Xilinx® Spartan™-6 LX16 Evaluation Kit PSoC Firmware User Guide

Contents

1 Introduction 3
2 Firmware Description 4
2.1 Initialization 4
3 Commands 6
3.1 Command Format 6
3.2 Command Parser 6
3.3 Supported Commands 8
3.3.1 load_config 8
3.3.2 3.2.2 drive_prog 9
3.3.3 3.2.3 drive_mode 9
3.3.4 3.2.4 spi_mode 9
3.3.5 3.2.5 read_init 10
3.3.6 3.2.6 read_done 10
3.3.7 3.2.7 sf_transfer 10
3.3.8 3.2.8 ss_program 11
3.3.9 3.2.9 jtag_mux 12
3.3.10 3.2.10 get_config 12
3.3.11 3.2.11 usb_bridge 12
3.3.12 3.2.12 fpga_rst 13
3.3.13 3.2.13 get_ver 13
4 4.0 Conclusion 14
5 Revisions 15
6 Getting Help and Support 16
7 Appendix B – PSoC Port Configuration 17

Figures

Figure 1 – Firmware Operation Flow Diagram 5
Figure 2 – Command Strings 7

Tables

Table 1 – Default I/O Configuration 4

1 Introduction

The Spartan-6 LX16 Evaluation Kit provides a complete hardware environment for designers to accelerate their time to market. The kit delivers a stable platform to develop and test designs targeted to the low-cost and low-power Xilinx Spartan-6 FPGA family and the Cypress PSoC 3 Mixed Signal Array devices. The installed Spartan-6 LX16 device offers a prototyping environment to effectively demonstrate the enhanced benefits of low-cost Xilinx FPGA solutions.

Included in this kit is the capability to program the FPGA, program a serial and parallel flash on the board as well as provide bridging from the USB to a MicroBlaze controlled UART peripheral from the FPGA. All of these functions are made possible by the Cypress PSoC CY8C3866AXI-40 programmable MCU device. In order to carry out these functions the PSoC is controlled by the PC application GUI called AvProg via a USB interface. The firmware loaded at the factory provides all of the listed functionality and this document outlines the firmware operation of the PSoC.

2 Firmware Description

The PSoC firmware is responsible for controlling all of the programming/reading of the Spansion serial flash, the FPGA programming and all of the data transfers between the host and the Spartan 3A Evaluation board for USB to UART bridging. This section will outline the organization of the PSoC firmware.

2.1 Initialization

The PSoC peripherals are initialized at the start of the application. The first task for the firmware is to set the default I/O configurations. This is accomplished via the Config_IO routine. The default configuration for the PSoC I/O is shown below.

Table 1 – Default I/O Configuration

Signal	Direction	Default Value
PSOC_FPGA_PROG	output	1
FPGA_DONE	input	na
FPGA_INIT_B	input	na
PSOC_FPGA_M[2:0]	output	Z
PSOC_SPI_MODE	output	0
PSOC_SPI_SEL#	output	Z
PSOC_MISO	input	na
PSOC_MOSI	output	1
SPI_CLK	output	Z
JTAG_MUX_CTL	output	1
PSOC_Px_xx	I/O	Z
UART_RXD	input	na
UART_TXD	output	0
FPGA_PUSH A/B	output	0
FPGA_RESET	output	0
IIC_SCL	I/O	Z
IIC_SDA	I/O	Z
TMS	output	1
TDI	output	1
TDO	input	na
TCK	output	1
CLK_12MHZ	output	12 MHz clock

All I/O at start-up are configured to the values above. The next task for the firmware is to load the default configuration and start the peripherals. The term “Start” is used as part of the Cypress APIs to initialize and enable associated peripherals. In this application the default configuration is called the UARTConfig. After loading the UART configuration the peripherals are started. The CapSense peripheral is started and the baseline values for the capacitance of the pads are measured. After that the threshold value for the measurements are set and the peripheral is operational. The UART is initialized as part of the main application loop.

The main application loop continually monitors multiple aspects of the PSoC operation. A flow diagram is shown below to outline the firmware operation.

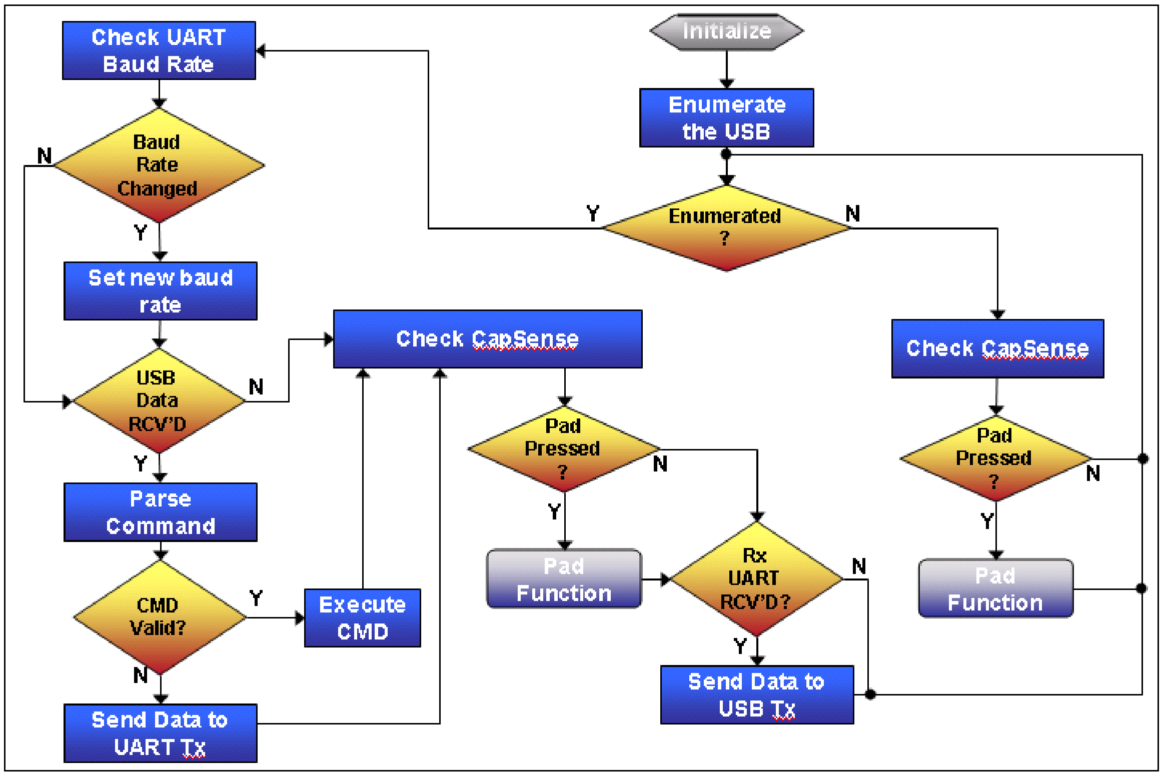


Figure 1 – Firmware Operation Flow Diagram

The first thing the loop must do is enumerate the USB. The only peripheral that is active prior to the USB being enumerated is the CapSense pad monitoring. Once the USB is enumerated, then the monitoring of the UART baud rate and checking for received data and sending data can be started. The UART baud rate is determined by the settings generated within the AvProg GUI and is sent via the SetLineCoding request over the USB to the PSoC. These values are sent once the user presses the “Connect to Com X” button in the GUI. If the current value set for the UART has changed then the rate is modified in the firmware.

Now that the USB has been enumerated and the UART rate has been configured, the PSoC is ready to receive commands or bridge the data between the FPGA and the host.

3 Commands

In order to initiate any sequence to program or read back anything from the Spartan3A Evaluation board a command protocol was developed for the AvProg GUI to use. This is an acknowledgment based protocol that provides a robust communications sequence for all commands. The sequence of events varies based on the command as to when the host expects the acknowledgments and will be described in this section.

3.1 Command Format

The commands follow a specific format. Each command is a text string consisting of one or more arguments. The first argument of the string is the actual command and any remaining arguments provide information associated with the command. The command format is shown below.

Argument0 Argument1 … ArgumentN <*NULL*>

Where:

Argument0 is the actual command
Argument1 identifies specific parameters for the command
<*NULL*> identifies the end of string character.

As of this writing the existing commands have at most two arguments (0 and 1). In the firmware the arguments are saved to a global variable array called ArgV. The Argument0 value is saved to location 0 in the array and the subsequent arguments are saved to the subsequent index to the array. So ArgV[0] in the firmware contains the string for the actual command. ArgV[1] contains the first argument for the command.

3.2 Command Parser

The command parser is responsible for evaluating the incoming USB data, determining whether that data has a valid command and then branching to the appropriate command function. This is done using function pointers found within a structure that identifies the associated text for the command. There are several elements to the command parser.

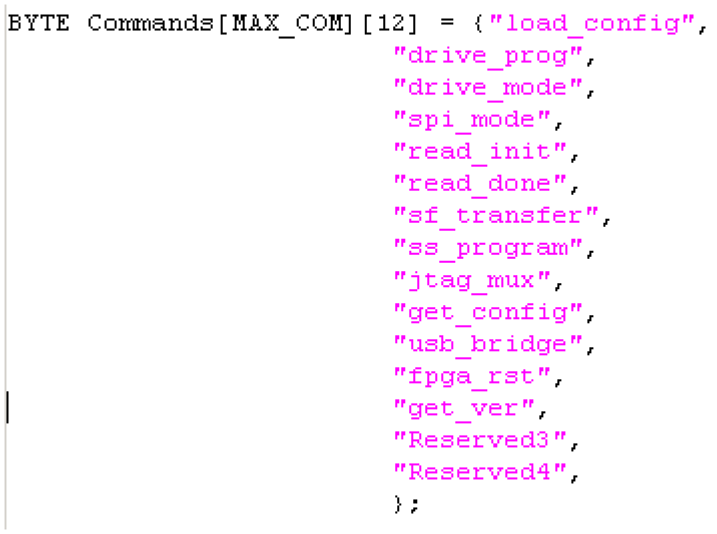
The first aspect required by the parser is the ability to sort out all of the arguments for each command string received from the host. The function responsible for reading and storing these arguments is get_command. This function parses each character and stores it into the ArgV array while looking for the space as an argument delimiter. Get_command is called from Main in the application. The get_command function prototype is shown below:

BYTE GetCommand(BYTE *CmdStr);

Usage: ArgC = GetCommand(&pData[0]);

Where: ArgC is the number of arguments returned
pData is the array where the USB functions returned data.

The next aspects are the variable declarations for the command strings that will be received by the host application. Figure 2 shows the variable declarations for these strings. These are the strings that AvProg passes to the PSoC if it wants to carry out one of these functions.



```
BYTE Commands[MAX_COM][12] = {"load_config",
                              "drive_prog",
                              "drive_mode",
                              "spi_mode",
                              "read_init",
                              "read_done",
                              "sf_transfer",
                              "ss_program",
                              "jtag_mux",
                              "get_config",
                              "usb_bridge",
                              "fpga_rst",
                              "get_ver",
                              "Reserved3",
                              "Reserved4",
                              };
```

Figure 2 – Command Strings

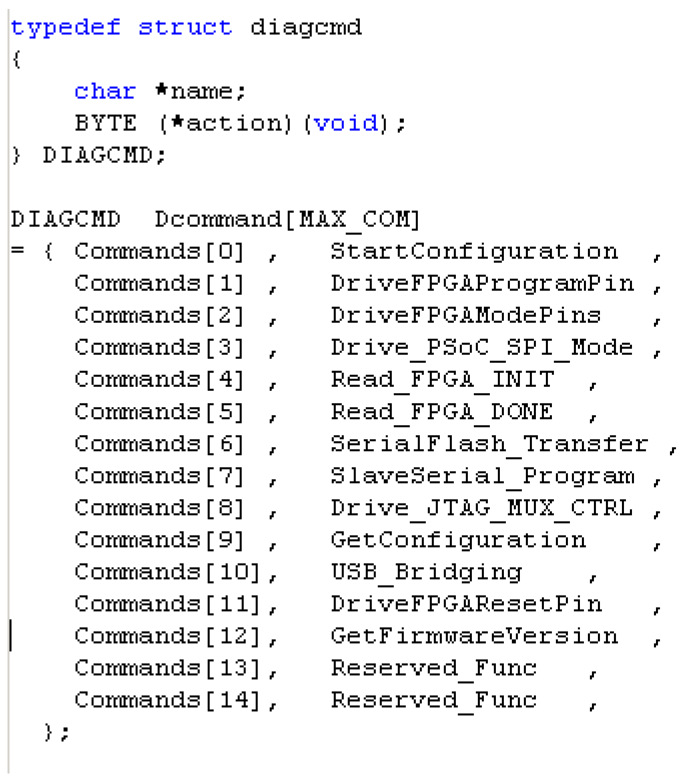
When the string is received from the host, it is compared by the PSoC to determine if the command is valid or not. If the string received matches one of the strings in the variable list shown above then the associated function is called. If there isn't a match then the data is sent to the USB bridge function.

Once we have parsed the string using get_command we can now match it to our string list above. This is accomplished via the ParseCommand function. The usage is shown below:

BYTE ParseCommand(void);

Usage: Result = ParseCommand();
Where:
Result indicates whether the function completed successfully or not.

ParseCommand uses a structure that defines a function pointer to the associated functions.



```
typedef struct diagcmd
{
    char *name;
    BYTE (*action)(void);
} DIAGCMD;

DIAGCMD  Dcommand[MAX_COM]
= { Commands[0] ,   StartConfiguration  ,
    Commands[1] ,   DriveFPGAProgramPin ,
    Commands[2] ,   DriveFPGAModePins   ,
    Commands[3] ,   Drive_PSoC_SPI_Mode ,
    Commands[4] ,   Read_FPGA_INIT  ,
    Commands[5] ,   Read_FPGA_DONE  ,
    Commands[6] ,   SerialFlash_Transfer ,
    Commands[7] ,   SlaveSerial_Program ,
    Commands[8] ,   Drive_JTAG_MUX_CTRL ,
    Commands[9] ,   GetConfiguration    ,
    Commands[10],   USB_Bridging    ,
    Commands[11],   DriveFPGAResetPin   ,
    Commands[12],   GetFirmwareVersion  ,
    Commands[13],   Reserved_Func   ,
    Commands[14],   Reserved_Func   ,
  };
```

There is a relationship between the variable with the string declarations and the function names to be called. The ParseCommand function cycles through the strings to obtain the command match (ArgV[0]) and calls the function with the same index value. For example, if the host sends the command:

load_config 0<***NULL***>

where:

ArgV[0] = load_config<***NULL***>
ArgV[1] = 0<***NULL***>

The ParseCommand function compares the ArgV[0] with the Command variable and detects a match at index 0. This in turn points the function call to the address at index 0 which is the StartConfiguration routine. Note that the <***NULL***> characters are added as part of the get_command parsing.

The main application calls the ParseCommand function within an "if" conditional. If the return value from the ParseCommand is SUCCESS then the main application will generate an "ack."

3.3 Supported Commands

The following section describes the supported commands between the AvProg GUI and the PSoC firmware.

3.3.1 load_config

Since the PSoC device is completely reprogrammable, it allows the firmware designer the ability to change configurations dynamically, or in other words, while the device is operating. The concept of dynamic reconfiguration and the configurations were discussed earlier in Section 1.1 and Section 1.2. In order to support dynamic reconfiguration the firmware must remain cognizant of the current loaded configuration. In order to load a configuration the firmware must first disable the peripherals in the current configuration and then unload the current configuration. Once that is complete, then the firmware can load the new configuration and enable the associated peripherals that are part of that configuration. The load_config command sets the current configuration. The syntax is shown below:

```
load_config <arg>

where valid values for <arg> are:
        0x30 - UART
        0x31 - SPI
        0x32 - JTAG
```

This command returns an ack for successful command completion. If the command is not successful then no response is generated.

3.3.2 3.2.2 drive_prog

The FPGA has a pin called PROG that initiates a new FPGA program cycle by erasing the current contents of the RAM and initiating the program sequence. The drive_prog command drives the pin from the PSoC connected to that FPGA pin. The syntax is shown below:

```
drive_prog <arg>
```

where valid values for arg are 0x30 to drive the pin low or 0x31 to drive the pin high.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.3 3.2.3 drive_mode

The FPGA has 3 mode pins M2:M0. These pins are used to determine what configuration mode the FPGA will use to load the internal RAM. The drive_mode command drives the pin from the PSoC connected to those FPGA pins. The syntax is shown below:

```
drive_mode <arg>
```

where valid values for arg range from 0x30 to 0x37 to drive the pins to the binary equivalent and 0x38 to tri-state the pins. For example, drive_mode 0x32 drives M2:M0 as 010.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.4 3.2.4 spi_mode

There are multiple sources that can drive the SPI on the Spartan 3A Evaluation board. In order to control what device gains access to the SPI a control pin from the PSoC was added. The syntax is shown below:

```
spi_mode <arg>
```

where valid values for arg are 0x30 to drive the pin low or 0x31 to drive the pin high.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.5 3.2.5 read_init

There is a pin on the FPGA called INIT that is used as part of the configuration sequence of the FPGA. This pin is used to latch the value of the mode pins into the FPGA to set the configuration mode used. The read_init command allows the host to read the value of the INIT pin for programming operation. The syntax is shown below:

```
read_init
```

There are no arguments for this command. The command returns an ack to signify a valid command has been received followed by the pin value. 0x00 represent a low pin condition and 0x01 represents a high pin voltage. An ack is generated upon successful completion. . If the command is not successful then no response is generated.

3.3.6 3.2.6 read_done

There is a pin on the FPGA called DONE that is used as part of the configuration sequence of the FPGA. This pin is used to signify that the FPGA programming sequence has been completed. The read_done command allows the host to read the value of the DONE pin to determine if the FPGA programming has been successful. The syntax is shown below:

```
read_done
```

There are no arguments for this command. The command returns an ack to signify a valid command has been received followed by the pin value. 0x00 represent a low pin condition and 0x01 represents a high pin voltage. An ack is generated upon successful completion. . If the command is not successful then no response is generated.

3.3.7 3.2.7 sf_transfer

The sf_transfer command is used to control the interface to the Spansion serial flash. The PSoC firmware is set-up such that the data is passed from the USB to the associated peripheral. In this case the data is passed from the USB to the SPI port. Sf_transfer controls the CS to the flash as well. The initial condition for the CS pin is tri-stated. This allows of board programming of flash devices as well as the ability to program other off board devices using the AvProg utility. Sf_transfer is responsible for setting the drive strength of the pin to the associated low logic level when the serial flash is to be programmed. There is a second argument associated with this command which tells the PSoC how many data bytes it will be receiving. This value is used to set the loop for a successful program sequence. If this value is not met or a timeout occurs then the sequence is aborted and must be restarted. The syntax of the command is as follows:

```
sf_transfer <arg>
```

where <arg> is an ASCII representation of the number bytes to be sent during the data phase of the transfer.

An example of this command would be “sf_transfer 21” as sent by the host. This would tell the PSoC that 21 bytes would be sent from the host and required to be read from the serial flash and passed back.

This command will send an “ack” to signify a valid command has been received. At this point it enters the data phase of the command and is waiting for the data from the host. There is a timeout feature in order to recover if the host stops sending data for some reason. The timer function is started upon entry into the data phase of the sf_transfer command. As packets of data are received from the host, the PSoC will send the data to the SPI flash and read the returned data as well. Once the packet has been transmitted and received via the SPI it is then sent to the host, an “ack” is sent and the routine waits for another packet. If the last packet had been received then the sf_transfer routine raises the CS line, tri-states the CS line and exits.

The command returns an ack upon successful completion. If the command is not successful then no response is generated.

3.3.8 3.2.8 ss_program

The ss_program command is used to control the interface to the FPGA for slave serial configuration. This mode is entered when the mode is set to M2:M0 = 111. The PSoC firmware is set-up such that the data is passed from the USB to the associated peripheral. In this case the data is passed from the USB to the SPI port. Ss_program controls the CS line as well. This isn’t really important in this case as the FPGA does not use the CS line, however, it is controlled and maintained in the inactive state to avoid contention with any other SPI devices. The initial condition for the CS pin is tri-stated. This allows of board programming of flash devices as well as the ability to program other off board devices using the AvProg utility. Ss_program is responsible for setting the drive strength of the pin to the associated low high level when the FPGA is to be programmed. There is a second argument associated with this command which tells the PSoC how many data bytes it will be receiving. This value is used to set the loop for a successful program sequence. If this value is not met or a timeout occurs then the sequence is aborted and must be restarted. The syntax of the command is as follows:

```
ss_program <arg>
```

where <arg> is an ASCII representation of the number bytes to be sent during the data phase of the transfer.

An example of this command would be “ss_program 235123” as sent by the host. This would tell the PSoC that 235123 bytes would be sent from the host and required to be read from the serial flash and passed back.

This command will send an “ack” to signify a valid command has been received. At this point it enters the data phase of the command and is waiting for the data from the host. There is a timeout feature in order to recover if the host stops sending data for some reason. The timer function is started upon entry into the data phase of the sf_transfer command. As packets of data are received from the host, the PSoC will send an “ack” to the host and then sends the data to the FPGA. Once the packet has been transmitted via the SPI the routine waits for another packet. If the last packet had been received then the ss_program routine raises the CS line, tri-states the CS line and exits.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.9 3.2.9 jtag_mux

There are multiple sources that can drive the JTAG interface on the Spartan 3A Evaluation board. In this programming mode the FPAG can be configured by the Xilinx USB JTAG interface cable or the PSoC. Note that as of this writing this is a future feature of the PSoC. In order to control what device gains access to the JTAG port a control pin from the PSoC was added. The syntax is shown below:

```
jtag_mux <arg>
```

where valid values for arg are 0x30 to drive the pin low or 0x31 to drive the pin high.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.10 3.2.10 get_config

The get_config command was added as a test command. This command is useful if the user or GUI ever needs to see what configuration is loaded in the PSoC. The loaded configuration determines what peripherals are active. The syntax for the get_config command is shown below:

```
get_config
```

There are no arguments for this command. The return data represents the loaded configuration as follows:

0x30 - UART
0x31 - SPI
0x32 - JTAG
0x33 – None

An ack is generated upon successful completion. . If the command is not successful then no response is generated.

3.3.11 3.2.11 usb_bridge

This command should never be needed as all bridging occurs as a part of the complete data reception mechanism implemented via the firmware. The usb_bridge function disables all main loop application code with the exception of the USB to UART bridging and the CapSense for controlling the pads. There is no way to recover the command parsing using this command without power cycling the board. The syntax is shown below:

```
usb_bridge <arg>
```

where valid values for arg are 0x31 to disable all main application code.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.12 3.2.12 fpga_rst

The fpga_rst command is used to drive the reset pin input to the FPGA. There are two ways to control this pin, from the CapSense pad EF4 and from the GUI. This command sets a flag to be read by the main loop application and it is combined with the control from the CapSense APIs. When issuing this command from the GUI the FPGA can be held in reset as long as the GUI needs to maintain that condition. The syntax of the fpga_rst command is as follows:

```
Fpga_rst <arg>
```

where valid values for arg are 0x30 to drive the pin low or 0x31 to drive the pin high.

The command returns an ack upon successful completion. . If the command is not successful then no response is generated.

3.3.13 3.2.13 get_ver

The PSoC version number is set as a constant string in the firmware. The get_ver command reads the value of the string and passes it the host. The AvProg GUI compares the value read and compares it to the known compatible PSoC firmware versions and determines if the communications will be operational. The syntax is shown below:

```
get_ver
```

There are no arguments for this command. The command returns the string associated with the version number. An ack is generated upon successful completion. If the command is not successful then no response is generated.

4 4.0 Conclusion

Other commands may be defined in the future as expansion of the capabilities grows. The structure type and the command parsing was done in such a way that new commands can be easily added to the firmware to increase the PSoC firmware capability.

5 Revisions

Date	Version	Revision
20 Aug 2010	C	Initial Release

6 Getting Help and Support

Evaluation Kit home page with Documentation and Reference Designs

- **http://em.avnet.com/spartan6LX16-evl**

Avnet Spartan-6 LX16 Evaluation Kit forum

- **http://community.em.avnet.com/t5/Spartan-6-LX16-Evaluation-Kit/bd-p/Spartan-6_S6LX16_Evaluation_Kit**

For Xilinx technical support, you may contact Xilinx Online Technical Support at www.support.xilinx.com. On this site you will also find the following resources for assistance:

- Software, IP, and Documentation Updates
- Access to Technical Support Web Tools
- Searchable Answer Database with Over 4,000 Solutions
- User Forums
- Training - Select instructor-led classes and recorded e-learning options

Contact Avnet Support for any questions regarding the Spartan-6 LX16 Evaluation Kit reference designs or kit hardware

- **http://www.em.avnet.com/techsupport**

You can also contact your local Avnet/Silica FAE.

7 Appendix B – PSoC Port Configuration

This section provides the complete Port Configuration report for the default S6LX16 firmware (v3.0.2).

```
-------------------------------------------------------------
Port Configuration report
-------------------------------------------------------------
     |      |       | Interrupt |                   |                            |
Port | Pin  | Fixed |      Type |        Drive Mode |                       Name | Connections
-----+------+-------+-----------+-------------------+----------------------------+-----------------------------------------------------------------
   0 |    0 |     * |      NONE |       HI_Z_ANALOG |                  VS1_P(0) | Analog(Net_128)
     |    1 |     * |      NONE |       HI_Z_ANALOG |                  VS1_N(0) | Analog(Net_129)
     |    2 |     * |      NONE |       HI_Z_ANALOG |                  VS2_P(0) | Analog(Net_130)
     |    3 |     * |      NONE |       HI_Z_ANALOG |                  VS2_N(0) | Analog(Net_133)
     |    4 |     * |      NONE |       HI_Z_ANALOG |                  VS3_P(0) | Analog(Net_131)
     |    5 |     * |      NONE |       HI_Z_ANALOG |                  VS3_N(0) | Analog(Net_134)
     |    6 |     * |      NONE |       HI_Z_ANALOG |                VS3_DIV(0) | Analog(Net_132)
     |    7 |     * |      NONE |     OPEN_DRAIN_HI |            FPGA_SUSPEND(0) |
-----+------+-------+-----------+-------------------+----------------------------+-----------------------------------------------------------------
   1 |    2 |     * |      NONE |          CMOS_OUT |      FPGA_Push_Button_D(0) |
     |    4 |     * |      NONE |          CMOS_OUT |      FPGA_Push_Button_B(0) |
     |    5 |     * |      NONE |          CMOS_OUT |      FPGA_Push_Button_C(0) |
     |    6 |     * |      NONE |      HI_Z_DIGITAL |             BC_USB_STAT(0) |
     |    7 |     * |      NONE |       HI_Z_ANALOG | \CapSense:sbCSD:cCmod(0)\ | Analog(\CapSense:sbCSD:Net_1889\)
-----+------+-------+-----------+-------------------+----------------------------+-----------------------------------------------------------------
   2 |    0 |     * |      NONE |      HI_Z_DIGITAL |              BC_AC_STAT(0) |
     |    1 |     * |      NONE |          CMOS_OUT |      FPGA_Push_Button_A(0) |
     |    2 |     * |      NONE |      HI_Z_DIGITAL |               FPGA_DONE(0) |
     |    3 |     * |      NONE |          CMOS_OUT |                FPGA_CLK(0) | In(Net_52_local)
     |    4 |     * |      NONE |      HI_Z_DIGITAL |           FPGA_SPI_CCLK(0) | FB(Net_167)
     |    5 |     * |      NONE |      HI_Z_DIGITAL |           FPGA_SPI_MOSI(0) | FB(Net_166)
     |    6 |     * |      NONE |          CMOS_OUT |           FPGA_SPI_MISO(0) | In(Net_169)
     |    7 |     * |      NONE |      HI_Z_DIGITAL |            FPGA_SPI_SEL(0) | FB(Net_168)
-----+------+-------+-----------+-------------------+----------------------------+-----------------------------------------------------------------
   3 |    0 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(4)\ |
     |    1 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(5)\ |
     |    2 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(6)\ |
     |    3 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(7)\ |
     |    4 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(8)\ |
     |    5 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(9)\ |
     |    6 |     * |      NONE |       HI_Z_ANALOG | \Avnet_LCD_12x12:Com(10)\ |
     |    7 |     * |      NONE |       HI_Z_ANALOG | \Avnet_LCD_12x12:Com(11)\ |
-----+------+-------+-----------+-------------------+----------------------------+-----------------------------------------------------------------
   4 |    0 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(0)\ |
     |    1 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(1)\ |
     |    2 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(2)\ |
     |    3 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(3)\ |
     |    4 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(4)\ |
     |    5 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(5)\ |
     |    6 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(6)\ |
     |    7 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(7)\ |
-----+------+-------+-----------+-------------------+----------------------------+-----------------------------------------------------------------
   5 |    0 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(8)\ |
     |    1 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Seg(9)\ |
     |    2 |     * |      NONE |       HI_Z_ANALOG | \Avnet_LCD_12x12:Seg(10)\ |
     |   3 |     * |      NONE |       HI_Z_ANALOG | \Avnet_LCD_12x12:Seg(11)\ |
     |   4 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(0)\ |
     |   5 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(1)\ |
     |   6 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(2)\ |
     |   7 |     * |      NONE |       HI_Z_ANALOG |  \Avnet_LCD_12x12:Com(3)\ |
-----+-----+-------+-----------+------------------+---------------------------+-------------------------------------------------------------
   6 |   0 |     * |      NONE |     OPEN_DRAIN_LO | \CapSense:sbCSD:cPort(0)\ | In(\CapSense:Net_128\), Analog(\CapSense:sbCSD:Net_1892_0\)
     |   1 |     * |      NONE |     OPEN_DRAIN_LO | \CapSense:sbCSD:cPort(1)\ | In(\CapSense:Net_128\), Analog(\CapSense:sbCSD:Net_1892_1\)
     |   2 |     * |      NONE |     OPEN_DRAIN_LO | \CapSense:sbCSD:cPort(2)\ | In(\CapSense:Net_128\), Analog(\CapSense:sbCSD:Net_1892_2\)
     |   3 |     * |      NONE |     OPEN_DRAIN_LO | \CapSense:sbCSD:cPort(3)\ | In(\CapSense:Net_128\), Analog(\CapSense:sbCSD:Net_1892_3\)
     |   4 |     * |      NONE |      HI_Z_DIGITAL |               BC_STAT1(0) |
     |   5 |     * |      NONE |      HI_Z_DIGITAL |               BC_STAT2(0) |
     |   6 |     * |      NONE |       RES_PULL_UP |                UART_RX(0) | FB(Net_22)
     |   7 |     * |      NONE |          CMOS_OUT |                UART_TX(0) | In(Net_17)
-----+-----+-------+-----------+------------------+---------------------------+-------------------------------------------------------------
  12 |   0 |     * |      NONE |          CMOS_OUT |            Module_LEDs(0) |
     |   1 |     * |      NONE |          CMOS_OUT |            Module_LEDs(1) |
     |   2 |     * |      NONE |          CMOS_OUT |            Module_LEDs(2) |
     |   3 |     * |      NONE |          CMOS_OUT |            Module_LEDs(3) |
     |   4 |     * |      NONE |     OPEN_DRAIN_LO |             I2C_FG_SCL(0) | FB(\FUEL_GAUGE_I2C:Net_414\), In(\FUEL_GAUGE_I2C:Net_405_0\)
     |   5 |     * |      NONE |     OPEN_DRAIN_LO |             I2C_FG_SDA(0) | FB(\FUEL_GAUGE_I2C:Net_458\), In(\FUEL_GAUGE_I2C:Net_405_1\)
     |   6 |     * |      NONE |     OPEN_DRAIN_LO |             BRD_PWR_EN(0) |
     |   7 |     * |      NONE |      HI_Z_DIGITAL |            FG_BATT_LOW(0) |
-----+-----+-------+-----------+------------------+---------------------------+-------------------------------------------------------------
  15 |   0 |     * |      NONE |     OPEN_DRAIN_LO |              FPGA_PROG(0) |
     |   1 |     * |      NONE |      HI_Z_DIGITAL |              FPGA_INIT(0) |
     |   2 |     * |      NONE |          CMOS_OUT |             FPGA_MODE1(0) |
     |   3 |     * |      NONE |      HI_Z_DIGITAL |             FPGA_AWAKE(0) |
     |   4 |     * |      NONE |      HI_Z_DIGITAL |              FPGA_CCLK(0) | In(Net_91)
     |   5 |     * |      NONE |      HI_Z_DIGITAL |               FPGA_DIN(0) | In(Net_90)
     |   6 |     * |      NONE |       HI_Z_ANALOG |           \USB_CDC:dp(0)\ | Analog(\USB_CDC:Net_548\)
     |   7 |     * |      NONE |       HI_Z_ANALOG |           \USB_CDC:dm(0)\ | Analog(\USB_CDC:Net_522\)
--------------------------------------------------------------------------------------------------------------------------------------------
```