CP2501 Programmer's Guide and API Specification

1. Introduction

The CP2501 devices are programmable, 8051-based devices that add a Windows-compatible, HID USB touch-screen interface to multi-touch devices, including touch-screens and pen inputs. The CP2501 devices include pre-programmed System Firmware that provide an easy-to-use API to initialize the device, transfer USB data and use the hardware peripherals to interface to multi-touch devices.

This application note details the System Firmware API and how to use it to develop a multi-touch product. Also discussed are the CP2501 Configuration Wizard and the CP2501 USB Bootloader PC applications.

System development with the CP2501 assumes a basic knowledge of HID touch screens and HID usage tables. The following documents are useful for understanding HID touch-screens:

1. Device Class Definitions For HID Devices:
   http://www.usb.org/developers/devclass_docs/HID1_11.pdf
2. HID Usage Tables:
   http://www.usb.org/developers/devclass_docs/Hut1_12.pdf

For more information regarding the CP2501, see the following documents:

1. CP2501 Data Sheet
2. CP2501 Development Kit User's Guide
   Both documents are available on the Development Kit CD and at www.silabs.com.

2. API Function Overview

The System Firmware provides a set of functions, memory buffers, and system flags that implement an application programming interface (API). These functions allow the user firmware to configure the hardware and they provide access to the communications interfaces available on the CP2501 devices. All low-level hardware details and protocols are handled by the API and do not require management by the user firmware.

3. Getting Started

Starting a new project with the CP2501 devices is simple. The CP2501 Configuration Wizard generates a base project that enumerates the CP2501 device with user-selected descriptors. The custom user firmware is added to this project and its primary purpose is to use the API to retrieve touch data from the touch-screen and transfer it to the USB host.The following sections describe how to use the Configuration Wizard to generate a project and how the generated files are organized and modified.

3.1. Configuration Wizard Guide

Once installed, start the CP2501 Configuration Wizard from: Start→All Programs→Silicon Laboratories→CP2501 Configuration Wizard. The Configuration Wizard includes a separate configuration tab for the different CP2501 features: Device, Communication, Screen, Touch, Pen, Mouse and GPIO. The following sections provide more details about the tabs and show how to generate a project once all of the options are configured.

3.1.1. Device Tab

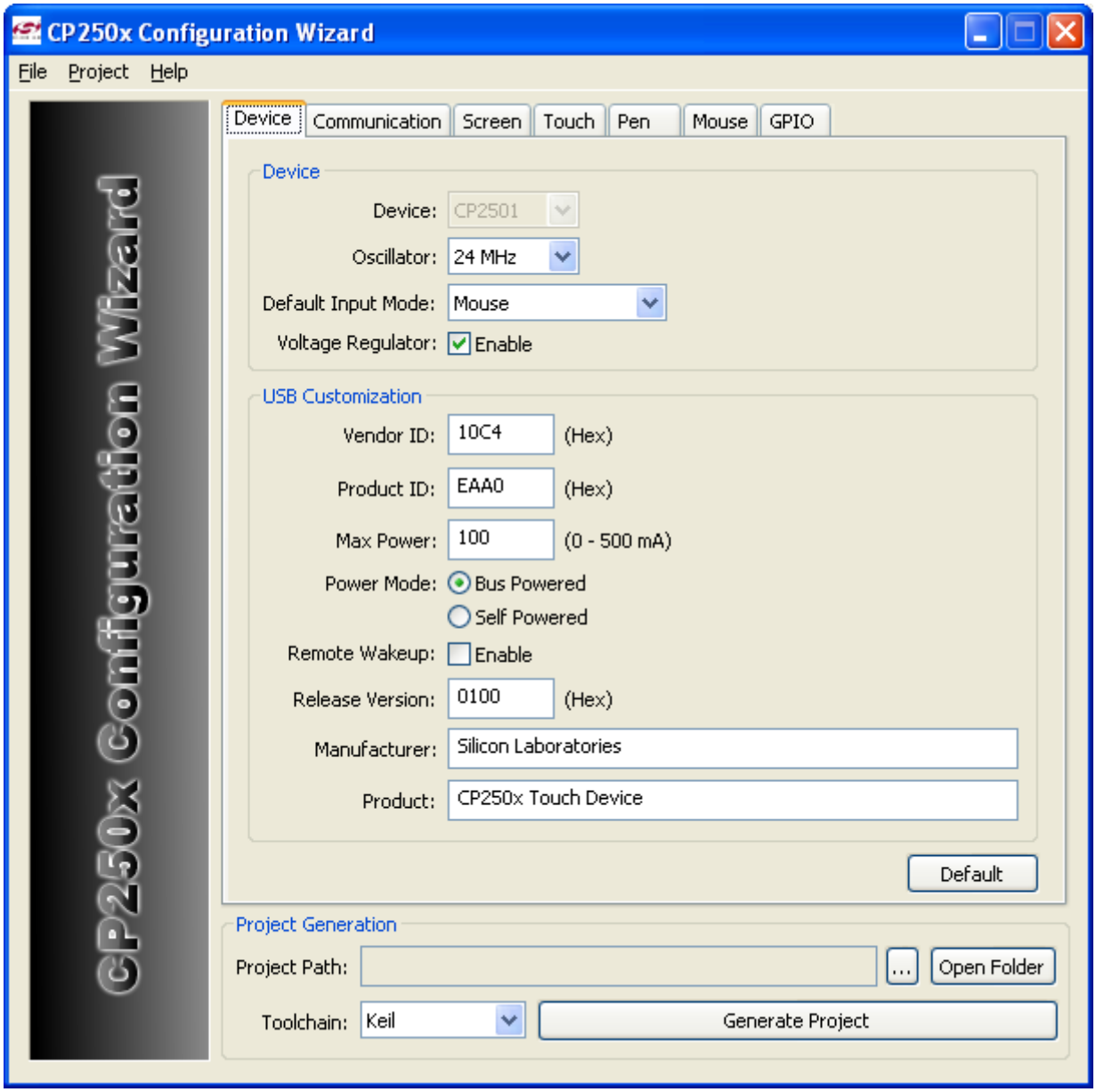

Figure 1. CP2501 Configuration Wizard Device Tab

The Device tab includes the configuration options for the CP2501 oscillator, regulator and USB descriptors. See the CP2501 data sheet for more details.

The Default Input Mode selection chooses between “Mouse” and ‘Single-Point Touch” and is used for Windows XP/ Vista systems. Unlike Windows 7, Windows XP/Vista do not explicitly configure the device to switch between Mouse mode and Single-Point Touch. This configuration option indicates what the default CP2501 behavior should be once enumeration is complete.

The default VID/PID combination is provided as an example and should not be used for your product. Contact Silicon Labs support to obtain a free, unique VID/PID for your CP2501 product.

Remote Wakeup allows the device to wake up the USB host from suspend mode from a user generated event. The Manufacturer and Product strings are each limited to 62 ASCII characters.

3.1.2. Communication Tab

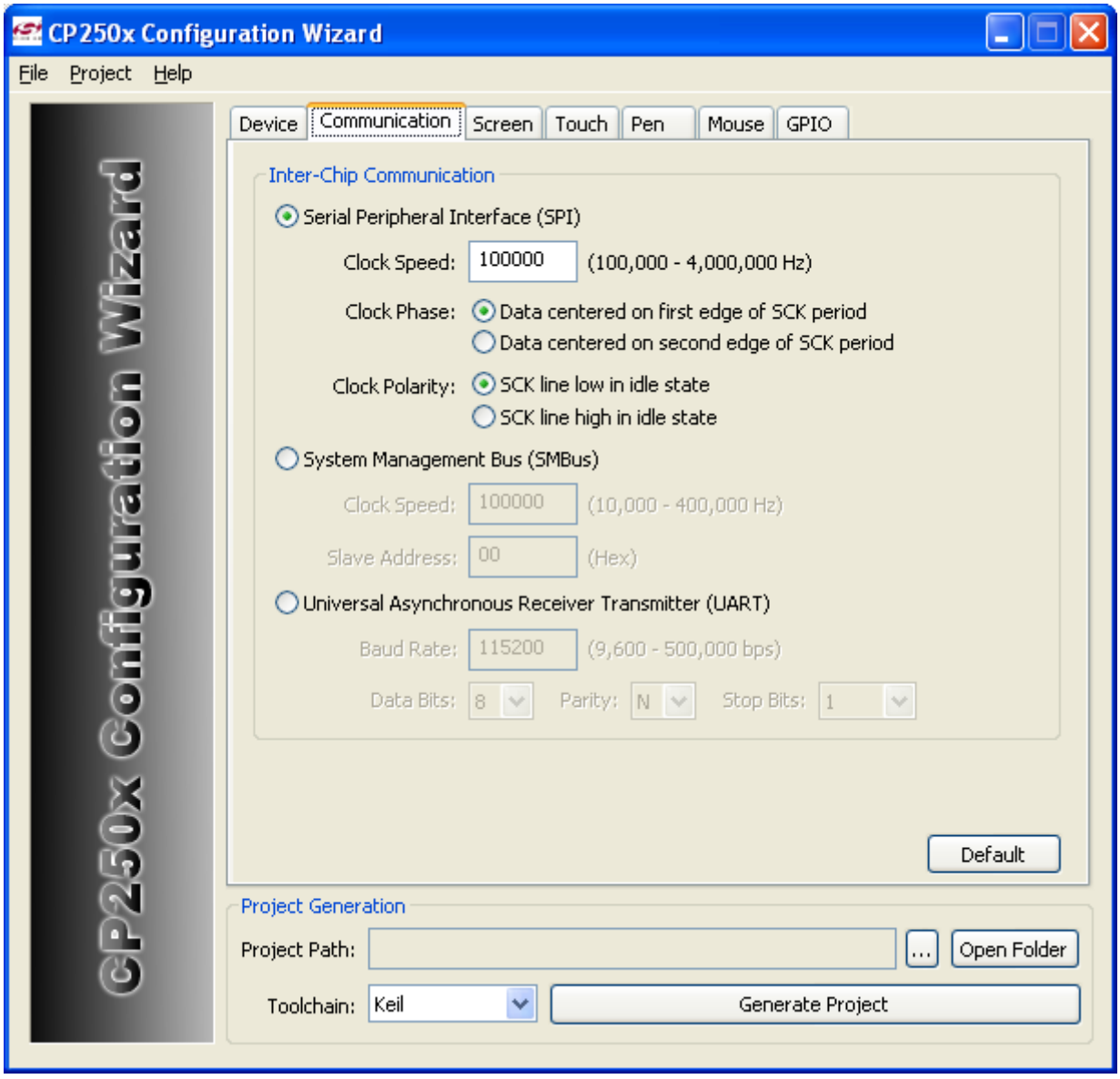

Figure 2. CP2501 Configuration Wizard Device Tab

This tab configures the communication interface to use between the CP2501 device and the touch screen module. Only one communication interface can be active at any time as the memory buffers are shared between the interfaces.

3.1.3. Screen Tab

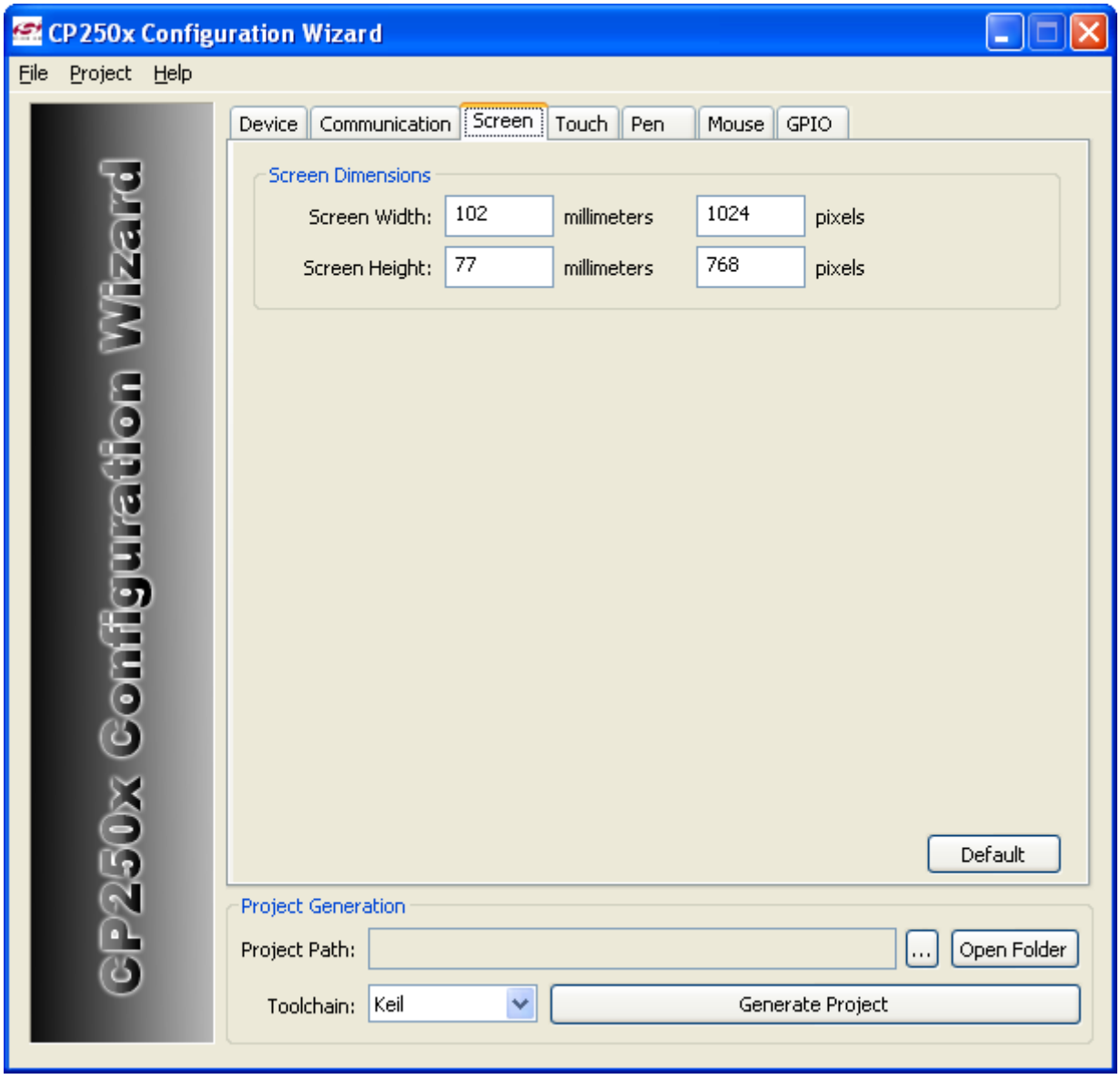

Figure 3. CP2501 Configuration Wizard Screen Tab

The Screen Width and Screen Height refer to the target display of the USB host, and not the touch screen itself. Windows requires these dimensions as part of the USB descriptor.

3.1.4. Touch Tab

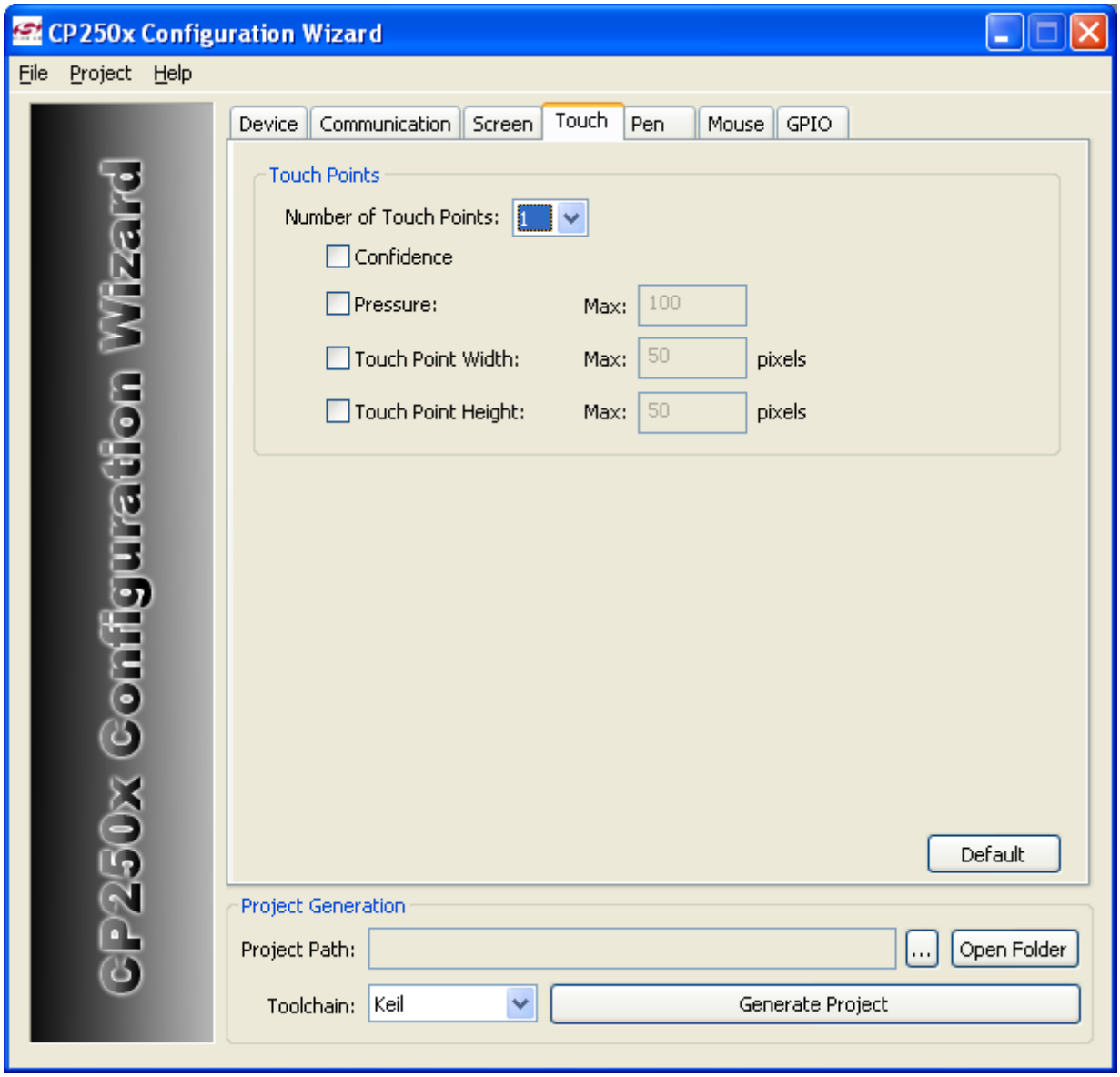

Figure 4. CP2501 Configuration Wizard Touch Tab

This tab configures the maximum number of touch points supported by the touch screen module. If the touch screen has the capability to report extra usage information, such as Confidence or Pressure, those options are selectable here and are added to the USB descriptor. If the touch screen supports usage information not available in Configuration Wizard, those usages can be manually added to the USB descriptor once the project is generated.

If the Default Input Mode on the Device tab is set to “Single-Point Touch”, the system must include at least one touch point.

3.1.5. Pen Tab

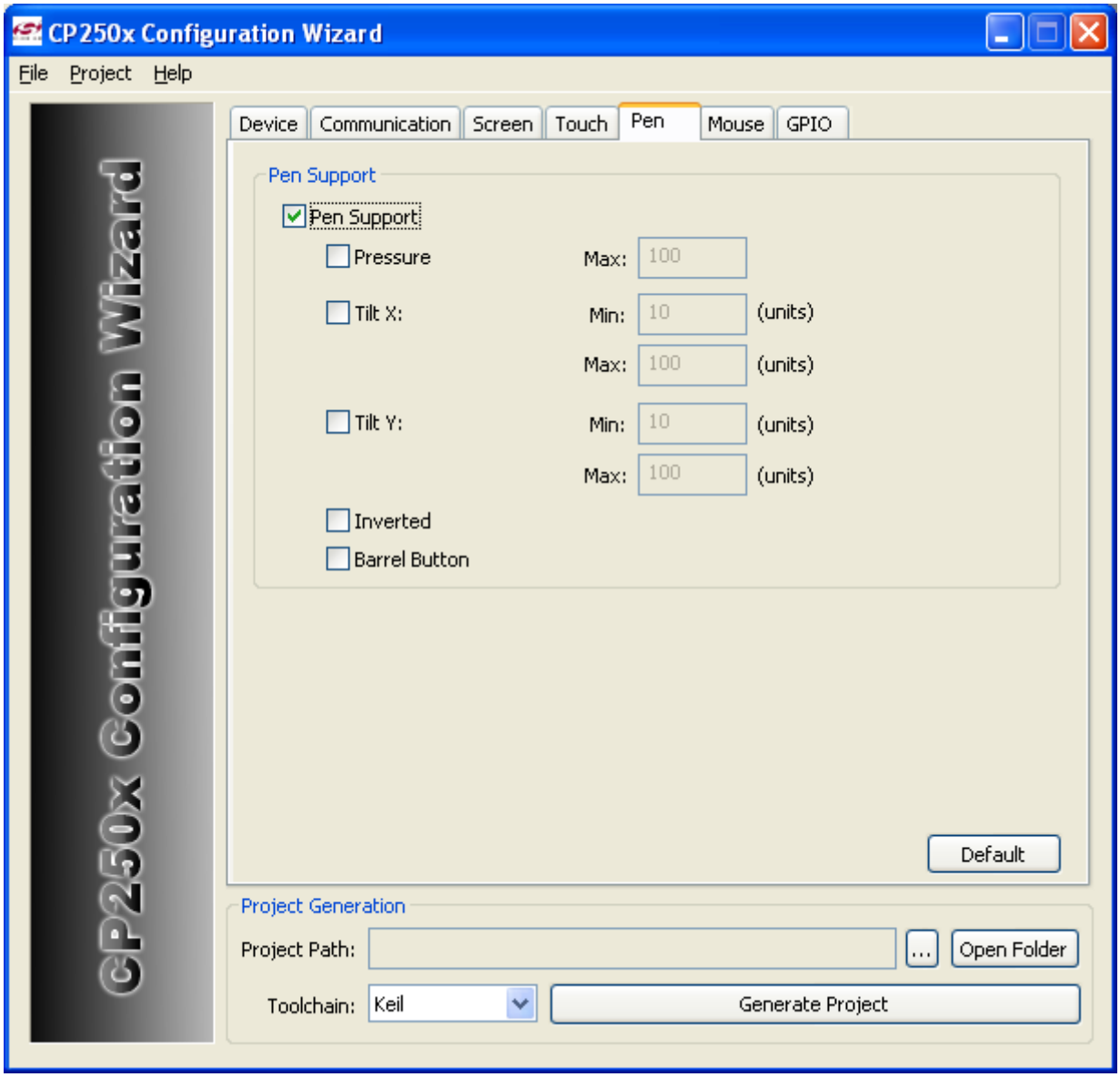

Figure 5. CP2501 Configuration Wizard Pen Tab

This tab enables a pen input for the touch screen module. If the touch screen has the capability to report extra usage information, such as Tilt or Pressure, those options are selectable here and are added to the USB descriptor. If the touch screen supports usage information not defined in Configuration Wizard, those usages can be manually added to the USB descriptor once the project is generated.

3.1.6. Mouse Tab

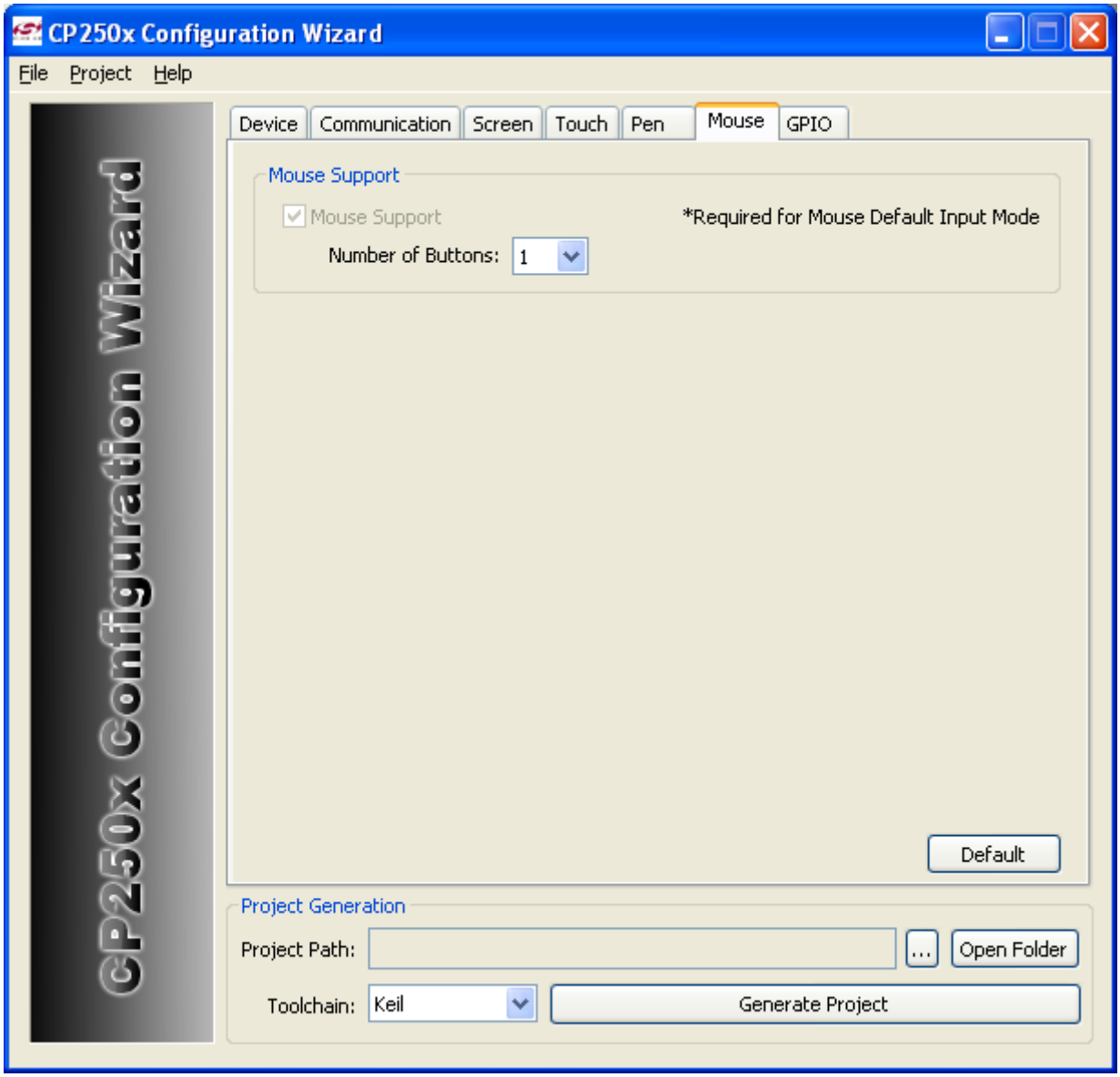

Figure 6. CP2501 Configuration Wizard Mouse Tab

This tab configures the USB descriptor to include mouse support and select the number of buttons supported by the mouse. Even if the final product does not support mouse functionality, it is safe to include the mouse descriptor.

If the Default Input Mode on the Device tab is set to “Mouse”, the mouse support must be selected.

3.1.7. GPIO Tab

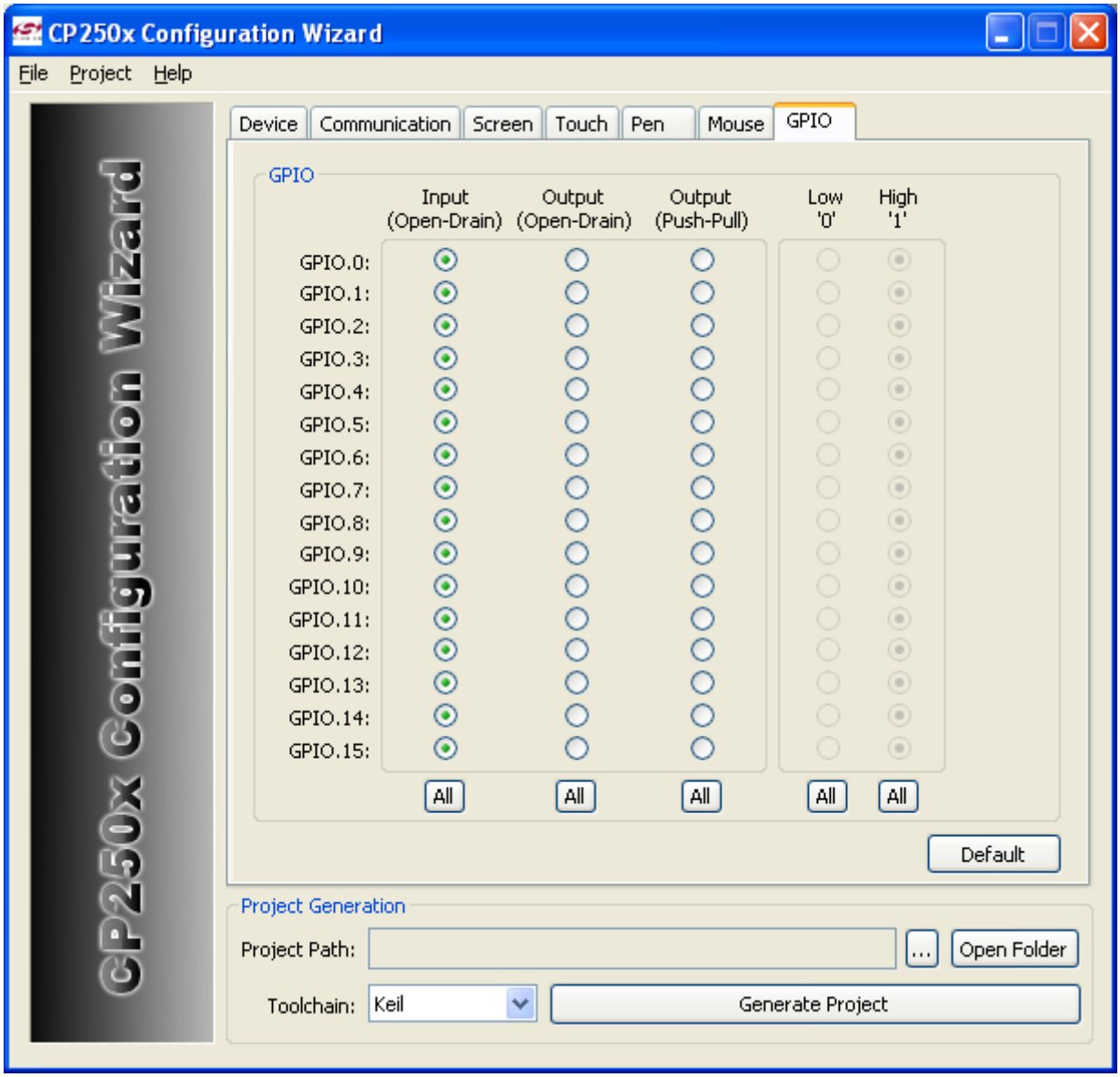

Figure 7. CP2501 Configuration Wizard GPIO Tab

The GPIO tab configures the mode of each GPIO pin. For pins that are selected as outputs, the initial latch value is also configurable. Set unused GPIO pins to Input (Open-Drain). The different modes are described in more detail in the GPIO section of the CP2501 data sheet.

3.1.8. Project Generation

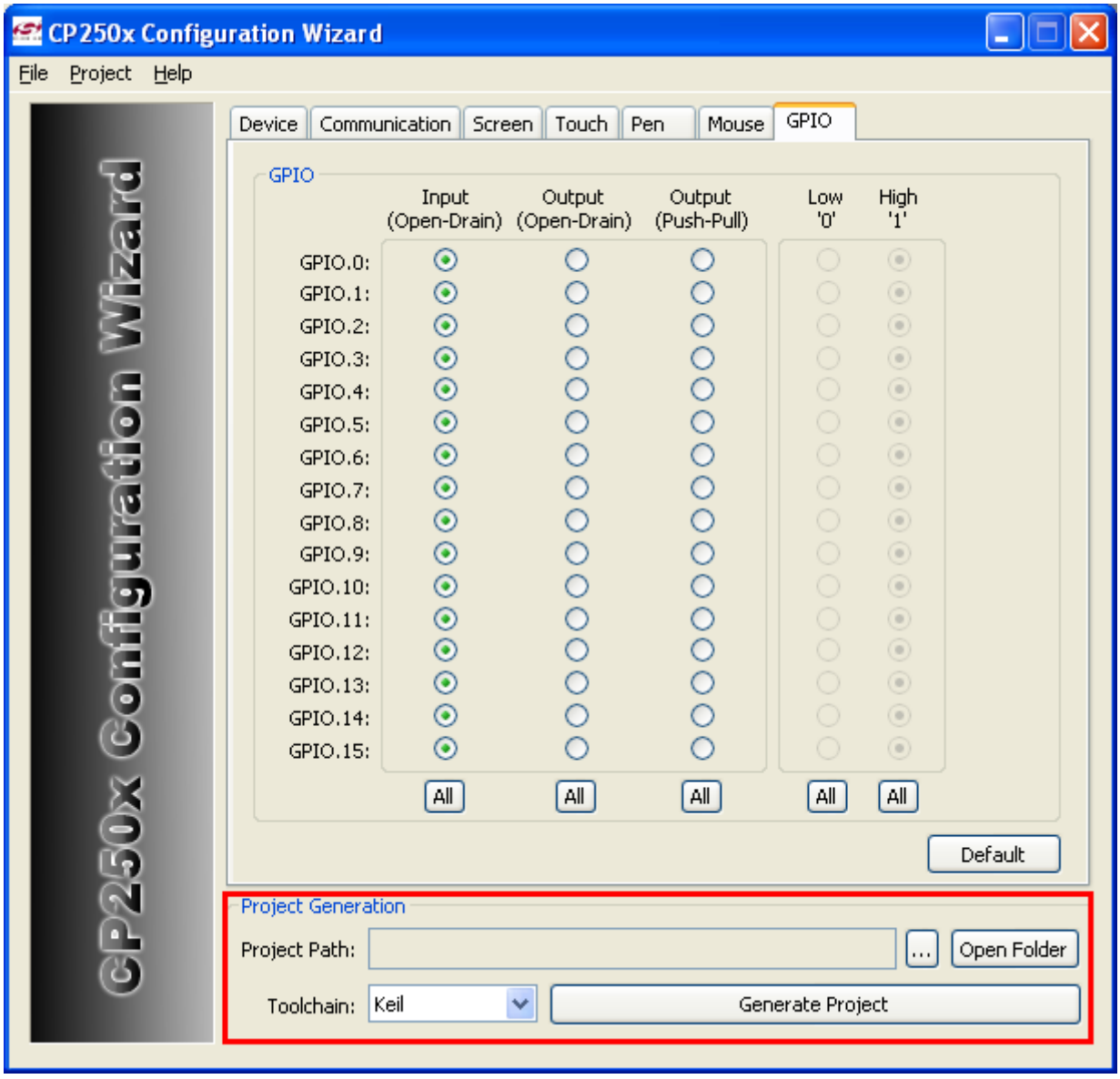

Figure 8. CP2501 Configuration Wizard Project Generation

Once all of the options are configured, the project is ready to be generated. First, select the desired Toolchain from the drop-down menu. The choices are Keil, Raisonance, and SDCC. In addition to generating code compatible with the selected toolchain's syntax, the Configuration Wizard also customizes the Silicon Labs IDE project workspace file to set the appropriate code, RAM, and XRAM boundaries for user firmware. Next, select a folder and click "Generate Project."

The following sections describe the files generated by the Configuration Wizard and how to use them to develop the system.

3.2. Project Directory Structure

The output of the Configuration Wizard includes the files listed in Table 1.

Table 1. Configuration Wizard Output Files

File Name	Description	User Modify
compiler_defs.h	Includes basic type and prototype definitions that enable compiler independent code.	No
CP250x_API.h	Includes all of the prototypes, function parameters, function return codes, and GPIO pin definitions presented by the System Firmware API.	No
CP250x_Buffer_Struct.h	Includes the structures that show how the touch, pen, and mouse data should be formatted in the USB_Position_Buffer. This file is dynamically generated.	No
CP250x_Configuration.c	Includes the HID Report IDs and the application version and validation signature constants. This file is dynamically generated.	No
CP250x_Configuration.h	Header file for CP250x_Configuration.c.	No
CP250x_Main.c	Main source file that includes the CP2501 device initialization, the touchscreen interface and sends the data to the System Firmware to pass to the USB host.	Yes
CP250x_USB_Descriptor.c	Includes the USB descriptors that specify the capabilities of the touch device. This file is dynamically generated.	Yes
STARTUP.A51	Startup file for Raisonance and Keil toolsets. Not generated for SDCC projects.	No
CP250x_<Toolset>.wsp	Silicon Labs IDE workspace file. The <Toolset> name is replaced by Raisonance, SDCC or Keil.	No

The CP250x_Main.c file is the only generated file that requires user modification. The file provides a basic structure for user firmware. In the example projects, all user developed code is localized to this file but additional files can be added to the project as necessary.

The selected configuration options, such as the number of touch points and the usages, determine the USB Descriptor. The organization of the USB descriptor determines the format of the data sent in the USB packet. This format is defined in CP250x_Buffer_Struct.h. If a specific device usage is not supported natively by the Configuration Wizard, it can be manually added to the USB descriptor in CP250x_USB_Descriptor.c. In addition to adding the usage information to the HID Report Descriptor, HidReportDesc[], the length of the descriptor must also be modified in the HID Configuration Descriptor, HidConfigDesc[]. Adding as usage will also change the format of the USB packet and a change in the structs found in CP250x_Buffer_Struct.h.

Without any modifications, the project generated by the Configuration Wizard will enumerate properly on a Windows machine. The device appears in Device Manager as an HID-device. In Windows Vista and Windows 7, the computer properties will show that a single-touch or multi-touch device is connected.

3.3. Code Flow Diagrams

The primary requirements of the user firmware are to retrieve data from the touch screen module, calculate the touch/mouse coordinates and any corresponding usage information such as IN_RANGE and TIP_SWITCH, and copy the data to the USB buffer in the USB packet format. Figure 9 below is a basic version of the code flow of the user firmware.

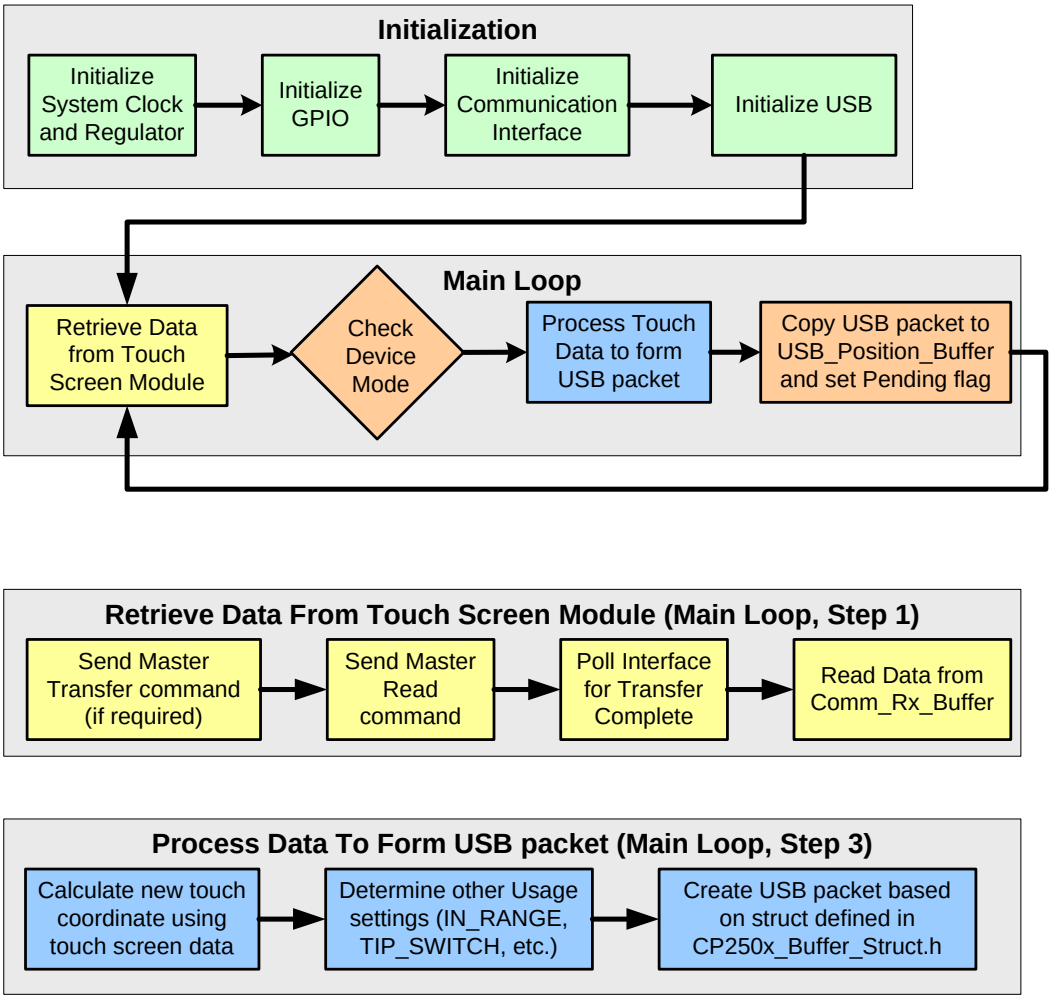

Figure 9. Simplified Code Flow Block Diagram (User Firmware)

The USB packets that are sent to the host PC need to follow the certain specifications in order to be compatible with the various Windows operating systems. Two useful documents are:

1. "Digitizer Drivers for Windows Touch and Pen-Based Computers"
   http://www.microsoft.com/whdc/device/input/DigitizerDrvs_touch.mspx
2. "Human Interface Device (HID) Extensions and Windows/Device Protocol for Multi-Touch Digitizers"
   <URL pending>

There are multiple operating modes for reporting touch data for multi-touch digitizers. The CP2501 implements serial mode, instead of the parallel and hybrid modes. In serial mode, each USB packet contains the data for only one touch point. Also note that in order to interpret USB touch packets correctly, Windows requires that the IN_RANGE and TIP_SWITCH fields are set according to a certain state machine. These two documents provide more detail about reporting touch data. The example projects listed in "3.4. Using the CP2501 Examples" follow the guidelines listed in these documents.

3.4. Using the CP2501 Examples

The CP2501 installation package includes an \Examples\ directory which includes basic firmware projects that show how to use the three different interface options (UART, SPI, and SMBus).

A fourth example, CP2501_NoScreen, is also included. This example simulates a single touch point which touches the screen at the top-left corner and drags the simulated finger towards the middle of the screen before releasing contact. This example runs directly on the CP2501 Development Kit without any additional hardware, and can be connected to any Windows Vista or Windows 7 machine.

4. CP2501 API Reference

4.1. Index of API functions

Table 2 includes all of the functions provided by System Firmware.

Table 2. API Function List

Name	Page
System Configuration	
CP250x_System_Init()	18
CP250x_USB_Init()	18
CP250x_GPIO_Init()	19
USB Suspend Functions	
CP250x_USB_Suspend	20
CP250x_USB_Suspend_For_Remote_Wakeup()	20
CP250x_Remote_Wakeup()	21
Interface Functions	
CP250x_UART_Init()	22
CP250x_UART_Write()	22
CP250x_UART_Dequeue()	23
CP250x_UART_Poll()	23
CP250x_SPI_Master_Init()	24
CP250x_SPI_Master_Transfer()	24
CP250x_SPI_Poll()	25
CP250x_SMBus_Master_Init()	25
CP250x_SMBus_Master_Transfer()	26
CP250x_SMBus_Poll()	26
Counter Functions	
CP250x_Start_Counter()	27
CP250x_Stop_Counter()	27

4.2. Memory Buffers and System Variables

The following memory buffers and variables are shared between the user firmware and System Firmware. These variables are absolutely located and the user firmware should not use these memory locations.

4.2.1. Memory Buffers

Table 3. Memory Buffers

Name	Size (Bytes)	Memory Space	Address
Comm_Rx_Buffer	128	XDATA	0x0000
Comm_Tx_Buffer	128	XDATA	0x0080
USB_Position_Buffer	64	XDATA	0x0100
USB_Control_Buffer	64	XDATA	0x0140

Comm_Rx_Buffer and Comm_Tx_Buffer are the shared buffers for the UART, SPI and SMBus communications interfaces. Since only one of these interfaces is active at any one time, the buffers are shared. Comm_TX_Buffer data is sent from the CP2501 to the touch screen module. Data received from the touch screen module is stored in Comm_Rx_Buffer.

User firmware stores the formatted and processed touch data in the USB_Position_Buffer. The System Firmware pulls the touch data directly from this buffer and sends it to the USB host.

The USB_Control_Buffer is a bi-directional buffer used to pass data between the USB host and the user firmware using the custom control report. Typically applications will not need to use this buffer as all touch data communication must pass through the USB_Position_Buffer.

4.2.2. Shared Variables

Table 4 is a list of all variables shared between the System Firmware and user firmware. User firmware should only write to the bytes/bits that are marked as Write.

Table 4. System Firmware Variables

Name	User R/W	Size	Memory Space	Address
CP250x_System_Flags		1 byte	RAM - BDATA	0x20
CP250x_USB_Touch_Send_Pending	Write	bit	bit-addressable	bit 0 (0)
CP250x_USB_Mouse_Send_Pending	Write	bit	bit-addressable	bit 1 (1)
CP250x_USB_Pen_Send_Pending	Write	bit	bit-addressable	bit 2 (2)
CP250x_UART_RX_Data_Ready	Read/Write	bit	bit-addressable	bit 3 (3)
CP250x_USB_Enter_Suspend	Read	bit	bit-addressable	bit 4 (4)
CP250x_System_Modes		1 byte	RAM - BDATA	0x21
CP250x_Mouse_Mode	Read	bit	bit-addressable	bit 0 (8)
CP250x_SingleTouch_Mode	Read	bit	bit-addressable	bit 1 (9)
CP250x_MultiTouch_Mode	Read	bit	bit-addressable	bit 2 (10
CP250x_SinglePen_Mode	Read	bit	bit-addressable	bit 3 (11)
CP250x_MultiPen_Mode	Read	bit	bit-addressable	bit 3 (12)
CP250x_Control_Flags		1 byte	RAM - BDATA	0x22
CP250x_USB_Control_Write	Read	bit	bit-addressable	bit 0 (16)
CP250x_USB_Control_Read	Read	bit	bit-addressable	bit 1 (17)
CP250x_Control_Data_Ready	Read/Write	bit	bit-addressable	bit 2 (18)
CP250x_Status	Read	1 byte	XRAM	0x01F0
CP250x_Param1	Read/Write	1 byte	XRAM	0x01F1
CP250x_Param2	Write	1 byte	XRAM	0x01F2
CP250x_Param3	Write	1 byte	XRAM	0x01F3
CP250x_Param4	Write	1 byte	XRAM	0x01F4
CP250x_Param5	Write	1 byte	XRAM	0x01F5
CP250x_Param6	Write	1 byte	XRAM	0x01F6
CP250x_Param7	Write	1 byte	XRAM	0x01F7
CP250x_Counter	Read/Write	1 byte	XRAM	0x01F8
CP250x_Counter_Status	Read/Write	1 byte	XRAM	0x01F9

4.2.2.1. CP250x_System_Flags

The CP250x_System_Flags are used to indicate when certain types of data are ready for processing by the user firmware or by the System Firmware.

CP250x_USB_Touch_Send_Pending: User firmware should set this bit once new touch data has been copied to the USB_Position_Buffer. The System Firmware will clear this bit once it has copied the touch data to the internal USB FIFO. User firmware should not write to the USB_Position_Buffer until this bit has been cleared.

CP250x_USB_Mouse_Send_Pending: User firmware should set this bit once new mouse data has been copied to the USB_Position_Buffer. The System Firmware will clear this bit once it has copied the mouse data to the internal USB FIFO. User firmware should not write to the USB_Position_Buffer until this bit has been cleared.

CP250x_USB_Pen_Send_Pending: User firmware should set this bit once new pen data has been copied to the USB_Position_Buffer. The System Firmware will clear this bit once it has copied the pen data to the internal USB FIFO. User firmware should not write to the USB_Position_Buffer until this bit has been cleared.

CP250x_UART_RX_Data_Ready: This bit indicates to user firmware that new UART data is available. When using the UART interface, user firmware should poll this bit as part of the main() loop. User firmware should use the CP250x_UART_Dequeue() to retrieve the data from the UART receiver buffer. If more new data is available in the UART receive buffer after calling CP250x_UART_Dequeue(), the CP250x_UART_RX_Data_Ready bit remains set, otherwise it is cleared by the System Firmware.

CP250x_USB_Enter_Suspend: This bit indicates to user firmware that the System Firmware has received a Suspend command from the USB host. The user firmware should poll this bit as part of the main() loop and call CP250x_USB_Suspend() or CP250x_USB_Suspend_Remote_Wakeup() when it is set. Upon exiting Suspend mode, this bit is cleared by the System Firmware.

4.2.2.2. CP250x_System_Modes

This bit-addressable variable indicates which mode or modes the operating system has placed the device in. Depending on which mode is active, the user firmware should send only that type of data to the host. If none of the bits are set, the operating system has not set an operating mode and no touch, pen, or mouse data should be sent to the host

CP250x_Mouse_Mode: When set, the user firmware should only send mouse position and button data.

CP250x_SingleTouch_Mode: When set, the user firmware can send data for one touch point.

CP250x_MultiTouch_Mode: When set, the firmware can send data for multiple touch points. Since the CP2501 operates in serial mode, the format of the data in the USB_Position_Buffer is exactly the same as the format for a single touch point. The only difference is that user firmware can change the Contact ID field to different values.

CP250x_SinglePen_Mode: When set, the user firmware can send data for one pen input.

CP250x_MultiPen_Mode: When set, the user firmware can send data for multiple pen inputs. Since the CP2501 operates in serial mode, the format of the data in the USB_Position_Buffer is exactly the same as the format for a single pen input. The only difference is that user firmware can change the Contact ID field to different values.

4.2.2.3. CP250x_Control_Flags

CP250x_USB_Control_Write: When set, indicates to user firmware that the USB host has sent data to the device. The received data is stored in USB_Control_Buffer. User firmware should clear this bit once the data is read from the USB_Control_Buffer.

CP250x_USB_Control_Read: When set, indicates to user firmware that the USB host has requested control data from the device.

CP250x_USB_Control_Data_Ready: User firmware should set this bit once new control data has been copied to the USB_Control_Buffer. The System Firmware will clear this bit once it has copied the control data to the internal USB FIFO. User firmware should not write to the USB_Control_Buffer until this bit has been cleared.

4.2.2.4. CP250x_Status

This variable returns the status of the System Firmware function. User firmware should check this variable after each API function call to determine if the function completed successfully. The full list of return codes is provide in Table 5, “Error Code Descriptions,” on page 28.

4.2.2.5. CP250x_Param1 through CP250x_Param7

These variables are used to pass parameters to the System Firmware function calls from user firmware. The function calls provided by CP250x_API.h automatically copy the parameters to these variables. Typical user firmware will not need to directly access these variables.

4.2.2.6. CP250x_Counter and CP250x_Counter_Status

If the Counter is running, the CP250x_Counter variable is incremented every 500 us. CP250x_Counter_Status contains two active bits. Bit 7 is the Run bit. If Run is set to 1, the Counter is running, else it is stopped. Bit 0 is the Overflow bit, which is set when the Counter overflows from 255 to 0. User firmware can clear the Overflow bit as required. CP250x_Counter_Status is not bit-addressable and the individual bits must be accessed using C-language mask operations.

4.3. System Configuration Functions

The function prototypes defined in the following sections are macro calls which are defined in CP250x_API.h. The macro calls copy the function parameters to specific XRAM variables CP250x_Param1 through CP250x_Param7 as required. The return value for all System Firmware functions is returned in the XRAM variable CP250x_Status. The XRAM variables are defined in CP250x_Main.c The #defines used for function parameters and return values are defined in CP250x_API.h.

4.3.1. CP250x_System_Init

Description: Initializes the regulator and system clock.

Prototype: `void CP250x_System_Init (U8 regulator, U8 sysclk);`

Example Call: `CP250x_System_Init (BUSPOW, OSC_48);`

Parameters:

1. *regulator*—BUSPOW, SELFPOW_VREGOFF, or SELFPOW_VREGON

   BUSPOW—The on-chip regulator is turned on. During USB enumeration, the device requests current from the USB host. The amount of current requested is listed in the USB descriptor, which is generated by the CP2501 Configuration Wizard.

   SELFPOW_VREGOFF—The on-chip regulator is turned off. During USB enumeration, no current is requested from the USB host. The device is powered with a 3 V source applied to VDD.

   SELFPOW_VREGON—The on-chip regulator is turned on. During USB enumeration, no current is requested from the USB host. The device is powered with a 5 V source applied to VREGIN.

2. *sysclk*—OSC_48 or OSC_24

   OSC_48—The 8051 core operates at 48 MHz.
   OSC_24—The 8051 core operates at 24 MHz.

Return Value: CP250x_Status = FUNC_SUCCESS
FUNC_OUT_OF_RANGE

4.3.2. CP250x_USB_Init

Description: Initializes the USB interface and starts the 500 µs Counter. This function should only be called after all other peripherals are initialized.

Prototype: *`void CP250x_USB_Init ();`*

Example Call: *`CP250x_USB_Init ();`*

Return Value: CP250x_Status **=** FUNC_SUCCESS

4.3.3. CP250x_GPIO_Init

Description: Initializes the GPIO pin modes and their initial values.

Prototype: `void CP250x_GPIO_Init (U8 gpio70_mode, U8 gpio158_mode,U8 gpio70_latch, U8 gpio158_latch);`

Example Call: `CP250x_GPIO_Init (0xFF, 0xFF, 0x00, 0x00);`

Parameters:

1. *gpio70_mode*—Configures the mode for GPIO pins 7:0. Set corresponding bit to 1b for push-pull mode, and 0b for open-drain and input modes.
2. *gpio158_mode*—Configures the mode for GPIO pins 15: 8. Set corresponding bit to 1b for push-pull mode, and 0b for open-drain and input modes.
3. *gpio70_latch*—Configures the initial latch value for GPIO pins 7:0 configured as open-drain or push-pull outputs. Set corresponding bit to 1b for GPIO pins configured as inputs.
4. *gpio158_latch*—Configures the initial latch value for GPIO pins 15:8 configured as open-drain or push-pull outputs. Set corresponding bit to 1b for GPIO pins configured as inputs.

Return Value: CP250x_Status = FUNC_SUCCESS

4.4. USB Suspend Functions

4.4.1. CP250x_USB_Suspend

Description: When the USB host signals the CP2501 device to enter USB suspend mode, the System Firmware sets the *CP250x_USB_Enter_Suspend* bit which should be polled as part of the main() loop. Once this bit is set, the user firmware has a choice of calling this function or CP250x_USB_Suspend_Remote_Wakeup().

This function puts the device in a low-power state, and code execution will remain in this function until the device is signaled by the USB host to exit USB suspend mode. User firmware should set the GPIO pins to reduce current consumption before calling this function.

Prototype: `void CP250x_USB_Suspend ();`

Example Call: `CP250x_USB_Suspend ();`

Return Value: CP250x_Status **=** FUNC_SUCCESS

4.4.2. CP250x_USB_Suspend_Remote_Wakeup

Description: When the USB host signals the CP2501 device to enter USB suspend mode, the System Firmware sets the *CP250x_USB_Enter_Suspend* bit which should be polled as part of the main() loop. Once this bit is set, user firmware has a choice of calling this function or CP250x_USB_Suspend().

This function puts the USB transceiver in a suspend state, and returns control back to user firmware. The purpose of this function is to allow the user firmware to remotely wake up the USB host once it determines a wake condition has occurred. The USB host can also wake up the device from this suspend mode.

Note: The device will not meet the USB Suspend current requirements in this suspend mode. This function should be called only when the device is in self-powered mode.

Prototype: `void CP250x_USB_Suspend_Remote_Wakeup ();`

Example Call: `CP250x_USB_Suspend_Remote_Wakeup ();`

Return Value: CP250x_Status **=** FUNC_SUCCESS

4.4.3. CP250x_USB_Remote_Wakeup

Description: User firmware calls this function to wake up the USB host using remote wakeup signalling. This function should only be called when the device is in a Suspend state entered by calling CP250x_USB_Suspend_Remote_Wakeup(). Once the USB host wakes up due to the signalling, it will clear the Suspend state of the device and the System Firmware will clear *CP250x_USB_Enter_Suspend* bit. This function returning FUNC_SUCCESS only indicates that remote wakeup signalling was sent, but does not indicate that the USB host has woken up. User firmware can continue to call CP250x_USB_Remote_Wakeup() until the *CP250x_USB_Enter_Suspend* bit is cleared. If this function is called when the USB host in not in a suspended state, it will return FUNC_REMOTE_WAKE_DISABLE.

Prototype: `void CP250x_USB_Remote_Wakeup ();`

Example Call: `CP250x_USB_Remote_Wakeup ();`

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_REMOTE_WAKE_DISABLE

4.5. Interface Functions

4.5.1. CP250x_UART_Init

Description: Initializes the UART peripheral and resets the transmit and receive buffers.

Prototype: `void CP250x_UART_Init (U32 baud_rate, U8 num_bits, U8 parity, U8 stop_bits);`

Example Call: `CP250x_UART_Init (115200, EIGHT_DATA_BITS, NO_PARITY, ONE_STOP_BIT);`

Parameters:

1. *baud_rate*—The speed in bits per second at which the UART transmits and receives data. The valid range for *baud_rate* is 9600 baud to 230,400 baud.
2. *num_bits*—The number of bits in each byte of data. The valid options are FIVE_DATA_BITS, SIX_DATA_BITS, SEVEN_DATA_BITS, and EIGHT_DATA_BITS.
3. *parity*—The type of parity bit appended to each data byte. The valid options are NO_PARITY, ODD_PARITY, EVEN_PARITY, MARK_PARITY, and SPACE_PARITY. If NO_PARITY is selected, no extra bit is appended.
4. *stop_bits*—The number of stop bits used for each data byte. The valid options are ONE_STOP_BIT and TWO_STOP_BITS. If *num_bits* is set to FIVE_DATA_BITS, the TWO_STOP_BITS option will appear as 1.5 stop bits.

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_OUT_OF_RANGE

4.5.2. CP250x_UART_Write

Description: Transmits data using UART from the Comm_Tx_Buffer. The user firmware must first copy the bytes to the Comm_Tx_Buffer before calling this function. UART_TX_Idle bit is set to 0 if a UART transmission is in progress and the user firmware should poll this bit before calling the function. If this function is called when the UART is not idle, the function will return FUNC_BUSY. User firmware should not write to the Comm_Tx_Buffer until all data is transmitted and UART_TX_Idle is back to 1b.

Data transmission can still be in progress when the function returns. User firmware should poll the UART_TX_Idle to determine when all the has been transmitted.

Prototype: `void CP250x_UART_Write (U8 length);`

Example Call: `CP250x_UART_Init (20);`

Parameters:

1. *length*—The number of bytes to transmit.

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_BUSY
FUNC_OUT_OF_RANGE

4.5.3. CP250x_UART_Dequeue

Description: Returns one byte from the Comm_Rx_Buffer, which is a circular FIFO. The System Firmware will set the CP250x_UART_RX_Data_Ready bit if new data is ready. This CP250x_UART_RX_Data_Ready bit will stay set to 1b until no new data remains in the Comm_Rx_Buffer. If this function is called when there is no data in the buffer, FUNC_UART_NO_DATA is returned.

The new data is returned in the CP250x_Param1 variable.

Prototype: *void CP250x_UART_Dequeue ();*

Example Call: *CP250x_UART_Dequeue ();*

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_UART_NO_DATA

4.5.4. CP250x_UART_Poll

Description: Returns the status of the UART communications interface. If the status is FUNC_UART_IDLE, user firmware can safely start a data transfer.

Prototype: *void CP250x_UART_Poll ();*

Example Call: *CP250x_UART_Poll ();*

Return Value: CP250x_Status **=** FUNC_BUSY
FUNC_UART_IDLE

4.5.5. CP250x_SPI_Master_Init

Description: Initializes the SPI peripheral and resets the transmit and receive buffers.

Prototype: *void CP250x_SPI_Master_Init (U32 clock_rate, U8 polarity_phase);*

Example Call: *CP250x_SPI_Master_Init (1000000, PH1_POLL0);*

Parameters:

1. *clock_rate*—The speed in bits per second at which the SPI transmits and receives data. The valid range for *clock_rate* is 10,000 to 4,000,000 bps.
2. *polarity_phase*—This parameter configures two features of the SPI peripheral: clock polarity and clock phase. The valid options are
   PH1_POLL0—Phase = 0 and Polarity = Low
   PH1_POLL1—Phase = 1and Polarity = High
   PH2_POLL0—Phase = 0 and Polarity = Low
   PH2_POLL1—Phase = 1and Polarity = High

   The clock phase selects which of two phases to clock the data (rising edge or falling edge). The clock polarity selects the idle value of the SPI_SCLK pin. Both the master CP2501 device and the slave device must be configured to use the same polarity and phase.

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_OUT_OF_RANGE

4.5.6. CP250x_SPI_Master_Transfer

Description: Transmits and receives data using the SPI interface. The function can Write data, Read data, or Write-and-Read data. Data is transmitted from the Comm_Tx_Buffer. If the function is configured to Write or Write-and-Read, data must be copied to the Comm_Tx_Buffer before calling the function. When data is read from the SPI slave device, it is stored into the Comm_Rx_Buffer starting at index 0.

SPI_TXRX_Idle bit is set to 0 if a SPI transfer is in progress. User firmware should poll this bit before calling the function. If this function is called when the SPI is not idle, the function will return FUNC_BUSY. User firmware should not write to the Comm_Tx_Buffer until all data is transmitted and SPI_TXRX_Idle is set back to 1b.

Data transmission can still be in progress when the function returns. User firmware should poll the SPI_TXRX_Idle to determine when all the has been transmitted or received.

Prototype: *void CP250x_SPI_Master_Transfer (U8 transfer_type, U8 length, U8 dummy_byte);*

Example Call: *CP250x_SPI_Master_Transfer (SPI_WRITE, 20, 0x00);*

Parameters:

1. *transfer_type*—The valid options are SPI_WRITE, SPI_READ, and SPI_WRITEREAD.

   SPI_WRITE—The function transfers data from the Comm_Tx_Buffer to the SPI slave device. The *DummyByte* is ignored in this configuration. The Comm_Rx_Buffer is not changed.

   SPI_READ—The function initiates a read from the SPI slave device and stores the received data in the Comm_Rx_Buffer. The CP2501 will transmit the *DummyByte* value over the SPI_MOSI pin when sending a clock to initiate the SPI reads.

   SPI_WRITEREAD—The function transfers data from the Comm_Tx_Buffer to the SPI slave

device, and also receives data from the SPI slave into the Comm_Rx_Buffer. The *DummyByte* is ignored in this configuration.

2. *length*—The number of bytes to transfer to/from the SPI slave device.
3. *dummy_byte*—This byte is used only during a SPI_READ and is transferred to the SPI slave device to initiate a SPI read.

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_BUSY
FUNC_OUT_OF_RANGE
FUNC_SPI_NSS_BUSY

4.5.7. CP250x_SPI_Poll

Description: Returns the status of the SPI communications interface. If the status is FUNC_SPI_IDLE, user firmware can safely start a data transfer.

Prototype: `void CP250x_SPI_Poll ();`

Example Call: `CP250x_SPI_Poll ();`

Return Value: CP250x_Status **=** FUNC_BUSY
FUNC_SPI_IDLE
FUNC_SPI_MODE_FAULT

4.5.8. CP250x_SMBus_Master_Init

Description: Initializes the SMBus peripheral and resets the transmit and receive buffers.

Prototype: `void CP250x_SMBus_Master_Init (U32 clock_rate);`

Example Call: `CP250x_SMBus_Master_Init (400000);`

Parameters:

1. *clock_rate*—The speed in bits per second at which the SMBus transmits and receives data. The valid range for *clock_rate* is 10,000 to 400,00 bps.

Return Value: CP250x_Status **=** FUNC_SUCCESS
FUNC_OUT_OF_RANGE

4.5.9. CP250x_SMBus_Master_Transfer

Description: Transmits and receives data using the SMBus interface. The function can Write data or Read data. Data is transmitted from the Comm_Tx_Buffer. Data must be copied to the Comm_Tx_Buffer before calling the function. When data is read from the SMBus slave device, it is stored into the Comm_Rx_Buffer starting at index 0.

SMBus_Idle bit is set to 0 if a SMBus transfer is in progress. User firmware should poll this bit before calling the function. If this function is called when the SMBus is not idle, the function will return FUNC_BUSY. User firmware should not write to the Comm_Tx_Buffer until all data is transmitted and SMBus_Idle is back to 1b.

Data transmission can still be in progress when the function returns. User firmware should poll the SPI_TXRX_Idle to determine when all the has been transmitted or received.

Prototype:
```
void CP250x_SMBus_Master_Transfer (U8 transfer_type, U8 length, U8 slave_address);
```

Example Call:
```
CP250x_SMBus_Master_Transfer (SMBUS_FUNC_WRITE, 20, 0xA0);
```

Parameters:

1. *transfer_type* —The valid options are SMBUS_FUNC_WRITE and SMBUS_FUNC_READ.
   SMBUS_FUNC_WRITE—The function transfers data from the Comm_Tx_Buffer to the SMBus slave device.
   SMBUS_FUNC_READ—The function initiates a read from the SMBus slave device and stores the received data in the Comm_Rx_Buffer, starting at index 0.
2. *length*—The number of bytes to transfer to/from the SMBus slave device.
3. *slave_address*—This byte is the slave address to which data is transferred or data is read from.

Return Value:
CP250x_Status **=** FUNC_SUCCESS
FUNC_BUSY
FUNC_OUT_OF_RANGE

4.5.10. CP250x_SMBus_Poll

Description: Returns the status of the SPI communications interface. If the status is FUNC_SMBUS_IDLE, user firmware can safely start a data transfer.

Prototype:
```
void CP250x_SMBus_Poll ();
```

Example Call:
```
CP250x_SMBus_Poll ();
```

Return Value:
CP250x_Status **=** FUNC_BUSY
FUNC_SMBUS_IDLE
FUNC_SMBUS_ARB_LOST
FUNC_SMBUS_NO_ACK
FUNC_SMBUS_SCL_TO

4.6. Counter Functions

4.6.1. CP250x_Start_Counter

Description: Restarts the counter if it is stopped and sets the RUN bit in CP250x_Counter_Status. Also resets the internal timer so that the next increment of the CP250x_Counter variable will occur 500 µs after this function call. The value of CP250x_Counter is not changed.

Prototype: *void CP250x_Start_Counter()*

Example Call: *CP250x_Start_Counter();*

Return Value: CP250x_Status **=** FUNC_SUCCESS

4.6.2. CP250x_Stop_Counter

Description: Stops CP250x_Counter from incrementing and clears the RUN bit in CP250x_Counter_Status.

Prototype: *void CP250x_Stop_Counter()*

Example Call: *CP250x_Stop_Counter()*

Return Value: CP250x_Status **=** FUNC_SUCCESS

4.7. CP250x_Status Return Values

Each System Firmware API function returns a value in CP250x_Status after the function to indicate that the function returned successfully or to describe an error. describes each error code.

Note: All functions share the same variable, CP250x_Status, to return the status. If one API function is called while another API function is already in progress, the status value of the second function will overwrite the FUNC_BUSY status of the first function. To prevent this, no API function should be called while CP250x_Status is set to FUNC_BUSY.

Table 5. Error Code Descriptions

Definition	Value	Description
FUNC_SUCCESS	0x00	Function completed successfully.
FUNC_BUSY	0x01	The communication interface is transferring data.
FUNC_OUT_OF_RANGE	0x02	One of the parameters passed to the function is out of range. The function returned without performing any operation.
FUNC_SPI_IDLE	0x10	SPI Interface is idle and ready for data transfer
FUNC_SPI_MODE_FAULT	0x11	SPI interface experienced a mode fault. User firmware should reattempt the data transfer.
FUNC_SPI_NSS_BUSY	0x12	Another SPI master is transmitting on the bus. User firmware should attempt the transfer at a later time.
FUNC_SMBUS_IDLE	0x20	SMBus Interface is idle and ready for data transfer.
FUNC_SMBUS_SCL_TO	0x21	SMBus interface experienced an SCL low timeout. Any pending transfer is cancelled.
FUNC_SMBUS_NO_ACK	0x22	A byte transmitted by the SMBus master was not acknowledged by a slave device.
FUNC_SMBUS_ARB_LOST	0x23	Another SMBus master won the address arbitration. User firmware should attempt the transfer at a later time.
FUNC_UART_IDLE	0x30	UART Interface is idle and ready for data transfer.
FUNC_UART_NO_DATA	0x31	Indicates that there is no new data in Comm_RX_Buffer from the UART interface.
FUNC_REMOTE_WAKE_DISABLE	0x40	Remote Wakeup is currently disabled by the USB host. User Firmware should wait till CP250x_USB_Enter_Suspend is set.

5. CP2501 Bootloader

The CP2501 devices' user firmware is updatable in-system through the debug pins or through the System Firmware's USB bootloader. The CP2501 Bootloader application is a Windows program that accepts a standard Intel hex file as the input. By default, a Silicon Labs project generated by the CP2501 Configuration Wizard is configured to generate a hex file when the project is built. Figure 10 is a screenshot of the program with a CP2501 device already connected to the PC.

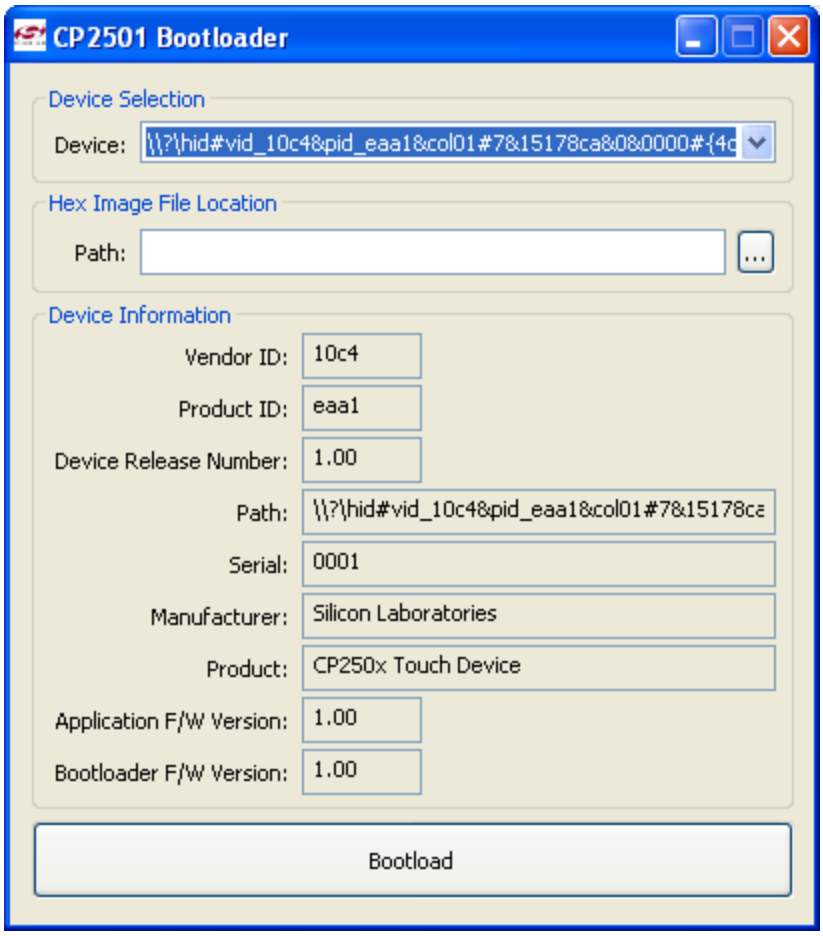

Figure 10. CP2501 Bootloader

Once the application is loaded, all available HID devices in the system are listed in the Device Selection drop-down menu. **Search** for the first instance of the Vendor ID (VID) and Product ID (PID) that matches the desired CP2501 device. Once selected, the relevant device information is populated the Device Information section.

Next, **select** the hex image file under Hex Image File Location and **click Bootload**. The CP2501 Bootloader application will automatically put the device in bootloader mode, update the user firmware image, and reset the device once the update is complete. If the selected image file includes bytes outside of the valid programming range, the bootloader will indicate an error and not program the image.

The CP2501 Bootloader does not automatically program the Validation Signature bytes. These bytes must be included as part of the hex image. If the Validation Signature bytes are not programmed, the System Firmware will not jump to the user firmware after a device reset. A Configuration Wizard generated project includes the Validation Signature Bytes.

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.