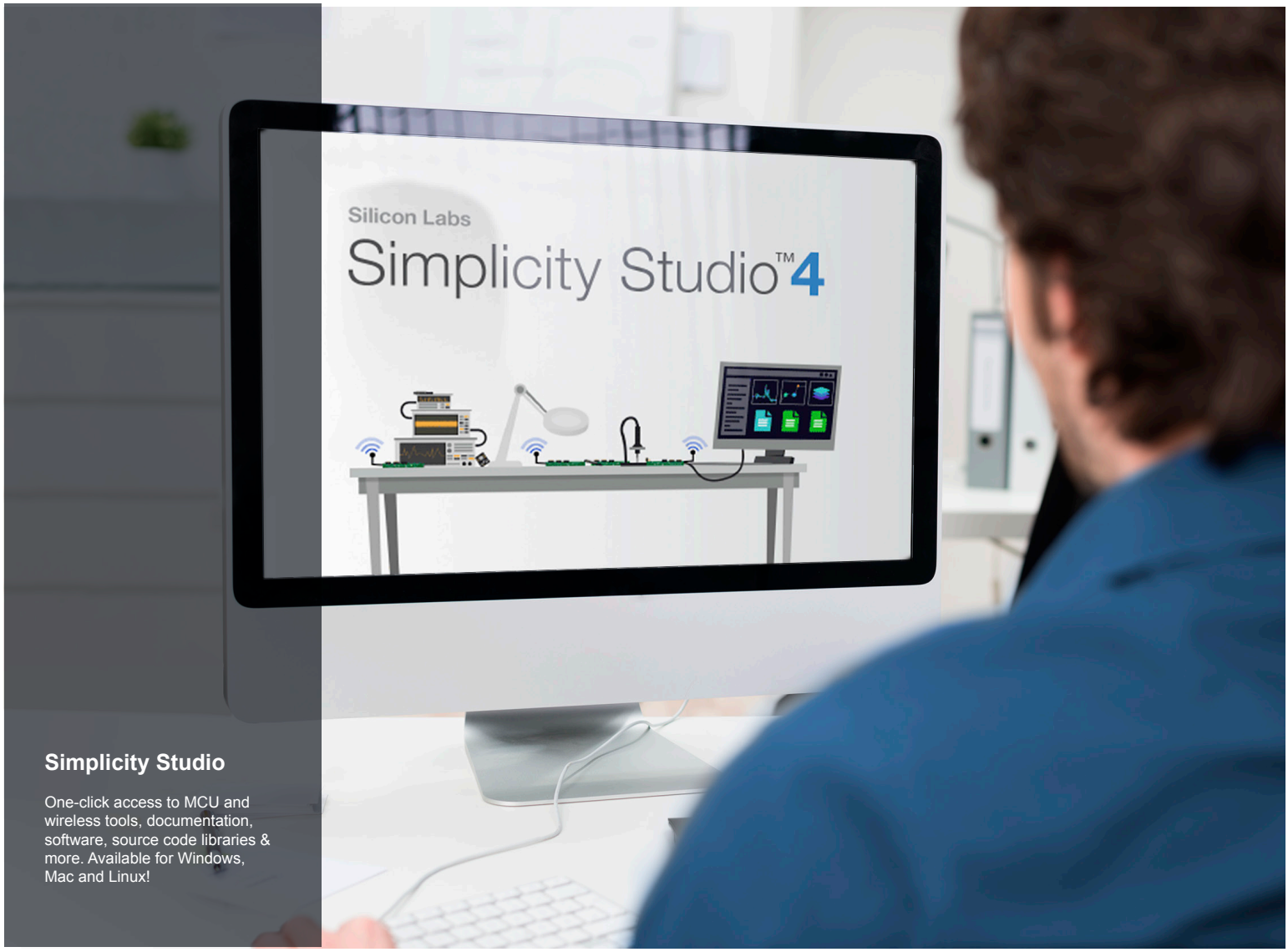
Removed `values` from `accumulation_values` and `gain_values` elements.

Updated the 4.1 Step 1 — Generating the Profile section to match the output from Xpress Configurator.

5.3 Revision 0.1

May 26th, 2017

Initial revision.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com