CY8C58LP/CY8C56LP/CY8C54LP/CY8C52LP

PSoC® 5LP Device Programming Specifications

Document #: 001-81290 Rev. *F

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com

Copyrights

Contents

1. Introduction

PSoC® 5LP device programming refers to the programming of nonvolatile memory in PSoC 5LP using an external host programmer. In the context of external host programmers, nonvolatile memory includes device configuration nonvolatile latch (NVL) flash memory, EEPROM, and write once NVL. PSoC 5LP supports programming through the Serial Wire Debug (SWD) interface or Joint Test Action Group (JTAG) interface. The data to be programmed is stored in a hex file. This programming specifications document explains the hardware connections, programming protocol, programming vectors, and the timing information for developing programming solutions for a PSoC 5LP device.

1.1 Host Programmer

The host programmer can be the MiniProg3 Programmer supplied by Cypress, a "third-party programmer", or a hardware device such as a microcontroller or an FPGA. The MiniProg3 programmer is used in the prototype stage of application development for both programming and debugging PSoC 5LP devices on board. Third-party programmers are used for production programming of PSoC 5LP in large numbers. They are used when the design is finalized and the application needs to go in for mass production. Apart from this, custom-developed host programmers such as FPGA or an external microcontroller can be used to perform in-system programming of the PSoC 5LP device either for complete programming or partial firmware upgrade.

The host programmer programs the PSoC 5LP device with the program image contained in the *<Project_Name>.hex* file, which is generated by the PSoC Creator™ software. See the General PSoC Programming web page for complete information on PSoC programming-related documents, software, and a list of supported third-party programmers.

1.2 Hardware Connections

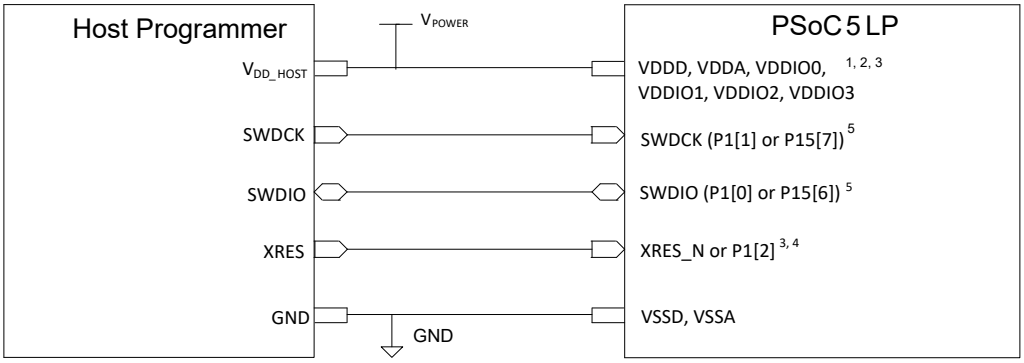
This section discusses hardware connections between the host programmer and the PSoC 5LP device for programming through the SWD and JTAG interfaces. Only programming related connections are discussed. For a complete schematic of the PSoC 5LP device, including the PSoC 5LP regulator output pins (VCCD and VCCA), see ""Example Schematic" on page 84". The PSoC 5LP device datasheet has information on device operating conditions, specifications, and pinouts for the different PSoC 5LP packages.

1.2.1 SWD Interface

Figure 1-1 on page 6 shows the hardware connections between the host programmer and the target PSoC 5LP device to program through the SWD interface.

PSoC 5LP has two pairs of pins that support SWD: P1[0] SWDIO and P1[1] SWDCK, or P15[6] USB D+ (SWDIO) and P15[7] USB D– (SWDCK) pins. No device configuration setting is required to choose between these two pairs. The internal device logic chooses between these pins automatically by detecting activity (clock transition on SWDCK lines) after the device comes out of reset. To reset the PSoC 5LP device for programming, either the XRES pin or power cycle mode must be used. Power cycle mode programming involves toggling power to the VDDD, VDDA, and VDDIO pins of PSoC 5LP to reset the device. All SWD interface programmers support programming using the XRES pin, but only some of them support power cycle mode. If power cycle mode programming is needed, make sure it is supported by the programmer being used.

Figure 1-1. SWD Programming Interface Connections between Host Programmer and PSoC 5LP



Notes for Figure 1-1:

1. The voltage level of the host programmer and the supply voltage for PSoC 5LP I/O pins used in programming should be the same. Port 1 SWD pins and XRES (XRES_N or P1[2] as XRES) pin in PSoC 5LP are powered by the VDDIO1 pin. USB SWD pins are powered by the VDDD pin.
   a. To program using the Port 1 SWD pins (P1[0], P1[1]) and XRES pin (XRES_N or P1[2] as XRES), the host voltage level (V_{DD_HOST}) should be the same as VDDIO1 pin of PSoC 5LP. The remaining PSoC 5LP power supply pins (VDDD, VDDA, VDDIO0, VDDIO2, and VDDIO3) need not be at the same voltage level as the host programmer.
   b. To program using the USB SWD pins (P15[6], P15[7]) and XRES pin, the host voltage level (V_{DD_HOST}) should be the same as the VDDD and VDDIO1 pins of PSoC 5LP. The remaining PSoC 5LP power supply pins (VDDA, VDDIO0, VDDIO2, VDDIO3) need not be at the same voltage level as the host programmer.
2. VDDA must be greater than or equal to all other power supplies (VDDD and VDDIOs) in PSoC 5LP.
3. For power cycle mode programming, the XRES pin is not required. The VDDD, VDDA, VDDIO0, VDDIO1, VDDIO2, and VDDIO3 pins of PSoC 5LP should be tied together to the same power supply; power to these pins should be toggled to reset the device. Ensure that the programmer used supports power cycle mode. MiniProg3 (rev 7 and later versions) supports power cycle mode.
4. The XRES pin can either be the dedicated XRES pin (XRES_N) or the optional XRES pin (P1[2]). P1[2] is configured as XRES pin by default only for 48-pin devices (which do not have a dedicated XRES pin). For devices with a dedicated XRES pin (XRES_N), P1[2] is a GPIO pin by default. Use P1[2] as reset pin only for 48-pin devices, but use the dedicated XRES pin for other devices.
5. USB SWD pins (P15[6], P15[7]) are not present in devices without USB functionality.

Table 1-1 lists the host programmer hardware requirements for PSoC 5LP pins involved in SWD interface programming.

Table 1-1. Host Programmer Requirements for PSoC 5LP Programming

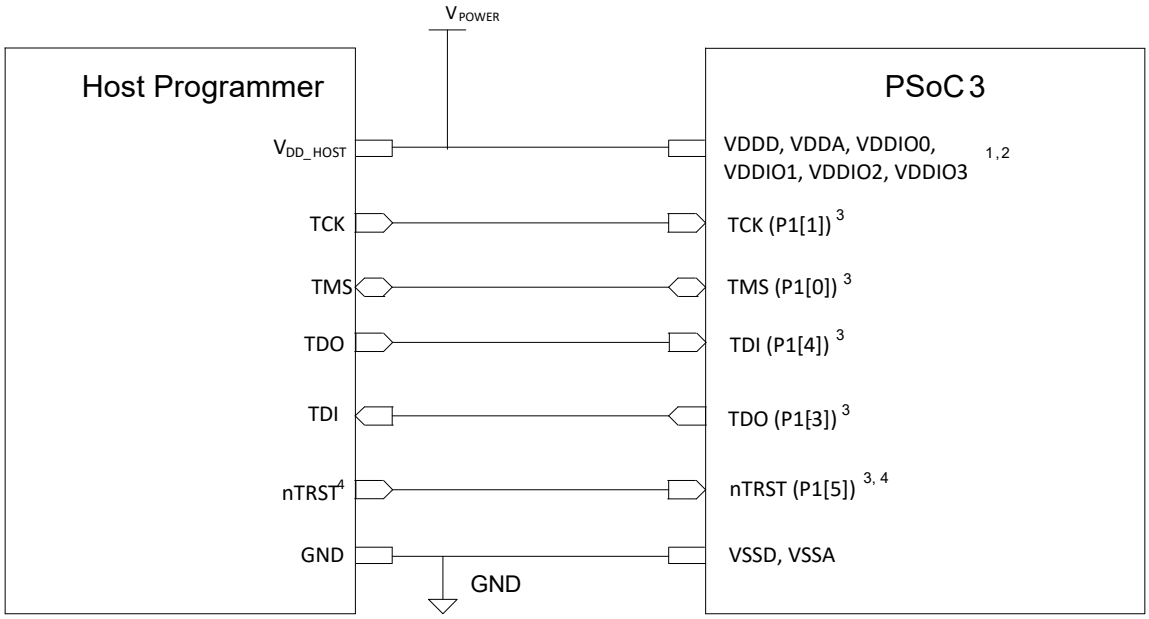
Pin	Host Programmer Requirement	PSoC 5LP Function	Comment
SWDCK (SWD Clock)	Strong drive (CMOS drive) digital output	P1[1] SWDCK pin - Digital input with internal 5.6 kΩ pull-down resistance P15[7] SWDCK pin - High-impedance digital input	The internal 5.6 kΩ pull-down resistor on the P1[1] SWDCK pin (not on P15[7]) is for internal device Port Acquire logic. No external resistor is needed on the SWDCK line. SWDCK should always be in Strong drive (CMOS drive) mode on the host programmer side.
SWDIO (SWD Data)	Write operation: Strong drive (CMOS drive) digital output Read operation: High-impedance digital input	Write operation: Strong drive (CMOS drive) digital output Read operation: High-impedance digital input	PSoC 5LP changes between two drive modes for read and write operations on the SWDIO line using the Turnaround (TrN) phase of SWD protocol. The host must also change the drive mode of the SWDIO line during this TrN phase. When the host writes to SWDIO, PSoC 5LP reads from SWDIO and vice-versa.
XRES	Strong drive (CMOS drive) digital output	Digital input with internal 5.6 kΩ resistive pull-up to VDDIO1	The XRES pin or P1[2] as XRES in PSoC 5LP is active low input and there is an internal 5.6 kΩ pull-up resistor to VDDIO1.
VDDA, VDDD, VDDIO	Positive voltage	Digital, analog, and I/O power supply	For power cycle mode, tie the VDDD, VDDA, and VDDIO pins of PSoC 5LP to the same power supply. Toggle power to these pins to reset the device. See the PSoC 5LP device datasheet for specifications on power pins (VDDD, VDDA, VDDIOs) and ground pins (VSSD, VSSA).
VSSD, VSSA	Low-resistance ground connection	Ground for all analog peripherals (VSSA), digital logic, and I/O pins (VSSD)	Toggle power to these pins to reset the device. See the PSoC 5LP device datasheet for specifications on power pins (VDDD, VDDA, VDDIOs) and ground pins (VSSD, VSSA).

1.2.2 JTAG Interface

Figure 1-2 shows the hardware connections between the host programmer and PSoC 5LP device to program through the JTAG interface.

There are fixed port pins to program PSoC 5LP through the JTAG interface: P1[0] (TMS), P1[1] (TCK), P1[3] (TDO), P1[4] (TDI), P1[5] (nTRST). The nTRST pin is an optional connection for JTAG interface. It is not functional during PSoC 5LP device programming, but can be enabled for debugging operations by programming the device configuration NVL with a five-wire JTAG setting (default factory setting is four-wire JTAG).

Figure 1-2. JTAG Programming Interface Connections between Host Programmer and PSoC 5LP



Notes for Figure 1-2:

1. The voltage level of the host programmer and the supply voltage for PSoC 5LP I/O pins involved in programming should be the same. PSoC 5LP JTAG pins are powered by VDDIO1. The host voltage level (V_{DD_HOST}) should be the same as VDDIO1 pin. The remaining PSoC 5LP power supply pins (VDDD, VDDA, VDDIO0, VDDIO2, and VDDIO3) need not be at the same voltage level as host programmer.
2. VDDA must be greater than or equal to all other power supplies (VDDD and VDDIOs) in PSoC 5LP.
3. PSoC 5LP programming using third-party JTAG programmers is not possible if the Debug Port Select (DPS) setting in NVL is configured for ‘Debug Port Disabled’ or ‘SWD’. The necessary DPS settings for JTAG interface programming are ‘4-wire JTAG’ or ‘5-wire JTAG’; the default factory setting is ‘4-wire JTAG’. If the DPS setting is changed to a different state, the JTAG port can reprogram the DPS setting for JTAG using the MiniProg3's SWD interface and XRES pin (or power cycle mode) as given in Figure 1-1 on page 6. The PSoC 5LP is compliant to IEEE 1149.1 standard if “4-/5- pins wire JTAG” DPS is set in NVL. Otherwise, it is required to use combined SWD and JTAG interfaces for programming.
4. The nTRST pin is an optional connection for the JTAG interface. It is not functional during PSoC 5LP device programming, but it can be enabled for debugging operations by programming the device configuration NVL with 5-wire JTAG setting.

Table 1-2 lists the host programmer hardware requirements for PSoC 5LP pins involved in JTAG interface programming.

Table 1-2. Host Programmer Requirements for PSoC 5LP JTAG Interface Programming

Pin	Host Programmer Requirement	PSoC 5LP Functionality	Comment
JTAG Clock (TCK)	Strong drive (CMOS drive) digital output	Digital input with internal 5.6 kΩ pull-down resistance	Pull-down resistor on TCK ensures that no spurious clock signals are present when the TCK input is not driven by host.
JTAG TDI (TDI)	High-impedance digital Input	Digital input with internal 5.6 kΩ pull-up resistance to VDDIO1	TDI of the host is connected to TDO of PSoC 5LP and vice-versa. TDI input in PSoC 5LP has a pull-up resistor so that the pin is in known state (logic high) when not driven by host.
JTAG TDO (TDO)	Strong drive (CMOS drive) digital output	Strong drive (CMOS drive) digital output	TDI of the host is connected to TDO of PSoC 5LP and vice-versa.
JTAG TMS (TMS)	Strong drive (CMOS drive) digital output	Digital input with internal 5.6 kΩ pull-up resistance to VDDIO1	TMS input in PSoC 5LP has a pull-up resistor to ensure that the pin is in known state (logic high) when not driven by the host.
JTAG Reset (nTRST) (Optional)	Strong drive (CMOS drive) digital output	Digital input with internal 5.6 kΩ pull-up resistance to VDDIO1	nTRST pin is an optional connection for the JTAG interface. It is not functional during programming of the PSoC 5LP device. Use the TMS and TCK pins to reset the JTAG TAP controller.
VDDA, VDDD, VDDIOs	Positive voltage	Digital, analog, I/O power supply	See the PSoC 5LP device datasheet for specifications on power pins (VDDD, VDDA, VDDIO0, VDDIO1, VDDIO2, and VDDIO3) and ground pins (VSSD and VSSA).
VSSD, VSSA	Low-resistance ground connection	Ground for all analog peripherals (VSSA), all digital logic, and I/O pins (VSSD)	

Document Revision History

Document Title: CY8C58LP/CY8C56LP/CY8C54LP/CY8C52LP, PSoC® 5LP Device Programming Specifications

Document Number: 001-81290

Revision	Issue Date	Origin of Change	Description of Change
**	07/17/2012	DISM / ANDI / VVSK	Initial revision
*A	07/22/2012	VVSK	Updated SWD and JTAG Vectors for Programming chapter on page 47: Updated code in all instances in the chapter.
*B	11/28/2012	DISM	Updated PSoC 5LP Programming Flow chapter on page 25: Updated “Step1: Enter Programming Mode” on page 26: Updated “SWD Universal Acquisition” on page 26: Updated Figure 3-3. Updated “SWD Programming using Bit Banging Host Programmers:” on page 29: Updated Figure 3-6. Added “Step 12: Program EEPROM (Optional)” on page 41. Added “Step 13: Verify EEPROM (Optional)” on page 42. Updated Programming Specifications chapter on page 43: Updated “Programming Mode Entry Specifications” on page 45: Updated Table 4-3. Updated SWD and JTAG Vectors for Programming chapter on page 47: Added “Step 12: Program EEPROM (Optional)” on page 72. Added “Step 13: Verify EEPROM (Optional)” on page 75. Updated Appendix chapter on page 77: Updated “Intel Hex File Format” on page 77: Updated A.1.1 Organization of Hex File Data: Updated description. Updated Figure A-2.
*C	12/12/2014	ANDI	Updated SWD and JTAG Vectors for Programming chapter on page 47: Updated “Step 2: Configure Target Device” on page 49: Updated code.
*D	06/29/2015	ANDI	No technical updates. Completing Sunset Review.
*E	04/25/2017	AESATMP8	Updated logo and Copyright.
*F	07/23/2018	STPP	No technical updates. Completing Sunset Review.

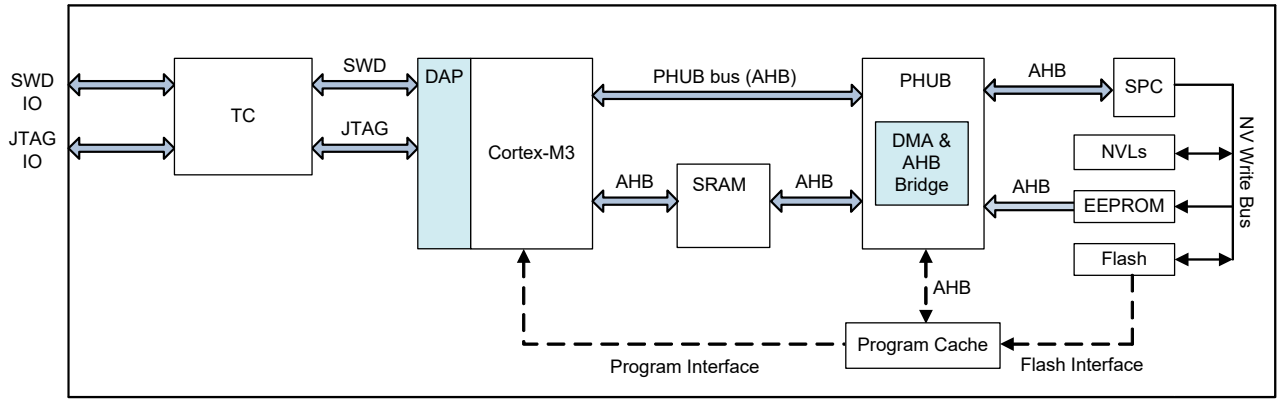
2. PSoC 5LP Programming Interface

This section explains the programming interface in PSoC 5LP and the registers used for programming PSoC 5LP. An overview of the SWD and JTAG interface is also provided. See section "Nonvolatile Memory Organization in PSoC 5LP" on page 80" for details. The section also provides some advanced information about the silicon's communication interface, which should help to better understand the programming algorithm in later sections.

2.1 Programming Interface Architecture

This section outlines the silicon's architecture related to nonvolatile subsystem. It simplifies understanding of the programming algorithm described later. Figure 2-1 shows the necessary blocks involved in programming.

Figure 2-1. Programming Interface Architecture



The abbreviations used on Figure 2-1:

TC - Test Controller.

DAP - Debug Access Port of Cortex-M3 CPU (ARM).

AHB - Advanced High-Performance Bus, def acto standard from ARM.

PHUB - Peripheral Hub, advanced multi spoke bus controller, which allows many different functional blocks to communicate without involving of CPU for setting up the bus transaction.

DMA - Direct Memory Access controller.

SRAM - Static Random Access Memory.

SPC - System Performance Controller implements R/W interface with nonvolatile memory.

NVL - Nonvolatile Latch.

There are three types of nonvolatile memory in the silicon, which can be programed by users:

- Flash - contains user's code and data 288 KB (256 KB code + 32 KB user data).
- EEPROM - up to 2K of user data.
- NVLs - nonvolatile latches, which are split into two groups containing 4 bytes each: Custom NVLs and Write Once NVLs.

For this specification document, only programming of Flash and NVLs is considered. More information about nonvolatile subsystem is available in the “Appendix” on page 77.

The only block which physically executes read/write operations with nonvolatile memory is SPC. It is connected to NVL, EEPROM, and Flash via a dedicated NVL Write Bus. The external programmer configures SPC via SWD/JTAG and then calls its APIs to access NV memory. Note that CPU is not involved in Flash/NVL/EEPROM programming. This operation is completed locally by SPC block through the NVL Write Bus. Moreover, the programming algorithm has the advantage of a DMA controller to increase performance of programming.

The external host puts data into SRAM, then configures DMA to transfer parameters from SRAM to SPC. It then triggers the DMA transfer. While this transfer is in progress (which programs even flash row), the host puts programming parameters in SRAM for the odd flash row. When programming of even row is completed, DMA transfer for odd row is triggered. So, programming of flash and transferring on SWD/JTAG bus are run in parallel.

The access to PSoC 5LP’s resources by the external programmer is controlled via the Test Controller block. It is the gateway to the Debug Access Port of the CPU (Cortex-M3). DAP manages all requests to the silicon’s resources without use of the CPU’s time. PSoC 5LP incorporates standard ARM’s Cortex-M3 CPU along with its debug subsystem (DAP).

The Test Controller is Cypress’s proprietary block, which grants access to DAP. It is indispensable for the programmer to know how to communicate with DAP via the Test Controller.

2.2 Test Controller Block

The Test Controller (TC) interfaces any external devices used to program, configure, or debug the chip. Its purpose is to implement communication logic between the Cortex-M3 DAP and the Programmer, considering some external and internal conditions.

TC’s functionality depends on the following chip settings:

- “Debug Port Settings” (DPS) in Custom NVLs (see “Device Configuration NVLs” on page 82). This initializes the Debug port to SWD, JTAG, or GPIO upon reset or power on. For JTAG compliant programming, the DPS can only be 4-wire JTAG or 5-wire JTAG.
- “Debug_Enable” setting in Custom NVL. If it is ON, then DAP is connected to the debug pins upon reset (or power on). This means that the external programmer can have access to debug the subsystem any time, if DPS = SWD/JTAG. This access is described in detail in “Step 1: Enter Programming Mode” on page 47. If the “Debug_Enable” option is OFF, then the programmer must write a special acquire key in TC to enable access to DAP. For JTAG compliant programming, the “Debug_Enable” should be ON.
- “Write Once NVL” content. It disables access to DAP permanently, if the correct key is written during the last programming cycle. This is a special security mechanism, which disables SWD/JTAG interface in the silicon permanently. It is an irreversible change, which leaves the silicon without any way of failure analysis or reprogramming. This programming step is described later.

TC also selects an active SWD pair (on USB pins P15[6]/P15[7] or Port 1 pins P1[0]/P1[0]) depending on the activity on these pins upon reset or power on. The pair on which the correct acquire key is detected during boot window becomes active. This behavior is independent of current DPS setting (SWD, JTAG, or GPIO). If the current DPS setting is SWD, and no correct key provided during boot time, then the SWD pins default to Port 1. Therefore, there is a very short (400 microsecond) window when USB pins can be acquired for SWD programming.

Figure 2-2 on page 13 shows internal details of the TC block and its bridging interface between the debug pins and CPU (DAP).

Figure 2-2. Programming Interface Architecture

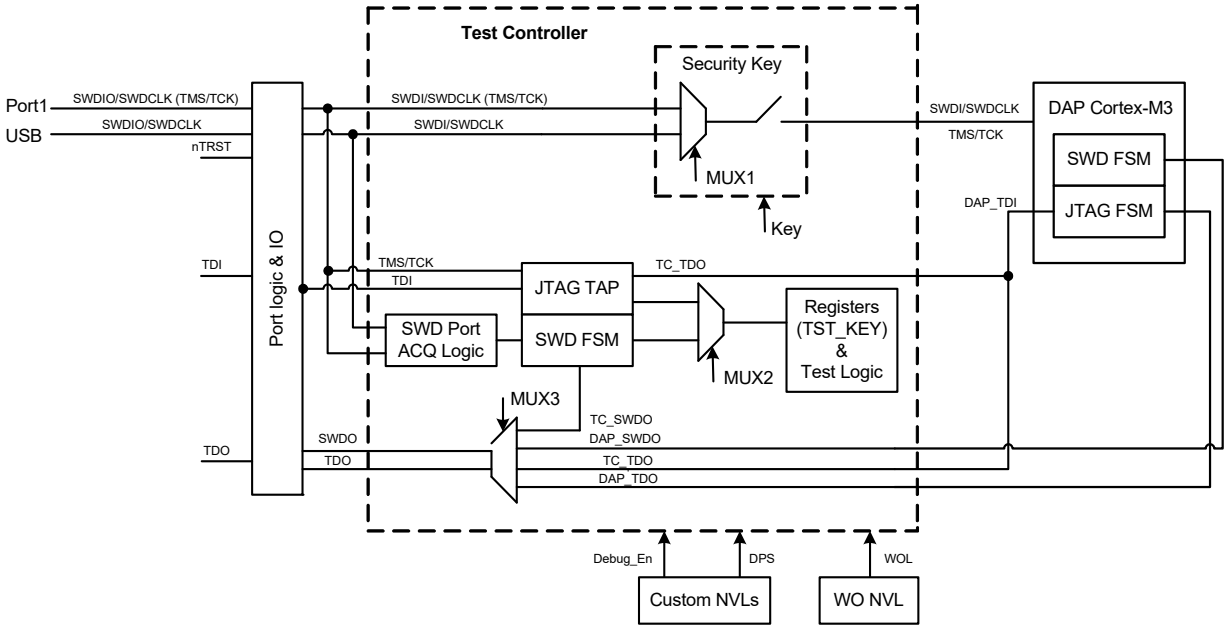


Figure 2-2 shows details of the communication logic of the TC; it clarifies how the TC controls access to debug the subsystem of CPU (DAP). The TC has its own JTAG TAP and SWD FSM, through which the external programmer configures connection logic. The external programmer must have access to the DAP for successful programming. To do this:

- SWD or JTAG debug port (pins) must be enabled either by DPS setting or during the acquire window. The acquire window is necessary if DPS is set to GPIO.
- “Security Key” must be closed, so SWDI/SWDCLK (TMS/TCK) signals are routed to the DAP.
- WOL must not be locked.

Three muxes and one key on Figure 2-2 configure different working modes of the TC. The TC is configured automatically if instructions from “Step 1: Enter Programming Mode” on page 47 are executed. This is a transparent process for the programmer.

During programming only two configurations of the TC are really used: for SWD and for JTAG access. Figure 2-3 and Figure 2-4 on page 14 show the schematic of these TC configurations.

Figure 2-3. SWD Configuration During Programming

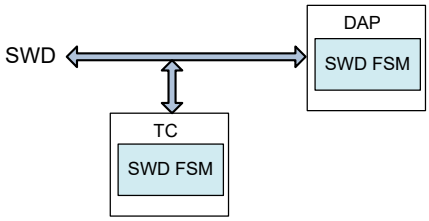
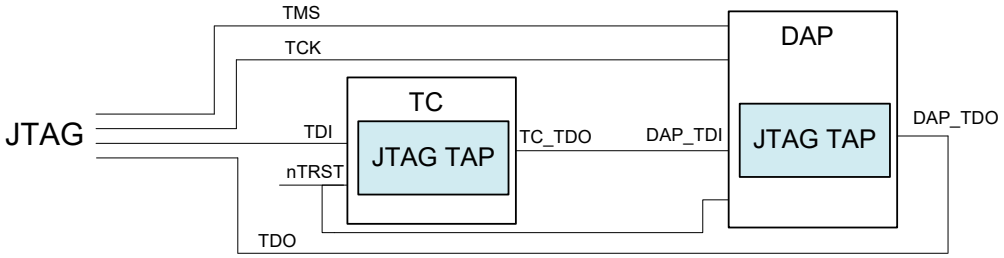


Figure 2-4. JTAG Configuration During Programming



First, the programmer configures the TC to get access to the DAP. For SWD access (Figure 2-3 on page 13), the DAP's and TC's FSMs are connected in parallel. All transactions are executed by the DAP, but the TC always monitors the SWD signals. TC only accepts transactions addressed to its registers space. The parallel connection of SWD FSM is possible because TC's and DAP's address spaces do not intersect. Programmer writes to the TC's register (for example, TST_KEY) to enable access to the DAP; it writes to the DAP's registers (PSoC 5LP resources) for programming.

For JTAG access (Figure 2-4) two TAPs are connected in series. The DAP's TAP can be disabled after power on/reset (by Debug_En bit in Custom NVLs). In this case, the programmer must enable it by writing the correct key in the TST_KEY register via the TC's TAP. During programming, the TC's TAP must be set to BYPASS mode.

More details about JTAG and SWD programming protocols are explained in next four chapters.

For an advanced understanding of the configuration schemes on Figure 2-2 on page 13, the details of forming MUX1, KEY, MUX2 and MUX3 signals are provided:

1. MUX1 =
   - *Port1 (SWD/JTAG)*, if (DPS == SWD) || (DPS == JTAG) || (SWD activities detected during boot on P1[0]/[1]);
   - *USB (SWD)*, if (SWD activity detected on P15[6]/[7] during boot window);

2. KEY = (DEBUG_EN) || (TST_KEY written during boot) && (WOL not Locked);

3. MUX2 =
   - *SWD*, if (DPS == SWD) || (JTAG_2_SWD sequence detected) || (boot time);
   - *JTAG*, if (DPS == JTAG) || (SWD_2_JTAG sequence detected);

4. MUX3 =
   - *tc_swdo*, (boot window) || ([address in TC's space] && [SWD mode]);
   - *tc_tdo*, (KEY is open) && (JTAG mode);
   - *dap_swdo*, (KEY is closed) && ([address in DAP's space] && [SWD mode]);
   - *dap_tdo*, (KEY is closed) && (JTAG mode);

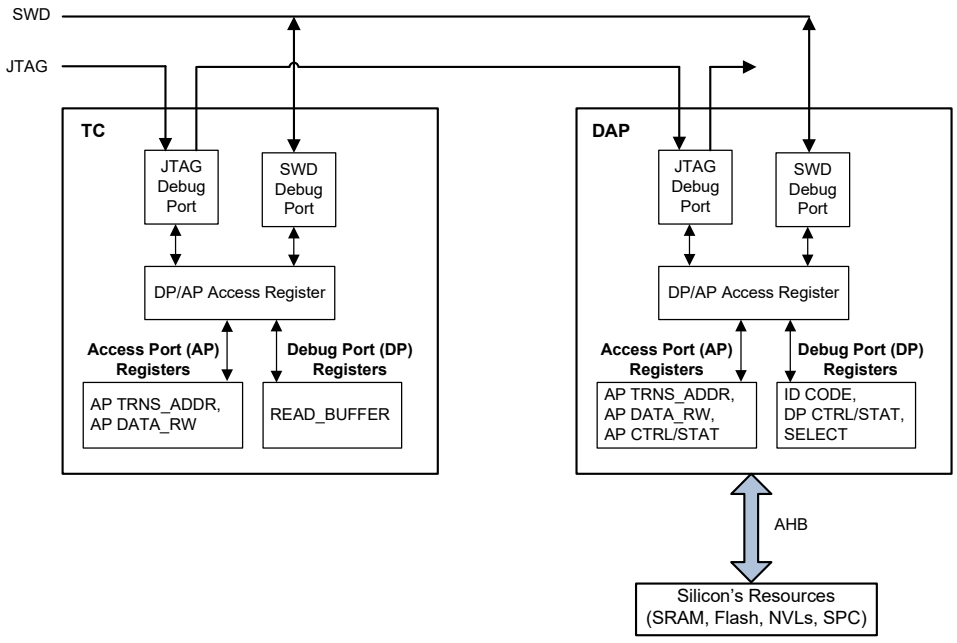
Note that if DPS = PI, then "Port Logic and IO" block on Figure 2-2 on page 13 configures the debug pins to GPIO after the boot window is completed (400 microseconds elapsed after reset or power-up). So, the external programmer cannot access TC or DAP via SWD/JTAG after that time window. In this case, the specific timing requirements must be met by the Programmer to get SWD access to the TC (see section "Step 1: Enter Programming Mode" on page 47). The TC's design allows for the silicon to be acquired following a reset regardless of the DPS settings via SWD interface. In such a scenario, the external programmer can also switch from SWD to JTAG interface by sending a SWD to JTAG switching sequence.

2.3 Programming Interface Registers

2.3.1 Debug Port/Access Port (DP/AP) Access Register

The PSoC 5LP TC has a DP/AP access register that is 35 bits wide. This register, which is part of both the TC and DAP interfaces, transfers data between JTAG/SWD bus and the Debug Port/Access Port registers. You can treat this register as the buffer through which all IN/OUT traffic is moving. The SWD interface enables direct reads and writes of the DP/AP Access register. The JTAG interface uses the APACC and DPACC instructions. The Access Port (AP) registers are used to read data from the specific or write data to the specific address. The Debug Port (DP) registers contain the Debug Port configuration such as byte size of AP register memory access and device JTAG ID. Figure 2-5 depicts access architecture:

Figure 2-5. PSoC 5LP Programming Interface

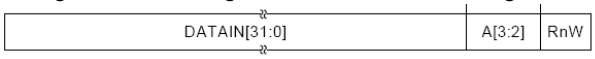


During programming all commands are directed to the DAP, only during chip acquiring stage is the TC's interface used. When the chip is acquired, the traffic is commutated to the DAP. All this happens automatically in the silicon during programming process.

2.3.1.1 Writing to the DP/AP Access Register

Figure 2-6 shows the structure when writing to the DP/AP Access register from the SWD or JTAG interfaces. For the JTAG, this register is written during Update_DR state of FSM.

Figure 2-6. Writing to the DP/AP Access Register

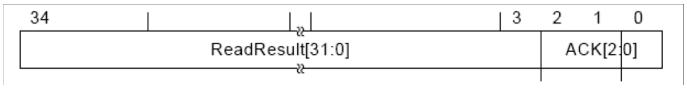


- Bits 34 to 3: (32 bits of data). If the register is less than 32-bits wide, zero-padding must be done for the remaining bits that are sent to PSoC 5LP
- Bits 2 to 1: 2-bit address for selecting DP or AP registers. These address bits are listed in Table 2-2 on page 16
- Bit 0: RnW – 1 = read (from PSoC 5LP to host programmer); 0 = write (to device from debug host)

2.3.1.2 *Reading from the DP/AP Access Register*

Figure 2-7 shows the structure of the 35-bit data register when reading the DP/AP Access register from the SWD or JTAG interfaces. For JTAG, this is data shifted out to TDO line during Update_DR state of FSM.

Figure 2-7. Reading from the DP/AP Access Register



- Bits 34 to 3: (32 bits of data): If the register is less than 32-bit wide (N-bit), it is still required to read the entire 32 bits to complete the transaction. Only the least N-bit data should be considered of the 32-bits read from device.
- Bits 2 to 0: (ACK response code): Depending on the interface, the ACK response is as indicated in Table 2-1. This ACK response is for the previous SWD transfer; if there is an error, it indicates that the previous transfer must be done again.

Table 2-1. ACK Response for SWD Transfers

ACK[2:0]	SWD
OK	001
WAIT	010
FAULT	100

2.3.2 Debug Port (DP)/Access Port (AP) Registers

The DP and AP registers listed in Table 2-2 are part of the ARM Cortex-M3 Debug Access Port (DAP). All the DP/AP registers are 32-bit registers. In the PSoC 5LP Cortex-M3, the DAP consists of the SWD Debug Port (SW-DP) and the AHB Access Port (AHB-AP). Note that Table 2-2 does not list all the DP/AP registers; it lists only those DP/AP registers that are required to program PSoC 5LP. For more information on these ports and their registers, see the ARM Debug Interface Architecture Specification (for SW-DP) and ARM Cortex-M3 Technical Reference Manual (for AHB-AP), available at http://www.arm.com.

Note that the TC also implements several AP/DP registers, which are used during the first step of programming (see Figure 2-5 on page 15). This is Cypress's implementation of the DP/AP access port; its main goal is to connect the external programmer to the DAP.

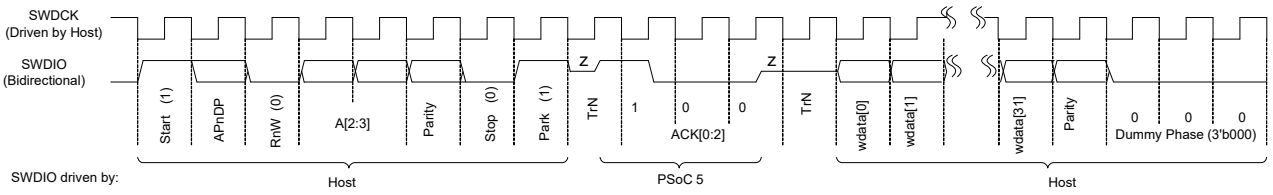
Table 2-2. Debug Port and Access Port Registers (PSoC 5LP)

Register Name	Register Type	Address (A[3:2])	Function
IDCODE	DP	00	32-bit Device IDCODE register.
DP CTRL/STAT	DP	01	Debug port control/status register. CTRLSEL bit in the SELECT register should be '0' to access this register.
SELECT	DP	10	Access port select – The MS byte of the SELECT register selects which Access Port (AP) is used on AP accesses. Bits [7:4] select which register in the AHB-AP is accessed.
READBUFF	DP	11	Port Acquire key is written to this 32-bit register to acquire port through SWD interface. Used in TC only.
AP Control Status (AP CTRL/STAT)	AP	00 (SELECT[7:4] = 0)	AHB-AP control/status register.
AP Transfer Address	AP	01 (SELECT[7:4] = 0)	AHB-AP transfer address register. This register holds the 32-bit address that is used for device register access. Also used in TC.
AP Data Read/Write	AP	11 (SELECT[7:4] = 0)	AHB-AP data read/write register. This 32-bit register holds the data to be read from/written to the address specified by the AP Transfer Address register. Also used in TC.

2.4 SWD Interface

PSoC 5LP supports programming through the serial wire debug (SWD) interface. There are two signals in the SWD interface: data signal (SWDIO) and a clock for data signal (SWDCK). The host programmer always drives the clock line, whereas either the programmer or the PSoC 5LP device drives the data line. The timing diagram for the SWD protocol is given in "Programming Specifications chapter on page 43. The host programmer and PSoC 5LP device communicate in packet format through the SWD interface. 'Write packet' refers to the SWD packet transaction in which the host writes data to PSoC 5LP. 'Read packet' refers to the SWD packet transaction in which the host reads data from PSoC 5LP. The format of the write packet and read packet are illustrated in Figure 2-6 and Figure 2-7, respectively.

Figure 2-8. SWD "Write Packet" Timing Diagram



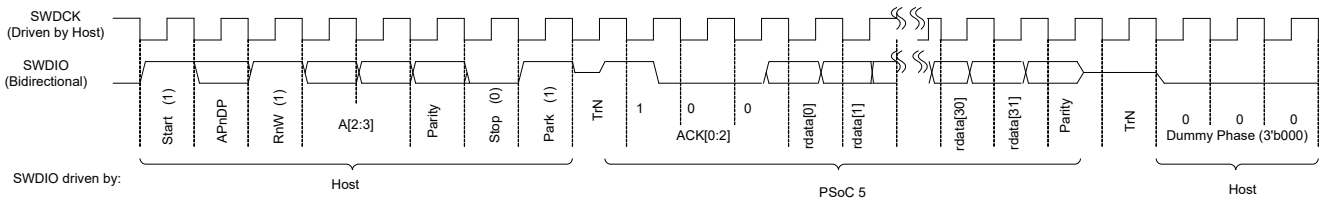
a.) Host Write Operation: Host sends data on the SWDIO line on falling edge of SWDCK and PSoC 5 reads that data on the next SWDCK rising edge (for example, 8-bit header data, Write data(wdata[31:0]), Dummy phase (3'b000))

b.) Host Read Operation: PSoC 5 sends data on the SWDIO line on the rising edge of SWDCK and the host should read that data on the next SWDCK falling edge (Ex: ACK data (ACK[2:0])

c.) The host should not drive the SWDIO line during TrN phase. During first TrN phase (½ cycle duration) of SWD packet, PSoC 5 starts driving the ACK data on the SWDIO line on the rising edge of SWDCK. The host should read the data on the subsequent falling edge of SWDCK. The second TrN phase is 1.5 SWDCK clock cycles. Both PSoC 5 and the Host will not drive the line during the entire second TrN phase (indicated as 'z'). Host should start sending the Write data (wdata) on the next falling edge of SWDCK after second TrN phase.

d.) "DUMMY" phase is three SWD clock cycles with SWDIO line low. This DUMMY phase is not part of SWD protocol. The three extra clocks with SWDIO low are required for the Test Controller in PSoC 5 to complete the Read/Write operation when the SWDCK clock is not free-running. For a reliable implementation, include three IDLE clock cycles with SWDIO low for each packet. According to the SWD protocol, the host can generate any number of SWD clock cycles between two packets with SWDIO low.

Figure 2-9. SWD "Read Packet" Timing Diagram



a.) Host Write Operation: Host sends data on the SWDIO line on falling edge of SWDCK and PSoC 5 reads that data on the next SWDCK rising edge (for example, 8-bit header data, dummy phase (3'b000))

b.) Host Read Operation: PSoC 5 sends data on the SWDIO line on rising edge of SWDCK and the host should read that data on the next SWDCK falling edge (for example, ACK data (ACK[2:0], Read data (rdata[31:0]))

c.) The host should not drive the SWDIO line during TrN phase. During first TrN phase (½ cycle duration) of SWD packet, PSoC 5 starts driving the ACK data on the SWDIO line on the rising edge of SWDCK. The host should read the data on the subsequent falling edge of SWDCK. The second TrN phase is 1.5 SWDCK clock cycles. Both PSoC 5 and the host will not drive the line during the entire second TrN phase (indicated as 'z'). Host should start sending the Dummy phase (3'b000) on the next falling edge of SWDCK after second TrN phase.

d.) "DUMMY" phase is three SWD clock cycles with SWDIO line low. This phase is not part of the SWD protocol. The three extra clocks with SWDIO low are required for the Test Controller in PSoC 5 to complete the Read/Write operation when the SWDCK clock is not free-running. For a reliable implementation, include three IDLE clock cycles with SWDIO low for each packet. According to the SWD protocol, the host can generate any number of SWD clock cycles between two packets with SWDIO low.

A complete data transfer requires 46 clocks (not including the optional three dummy clock cycles in Figure 2-8 and Figure 2-9). Each data transfer consists of three phases:

- **Packet request** – External host programmer issues a request to the PSoC 5LP device.
- **Acknowledge response** – PSoC 5LP sends an acknowledgement to the host.
- **Data** – Data is valid only when a packet request is followed by a valid (OK) acknowledge response.

The data transfer is either:

- PSoC 5LP to host, following a read request – RDATA
- Host to PSoC 5LP, following a write request – WDATA

In Figure 2-8 and Figure 2-9, the following sequence occurs:

1. The start bit initiates a transfer; it is always logic ‘1’.
2. The APnDP bit determines whether the transfer is an AP access, ‘1’, or a DP access, ‘0’.
3. The next bit is RnW, which is ‘1’ for a read from the PSoC 5LP device or ‘0’ for a write to the PSoC 5LP device.
4. The ADDR bits (A[3:2]) are register select bits for the access port or debug port. See Table 2-2 on page 16 for address bit definitions.
5. The parity bit has the parity of APnDP, RnW, and ADDR. This is an even parity bit. If the number of logical 1s in these bits is odd, then parity must be ‘1’, otherwise it is ‘0’.

   If the parity bit is not correct, the header is ignored by the target device; there is no ACK response. For the host implementation, the programming operation should be stopped and tried again by doing a device reset.
6. The stop bit is always logic ‘0’.
7. The park bit is always logic ‘1’ and should be driven high by the host.
8. The ACK bits are the device-to-host response.

   Possible values are shown in Table 2-1 on page 16. Note that the ACK in the current SWD transfer reflects the status of the previous transfer. OK ACK means the previous packet is successful. WAIT response indicates that the previous packet transaction is not yet complete. For a Fault operation, the programming operation should be aborted immediately.

   a. For a WAIT response, if it is a read transaction, the host should ignore the data read in the data phase. PSoC 5LP does not drive the line and the host must not check the parity bit.
   b. For a WAIT response, if it is a write transaction, the data phase is ignored by the PSoC 5LP device. But the host must still send the data to be written from an implementation standpoint. The parity data corresponding to the data should also be sent by the host.
   c. For a WAIT response, it means that the PSoC 5LP device is processing the previous transaction. The host can try for a maximum of four continuous WAIT responses to see if an OK response is received, failing which, it can abort the programming operation and try again.
   d. For a FAULT response, the programming operation should be aborted and retried by doing a device reset.
9. The data phase includes a parity bit (even parity, similar to the packet request phase).
   a. For a read data packet, if the host detects a parity error, then it must abort the programming operation and restart.
   b. For a write data packet, if the PSoC 5LP detects a parity error in the data packet sent by the host, it generates a FAULT ACK response in the next packet.
10. Turnaround (TrN) phase: According to the SWD protocol, the TrN phase is used both by the host and the PSoC 5LP device to change the Drive modes on their respective SWDIO line. There are two TrN phases in each SWD packet. During the first TrN phase after packet request, PSoC 5LP drives the ACK data on the SWDIO line on the rising edge of SWDCK in TrN phase. This ensures that the host can read the ACK data on the next falling edge. Thus, the first TrN cycle is only for half cycle duration, as shown in Figure 2-8 and Figure 2-9. The location of the second TrN phase is different for read and write packets. The second TrN phase of the SWD packet is one-and-a-half cycle long. Neither the host nor PSoC 5LP should drive SWDIO line during both the TrN phases as indicated by ‘z’ in Figure 2-8 and Figure 2-9.
11. The address, ACK, and read and write data are always transmitted least significant bit (lsb) first.
12. At the end of each SWD packet in Figure 2-8 and Figure 2-9, there is a “DUMMY” phase, which is three SWD clock cycles with SWDIO line held low. This DUMMY phase is not part of the SWD protocol. The three extra clocks with SWDIO low are required for the Test Controller in PSoC 5LP to complete the read/write operation when the SWDCK clock is not free-running. For a reliable implementation, include three IDLE clock cycles with SWDIO low for each packet. According to the SWD protocol, the host can generate any number of SWD clock cycles between two packets with SWDIO low.

Note The SWD interface can be reset anytime during programming by clocking 51 or more cycles with SWDIO high. To return to the idle state, SWDIO must be clocked low for three or more cycles. The host programmer can begin a new SWD packet transaction from the idle state.

2.4.1 Register Access Using SWD Interface

To access the registers using the SWD interface, in the 8-bit transfer request packet, set the APnDP bit and select the corresponding ADDR bits, as shown in Table 2-2 on page 16. Table 2-3 shows the 8-bit transfer request packet to access the DP and AP registers for read or write operation. The 8-bit transfer request data in Table 2-3 is transmitted least significant bit first. The 'Start' bit is the least significant bit (LSb) and the 'Park' bit is the most significant bit (MSb) in Table 2-3. Use Table 2-3 and vectors given in the section "SWD and JTAG Vectors for Programming chapter on page 47" to implement PSoC 5LP programming.

Table 2-3. SWD Transfer Request Data Packet for DPACC and APACC Register Access in DAP and TC

Pseudo Code	Register Name	Type of Operation	SWD Transfer Request Data (LSB sent first)	
			Binary	Hex
DPACC IDCODE Read	IDCODE	Read	8'b10100101	8'hA5
DPACC DP CTRLSTAT Write	DP CTRL/STAT	Write	8'b 10101001	8'hA9
DPACC DP SELECT Write	SELECT	Write	8'b10110001	8'hB1
DPACC READBUFF Write	READBUFF	Write	8'b10011001	8'h99
APACC AP CTRLSTAT Write	AP CTRL/STAT	Write	8'b10100011	8'hA3
APACC ADDR Write	AP Transfer Address	Write	8'b10001011	8'h8B
APACC DATA Read	AP Data Read/Write	Read	8'b10011111	8'h9F
APACC DATA Write	AP Data Read/Write	Write	8'b10111011	8'hBB

The 'AP Transfer Address' register holds the PSoC 5LP memory address that needs to be accessed. To read or write PSoC 5LP's internal registers or SRAM, first write the address to the 'AP Transfer Address' register (Pseudo Code – APACC ADDR Write). For a write operation, write data to the 'AP Data Read/Write' register (Pseudo Code – APACC DATA Write). If it is a read operation, read the 'AP Data Read/Write' register twice (Pseudo Code – APACC DATA Read); the test controller (TC) reads out data through the data line.

For example, to write 32'hB6 to the target device internal register at address 32'h40004720, the following SWD transfers are necessary:

APACC ADDR WRITE [0x40004720]

APACC DATA WRITE [0x000000B6]

The binary data for the two SWD packets, with the bit pattern being least significant bit to most significant bit (from left to right), are as follows.

11010001 (ACK) 00000100111000100000000000000010(0)

11011101 (ACK) 01101101000000000000000000000000(1)

'(ACK)' indicates waiting for ACK from the target device. This '(ACK)' is for the previous SWD transfer as explained earlier. The last bit in data phase (enclosed in brackets above) is the parity bit for the 32-bit data.

SWD register read is similar to SWD write operation, except that the read operation should be done twice to get the correct data. First, host should write the address to the APACC ADDR register address. Then, it should read the DATA_RW register twice. The first read initiates the command to the DAP interface and the second read returns the requested value.

For example, to read from address 32'h40004720, the following transfers need to be done:

APACC ADDR Write [0x40004720]

Dummy_data = APACC DATA Read //dummy SWD read

Data = APACC DATA Read //returns actual data

Note The previous two examples do not consider the three dummy clocks cycles required at the end of each SWD packet. They should be appended, as shown in Figure 2-8 and Figure 2-9, if the SWDCK clock is not free running.

To simplify the process, the programmer can have a SWD command interpreter that implements Table 2-3 on page 19 and outputs data in binary format. An example follows. The SWD_packet function recognizes the SWD transfer that is given and puts the corresponding binary data into the outgoing data buffer for transmission.

SWD_packet (APACC_ADDR, 32'h40004720)

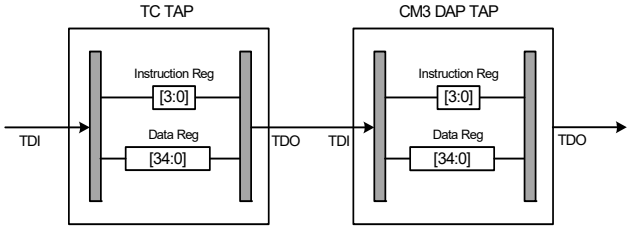
SWD_packet (APACC_DATA_WRITE, 32'hB6)

Data = SWD_packet (APACC_DATA_READ)

2.5 JTAG Interface

The PSoC 5LP JTAG interface complies with the IEEE 1149.1-2001 specification and provides additional instructions. There are two TAPs in the silicon. One is in the TC and the other is in the Cortex-M3's DAP, which is used for device debug and programming. The two TAPs are connected in series, where the TDO of the TC TAP is connected to the TDI of the DAP TAP. This is illustrated in Figure 2-10.

Figure 2-10. TC/DAP TAP Connection



Each TAP consists of a 35-bit data register (called DP/AP access register) and a 4-bit instruction register. Refer to the "Test Controller" chapter of the "PSoC 5LP Architecture TRM" for details on the instructions supported by the JTAG interface and an explanation of the JTAG TAP controller state machine. The important instructions to program the device through JTAG are listed in Table 2-4. The timing diagrams are in the section Programming Specifications chapter on page 43.

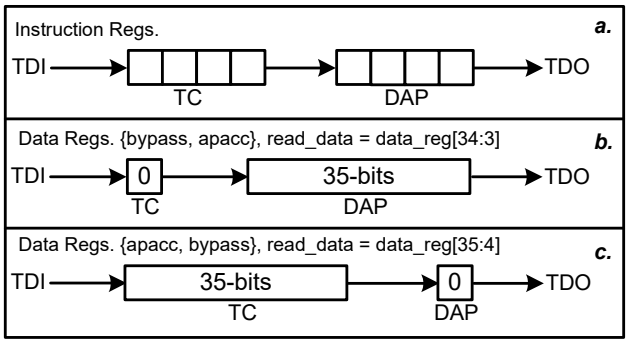
Table 2-4. PSoC 5LP JTAG Instructions

Bit Code [3:0]	Instruction	PSoC 5LP Function
1110	IDCODE	Connects TDI and TDO to the device 32-bit JTAG ID code.
1010	DPACC	Connects TDI and TDO to the DP/AP access register (35-bit), for access to the Debug Port registers.
1011	APACC	Connects TDI and TDO to the DP/AP access register (35-bit), for access to the Access Port registers.
1111	BYPASS	Bypasses the device, by providing 1-bit latch (bypass register) connected between TDI and TDO.

The 35-bit data register (DP/AP access register) is used for DPACC and APACC instructions. The 35-bit data register structure for JTAG write and read operations are as shown in Figure 2-6 and Figure 2-7, respectively.

Table 2-4 also lists which instructions are applicable for each TAP. If an instruction that is not applicable is shifted into a TAP, the TAP goes into bypass mode. In by_pass mode, the data register is only 1 bit long with the contents of 0. The bypass mode is used to isolate the target TAP. For example, if targeting the TC TAP, the DAP TAP is put in bypass mode by shifting in the BYPASS instruction into its instruction register and if targeting the DAP TAP, the TC TAP will be placed in bypass mode. See the examples of TAPs configuration in Figure 2-11.

Figure 2-11. TC/DAP TAP Configuration Examples



a) Instructions registers combined. 8 bits total.

b) Access the DAP's APACC registers for device debug and programming. TC TAP in bypass mode.

c) Access the TC's APACC registers for enabling test modes. DAP TAP in bypass mode.

2.5.1 Register Access Using JTAG Interface

The following steps show how to access an address using the JTAG interface. Note that DAP must be configured before the first three commands from "Step 2: Configure Target Device" on page 49 are executed.

1. Put TC's TAP into BYPASS mode.
2. Assume that the address value is 0x40007014 and data '0xDA' needs to be written to this register.
   a. Shift the APACC instruction into the instruction register.
   b. Shift a '0' (write) followed by '01' (selecting TRNS_ADDR register) followed by '0x40007014' (32-bit address), into the 35-bit data register. For each element, the LS bit is shifted out first.
   c. Shift a '0' (write) followed by '11' (selecting DATA_RW register) followed by a '0x000000DA' (8-bit data) into the 35-bit data register. For each element, the LSB is shifted first.
   d. The DAP initiates a write transfer request to the PSoC 5LP's memory.
3. Assume that the data to be read from the register has an address value 0x40007014.
   a. Shift the APACC instruction into the instruction register.
   b. Shift a '0' (write) followed by '01' (selecting TRNS_ADDR register) followed by '0x40007014' (32-bit address), into the 35-bit data register. For each element, the LSB is shifted first.
   c. Shift a '1' (read) followed by '11' (selecting DATA_RW register) into the 35-bit data register. For each element, the LSB is shifted first. Note that for read operation, the 32-bit data written is not used.
   d. The TC or DAP (depending on configuration) initiates a read transfer request to the PSoC 5LP's memory; the data read from DATA_RW is invalid in this cycle.
   e. Wait at least five TCK clock cycles to avoid a WAIT response.
4. Read the DATA_RW register again. The data is now valid.

2.6 Switching between JTAG and SWD Interfaces

PSoC 5LP supports programming through both the SWD and JTAG interfaces. It is also possible to switch from the SWD to JTAG protocol or vice-versa at any time. This switching is done by sending a specific key sequence on the SWDIO/TMS shared pin (referred to as SWDIOTMS) with the clock on the TCK/SWDCK shared pin (referred to as SWCLKTCK). This may be needed for JTAG interface programming.

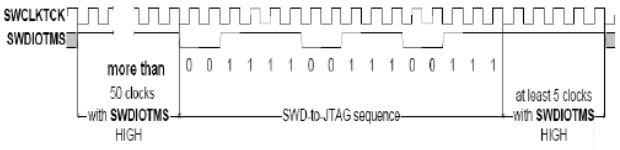
It is not recommended to use combined protocols to program a chip. This is useful only for specific cases of the JTAG chain. If there are several devices in the chain, and some of them configure debug pins to SWD, then the master must switch them all to the JTAG interface. In other cases, some of them configure debug pins to GPIO. In this case, the master must acquire all chips together by the SWD, and only then switch them to the JTAG mode. Normally this does not happen. For the multi-device JTAG chain, all devices must configure debug pins to the JTAG mode.

2.6.1 SWD to JTAG Switching

To switch programming interface from SWD to JTAG (4- wire) operation, use the following steps:

1. Send 51or more **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that the current interface is in its reset state. The serial wire interface detects the 16-bit SWD-to-JTAG sequence only when it is in the reset state.
2. Send the 16-bit SWD-to-JTAG select sequence on **SWDIOTMS**. The 16-bit SWD-to-JTAG select sequence is 0b0011_1100_1110_0111, MSB first. This can be represented as either:
   a. 0x3CE7 transmitted MSB first.
   b. 0xE73C transmitted LSB first.

Figure 2-12. SWD to JTAG Switching Sequence



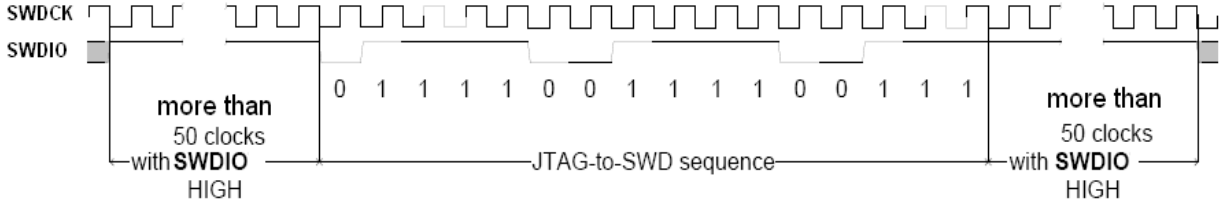
3. Send at least five **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that if the programming interface is already in JTAG operation before sending the select sequence, the JTAG TAP enters the Test-Logic-Reset state.

2.6.2 JTAG to SWD Switching

To switch DAP from JTAG to SWD operation, use the following steps:

1. Send 51 or more **SWDCK** cycles with **SWDIO** HIGH. This ensures that the current interface is in its reset state. The JTAG interface only detects the 16-bit JTAG-to-SWD sequence starting from the Test-Logic-Reset state.
2. Send the 16-bit JTAG-to-SWD select sequence on **SWDIO**. The 16-bit JTAG-to-SWD select sequence is 0b0111_1001_1110_0111, most-significant bit (MSB) first. This can be represented as either:
   a. 0x79E7 transmitted most-significant bit (MSb) first
   b. 0xE79E transmitted least-significant bit (LSb) first.

Figure 2-13. First Three Steps of JTAG to SWD Switching

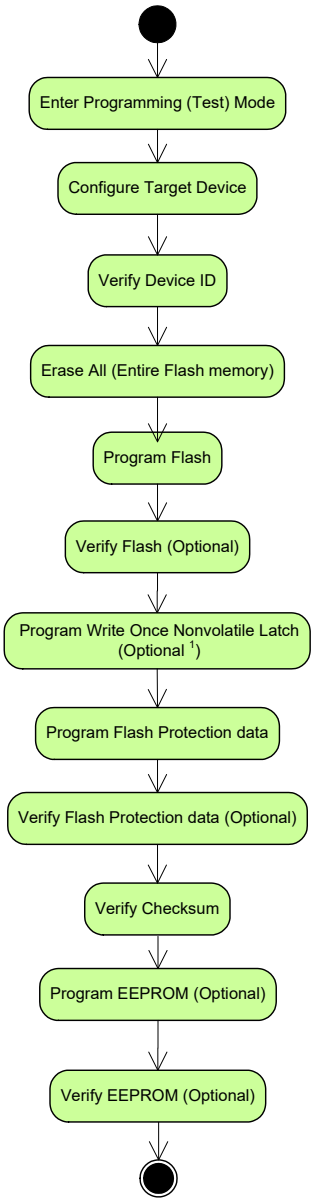


3. Send 51 or more **SWDCK** cycles with **SWDIO** HIGH. This ensures that if DAP is already in SWD operation before sending the select sequence, the SWD interface enters line reset state.
4. Send three or more **SWDCK** cycles with **SWDIO** low. This ensures that the SWD line is in the idle state before starting a new SWD packet transaction.
5. Send the **DPACC IDCODE READ** SWD read packet as given in Table 2-3 on page 19. It is not necessary to process the Device ID returned by the PSoC 5LP device for this read packet. Ignore the Device ID returned by PSoC 5LP in this step.

3. PSoC 5LP Programming Flow

Figure 3-1 shows the sequence of steps involved in programming a PSoC 5LP device. Each step is discussed in detail in later sections. All steps in Figure 3-1 must be completed successfully for a successful programming operation. The programming operation should be stopped if there is a failure in any of the steps. The SWD and JTAG packets for each step are provided in SWD and JTAG Vectors for Programming chapter on page 47.

Figure 3-1. PSoC 5LP Programming Flow



3.1 Step1: Enter Programming Mode

The first step in PSoC 5LP device programming is to enter the Programming mode, also called the Test mode. The host programmer must complete this step successfully for the remaining programming steps to be successful.

The procedure to enter the programming mode depends on the method used to reset the PSoC 5LP device. The two methods to reset PSoC 5LP are as follows:

- Using the device reset (XRES) pin: In this method, the host programmer drives the XRES pin of PSoC 5LP low to do a device reset.
- Power cycle mode: In this method, the host programmer toggles power to PSoC 5LP's power supply pins (Vddd, Vdda, and Vddios) to do a device reset.

There are several initial conditions on how the part can come out of reset and some of these scenarios require different initialization steps. These conditions depends on Debug_En and Debug Port Settings fields in Custom NVL memory (see Appendix).

In some scenarios, access to Cortex-M3 DAP is available and the programmer can start working with the nonvolatile memory right away (when DPS = SWD/JTAG and Debug_En = On). But in other cases, the programmer must execute special acquire sequence to get access to Cortex-M3 DAP (for example, when DPS = GPIO and Debug_En = OFF). In such cases, the programmer must reset a part and then send an acquire key during the 400 uS boot window to enable the SWD port and connect to DAP. This specification will consider two methods of entering into programming mode:

- **SWD universal acquisition**. This method can be used independently on current DPS or Debug_En settings. This method must be implemented by a third-party programmer to be 100 percent compatible with the PSoC 5LP device.
- **JTAG compliant acquisition**. This method is in field compliance with IEEE 1149.1 standard and requires only four JTAG wires (no XRES). This method is only available if DPS = JTAG and Debug_En = ON.

Both these methods are described in detail here.

3.1.1 SWD Universal Acquisition

Figure 3-2 on page 27 shows the sequence of steps to enter the programming mode (or test mode) of the PSoC 5LP using SWD interface; Figure 3-3 on page 27 shows the corresponding timing diagram. See Table 4-3 on page 45 for specifications of the timing parameters mentioned in Figure 3-2 on page 27 and Figure 3-3 on page 27. Figure 3-3 on page 27 shows both the XRES method and power cycle mode of programming. Each of these methods are explained in separate sections

Figure 3-2. Entering Programming (Test) Mode through SWD Interface

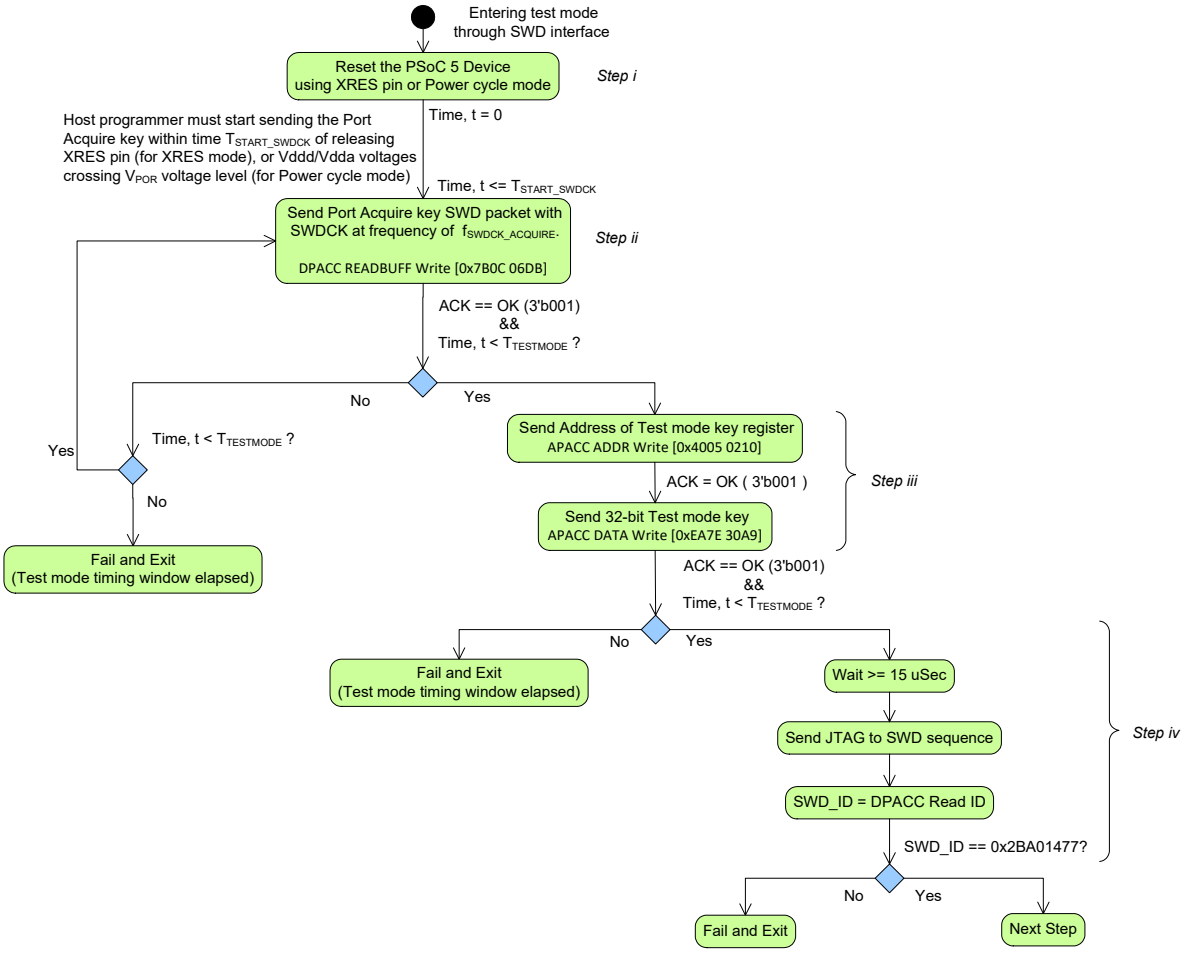
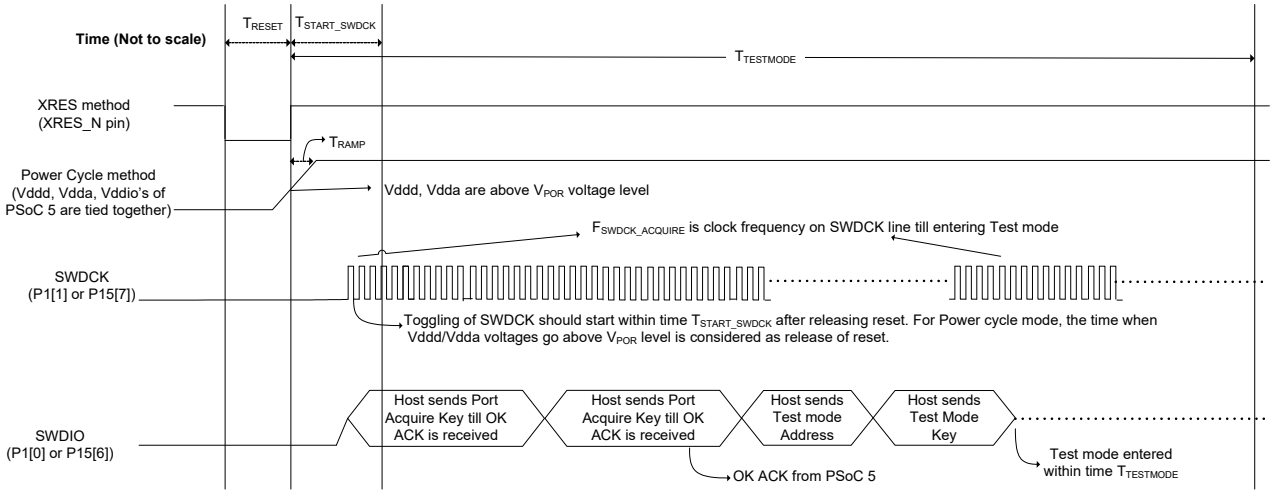


Figure 3-3. Timing Diagram to Enter Test Mode through SWD Interface



3.1.1.1 SWD Programming using XRES Pin

The sequence in Figure 3-3 on page 27 is using the SWD interface and XRES pin as follows.

1. The host programmer drives the XRES pin of PSoC 5LP low to cause a device reset. The reset signal is active low, and the reset pulse width is specified by the T_{RESET} timing parameter.
2. Within time T_{START_SWDCK} of releasing XRES signal, the host must start sending the Port Acquire key on SWDIO and SWDCK lines. The host must send this Port Acquire key continuously until an OK ACK is received from PSoC 5LP. The pseudo code is given here.

```
do
{
/* Write Port Acquire key, Use SWD ADDR =
       2'b11*/
    DPACC READBUFF Write [0x7B0C 06DB]
//Check port acquire retry time and
       whether OK ACK is received
} while (ACK != "OK" AND time_elapsed <
       T_TESTMODE)

// Exit on timeout
if (ACK != "OK" OR time_elapsed >
T_TESTMODE) then FAIL_EXIT
```

   If the debug port is disabled, PSoC 5LP ignores the first Port Acquire SWD packet sent after releasing reset. It does not return an OK ACK for the first packet. PSoC 5LP sends an OK ACK only during the second try of the Port Acquire SWD packet. Therefore, the port acquire sequence must be sent continuously on the SWD interface until an OK ACK is received. The timeout window for this loop is $T_{TESTMODE}$, the programming (test) mode entry window duration.

   Significance of SWDCK frequency $f_{SWDCK\_ACQUIRE}$: In Figure 3-2 on page 27 and Figure 3-3 on page 27, the SWDCK frequency during test mode entry is $f_{SWDCK\_ACQUIRE}$. The host programmer must meet this frequency specification to successfully enter PSoC 5LP programming mode. After device reset is released, the internal test controller logic in PSoC 5LP looks for the clock transitions on the SWDCK line. If the test controller logic notices eight SWDCK clock cycles within a time window of $T_{ACQUIRE}$, it extends the time to enter programming mode to $T_{TESTMODE}$. This time window can be anywhere within duration T_{BOOT} (68 µs) after device reset. T_{BOOT} is the time for PSoC 5LP boot to complete after device reset is released. By ensuring that the SWDCK line is always clocked at a frequency of $f_{SWDCK\_ACQUIRE}$, the host programmer can meet PSoC 5LP test mode entry timing requirements. Note that for bit banging host programmers, which cannot generate a constant clock frequency of $f_{SWDCK\_ACQUIRE}$ on the SWDCK line for entire SWDCK packet duration, an alternate acquire method is explained in a later section.
3. After the host programmer receives an OK ACK for the port acquire sequence, it must write the test mode key to the Test Mode Key register to enter PSoC 5LP programming mode. This key must be written within time $T_{TESTMODE}$, as shown in Figure 3-2 on page 27 and Figure 3-3 on page 27. By ensuring that SWDCK is clocked at a frequency of $f_{SWDCK\_ACQUIRE}$ during this step, the host programmer can enter PSoC 5LP programming mode within time $T_{TESTMODE}$. The pseudo code for this step is given here.

```
APACC ADDR Write [0x4005 0210] // Address of
the Test mode key register
APACC DATA Write [0xEA7E 30A9] // Write 32-
bit test mode key

/* Exit on timeout or reception of FAULT
response means the device did not enter
Programming mode within time T_TESTMODE. Retry
again by doing reset and restarting.*/
if (ACK != "OK" OR time_elapsed > T_TESTMODE
usec) then FAIL_EXIT
```

3.1.1.2 SWD Programming using Power Cycle Mode:

Power cycle mode programming is identical to XRES method from a programming algorithm standpoint, as shown in Figure 3-2 on page 27 and Figure 3-3 on page 27. The only difference is that, instead of driving the XRES pin, the host programmer toggles power to the PSoC 5LP power supply pins (Vddd, Vdda, Vddio0, Vddio1, Vddio2, and Vddio3) to cause a device reset.

The power cycle method is complex to implement compared to the XRES method because it requires special hardware design considerations for power toggling. The power cycle mode programming also requires that the Vdda, Vddd, and Vddio power supply pins in PSoC 5LP are tied to the same power supply and toggled at the same time, as shown in Figure 3-2 on page 27. It is recommended to implement the XRES method of programming because it is easier to implement. Power cycle mode programming is required in two case:

- When the optional XRES pin (P1[2]) in 68-pin SSOP parts is configured as a GPIO pin, the only way for the host programmer to do a device reset is to toggle power to PSoC 5LP. This is because there is no dedicated XRES pin in 68-pin parts unlike the other pin count packages. Note that this condition of disabling P1[2] as XRES for 68-pin parts is done only by the user and not by Cypress. The 68-pin parts coming from factory have the P1[2] pin configured as XRES by default. But if the user

programs a hex file that disables P1[2] as XRES, then XRES method is not available for subsequent tries of programming. The power cycle method must be used in such a case.

- If it is required to program PSoC 5LP using the SWD interface's USB pins (P15[6], P15[7]), then the host programmer can toggle power to USB interface's VBUS pin to cause a device reset and program using the USB SWD pins. In this case, the VBUS power pin in the USB interface powers the Vddd, Vdda, and Vddio power supply pins in PSoC 5LP.

Ramp Rate Requirements for Power Cycle Mode Programming

The maximum power supply ramp rate is specified in the PSoC 5LP device datasheet as parameter Svdd. There is no minimum ramp rate requirement specified for power cycle mode. A slower ramp rate requires special hardware considerations as follows:

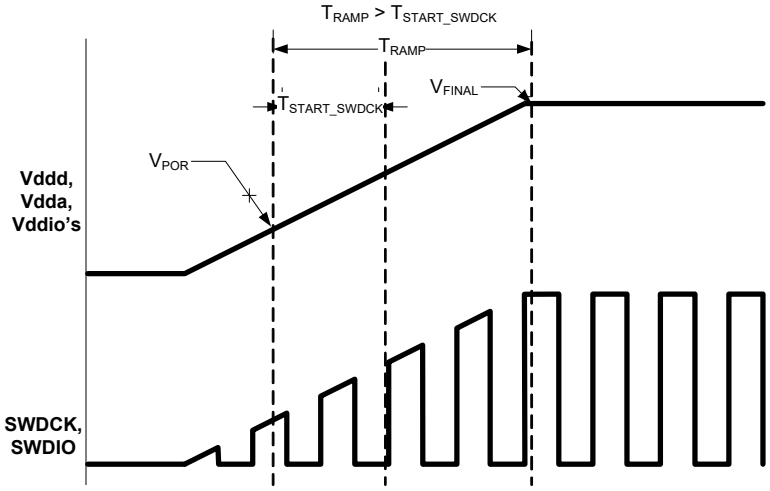
- When the power supply ramp duration (T_{RAMP}) from VPOR to final value is less than T_{START_SWDCK}.

  Figure 3-3 on page 27 shows that the host programmer must start sending the Port Acquire sequence within time duration T_{START_SWDCK} of Vddd and Vdda voltage levels crossing VPOR voltage level specification. If the time (T_{RAMP}) for power supplies to ramp from VPOR to final supply voltage is less than T_{START_SWDCK}, then the host programmer can start sending the Port Acquire sequence after Vddd, Vdda, and Vddio pins have reached final voltage value.

- When power supply ramp duration from VPOR to final value (T_{RAMP}) is more than T_{START_SWDCK}

  In this case, the host programmer cannot wait for power supplies to ramp to the final voltage value before sending the Port Acquire sequence. Otherwise, the host programmer cannot meet the timing requirements to enter PSoC 5LP programming mode. The host programmer should implement the power cycle mode shown in Figure 3-4. It should start sending the Port Acquire sequence even as the power supplies (Vddd, Vdda, Vddio) ramp up. Adjust the voltage levels of the SWDCK and SWDIO lines to match the instant value of the power supply pins. This method is implemented in Cypress's MiniProg3 programmer in which the ramp rate duration (T_{RAMP}) is greater than T_{START_SWDCK}. This implementation ensures that the PSoC 5LP's test controller is able to detect data (logic levels) on the SWDIO and SWDCK lines even when the power supply is ramping.

Figure 3-4. Power Cycle Mode Implementation for $T_{RAMP} > T_{START\_SWDCK}$



3.1.1.3 *SWD Programming using Bit Banging Host Programmers:*

Some host programmers implement the SWD interface as a bit banging implementation. Examples of such host programmers are microcontrollers in which the SWDIO and SWDCK signals are generated by writing to specific port registers of the microcontroller.

It is not possible for some of the bit banging programmers to generate the SWDCK clock signal at a constant frequency of $f_{SWDCK\_ACQUIRE}$ for the entire SWD packet, as shown in Figure 3-2 on page 27 and Figure 3-3 on page 27. A modified method of entering PSoC 5LP programming mode is given for these programmers. This method is applicable only for programmers that use the XRES pin. It is not applicable for power cycle mode programming due to the constraints it imposes on power supply ramp rates.

Figure 3-5 on page 31 shows the modified steps to enter test mode of PSoC 5LP; Figure 3-6 on page 31 shows the corresponding timing diagram. See Table 4-3 on page 45 for

specifications of timing parameters. The primary need for SWDCK clocking at frequency of $f_{SWDCK\_ACQUIRE}$ is to meet the condition of "8 SWDCK clock cycles in the time window $T_{ACQUIRE}$". On detection of these eight clocks, the time to enter test mode is extended to $T_{TESTMODE}$. The time window $T_{ACQUIRE}$ can occur anywhere during time T_{BOOT}. To simplify the implementation for bit banging programmers, the method in Figure 3-5 on page 31 requires the programmer to toggle SWDCK alone at a frequency of $f_{SWDCK\_ACQUIRE}$ with SWDIO held low. This ensures that the host programmer meets the initial test mode timing requirements. An example C code that implements Figure 3-5 on page 31 is given here.

```
/* Set LOOP_COUNT value based on number of
loop cycles needed to execute the "Initial
Port Acquire window" loop below for time
TBOOT */
#define LOOP_COUNT 240

uint16 j = 0; /* Variable to keep track of
       no. of times to generate SWDCK clock
       */

XRES_LOW; /* Generate active reset on XRES
       line for at least for time T_RESET */
XRES_HIGH; /* Release XRES */

SWDIO_LOW; /* Hold the SWDIO line low during
T_BOOT */

/*---------------Initial Port Acquire win-
dow, T_BOOT---------------------*/
do
{
/* Ensure that SWDCK frequency is greater
than f_SWDCK_ACQUIRE */
       SWD_CLOCK_LOW;
       SWD_CLOCK_HIGH;
       j++;
       }while(j < LOOP_COUNT);
/*---------------End of Initial Port Acquire
       window---------------*/

/*Send Port Acquire key, Test mode address,
Test mode key SWD packets at frequency of
f_SWDCK_BITBANG to complete all steps within
time T_TESTMODE*/
```

After time T_{BOOT}, the programmer must send the port acquire, test mode key SWD packets. These SWD packets should be sent within time $T_{TESTMODE}$.

Figure 3-5. Enter Test Mode through SWD Interface (for bit banging programmers)

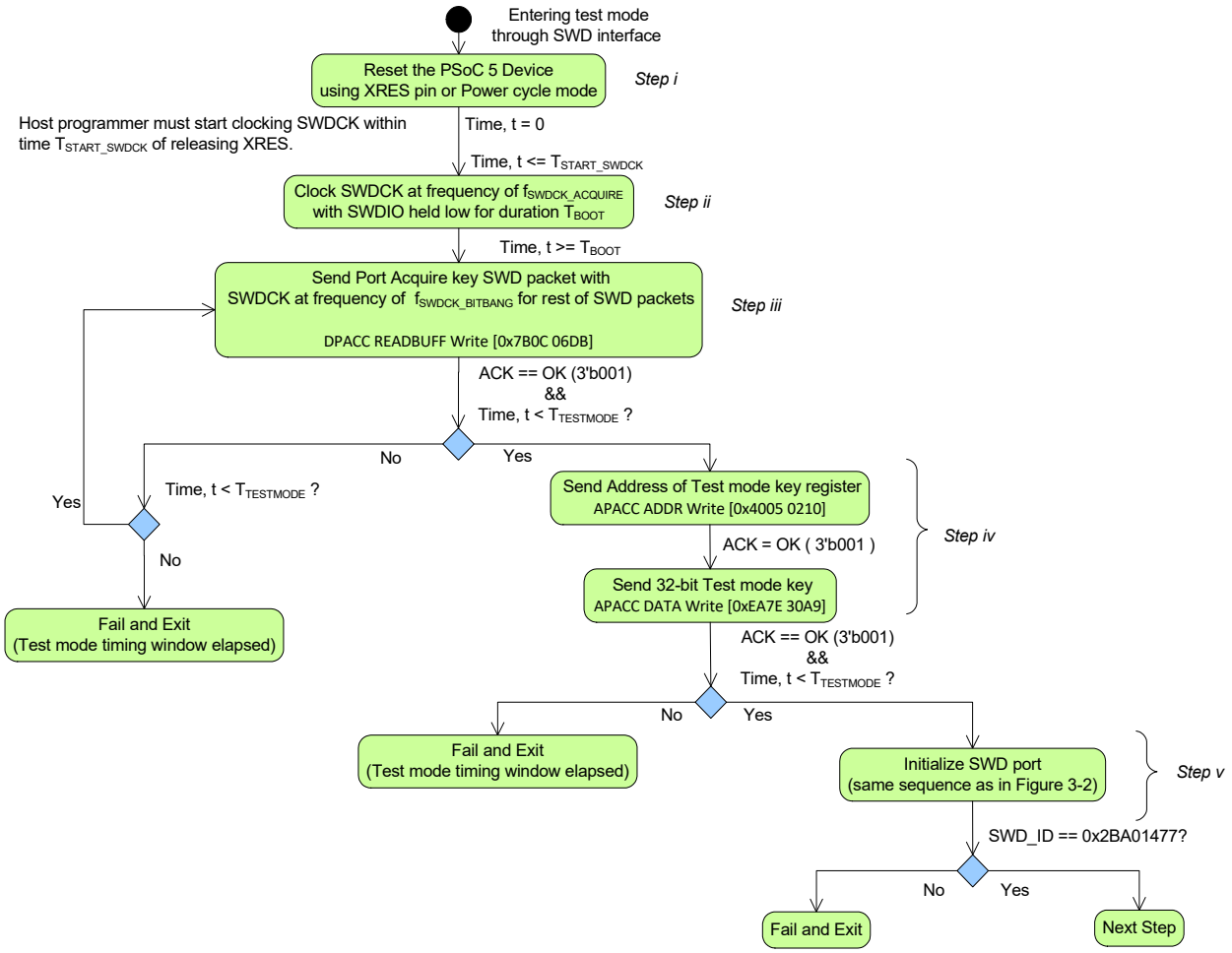
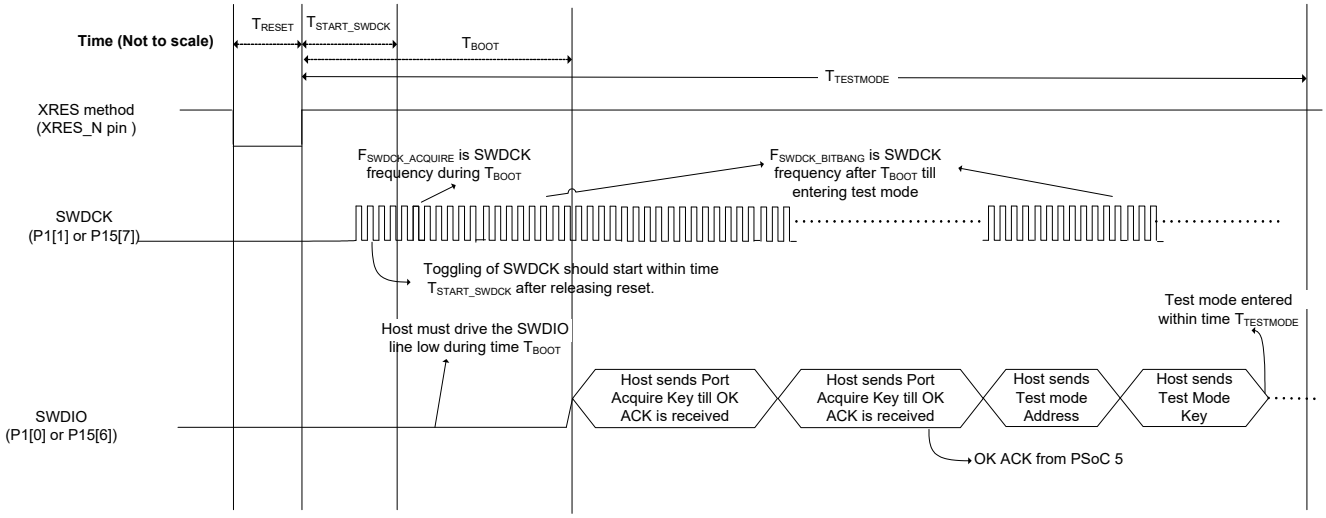


Figure 3-6. Timing Diagram to Enter Test Mode through SWD Interface (for bit banging programmers)



3.1.1.4 Determine $f_{SWDCK\_BITBANG}$:

In Figure 3-5 on page 31, the programmer must send the SWD packets after time T_{BOOT} at a frequency of $f_{SWDCK\_BITBANG}$. This frequency requirement is to meet the $T_{TESTMODE}$ timing requirement. The value of $f_{SWDCK\_BITBANG}$ depends on bit banging programmer implementation. An example calculation for $f_{SWDCK\_BITBANG}$ that assumes no overhead in sending SWD packets is given here.

In PSoC 5LP, a maximum of two Port Acquire SWD packet tries are required to get OK ACK. The test mode address and test mode key require another two SWD packets. A maximum of four SWD packets must be sent by the programmer within time ($T_{TESTMODE}$ - T_{BOOT}). The minimum value of $T_{TESTMODE}$ from Table 4-3 on page 45 is 395 µs, and T_{BOOT} is 68 µs; the difference factor is 327 µs. Each SWD packet requires 49 SWDCK clock cycles (including the three dummy clock cycles at end of each SWD packet); , hence, 196 SWDCK clock cycles are required for four SWD packets.

$T_{SWDCK\_BITBANG}$(no overhead) ≤ (327 µs/196) ≅ 1.6 µs

$f_{SWDCK\_BITBANG}$(no overhead) ≥ (1/1.6 µs) ≅ 0.7 MHz

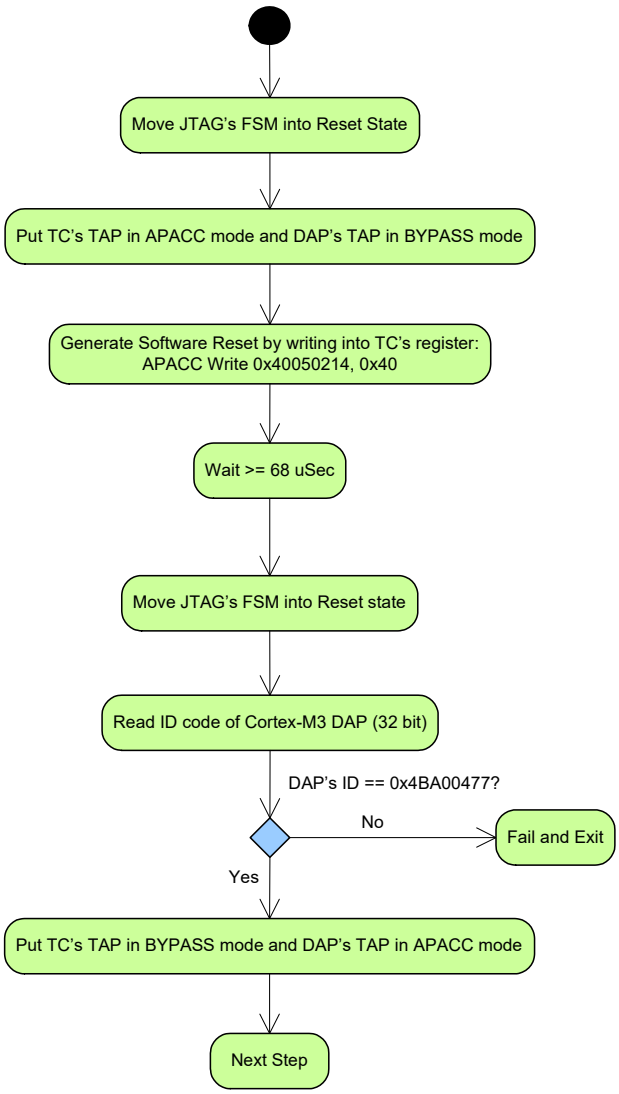
This example calculation assumes no overhead in sending the SWD packets on the host programmer side. The minimum frequency requirement increases with other additional overhead; this is specific to host programmer architecture.

The frequency parameter $f_{SWDCK\_BITBANG}$ refers to the average frequency of the SWDCK clock generated by host programmer. Bit banging programmers cannot generate constant frequency on the SWDCK line during the entire SWDCK packet. But the average SWDCK frequency must be greater than the minimum value of $f_{SWDCK\_BITBANG}$ so that the programming mode is entered within time $T_{TESTMODE}$.

3.1.2 JTAG Compliant Acquisition

The PSoC 5LP silicon can be programmed in full compliance with IEEE 1149.1 standard. For example, SVF or STAPL scripts for JTAG programming can be generated from the hex file and executed in the third-party JTAG tools. To be compatible with the JTAG standard, the default device factory settings for DPS is “4-wire JTAG” and for Debug_En is “Enabled”. It means that access to Cortex-M3 DAP is always available (during firmware execution) and the JTAG master can start communication with DAP any time. Figure 3-7 shows the steps to enter programming mode (or test mode) of PSoC 5LP in compliance with IEEE 1149.1 standard.

Figure 3-7. Enter Programming (Test) Mode through JTAG



Following are the details of Figure 3-7:

1. **Device Reset** - If reset mode is used to start programming, then it is recommended to reset the device before starting to program. Because the JTAG standard does not specify the XRES pin, this step can be considered optional. You should be able to start programming at any time of firmware execution. However, PSoC 5LP supports programmatic reset by setting a gen_tcr (0x40) bit of TC_PM_CTRL (0x40050214) register in the Test Controller block. The chip has the ability of software reset and should be used as a synchronization mechanism for the programmer and target.

2. **Wait for >= 68 uSec** - This is a necessary step to ensure that at least 68 uS are elapsed from the last device reset. It is recommended to have this delay in the milliseconds range (for example, 1 mS).

3. **Reset JTAG FSM** - This is needed to synchronize FSMs of all the JTAG devices on the chain, which can be in the unknown state at the start of programming.

4. **Reading/Checking ID** - This step reads ID of Cortex-M3 DAP's TAP. First, set TC's TAP in BYPASS mode and set ID CODE in DAP's IR. The DAP returns the ARM's ID of Cortex-M3 CPU. It is the same for all PSoC 5LP packages. The verification of the ID ensures that a proper connection with PSoC 5LP silicon is established and that it is correct to goon. The ID is returned in a single 32-bit word.

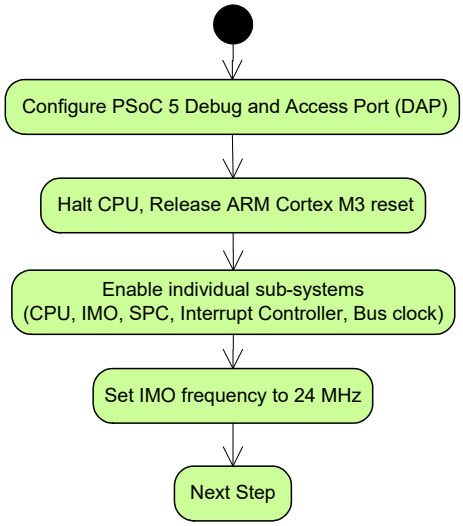
5. **Initialize IRs** - PSoC 5LP contains two TAPs and the programmer must initialize them correctly (see "Test Controller Block" on page 12 section). During programming only DAP's TAP is used, so the Test Controller's TAP must be put in BYPASS mode (see "JTAG Interface" on page 20 section). The IR's size for each TAP is four bits, and the DR's size is 35 bits. After IR is configured, the TC's DR will be 1 bit long (bypass latch), and the DAP's DR will be 35 bits long. Externally, PSoC 5LP will appear as one JTAG device with an 8-bit IR and a 36 bit DR.

Note that the JTAG compliant programming shown in Figure 3-7 is only available with certain settings of DPS and Debug_En fields in Custom NVLs. The third-party programmer must ensure that DPS and Debug_En are never programmed with other settings. The locations of these settings in the hex file is described in the Appendix chapter on page 77. The programmer software can throw an error message and abort operation if a hex file with different settings attempts to program PSoC 5LP. The default device factory settings for DPS = "4-wire JTAG" and for Debug_En = "Enable". They must not be changed to ensure the device is compliant with IEEE 1149.1 standard.

3.2 Step 2: Configure Target Device

Figure 3-8 shows the sequence to configure the target PSoC 5LP device before programming the device.

Figure 3-8. Configuring Target PSoC 5LP Device



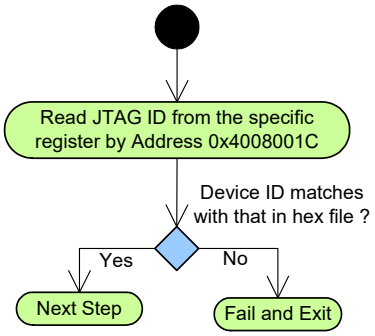
After entering Programming mode, the host programmer must do certain register writes to configure the target device. These are required to configure the PSoC 5LP Debug and Access Port (DAP), halt the CPU, activate debug mode, enable different sub-systems (IMO, bus clock, CPU), and configure clocks (IMO).

3.3 Step 3: Verify JTAG ID

To ensure that the target device corresponds to the device for which the hex file is meant, the device ID of the target device must be compared against the Device ID information in the hex file. This ensures that the hex file is completely compatible with the Device under Test (DUT). If there is a mismatch in the device IDs, the programming operation should be stopped. See “Intel Hex File Format” on page 77 for information on the location of the device ID in the hex file.

The PSoC 5LP JTAG ID is located in the special register of the CPU address space. The register address is 0x4008001C. The flow chart of this step is shown in Figure 3-9.

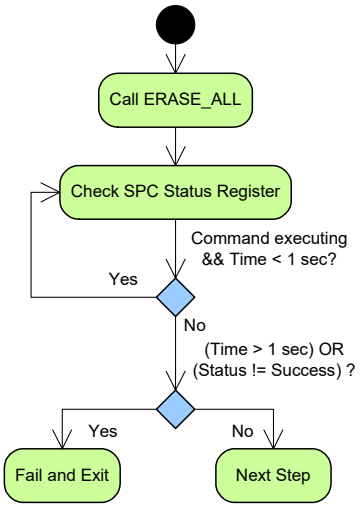
Figure 3-9. Verify Device ID of Target Device



3.4 Step 4: Erase Flash

Figure 3-10 demonstrates the Erase Flash process, which erases all flash data and configuration bytes, and all flash protection rows.

Figure 3-10. Erase Flash Sequence



All the nonvolatile memory (flash, EEPROM, NVL) erase and program operations are done through a simple command and status register interface. The Test Controller (TC) accesses programming operations by writing to the command data register (SPC_CPU_DATA) at the address 32’h40004720. After providing a valid command, the host should wait until the command is executed. When a command is completed, the status is available in the status register (SPC_SR). The status register can be polled to see if the command is executed successfully.

These details are explained in ““Nonvolatile Memory Programming” on page 80”. For more information on nonvolatile memory programming, refer to the “PSoC 5LP Architecture TRM”.

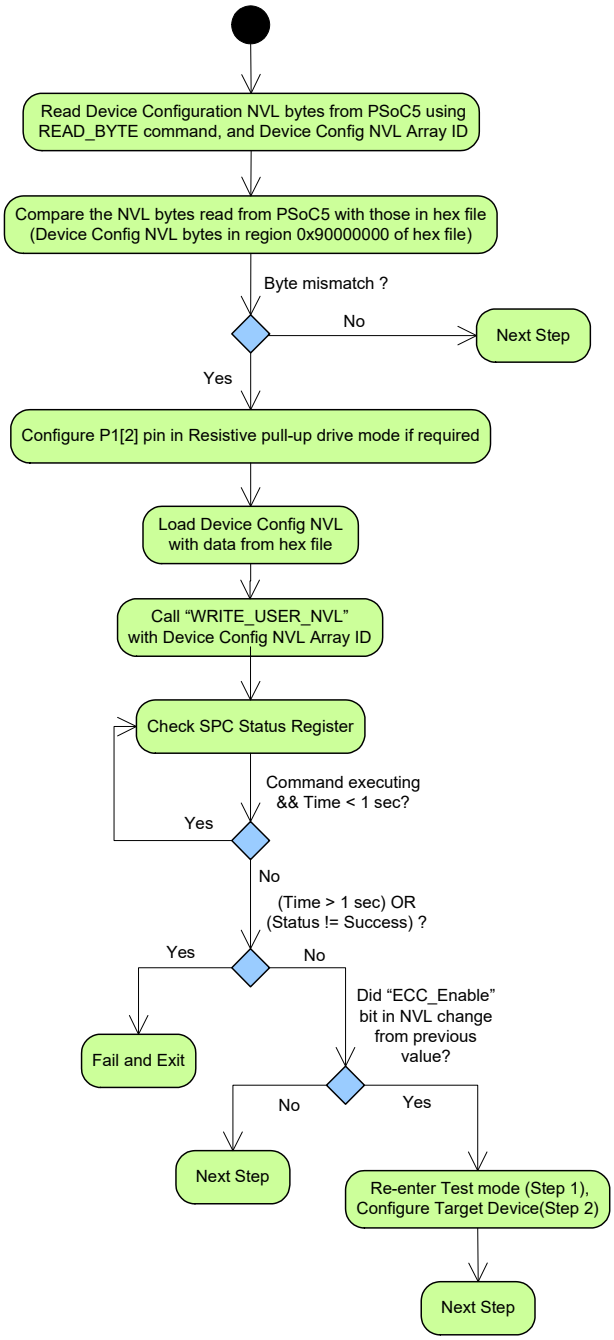
A single command requires several SWD writes to the command data register. The ERASE_ALL command has three parameters, and they should be written to the command data register. After calling ERASE_ALL, the target device starts erasing the entire flash. The ERASE_ALL command should not take longer than 1 second, otherwise an overtime error occurs.

3.5 Step 5: Program Device Configuration NVL

Figure 3-11 on page 35 shows the Program Device Configuration Nonvolatile Latch (NVL) setup flow. This step writes the 4-byte Device Configuration NVL (Custom NVLs). The data to be written to the NVL is located in address 32'h90000000 of the hex file. The LOAD_BYTE and WRITE_USER_NVL commands are used in this step. The LOAD_BYTE command loads the data one byte at a time to a 4-byte latch. The WRITE_USER_NVL command writes the four bytes of loaded data in the latch to NVL. Therefore, the LOAD_BYTE command needs to be called four times, followed by one WRITE_USER_NVL command. The SPC status register needs to be polled to check when the command finishes the write operation. The WRITE_USER_NVL command should not take longer than 1 second, otherwise an overtime error occurs.

Before writing the device configuration data, the P1[2] pin should be configured for resistive pull-up drive mode in special scenario (P1[2] can be either GPIO or XRES pin). This configuration of pins is needed because P1[2] is shared with XRES pin, which can be set during NVLs write. Due to silicon design the reset pulse is generated on P1[2] when XRES gets enable in NVLs. Having pull up enabled on this pin disables this pulse to penetrate into device. So, the programmer must enable pull up on P1[2] if NVL write is going to set XRES mode on this GPIO pin.

Figure 3-11. Program Device Configuration NVL



The NV latches in PSoC 5LP have a much lesser endurance compared to flash and EEPROM memory. Due to this, the user NVL is written only if new data needs to be programmed into the latch. This ensures that the latches are programmed only when there is change in the configuration data in the hex file, which in turn maximizes the endurance time.

When programming the user NVL, if the ECC Enable bit has changed from its previous value, then it is necessary to reset the chip and acquire it again and re-enter the Programming mode (repeat Step 1 and Step 2). This is because the modified ECC setting takes effect only when the chip is reset again; the modified value is needed for the Program Flash, Verify Flash, Program Flash Protection, and Verify Flash Protection steps.

3.6 Step 6: Program Flash

Flash memory in PSoC 5LP is programmed in rows. Each row has 256 code bytes and 32 ECC bytes. There is an option to use the ECC memory space to store configuration data. The row latch to program the flash row288 bytes (if ECC is enabled) or 288 bytes (if ECC is disabled). The flash data to be programmed comes from the hex file. See “Intel Hex File Format” on page 77 for information on the location of flash programming data in the hex file.

During the programming process, if the ECC feature is enabled, the row latch needs to be loaded with all the 256 bytes of data. In this scenario, the 256 code bytes should be fetched from main flash data region of hex file at address 0x0000 0000. If ECC is disabled, the 32 ECC bytes should be fetched from the configuration data region of hex file at address 0x8000 0000. The programmer software should concatenate these 32 bytes with the 256 bytes to form the 288 byte row data that needs to be loaded into the row latch. This step needs to be done to program all flash rows.

The ECC enabled/disabled setting is stored in bit 3 of byte 3 of device configuration NVL. This byte is stored in address 0x90000003 of hex file. The Programmer software must check this bit to determine the size of the flash row to be programmed.

There are three parameters to consider in the flash programming process.

- Number of flash arrays (K) of flash memory: The value of ‘K’ depends on flash memory size. The flash memory in PSoC 5LP is organized as flash arrays, where each flash array can have maximum size of 64 KB. Each flash array in turn is organized as rows, where the size of each row is 256 code bytes and 32 configuration bytes. The maximum flash size in the PSoC 5LP family is 256 KB, and hence the maximum number of flash arrays possible in PSoC 5LP is 4. Note that flash memory size given in device datasheet refers only to the code region of flash and not configuration region. A 256 KB flash memory implies that code region memory size is 256 KB.
  - K=1 for flash memory ≤ 64 KB,
  - K=2 for 64 KB < flash memory ≤ 128 KB,
  - K=3 for 128 KB < flash memory ≤ 192 KB,
  - K=4 for 192 KB < flash memory ≤ 256 KB.
- Number of rows (N) of flash memory: The value of ‘N’ depends on the flash memory size of the target device. For example, a 256 KB flash memory device has 1024 rows [(256K/256) = 1024 rows]. As mentioned previously, these rows are organized across multiple flash arrays depending on the flash memory size. A 256 KB

flash memory has the 1024 rows organized as four flash arrays of 256 rows each. Also, note that the flash size parameter does not consider the size of configuration bytes. For example, a 64 KB flash size means that the code region capacity is 64 KB. It does not include the configuration bytes because this region cannot be used for code space, only for configuration data.

- Number of bytes per row (L) of flash memory: Each row of flash has 256 code bytes and 32 bytes of ECC. There is an option to use the 32 ECC bytes to store configuration data instead of error correction.

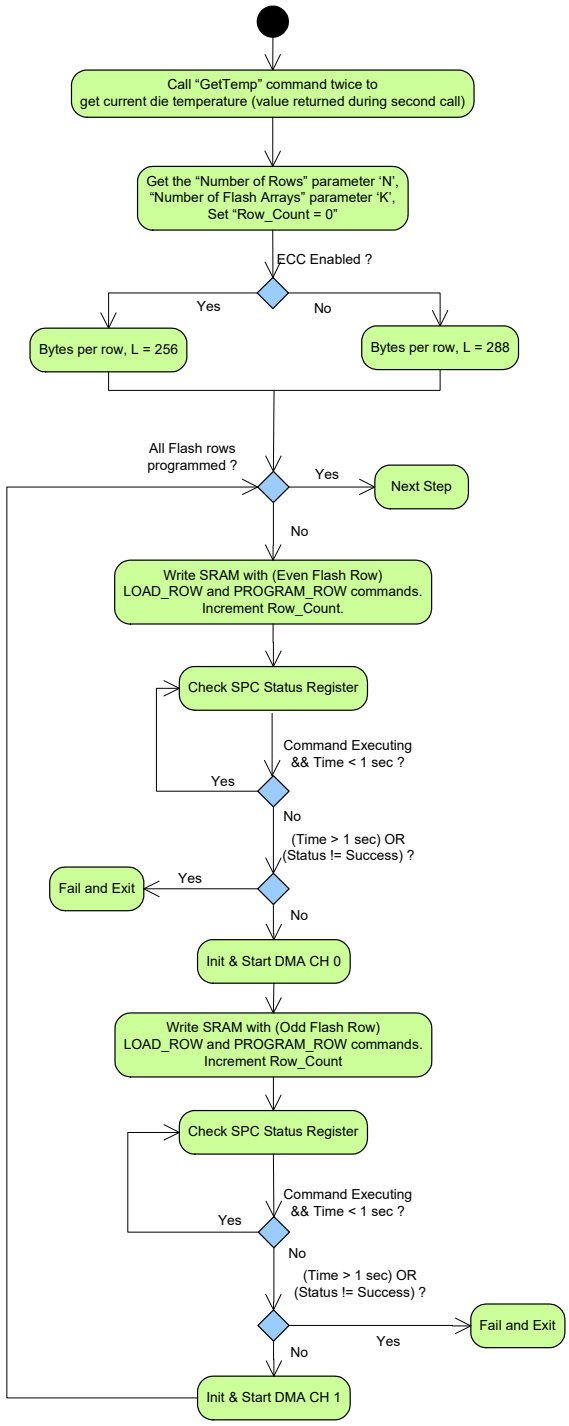
  L = 256 bytes, if ECC is enabled

  L = 288 bytes, if ECC is disabled

Figure 3-12 demonstrates the flash row programming process. Before programming flash, it is necessary to get the on-chip die temperature using the Get Temp command. This temperature value is passed as one of the parameters for the PROGRAM_ROW command. The Get Temp command should be called twice, after device comes out of reset to get an accurate temperature value. LOAD_ROW and PROGRAM_ROW commands are required to program flash. The LOAD_ROW command loads one row of flash data into the row latch and the PROGRAM_ROW command programs the latched data into the specified row of target flash. This process needs to be repeated for every row of flash array and for all flash arrays.

It takes time to load and then program each flash row. Direct Memory Access (DMA) can speed up this process, because the DMA runs in parallel with the flash operations. It can call commands through two DMA channels, such that one channel can load row data and then call PROGRAM_ROW, and the other channel can start loading data for the next row while the previous command is still programming.

Figure 3-12. Program Flash



3.7 Step 7: Verify Flash (Optional)

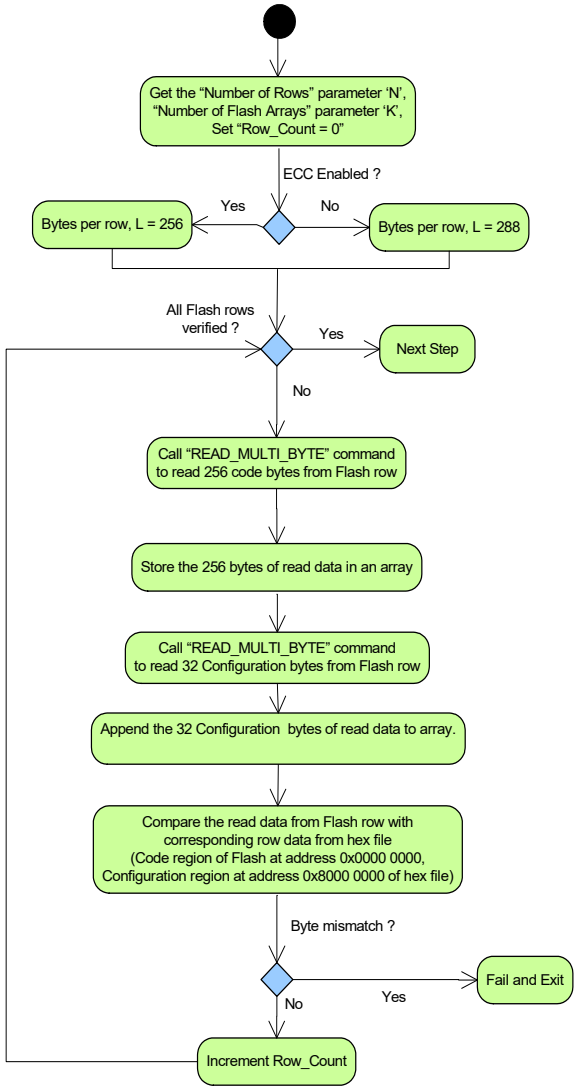
Figure 3-13 demonstrates the flash read process. This optional step allows reading back and verifying data programmed in the Program Flash step. This step should be done before the Checksum Validation step.

The READ_MULTI_BYTE command is used to read out all bytes in flash rows. Each read command can read out a maximum 256 code bytes. If ECC is disable, the 32 bytes of configuration data need to be read out. To read this data, call the READ_MULTI_BYTE command again addressed to point to that configuration data. The number of returned data should be set to 32. This cycle needs to be repeated for all flash rows in all flash arrays.

After reading the data for one flash row, it should be verified with the corresponding flash row data in the hex file. If there is a mismatch in even one of the bytes, the programming process should be stopped and restarted.

Note that in the hex file, the code region in flash row (256 bytes) starts at address 0x0000 0000 of hex file. If ECC is disabled, the 32 configuration bytes for flash row are present starting at address 0x8000 0000 of hex file. In this case the 256 bytes from the code region (0x0000 0000 of hex file) and 32 bytes from the configuration region (0x8000 0000 of hex file) must be concatenated to form a flash row.

Figure 3-13. Verify Flash Sequence

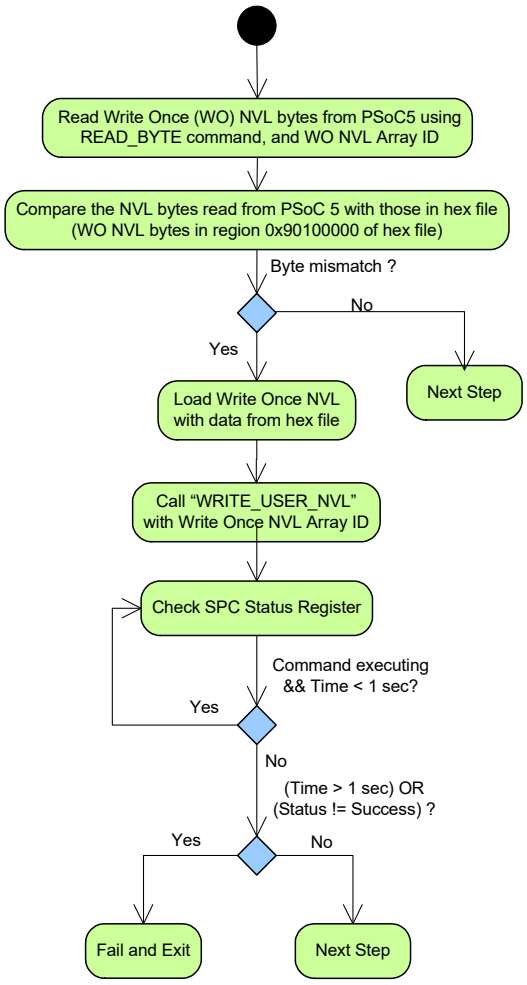


3.8 Step 8: Program WO NVL (Optional)

Figure 3-14 shows the Program Write Once Nonvolatile Latch setup flow. This step writes the 4-byte Write Once (WO) NVL. Note that programming WO NVL with the correct 32-bit key (0x50536F43) makes the device One Time Programmable (OTP). Any other key value does not have any impact on device security. Include this step after understanding its implications and only if it is required for the end application. It is recommended to have this step as an optional selection in your programmer software's graphical user interface in the form of a check box; by default, it should be cleared. See section "Nonvolatile Memory Organization in PSoC 5LP" on page 80 for details on the Device Security feature that is supported by WO NVL.

Figure 3-14 shows the Program Write Once Nonvolatile Latch setup flow. This step writes the 4-byte Write Once (WO) NVL. The data to be written to the NVL is located in address 32'h90100000 of the hex file. The LOAD_BYTE and WRITE_USER_NVL commands are used in this step. The LOAD_BYTE command loads the data one byte at a time to a 4-byte latch. The WRITE_USER_NVL command writes the four bytes of data in the latch to NVL. Therefore, the LOAD_BYTE command needs to be called four times, followed by one WRITE_USER_NVL command. The SPC status register needs to be polled to check when the command finishes the write operation. The WRITE_USER_NVL command should not take longer than 1 second, otherwise an overtime error occurs.

Figure 3-14. Program Write Once NVL

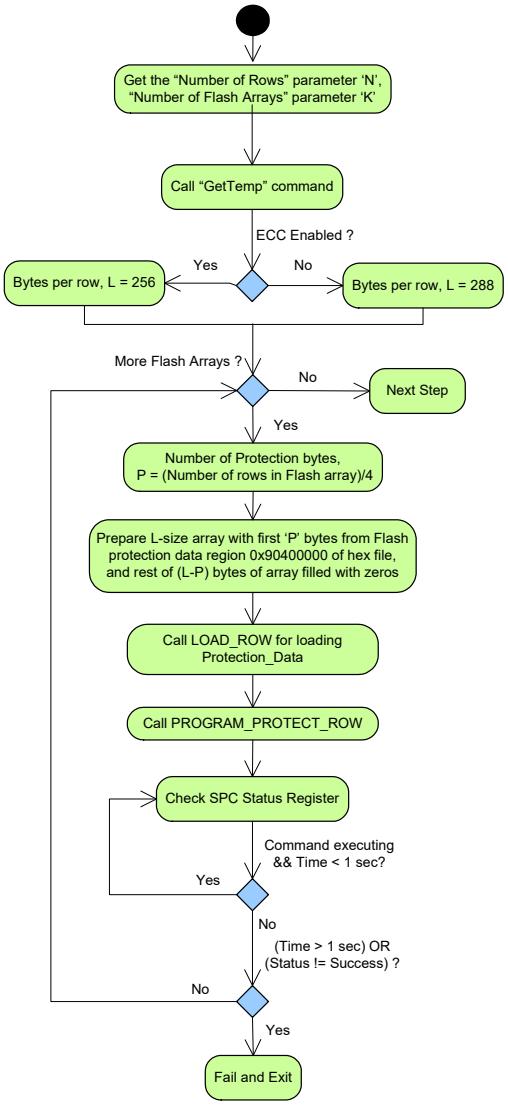


The NVLs in PSoC 5LP have much lesser endurance compared to flash and EEPROM memory. Due to this, the WO NVL is written only if new data needs to be programmed into the latch. This ensures that the latches are programmed only when there is a change in the 4-byte security key in hex file, which in turn maximizes the endurance time.

3.9 Step 9: Program Flash Protection

Figure 3-15 shows the sequence to program the protection rows in flash.

Figure 3-15. Program Flash Protection Sequence



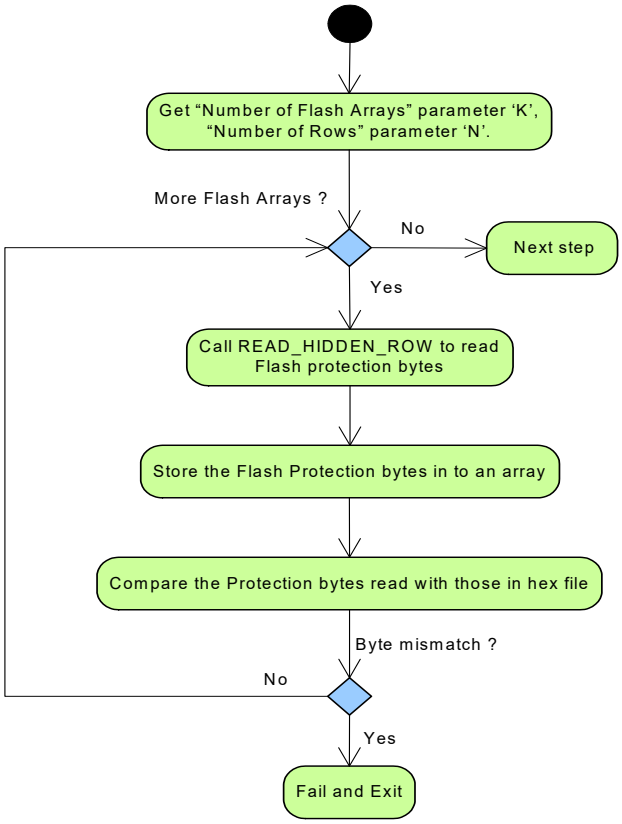
The protection rows start in address 32'h90400000 in the hex file, as shown in "Intel Hex File Format" on page 77. In this step, commands LOAD_ROW and PROGRAM_PROTECT_ROW are called to program flash protection data. Similar to "Step 6: Program Flash" on page 36, the Get Temp command is called initially to get the on-chip die temperature. This temperature is sent as one of the parameters for the PROGRAM_PROTECT_ROW command.

Each protection byte stores protection settings of four flash rows. Each flash array in PSoC 5LP can have a maximum of 256 flash rows and, hence, a maximum of 64 flash protection bytes. The remaining bytes ((L–P) bytes) needed for the LOAD_ROW command are initialized with zeros, as shown in Figure A-3 on page 80. This programming of flash protection data should be done for one flash array at a time and should be repeated for all flash arrays.

3.10 Step 10: Verify Flash Protection (Optional)

Figure 3-16 explains the flash protection data verification procedure. This step is optional, and it allows reading back and verifying the data programmed in the Program Flash Protection step. It is recommended that third party programmers include this step to validate the data programmed.

Figure 3-16. Verify Flash Protection Sequence



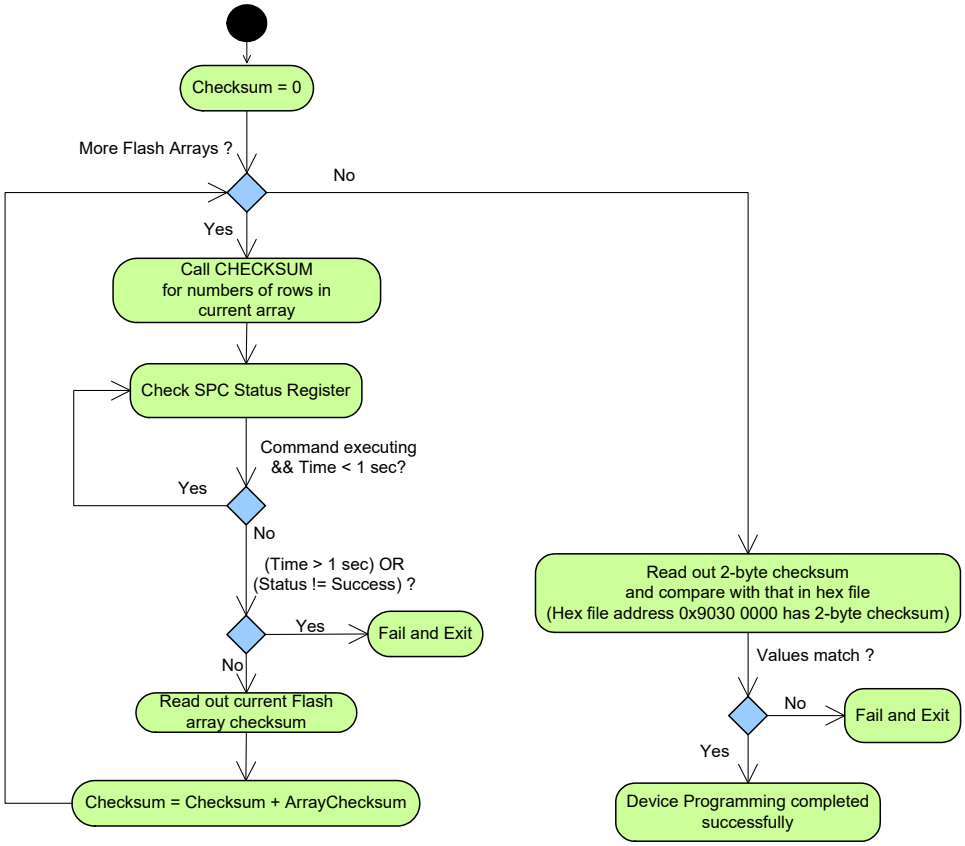
The READ_HIDDEN_ROW command is used to read out all bytes in the Flash Protection row. This command always returns 256 bytes irrespective of the ECC setting and the number of valid flash protection bytes. Each protection byte stores protection settings of four flash rows. Each flash array in PSoC 5LP can have a maximum of 256 flash rows and hence, a maximum of 64 flash protection bytes. The remain-

ing bytes returned by the READ_HIDDEN_ROW command should be ignored during the verification step. The step should be repeated for all the flash arrays.

3.11 Step 11: Checksum Validation

Figure 3-17 demonstrates the checksum validation step. This step validates that the programming operation is successful by doing a checksum on the flash memory data. The computed checksum is only for the code region and the configuration region of flash memory. Flash protection data is not included in the checksum computation. The programmer software needs to locally compute the checksum for all flash rows in all flash arrays, so that it can be compared to the value read out from the target device. The CHECKSUM command is used to compute and return the checksum value, which can be read out through the data register at 32'h40004720. The checksum is a 4-byte value, so four SWD read transfers are required. Only the lower two bytes of this 4-byte value returned from the target device should be taken for comparison as the hex file stores only 2-byte checksum. If the lower 2-byte checksum values mismatch, terminate the programming process. In the hex file, the 2-byte checksum of all flash rows is stored at address 0x9030 0000 of hex file (MSB byte first). This is explained in "Intel Hex File Format" on page 77.

Figure 3-17. Checksum Validation Sequence Block Diagram



3.12 Step 12: Program EEPROM (Optional)

EEPROM nonvolatile memory in PSoC 5LP is used to store constant data such as calibration data and look up table. Some applications may require the EEPROM memory in PSoC 5LP to be initialized as part of the device programming sequence. The programmer software provides a configuration option to the end user to select whether to include the EEPROM initialization as part of programming sequence. The Program EEPROM and Verify EEPROM steps can be included if that option is selected and EEPROM section is available in the hex file.

The number of rows in the EEPROM memory of the PSoC 5LP device can be calculated based on the EEPROM memory size in bytes given in the device datasheet. The

EEPROM is written row wise with the programming data coming from the EEPROM region of the hex file as explained in “Intel Hex File Format” on page 77.

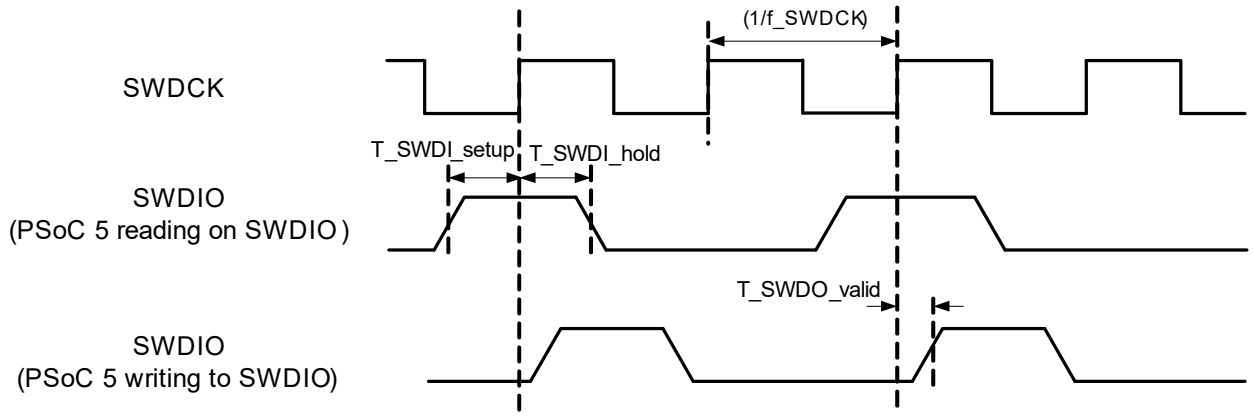
3.13 Step 13: Verify EEPROM (Optional)

This step verifies the integrity of the EEPROM program operation by ensuring the EEPROM data read from the device matches the data in the hex file. This step should be included only if the Program EEPROM step is also included. The EEPROM data is read from the device by directly accessing the EEPROM memory address through the Debug and Access Port (DAP) interface. The step is successful if all the EEPROM bytes read from the device matches with the corresponding hex file data.

4. Programming Specifications

4.1 SWD Interface Timing and Specifications

Figure 4-1. SWD Interface Timing



The external host programmer should do all read or write operations on the SWDIO line on the falling edge of SWDCK, and PSoC 5LP will do the corresponding write or read operations on SWDIO on rising edge of SWDCK.

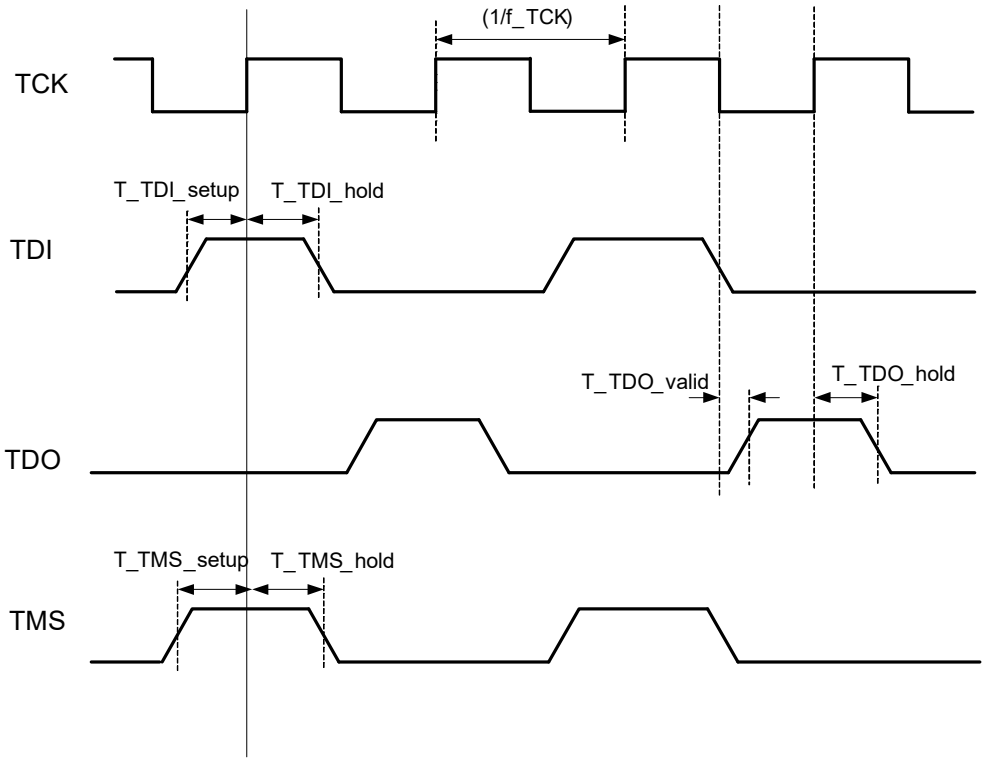
Table 4-1. SWD Interface AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
f_SWDCK	SWDCLK frequency	3.3 V ≤ Vddd ≤ 5 V	–	–	8[a]	MHz
		2.7 V ≤ Vddd < 3.3 V	–	–	7	MHz
		2.7 V ≤ Vddd < 3.3 V, SWD over USBIO pins	–	–	5.5	MHz
T_SWDI_setup	SWDIO input setup before SWDCK high	T = 1/f_SWDCK max	T/4	–	–	
T_SWDI_hold	SWDIO input hold after SWDCK high	T = 1/f_SWDCK max	T/4	–	–	
T_SWDO_valid	SWDCK high to SWDIO output	T = 1/f_SWDCK max	–	–	2T/5	

a. The maximum frequency of 8 MHz is less than device datasheet specification as the CPU clock frequency is configured for a fixed frequency of 24 MHz in the programming algorithm, and the f_SWDCK must be no more than 1/3 CPU clock frequency.

4.2 JTAG Interface Timing and Specifications

Figure 4-2. JTAG Interface AC Timing



The PSoC 5LP reads data on its TMS and TDI lines on the rising edge of TCK. The host should write to the TMS and TDI pins of PSoC 5LP on the falling edge of TCK. PSoC 5LP writes to its TDO line on the falling edge of TCK. The host should read from the TDO line of PSoC 5LP on the rising edge of TCK.

Table 4-2. JTAG Interface AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
f_TCK	TCK frequency	3.3 V ≤ VDDD ≤ 5 V	–	–	8[a]	MHz
		1.71 V ≤ VDDD ≤ 3.3 V	–	–	7	MHz
T_TDI_setup	TDI setup before TCK high	T = 1/f_TCK max	(T/10) – 5	–	–	ns
T_TMS_setup	TMS setup before TCK high	T = 1/f_TCK max	T/4	–	–	–
T_TDI_hold	TDI and TMS hold after TCK high	T = 1/f_TCK max	T/4	–	–	–
T_TDO_valid	TCK low to TDO valid	T = 1/f_TCK max	–	–	2T/5	–
T_TDO_hold	TDO hold after TCK high	T = 1/f_TCK max	T/4	–	–	–

a. The maximum frequency of 8 MHz is less than the device datasheet specification as the CPU clock frequency is configured for fixed frequency of 24 MHz in the programming algorithm, and f_TCK must be no more than 1/3 CPU clock frequency.

4.3 Programming Mode Entry Specifications

Table 4-3. PSoC 5LP Programming Mode Entry Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
T_{RESET}	Reset pulse width (active low)		1	–	–	µs
T_{START_SWDCK}	Maximum time from release of device reset to start of SWDCK signal clocking by host programmer		–	4	–	µs
T_{START_TCK}	Time from release of device reset to start of SWDCK signal clocking by host programmer		–	–	4	µs
$T_{ACQUIRE}$	Initial Port Acquire window		6.1	8	9	µs
T_{BOOT}	Time for device boot process to complete after releasing reset		–	68	–	µs
$T_{TESTMODE}$	Time window to enter Programming mode (Test mode)		395	420	430	µs
$f_{SWDCK\_ACQUIRE}$	SWDCK clock frequency during Port Acquire, Test mode entry	f_SWDCK max is from Table 4-1 on page 43	1.4	–	f_SWDCKmax	MHz
$f_{SWDCK\_BITBANG}$	Average SWDCK clock frequency during Port Acquire, Test mode entry for bit banging SWD interface programmers	f_SWDCK max is from Table 4-1 on page 43. The minimum frequency is assuming no overhead or delay between SWD packets.	0.7	–	f_SWDCKmax	MHz
$f_{TCK\_ACQUIRE}$	TCK clock frequency during Port Acquire, Test mode entry	f_TCK max is from Table 4-2 on page 44	1.4	–	f_TCKmax	MHz
V_{POR}	Vddd, Vdda rising trip voltage		1.64	–	1.68	V

See the PSoC 5LP device datasheet for other specifications such as minimum device operating voltage and nonvolatile memory specifications.

5. SWD and JTAG Vectors for Programming

5.1 Step 1: Enter Programming Mode

This is the first step in the programming procedure; the timing requirements are specified in Table 4-3 on page 45. Depending on the programming interface used, the appropriate method to enter PSoC 5LP's programming mode should be used from the following methods. A separate method is provided for bit banging programmers that need to program PSoC 5LP through SWD interface. Detailed information on all these methods are provided in "Step1: Enter Programming Mode" on page 26.

5.1.1 Method A

```
 /*--- Entering Programming mode through SWD Interface using XRES or Power cycle mode---*/
 /* --------For Programmers with Hardware SWDCK generation capability------------*/
 /* Based on Test mode entry flowchart given in Figure 3-2 on page 27, Table 4-3 on page 45
*/
```

Step i.) Reset device using the XRES pin or the Power Cycle mode.

```
time_elapsed = 0
```

Step ii) Start sending Port Acquire key within time T_{START_SWDCK} of releasing XRES pin high (for XRES mode) or Vddd, Vdda voltages crossing V_{POR} voltage level (for Power Cycle mode). SWDCK frequency during this step should be $f_{SWDCK\_ACQUIRE}$.

```
do
{
/* Write Port Acquire key, Use SWD ADDR = 2'b11*/
    DPACC READBUFF Write [0x7B0C 06DB]

} while (ACK != "OK" AND time_elapsed < T_TESTMODE)//Check port acquire retry time

if (ACK != "OK" OR time_elapsed > T_TESTMODE) then FAIL_EXIT // Exit on timeout
```

Step iii) Send SWD packets for entering test mode. SWDCK frequency during this step should be $f_{SWDCK\_ACQUIRE}$. This step should be completed within time $T_{TESTMODE}$, as given below.

```
APACC ADDR Write [0x4005 0210] // Address of the Test mode key register
APACC DATA Write [0xEA7E 30A9] // Write 32-bit test mode key

/* Exit on timeout or reception of FAULT response which means the device did not enter
Programming mode within time T_TESTMODE. Retry again by doing reset and restarting.*/
if (ACK != "OK" OR time_elapsed > T_TESTMODE) then FAIL_EXIT
```

Step iv) Wait for >= 15 uSec and send JTAG to SWD sequence. After that read SWD ID and compare it with 0x2BA01477. If the ID is not matched with expected then do next step otherwise fail and exit. See, below implementation:

```
time_start = time_current

while (time_elapsed < T15uSec) {time_elapsed = time_current - time_start}
/*
* Send the JTAG to SWD switching sequence on TMS, TCK pins.
* See, section 2.6.2 for details.
*/
exp_idcode = 0x2BA01477
if (DPACC IDCODE Read != exp_idcode) then FAIL_EXIT //Exit on JTAG ID mismatch
else NEXT_STEP /* Entered PSoC 5LP Programming mode */
```

5.1.2 Method B

```
 /* -----------Entering Programming mode through SWD Interface using XRES pin------------*/
 /* --------For Bit Banging Host Programmers ------------*/
/* Based on Test mode entry flowchart given in Figure 3-5 on page 31, Table 4-3 on page 45*/
```

Step i.) Reset device using the XRES pin.

```
time_elapsed = 0
```

Step ii.) Clock SWDCK at frequency of $f_{SWDCK\_ACQUIRE}$ for time T_{BOOT}. SWDIO pin of PSoC 5LP should be driven low by the Host during time T_{BOOT}. Host should start clocking SWDCK within time T_{START_SWDCK} of releasing XRES pin high.

```
time_elapsed = TBOOT
```

Step iii) Start sending Port Acquire key in a loop after time T_{BOOT}. Average SWDCK frequency during this step should be $f_{SWDCK\_BITBANG}$.

```
do
{
/* Write Port Acquire key, Use SWD ADDR = 2'b11*/
    DPACC READBUFF Write [0x7B0C 06DB]

} while (ACK != "OK" AND time_elapsed < TTESTMODE)//Check port acquire retry time

if (ACK != "OK" OR time_elapsed > TTESTMODE) then FAIL_EXIT // Exit on timeout
```

Step iv) Send SWD packets for entering test mode. Average SWDCK frequency during this step should be $f_{SWDCK\_BITBANG}$. This step should be completed within time $T_{TESTMODE}$ as given below.

```
APACC ADDR Write [0x4005 0210] // Address of the Test mode key register
APACC DATA Write [0xEA7E 30A9] // Write 32-bit test mode key

/* Exit on timeout or reception of FAULT response which means the device did not enter
Programming mode within time TTESTMODE. Retry by doing a reset and restarting.*/
if (ACK!= "OK" OR time-lapse > TTESTMODE) then FAIL_EXIT
```

Step v) Wait for >= 15 uSec and send JTAG to the SWD sequence. After that read SWD ID and compare it with 0x2BA01477. If the ID does not match as expected then do the next step; otherwise, fail and exit. See the following implementation:

```
time_start = time_current
```

```
while (time_elapsed < T15uSec) {time_elapsed = time_current - time_start}
/*
* Send the JTAG to SWD switching sequence on TMS, TCK pins.
* See, section 2.6.2 for details.
*/
exp_idcode = 0x2BA01477
if (DPACC IDCODE Read != exp_idcode) then FAIL_EXIT //Exit on JTAG ID mismatch
else NEXT_STEP /* Entered PSoC 5LP Programming mode */
```

5.1.3 Method C

```
       /* Entering Programming mode through JTAG Interface */
      /* Based on Test mode entry flowchart in Figure 3-7 on page 32, Table 4-3 on page 45 */

       a.) Move JTAG FSM in Reset state.
       b.) TC's IR = APACC //Set instruction register of Test Controller
       c.) DAP's IR = BYPASS //Set instruction register of Cortex-M3 DAP
       d.) APACC ADDR Write [0x4005 0214] //Set address of TC_PM_CTRL register
           APACC DATA Write [0x0000 0040] //Set the "gen_tcr" bit to generate reset
       e.) Wait for 1 ms. //This delay will cover Reset mode.
       f.) Move JTAG's FSM into Reset mode.g.) TC's IR = BYPASS //Set instruction register of
Test Controller
       h.) DAP's IR = ID CODE //Set instruction register of Cortex-M3 DAP
       i.) Shift out 32 bit ID from PSoC 5LP DAP.
       g.) if (DAP's != 0x4BA00477) then FAIL_EXIT
       k.) TC's IR = BYPASS
       l.) DAP's IR = APACC
```

5.2 Step 2: Configure Target Device

```
DPACC DP CTRLSTAT Write [0x50000000] //Configure DP Control & Status Register

DPACC DP SELECT Write [0x00000000] // Clear DP Select Register

APACC AP CTRLSTAT Write [0x22000002] // Set 32-bit transfer mode of DAP

APACC ADDR Write [0xE000 EDF0]
APACC DATA Write [0xA05F 0001]       //Activate Debug

APACC ADDR Write [0xE000 EDFC]
APACC DATA Write [0x0000 0001]        //Set VC_CORERESET, to halt CPU on reset release

APACC ADDR Write [0x4008 000C]
APACC DATA Write [0x0000 0002]       // Release Cortex-M3 CPU Reset

APACC ADDR Write [0x4000 43A0]
APACC DATA Write [0x0000 00BF]       // Enable individual sub-system of chip

APACC ADDR Write [0x4000 4200]
APACC DATA Write [0x0000 0002]       // IMO set to 24 MHz
```

5.3 Step 3: Verify JTAG ID

```
/* Compare the 4-byte Device ID in the hex file (exp_idcode) at address 0x90500002 of hex
file with the Target Device Jtag ID. Abort programming operation if Device ID's mismatch
```

```
4-byte Device ID in hex file is in Big-endian format. See "Intel Hex File Format" for details
*/
int32 exp_idcode, dummy, JtagID

APACC ADDR Write [0x4008 001C] //Set address of the Jtag ID register
dummy = APACC DATA Read   //Dummy Read - returns incorrect value, Next Read gives correct
Jtag ID value
JtagID = APACC DATA Read
if (JtagID != exp_idcode) then FAIL_EXIT // Exit on Jtag ID mismatch
```

5.4 Step 4: Erase All (Entire Flash Memory)

```
APACC ADDR Write [0x4000 4720]   // SPC data register address
APACC DATA Write [0x0000 00B6]   // First initiation key

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00DC]  // Second key:00DC(0xD3 + 0x09); 0x09 is Erase All opcode

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0009]  // ERASE_ALL opcode

/*Read SPC status register to check the status of SPC command. If "Command Success" status is
not received within 1 second, then exit the programming operation */
APACC ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value
do
{
     StatusReg = APACC DATA Read // Save status register value to a local variable
     StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
} while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec);

if (time_elapsed > 1 sec) then FAIL_EXIT
```

5.5 Step 5: Program Device Configuration Nonvolatile Latch

The data for this section is located in address 0x90000000 of the hex file.

```
/* The NV Latches have a lesser endurance, and hence should be written only when the data has
changed. First read the Device Configuration NVL bytes from target device and dump in to an
array, Data_Array. Compare the bytes read from the silicon to the NVL bytes in hex file at
address 0x90000000. Perform write operation only if there is a byte mismatch */

byte ByteRead = 0  //Variable to track number of bytes that have been read
byte Data_Array[4] //4-byte array to store the NVL data read from device

while (ByteRead < 0x0000 0004)
{
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 00B6] // First initiation key

    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 00D6] // Second key:00D6(0xD3 + 0x03); 0x03 is Read Byte opcode

    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 0003] //0x03 is Read Byte opcode

    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 0080] // Device Configuration NVL array ID

    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [ByteRead]    //Byte number of User NVL to be read

    // Poll status register bit till data is ready
    APACC ADDR Write [0x4000 4722]
   byte dummy = APACC DATA Read   //Dummy SWD Read, Next read gives correct status

    byte StatusReg //To store SPC_SR status register value
    time_elapsed = 0
    do
    {
    StatusReg = (byte) APACC DATA Read  // Save status register value
    StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte

    } while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

    if (time_elapsed > 1 sec) then FAIL_EXIT // Check if command execution time < 1 second

    APACC ADDR Write [0x4000 4720]
    byte dummy = APACC DATA Read           //Dummy SWD read, first byte read is garbage
    Data_Array[ByteRead] = (byte) APACC DATA Read /* Store the data read from device in to
                                                    array */
    ByteRead = ByteRead + 1

    //Check if SPC Idle bit is high
    time_elapsed = 0
    APACC ADDR Write [0x4000 4722]// SPC status register address
    byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

    do
    {
        StatusReg = (byte) APACC DATA Read// Save status register value
```

```
        StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte

    } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

    if (time_elapsed > 1 sec) then FAIL_EXIT
}

/*Compare the NVL bytes read from target device with those in hex file. Set "WriteFlag" if
there is change in NVL data even in one bit position. If "ECC Enable" bit in NVL (bit 3 of
byte 4 (last NVL byte)) has been changed from its previous value, "eccEnableChanged" flag is
set. If this flag has been set, a port acquire sequence (repeat of Step 1, Step 2) is done
again after completing NVL write operation. This is required for the new ECC settings to take
effect during subsequent Flash Programming, Read operations.*/

ByteRead = 0 /* Count of number of bytes read for comparison */

/*This flag determines whether the NV latch will be programmed or not. Flag is set when new
data needs to be written; otherwise reset */
byte WriteFlag=0
/*This flag, if set, indicates "ECC Enable"bit in User NVL in hex file
is different from what is already programmed in target device */
byte eccEnableChanged = 0

while (ByteRead < 0x04)
{
    // Replace XX in below line with data at address (0x90000000 + ByteRead) of .hex file
    if(Data_Array[ByteRead] !=  XX)
    {
        WriteFlag=1 //Set the flag if NV latch needs to be programmed

        /* Set the "eccEnableChanged" flag if "ECC_Enable" bit(bit 3 of NVL
        byte-4 is ECC_Enable bit) in User NVL is different between hex file and the
        target device. */
        if (ByteRead == 0x03)
        {
             /* Replace XX in below line with data at address (0x90000000 + ByteRead) of
               .hex file */
             eccEnableChanged = ((( XX ^ Data_Array [3]) & 0x08) == 0x08);
        }
    }
    ByteRead = ByteRead + 1
}

//Check if the WriteFlag is set before programming User NVL

if (WriteFlag == 1)
{
   /* When writing the NV Latches, ensure that the GPIO/XRES pin P1[2] is configured to pull-
up drive mode when writing '1' to XRES NVL bit. */
/* Replace hexNvlByte2 in the following line with data at address 0x90000002 of the hex file.
If the XRESMEN bit (msb) is set in that byte, check if the chip is already in resistive pull-
up drive mode by checking the NVL data read from the device (Data_Array[0]). If it is not,
configure the chip in resistive pull-up drive mode before performing a NVL write. */
    pullupEnable = ((hexNvlByte2 & 0x80) == 0x80) && ((Data_Array[0] & 0x0C) != 0x08)
   if (pullupEnable == 1)
```

```
{
   APACC ADDR Write [0x4000 500A]
   Long dummy = APACC DATA READ
   Long PinState = APACC DATA READ //Read current state of P1[2]
   PinState = (PinState & 0x00F00000) | 0x00050000 //Set Pull-up Drive mode and High Data
   APACC ADDR Write [0x4000 500A]
   APACC DATA Write [PinState] //Apply new state for P1[2]
}

byte AddrCount = 0
while (AddrCount < 4)
{
     APACC ADDR Write [0x4000 4720]// Write to command data register
     APACC DATA Write [0x0000 00B6]// First initiation key

     APACC ADDR Write [0x4000 4720]
     APACC DATA Write [0x0000 00D3] // Second initiation key: 0xD3 + 0x00

     APACC ADDR Write [0x4000 4720]
     APACC DATA Write [0x0000 0000]// LOAD_BYTE opcode

     APACC ADDR Write [0x4000 4720]
     APACC DATA Write [0x0000 0080]// Array ID of "Device Config NVL"

     APACC ADDR Write [0x4000 4720]
     APACC DATA Write [AddrCount]// Current address: 0 - 3

     APACC ADDR Write [0x4000 4720]
   APACC DATA Write [0x0000 00XX]     // Replace XX with data located in
                                          // (0x90000000 + AddrCount) of .hex file

    time_elapsed = 0
    APACC ADDR Write [0x4000 4722]
    byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
    do             // Poll status register
    {
    StatusReg = (byte) APACC DATA Read // Save status register value
    StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
    } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

    if (time_elapsed > 1 sec) then FAIL_EXIT // Check if command execution time < 1 second


     AddrCount = AddrCount + 1 //Increment to load the next NVL byte
}

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00B6] // Call WRITE_USER_NVL command

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00D9]// Second initiation key: 0xD3 + 0x06

APACC ADDR Write [0x4000 4720]
```

```
    APACC DATA Write [0x0000 0006]// WRITE_USER_NVL opcode

    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 0080]// Array ID: Device Config NVL

    time_elapsed = 0
    APACC ADDR Write [0x4000 4722]
    byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
    do                // Poll status register
    {
    StatusReg = (byte) APACC DATA Read // Save status register value
    StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
    } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

    if (time_elapsed > 1 sec) then FAIL_EXIT// Check if command execution time < 1 second

    /* If "ECC Enable" bit changed from its previous value, do a Test mode entry again by
    repeating all of "Step 1: Enter Programming mode ", "Step 2: Configure Target Device ".
    This is necessary for the new ECC settings to take effect which in turn will be used in
    subsequent Flash Program, Read operations. */

   if (eccEnableChanged)
    {
        /* Repeat "Step 1: Enter Programming mode " */
        /* Repeat "Step 2: Configure Target Device" */
    }

} /* End of "WriteFlag ==1" loop */
```

5.6 Step 6: Program Flash

The data for this section is located in address 0x0000 0000 and 0x8000 0000 of the hex file. This step requires three parameters: K - number of flash arrays, N - total number of flash rows, and L - number of bytes in row. K and N are derived from the total flash memory size of the device, and the L value is fixed to 256 or 288, depending on ECC option. See the respective device datasheet for flash memory size of each device.

```
    /*Get the die temperature and store it in "Sign, Magnitude" bytes.
    Note that when this command is called the first time after device comes out of reset
    (which is in this step), it should be called twice. This is because the "Get Temp" command
    returns accurate value only from the second time it is called after device comes out of
    reset.*/

    /*********************************************************************/

    //Start of "Get_Temp" routine to get Die temperature

    byte Temp_Sign, Temp_Magnitude; //Die temperature - used in the PROGRAM_ROW instruction

    byte loop = 0; //This variable is used to do the Get_Temp routine twice.
   byte StatusReg //To store SPC_SR status register value

    while (loop <= 1)
    {

        APACC ADDR Write [0x4000 4720]
        APACC DATA Write [0x0000 00B6]  //SPC_KEY1
```

```
        APACC ADDR Write [0x4000 4720]
        APACC DATA Write [0x0000 00E1]  //SPC_KEY2 + SPC_GET_TEMP (0xD3+0x0E)

        APACC ADDR Write [0x4000 4720]
        APACC DATA Write [0x0000 000E]  //SPC_ GET_TEMP opcode

        APACC ADDR Write [0x4000 4720]
        APACC DATA Write [0x0000 0003] //Number of samples, valid values [1..5]

        //Wait until Temperature data is ready
        APACC ADDR Write [0x4000 4722]
        byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
        time_elapsed = 0
        do
        {
        StatusReg = (byte) APACC DATA Read // Save status register value
        StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
        } while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

        if (time_elapsed > 1 sec) then FAIL_EXIT

        APACC ADDR Write [0x4000 4720]
        byte dummy = APACC DATA Read // Dummy SWD read
        Temp_Sign =(byte) APACC DATA Read // First byte read is sign of temperature
        Temp_Magnitude =(byte) APACC DATA Read // Second byte read is magnitude of temperature

        //Wait for IDLE - just in case. Must be in idle state once data byte is read.
        APACC ADDR Write [0x4000 4722]// Poll status register
        byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
        time_elapsed = 0
        do
        {
        StatusReg = (byte) APACC DATA Read // Save status register value
        StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
        } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

        if (time_elapsed > 1 sec) then FAIL_EXIT

        loop++;
    }
/* End of "Get_Temp" routine to get Die temperature. The temperature value received
During second time of above loop is stored in Temp_Magnitude, Temp_Sign, and used in below
programming step */
/*********************************************************************/

//Calculating total number of rows 'N'
int32 N = (Total_Flash_Code_size)/256; //Each row has 256 code bytes, "Total_Flash_size" is
                                       //in bytes

//Calculating total number of Flash arrays 'K'
if (N % 256 == 0)
{
   byte K= (byte) N/256; //If rows are exact multiple of 256,quotient of (N/256) gives 'K'
}
else
{
```

```
   byte K= (byte)((N/256) + 1); //If rows are not exact multiple of 256,increment quotient of
                                // (N/256) by one for 'K's
}
int16 RowsPerArray; //Variable that hold the number of data rows in current Flash array

/* Setting AP Control/Status register configuration register for four-byte access to SRAM.
LSB 3bits: 2 - "4 byte", 1 - "2 byte",  0 - "1 byte" mode - already set during chip initial-
ization */
APACC AP CTRLSTAT WRITE [0x22000002]

// Program all Flash Arrays
for (byte ArrayCount = 0; ArrayCount < K; ArrayCount++)
{
  // Find number of rows in current array
   if (ArrayCount == (K-1))
   {
      RowsPerArray = N - (ArrayCount*256); //Last array may have less than 256 rows
   }
   else
   {
      RowsPerArray = 256; //Except last flash array, rest of them have 256 rows
   }

   int16 RowCount = 0;

   //Program Rows
   while (Row_Count < RowsPerArray)
   {
     //-------------Programming EVEN ROW ----------------------

     //"B6" - SPC_KEY1, "D5" - SPC_KEY2, "02" - LOAD_ROW opcode,
     // ArrayCount - Flash ArrayID
     APACC ADDR Write [0x2000 0000]// SRAM address- 32'h20000000
     APACC DATA Write [(0x0002 D5B6) | (ArrayCount << 24)] // 4 byte data

     int16 Byte_Count = 0

     /*Send Row data to SRAM from HEX file. Each row needs 288 bytes (256 Code bytes + 32
       Configuration bytes) for programming. The 256 code bytes for row are present
       starting at address 0x00000000 of hex file. The 32 Configuration bytes are present
       starting at address 0x80000000 of hex file. Thus a single row data is formed by
     concatenating these 256 code bytes and 32 configuration bytes to form a 288-byte row
       data. See "Intel Hex File Format" for more details. */
     while (Byte_Count < L) // Define L according to ECC settings
     {
        APACC ADDR Write [(0x20000000) + Byte_Count + 0x4]
        /* 4-bytes (d3d2d1d0) are from hex file starting at address (address of d0):
        i.) if Byte_Count < 256: (0x00000000 + (ArrayCount*65536)
  +(Row_Count * 256) + Byte_Count)
        ii.) if 256 <= Byte_Count < 288: Address of do = (0x80000000 +
  (ArrayCount*8192) + (Row_Count*32) + (Byte_Count - 256))
       The ii) address will be needed only if ECC is disabled.
        ECC data is 32 bytes per row.*/
        APACC DATA Write [d3d2d1d0] // Write 4 bytes at a time, 4-bytes are from hex file
        Byte_Count = Byte_Count + 4
     }

     //"00","00","00" - 3 NOPs for short delay, "B6" - SPC_KEY1
```

```
APACC ADDR Write [(0x20000000)+ (L - 1) + 0x05]
APACC DATA Write [0xB600 0000]

//"DA" - SPC_KEY1+SPC_PRG_ROW, "07" -SPC_PRG_ROW, " ArrayCount" -Flash Array ID,

//"00" - High Byte of RowCount, 'Temp_Sign' - temperature Sign, 'Temp_Magnitude' - temp
Magnitude
APACC ADDR Write [0x20000000 + (L - 1) + 0x09]
APACC DATA Write [(0x000007DA) | (ArrayCount << 16)]

APACC ADDR Write [0x20000000 + (L - 1) + 0xD]
APACC DATA Write [(0x00 << 24) | (Temp_Magnitude << 16) | (Temp_Sign << 8) |
                  (RowCount & 0xFF)] //Low byte of row number
                                     //and Die's temperature ('Temp_Sign', 'Temp_Magnitude')

//DMA operations

APACC ADDR Write [0x4000 7018]// PHUB_CH0_STATUS Register
APACC DATA Write [0x0000 0000]// Disable chain event, use TDMEM1_ORIG_TD0

APACC ADDR Write [0x4000 7010]// PHUB_CH0_BASIC_CFG register
APACC DATA Write [0x0000 0021] // Enable DMA CH 0

APACC ADDR Write [0x4000 7600]// PHUB_CFGMEM0_CFG0 register
APACC DATA Write [0x0000 0080]// DMA request is required for each burst

APACC ADDR Write [0x4000 7604]// PHUB_CFGMEM0_CFG1 register
APACC DATA Write [0x4000 2000] // Sets upper 16-bit address of destination/source

APACC ADDR Write [0x4000 7800]//PHUB_TDMEM0_ORIG_TD0 register
APACC DATA Write [(0x01FF 0000) + L + 15]  // Set TD transfer counts

APACC ADDR Write [0x4000 7804] // PHUB_TDMEM0_ORIG_TD1 register
APACC DATA Write [0x4720 0000] // Set lower 16-bit address of the destination/source

//Wait until SPC has done previous request
APACC ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value

do
{
    StatusReg = APACC DATA Read
    StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
} while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x4000 7014]// PHUB_CH0_ACTION register
APACC DATA Write [0x0000 0001]// This creates a direct DMA request for channel '0'

// DMA will transfer data from SRAM, and call LOAD_ROW and then WRITE_ROW
//When the DMA is transferring data using Channel '0', configure Channel '1'
//to speed up programming time

//-------------Programming ODD ROW ----------------------
```

```
    Row_Count = Row_Count + 1 // Increment row count and repeat process for the next row

    //"B6"-SPC_KEY1, "D5"-SPC_KEY2, "02"-LOAD_ROW opcode, "ArrayCount"-ArrayID
    APACC ADDR Write [0x2000 0200]// SRAM address 32'h20000200
    APACC DATA Write [0x0002 D5B6 | (ArrayCount << 24)]// 4-byte data as commented above

    /*Send Row data to SRAM from HEX file. Each row needs 288 bytes (256 Code bytes + 32
    Configuration bytes) for programming. The 256 code bytes for row are present
    starting at address 0x00000000 of hex file. The 32 Configuration bytes are present
    starting at address 0x80000000 of hex file. Thus a single row data is formed by
    concatenating these 256 code bytes and 32 configuration bytes to form a 288-byte row
    data. See "Intel Hex File Format" on page 77 for more details. */

    Byte_Count = 0
    while (Byte_Count < L) // Define L according to ECC settings
    {
        APACC ADDR Write [0x20000000 + Byte_Count + 0x204]
       /* 4-bytes (d3d2d1d0) are from hex file starting at address (address of d0):
       i.) if Byte_Count < 256: (0x00000000 + (ArrayCount*65536)
  +(Row_Count * 256) + Byte_Count)
        ii.) if 256 <= Byte_Count < 288: Address of do = (0x80000000 +
  (ArrayCount*8192) + (Row_Count*32) + (Byte_Count - 256))
       The ii) address will be needed only if ECC is disabled.
       ECC data is 32 bytes per row.*/
        APACC DATA Write [d3d2d1d0] // Write 4 bytes at a time, 4-bytes are from
                                    //hex file
        Byte_Count = Byte_Count + 4
    }

    //"00","00","00" - 3 NOPs for short delay, "B6" - SPC_KEY1
    APACC ADDR Write [0x20000000 + (L - 1) + 0x205]
    APACC DATA Write [0xB600 0000]

    //"DA" - SPC_KEY1+SPC_PRG_ROW, "07" -SPC_PRG_ROW, " ArrayCount" -Flash Array ID,

    //"00" - High Byte of RowCount, 'Temp_Sign' - temperature Sign, 'Temp_Magnitude' - temp
Magnitude
    APACC ADDR Write [0x20000000 + (L - 1) + 0x209]
    APACC DATA Write [(0x000007DA) | (ArrayCount << 16)]

    APACC ADDR Write [0x20000000 + (L - 1) + 0x20D]
    APACC DATA Write [(0x00 << 24) | (Temp_Magnitude << 16) | (Temp_Sign << 8) |
                      (RowCount & 0xFF)] //Low byte of row number
                                         //and Die's temperature ('Temp_Sign', 'Temp_Magnitude')

   //DMA operations
    APACC ADDR Write [0x4000 7028]// PHUB_CH1_STATUS Register
    APACC DATA Write [0x0000 0100] // Disable chain event, use TDMEM1_ORIG_TD1

    APACC ADDR Write [0x4000 7020]// PHUB_CH1_BASIC_CFG register
    APACC DATA Write [0x0000 0021]// Enable DMA CH 0

    APACC ADDR Write [0x4000 7608]// PHUB_CFGMEM1_CFG0 register
    APACC DATA Write [0x0000 0080]// DMA request is required for each burst

    APACC ADDR Write [0x4000 760C]// PHUB_CFGMEM1_CFG1 register
    APACC DATA Write [0x4000 2000]// Sets upper 16-bit address of
```

```
                                    //destination/source

      APACC ADDR Write [0x4000 7808]
      APACC DATA Write [(0x01FF 0000) + L + 15]

      APACC ADDR Write [0x4000 780C] // PHUB_TDMEM1_ORIG_TD1 register
      APACC DATA Write [0x4720 0200] // Set lower 16-bit address of the
                                    //destination/source

      //Wait until SPC has done previous request
      APACC ADDR Write [0x4000 4722]// SPC status register address
      dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

      time_elapsed = 0
      int32 StatusReg //To store SPC_SR status register value
      do  // Poll status register
      {
           StatusReg = APACC DATA Read
           StatusReg = (StatusReg >> 16) & 0xFF // Extract status code which is in 3rd byte
      } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

      if (time_elapsed > 1 sec) then FAIL_EXIT

      APACC ADDR Write [0x4000 7024]  // PHUB_CH1_ACTION register
      APACC DATA Write [0x0000 0001] //Creates a direct DMA request to Channel '1'.
                                    // DMA will transfer data from SRAM, and call
                                    //LOAD_ROW and then WRITE_ROW

      Row_Count = Row_Count + 1

   }    //Repeat for all rows of one Flash array

} //Repeat for all Flash arrays

//Make sure that last SPC request is completed
APACC ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value
do// Poll status register
{
   StatusReg = APACC DATA Read
   StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
} while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT
```

5.7 Step 7: Verify Flash (Optional)

This step requires three parameters: K - number of flash arrays, N - total number of flash rows, L - number of bytes in row (L = 288). K and N are derived from the total flash memory size of the device, and the L value is fixed to 288. See the respective device datasheet for flash memory size of each device.

```
//Calculating total number of rows 'N'
int32 N = (Total_Flash_Code_size)/256; //Each row has 256 bytes, "Total_Flash_Code_size" is
                                       //in bytes
//Calculating total number of Flash arrays 'K'
if (N % 256 == 0)
{
   byte K= (byte) N/256; //If rows are exact multiple of 256,quotient of (N/256) gives 'K'
}
else
{
   byte K= (byte)((N/256) + 1); //If rows are not exact multiple of 256,increment quotient of
                                // (N/256) by one for 'K'
}

int16 RowsPerArray; //Variable that hold the number of data rows in current Flash array
int16 byte_index = 0 //Variable to keep track of number of bytes read in a Flash row
byte Data_Array[L] //Array of size 'L' bytes to store one row of data read from device
int32 address

//Read Flash data bytes for all Arrays
for (byte ArrayCount = 0; ArrayCount < K; ArrayCount++)
{
   // Find number of rows in current array
   if (ArrayCount == (K-1))
   {
      RowsPerArray = N - (ArrayCount*256); //Last array may have less than 256 rows
   }
   else
   {
      RowsPerArray = 256; //Except last flash array, rest of them have 256 rows
   }

   int16 RowCount = 0;

   // Iterate through all rows of flash
   while (RowCount < RowsPerArray)
   {
      int32 address = RowCount * 256   //Starting address of Flash row

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 00B6]//First initiation key

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 00D7]//0xD7= (0xD3 + READ_MULTI_BYTE opcode)

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 0004] // READ_MULTI_BYTE opcode

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [ArrayCount]// Array ID

      APACC ADDR Write [0x4000 4720]
```

```
APACC DATA Write [(address >> 16) & 0xFF]//MSB byte2 of 3-byte address

APACC ADDR Write [0x4000 4720]
APACC DATA Write [(address >> 8) & 0xFF]//Byte1 of 3-byte address

APACC ADDR Write [0x4000 4720]
APACC DATA Write [(address >> 0) & 0xFF]//LSB Byte0 of 3-byte address

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00FF]// Number of bytes to be read minus one

//Wait until Data is ready
ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value

do
{
   StatusReg = APACC DATA Read
   StatusReg = (StatusReg >> 16) & 0xFF  //Extract status code which is in 3rd byte
}    while ((StatusReg != [0x0000 0001]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x4000 4720]
dummyByte = APACC DATA Read // Dummy SWD read

// Read 256 bytes of row data in to Data_Array
int16 ByteRead = 0, byte_index = 0
while (ByteRead <= 0x0000 00FF)
{
   Data_Array[byte_index] = APACC DATA Read // Save Flash data
   ByteRead = ByteRead + 1
   byte_index = byte_index + 1
}

// If ECC is disabled, row size is 288
If (L = 288)
{
  // Configuration (ECC) data is addressed as following. MSB bit is '1' to
  //specify that addressed memory is ECC (config) memory
  address = (RowCount * 32) | 0x00800000;

  // Call READ_MULTI_BYTE to read configuration data in ECC memory space

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 00B6]  //First initiation key

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 00D7] //0xD7= (0xD3 + READ_MULTI_BYTE opcode)

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 0004] // READ_MULTI_BYTE opcode

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [ArrayCount] // Array ID
```

```
      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [address >> 16) & 0xFF] //MSB Byte 2 of 3-byte address;

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [address >> 8) & 0xFF] //Byte 1 of 3-byte address

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [address >> 0) & 0xFF] //LSB Byte 0 of 3-byte address

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 001F]   //Each row has 32 ECC bytes to be read

      //Wait until Data is ready
      ADDR Write [0x4000 4722]// SPC status register address
      dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

      time_elapsed = 0
      int32 StatusReg //To store SPC_SR status register value

      do
      {
          StatusReg = APACC DATA Read
          StatusReg = (StatusReg >> 16) & 0xFF //Extract statuscode which is in 3rdbyte
      } while ((StatusReg != [0x0000 0001]) AND time_elapsed < 1 sec)

      if (time_elapsed > 1 sec) then FAIL_EXIT

      APACC ADDR Write [0x4000 4720]
      dummyByte = APACC DATA Read // Dummy SWD read

      ByteRead = 0
      while (ByteRead <= 0x000 0001F)
      {
           Data_Array[byte_index] = APACC DATA Read// Save configuration data
           ByteRead = ByteRead + 1
           byte_index = byte_index + 1
      }
    }
    /* Now, the array Data_Array contains a row of Flash data.
    Compare it with data in hex file to check if the correct data has been programmed
    in to Flash row. If there is data mismatch, Abort the Programming operation and
    retry again. Repeat for all Flash rows in all Flash arrays. */

    RowCount = RowCount + 1; // Next Flash row

  } //Repeat for all rows of Flash array

}//Repeat for all Flash arrays
```

5.8 Step 8: Program Write Once Nonvolatile Latch (Optional)

```
/* The NV Latches have a lesser endurance, and hence should be written only when the data has
changed. First read the Write Once NVL bytes from target device, and dump in to an array
(Data_Array). Compare the bytes read from the silicon to the NVL bytes in hex file at address
0x90100000. Perform write operation only if there is atleast one byte mismatch */

byte ByteRead = 0  //Variable to track number of bytes that are read
byte Data_Array[4] //4-byte array to store the NVL data read from device

while (ByteRead < 0x0000 0004)
{
  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 00B6] // First initiation key

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 00D6] //Second key:00D6(0xD3+0x03);0x03 is ReadByte opcode

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 0003] //0x03 is Read Byte opcode

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [0x0000 00F8] //Write Once NVL array ID

  APACC ADDR Write [0x4000 4720]
  APACC DATA Write [ByteRead]    //Byte number of Write Once NVL to be read

  // Poll status register bit till data is ready
  ADDR Write [0x4000 4722]// SPC status register address
  dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

  time_elapsed = 0
  int32 StatusReg //To store SPC_SR status register value

  do
  {
     StatusReg = APACC DATA Read
     StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
  } while ((StatusReg != [0x0000 0001]) AND time_elapsed < 1 sec);

  if (time_elapsed > 1 sec) then FAIL_EXIT //Check if command execution time < 1 sec

  APACC ADDR Write [0x4000 4720]
  dummyByte = APACC DATA Read   //Dummy SWD read, first byte read is garbage
  Data_Array[ByteRead] = APACC DATA Read //Store the data read from device in to
                                         //array
  ByteRead = ByteRead + 1

  //Check if SPC Idle bit is high. Must be in idle state once data byte is read.
  ADDR Write [0x4000 4722]// SPC status register address
   dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

   time_elapsed = 0
   int32 StatusReg //To store SPC_SR status register value
   do
   {
      StatusReg = APACC DATA Read   //Save status register value to a local variable
     StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
```

```
  } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

  if (time_elapsed > 1 sec) then FAIL_EXIT

}

//Compare the NVL bytes read from target device with those in hex file at address
//0x90100000

ByteRead = 0
byte WriteFlag=0  /* This flag determines whether the NV latch will be programmed or not.
          Flag is set when new data needs to be written; otherwise reset */

while (ByteRead < 0x00000004)
{
   // Replace XX in below line with data at address (0x90100000 + ByteRead) of .hex file
   if(Data_Array[ByteRead] !=  XX)
   {
       WriteFlag=1 //Set the flag if NV latch needs to be programmed
   }
   ByteRead = ByteRead + 1
}

//Check if the WriteFlag is set before programming Write Once NVL
if (WriteFlag == 1)
{
   byte AddrCount = 0
   while (AddrCount < 4)
   {
      APACC ADDR Write [0x4000 4720]// Write to command data register
      APACC DATA Write [0x0000 00B6]// First initiation key

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 00D3] // Second initiation key: 0xD3 + 0x00

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 0000]// LOAD_BYTE opcode

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 00F8]// Array ID of "Write Once NVL"

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [AddrCount]// Byte index in "Write Once NVL"

      APACC ADDR Write [0x4000 4720]
      APACC DATA Write [0x0000 00XX]        // Replace XX with data located in
                                           // (0x90100000 + AddrCount) of .hex file

      // Poll status register
      ADDR Write [0x4000 4722]// SPC status register address
      dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
      time_elapsed = 0
      int32 StatusReg //To store SPC_SR status register value
      do
      {
          StatusReg = APACC DATA Read
          StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
      } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)
```

```
        if (time_elapsed > 1 sec) then FAIL_EXIT // Check if command execution time < 1
                                                 //second
        AddrCount = AddrCount + 1 //Increment to load the next NVL byte
       }

       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 00B6] // SPC_KEY1

       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 00D9]// SPC_KEY2 + _WRITE_USER_NVL opcode

       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 0006]// SPC_WRITE_USER_NVL opcode

       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 00F8]//Array ID of "Write Once NVL"

       // Poll status register
       ADDR Write [0x4000 4722]// SPC status register address
       dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

       time_elapsed = 0
       int32 StatusReg //To store SPC_SR status register value
       do
       {
          StatusReg = APACC DATA Read
          StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
       } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

       if (time_elapsed > 1 sec) then FAIL_EXIT//Check if command execution time < 1
                                               //second
}
```

5.9 Step 9: Program Flash Protection Data

Flash protection data is located in address 32'h9040 0000 in the hex file. This step requires three parameters: K - number of flash arrays, N - total number of flash rows, L - number of bytes in row. K and N are derived from the total flash memory size of the device, and the L value is 256 or 288 depending on ECC option. See the respective device datasheet for flash memory size of each device.

```
//Start of "Get_Temp" routine to get Die temperature
byte Temp_Sign, Temp_Magnitude; //Die temperature - used in the PROGRAM_PROTECT_ROW instruc-
tion
byte StatusReg //To store SPC_SR status register value
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00B6]  //SPC_KEY1
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00E1]  //SPC_KEY2 + SPC_GET_TEMP (0xD3+0x0E)

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 000E]  //SPC_ GET_TEMP opcode

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0003] //Number of samples, valid values [1..5]

//Wait until Temperature data is ready
APACC ADDR Write [0x4000 4722]
```

```
byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
  StatusReg = (byte) APACC DATA Read // Save status register value
  StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
} while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x4000 4720]
byte dummy = APACC DATA Read // Dummy SWD read
Temp_Sign =(byte) APACC DATA Read // First byte read is sign of temperature
Temp_Magnitude =(byte) APACC DATA Read // Second byte read is magnitude of temperature

//Wait for IDLE - just in case. Must be in idle state once data byte is read.
APACC ADDR Write [0x4000 4722]// Poll status register
byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
  StatusReg = (byte) APACC DATA Read // Save status register value
  StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
} while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

/* End of "Get_Temp" routine to get Die temperature. The Temp_Magnitude and Temp_Sign are
used in below PROGRAM_PROTECT_ROW step */
/*******************************************************************/

//Calculating total number of rows 'N'
int32 N = (Total_Flash_Code_size)/256;  //Each row has 256 bytes, "Total_Flash_Code_size" is
                                        //in bytes
//Calculating total number of Flash arrays 'K'
if (N % 256 == 0)
{
   byte K= (byte) N/256; //If rows are exact multiple of 256,quotient of (N/256) gives 'K'
}
else
{
   byte K= (byte)((N/256) + 1); //If rows are not exact multiple of 256,increment quotient of
                                // (N/256) by one for 'K'
}

int16 RowsPerArray; //Variable that hold the number of data rows in current Flash array
byte protectionPerArray; //Variable that hold the number of security bytes in current
                         //Flash array
int16 Offset =0; //Offset address of current security byte from address 0x9040 0000 of
                 //hex file
//Program protection bytes for all Arrays
for (int ArrayCount = 0; ArrayCount < K; ArrayCount++)
{
   // Find number of rows in current array
   if (ArrayCount == (K-1))
   {
      RowsPerArray = N - (ArrayCount*256); //Last array may have less than 256 rows
   }
```

```
else
{
   RowsPerArray = 256; //Except last flash array, rest of them have 256 rows
}

protectionPerArray = (RowsPerArray/4) //Each Flash protection byte stores
                                      //protection data of 4 Flash rows
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00B6] // First initiation key
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00D5] // Second initiation key: 0xD3 + 0x02

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0002] // LOAD_ROW opcode

APACC ADDR Write [0x4000 4720]
APACC DATA Write [ArrayCount]//Flash Array ID

int16 ByteCount = 0
while (ByteCount < L) // Define L according to ECC settings
{
  APACC ADDR Write [0x4000 4720]

  if (ByteCount < protectionPerArray)
  {
     APACC DATA Write [XX]//Data at address (32'h90400000 + Offset) of
                          //hex file
     Offset = Offset+1; //Increment the offset address in hex file
  }
  else
  {
     APACC DATA Write [0x0000 0000]//Fill bytes greater than protection size with
                                   //zero
  }

  ByteCount = ByteCount + 1
}

// After loading the protection data, program it in to the Flash hidden rows
//using PROGRAM_PROTECT_ROW command
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00B6] // First initiation key

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00DE] // Second initiation key: 0xD3 + 0x0B

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 000B]// PROGRAM_PROTECT_ROW opcode

APACC ADDR Write [0x4000 4720]
APACC DATA Write [ArrayCount] //Flash array ID

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0000] //Row select value is always zero for protection
                               //data
APACC ADDR Write [0x4000 4720]
APACC DATA Write [Temp_Sign] //Send Sign byte of die temperature
```

```
   APACC ADDR Write [0x4000 4720]
   APACC DATA Write [Temp_Magnitude] //Send Magnitude byte of die temperature

   // Poll status register
   ADDR Write [0x4000 4722]// SPC status register address
   dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

   time_elapsed = 0
   int32 StatusReg //To store SPC_SR status register value
   do
   {
     StatusReg = APACC DATA Read
     StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
   } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

   if (time_elapsed > 1 sec) then FAIL_EXIT

 } //Repeat for all Flash arrays
```

5.10 Step 10: Verify Flash Protection Data (Optional)

Flash protection data is located in address 32'h9040 0000 in the hex file. This step requires two parameters: K - number of flash arrays, N - total number of flash rows. K and N are derived from the total flash memory size of the device. See the respective device datasheet for flash memory size of each device.

```
//Calculating total number of rows 'N'
int32 N = (Total_Flash_Code_size)/256;  //Each row has 256 bytes, "Total_Flash_Code_size" is
                                        //in bytes
//Calculating total number of Flash arrays 'K'
if (N % 256 == 0)
{
   byte K= (byte) N/256; //If rows are exact multiple of 256,quotient of (N/256) gives 'K'
}
else
{
   byte K= (byte)((N/256) + 1); //If rows are not exact multiple of 256,increment quotient of
                                // (N/256) by one for 'K'
}

int16 RowsPerArray; //Variable that hold the number of data rows in current Flash array
byte protectionPerArray; //Variable that hold the number of security bytes in current
                         //Flash array

int16 byte_index = 0 //Variable to keep track of number of bytes read

/* Array to store the protection bytes read from PSoC 5LP Flash array */
byte Data_Array[256];

for (int ArrayCount = 0; ArrayCount < K; ArrayCount++)
{
   // Find number of rows in current array
   if (ArrayCount == (K-1))
   {
      RowsPerArray = N - (ArrayCount*256); //Last array may have less than 256 rows
   }
   else
   {
      RowsPerArray = 256; //Except last flash array, rest of them have 256 rows
```

```
}

protectionPerArray = (RowsPerArray/4) //Each Flash protection byte stores
                                      //protection data of 4 Flash rows

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00B6]//First initiation key

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00DD]//0xDD= (0xD3 + READ_HIDDEN_ROW opcode)

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 000A]// READ_HIDDEN_ROW opcode

APACC ADDR Write [0x4000 4720]
APACC DATA Write [ArrayCount]// Flash Array ID

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0000]// RowID of Protection bytes row

//Wait until Data is ready
ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value
do

{
   StatusReg = APACC DATA Read
   StatusReg = (StatusReg >> 16) & 0xFF  //Extract status code which is in 3rd byte
} while ((StatusReg != [0x0000 0001]) AND time_elapsed < 1 sec)
if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x4000 4720]
dummyByte = APACC DATA Read // Dummy SWD read

/* Read 256 bytes of row data in to Data_Array. Even though the maximum number of
protection bytes is only 64 for a Flash array, it is still required to read all the
256 bytes in Flash protection row to ensure that the SPC returns back to the idle state.
*/
byte_index = 0
while (byte_index < 256)
{
    Data_Array[byte_index] = APACC DATA Read// Save data in to the array
    byte_index = byte_index + 1
}

/* Now, the array Data_Array contains a row of Flash protection data (256 bytes) read from
the device. Compare the first "protectionPerArray" bytes in the array with the protection
data in the hex file. In the hex file, the Flash protection bytes are present starting
from the address 32'h90400000 of the hex file. */

byte_index = 0
while (byte_index < protectionPerArray)
{
  /* hexData[i] is from address (32'h90400000 + (64* ArrayCount) + byte_index) of hex
  file*/
```

```
     if (Data_Array[byte_index] != hexData[i])
     {
        FAIL_EXIT /* Byte mismatch. Verify operation for Protection bytes failed. Abort
                Operation, Exit */
     }

     byte_index = byte_index + 1
    }

  /* Verify operation for Protection bytes passed. Go to next step */

  }//Repeat for all Flash arrays
```

5.11 Step 11: Verify Checksum

The data for this section is located in address 0x90300000 of the hex file. Only the lower two bytes of checksum are stored in the hex file. The MSB byte is stored at address 0x90300000, and the LSB byte is stored at address 0x90300001. This step requires two parameters: K - number of flash arrays, N - total number of flash rows.

```
//Calculating total number of rows 'N'
int32 N = (Total_Flash_Code_size)/256;  //Each row has 256 bytes, "Total_Flash_Code_size" is
                                        //in bytes
//Calculating total number of Flash arrays 'K'
if (N % 256 == 0)
{
   byte K= (byte) N/256; //If rows are exact multiple of 256,quotient of (N/256) gives 'K'
}
else
{
   byte K= (byte)((N/256) + 1); //If rows are not exact multiple of 256,increment quotient of
                               // (N/256) by one for 'K'
}

int16 RowsPerArray; //Variable that hold the number of data rows in current Flash array
int32 chipCheckSum = 0; //32-bit variable used to store the running checksum
//Calculate Checksum for all Arrays
for (byte ArrayCount = 0; ArrayCount < K; ArrayCount++)
{
   // Find number of rows in current array
  if (ArrayCount == (K-1))
   {
      RowsPerArray = N - (ArrayCount*256); //Last array may have less than 256 rows
   }
   else
   {
      RowsPerArray = 256; //Except last flash array, rest of them have 256 rows
   }

   APACC ADDR Write [0x4000 4720]
   APACC DATA Write [0x0000 00B6] //First initiation key

   APACC ADDR Write [0x4000 4720]
   APACC DATA Write [0x0000 00DF] //0xDF = 0xD3 + 0x0C

   APACC ADDR Write [0x4000 4720]
   APACC DATA Write [0x0000 000C] // GET_CHECKSUM opcode
```

```
APACC ADDR Write [0x4000 4720]
APACC DATA Write [ArrayCount] //Flash array ID is the current Flash array

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0000] //Starting row number (lower byte)

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0000] //Starting row number (higher byte)

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0000] //Number of rows minus one (higher byte which is always 0)

APACC ADDR Write [0x4000 4720]
APACC DATA Write [(RowsPerArray - 1)&0xFF] //Number of rows minus one (lower byte)

// Poll status register
ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value
do
{
  StatusReg = APACC DATA Read
  StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
}    while ((StatusReg != [0x0000 0001]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT
APACC ADDR Write [0x4000 4720]
dummyByte = APACC DATA Read// Dummy SWD read
b3 = APACC DATA Read // Checksum byte 4(MSB byte)
b2 = APACC DATA Read // Checksum byte 3
b1 = APACC DATA Read // Checksum byte 2
b0 = APACC DATA Read // Checksum byte 1(LSB byte)

// Add current array 4-byte checksum to running checksum
chipCheckSum = chipCheckSum + (b3 << 24) + (b2 << 16) + (b1 << 8) + (b0 << 0);

// Poll status register till SPC is IDLE
ADDR Write [0x4000 4722]// SPC status register address
dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

time_elapsed = 0
int32 StatusReg //To store SPC_SR status register value
do
{
  StatusReg = APACC DATA Read
  StatusReg = (StatusReg >> 16) & 0xFF  // Extract status code which is in 3rd byte
 } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)

 if (time_elapsed > 1 sec) then FAIL_EXIT

} //Repeat for all Flash arrays

chipCheckSum = chipCheckSum & (0xFFFF); //Extract only the lower 2-byte checksum

/* Compare with 2-byte checksum value in hex file (big endian format) at address
0x90300000. Only the lower two bytes of checksum are stored in the hex file. The MSB byte
```

```
  is stored at address 0x90300000, and the LSB byte is stored at address 0x90300001. */
  if (chipCheckSum != file_checksum) then FAIL_EXIT
```

5.12 Step 12: Program EEPROM (Optional)

The data for this section is located in address 0x90200000 of the hex file.

```
//Start of "Get_Temp" routine to get Die temperature
byte Temp_Sign, Temp_Magnitude; //Die temperature - used in the WRITE_ROW instruction
byte StatusReg //To store SPC_SR status register value
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00B6]  //SPC_KEY1
APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 00E1]  //SPC_KEY2 + SPC_GET_TEMP (0xD3+0x0E)

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 000E]  //SPC_ GET_TEMP opcode

APACC ADDR Write [0x4000 4720]
APACC DATA Write [0x0000 0003] //Number of samples, valid values [1..5]

//Wait until Temperature data is ready
APACC ADDR Write [0x4000 4722]
byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
  StatusReg = (byte) APACC DATA Read // Save status register value
  StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
} while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x4000 4720]
byte dummy = APACC DATA Read // Dummy SWD read
Temp_Sign =(byte) APACC DATA Read // First byte read is sign of temperature
Temp_Magnitude =(byte) APACC DATA Read // Second byte read is magnitude of temperature

//Wait for IDLE - just in case. Must be in idle state once data byte is read.
APACC ADDR Write [0x4000 4722]// Poll status register
byte dummy = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
  StatusReg = (byte) APACC DATA Read // Save status register value
  StatusReg = (StatusReg >> 16) & 0xFF   // Extract status code which is in 3rd byte
} while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT


/*Initialize EEPROM by setting 4-th bit in PM_ACT_CFG12 register*/
APACC ADDR Write [0x4000 43AC] //Read current value from PM_ACT_CFG12
byte dummy = APACC DATA Read // Dummy SWD Read
byte data = APACC DATA Read // Read actual value from the PM_ACT_CFG12
data = data | 0x10 // Set 4-th bit
APACC ADDR Write [0x4000 43AC]
APACC DATA Write [data] // Enable EEPROM
```

```
/* Get the number of rows in EEPROM based on the EEPROM memory size information in the device
datasheet. Each row has 16 bytes */
byte NumofRows
/* EEPROM_SIZE_IN_BYTES is given in the device datasheet */
NumofRows = EEPROM_SIZE_IN_BYTES / 16
/* Program EEPROM row one by one */
byte Row_Count = 0/* Variable to keep track of current row number */
byte Byte_Count = 0/* Variable to keep track of byte number in a row */
while(RowCount < NumOfRows)
{
       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 00B6]/* First SPC Key */
       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 00D5]/* Second SPC Key = 0xD3 + 0x02 */
       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 0002]  /* Load Row Opcode */
       APACC ADDR Write [0x4000 4720]
       APACC DATA Write [0x0000 0040]  /* EEPROM Array ID */
       /* Load the 16 bytes of EEPROM row one by one by reading from the hex file */
       for(ByteCount = 0; ByteCount < 16; ByteCount++)
       {
              /* EEPROMByteData is located in the hexfile at address (0x90200000 + (RowCount
               * 16) + ByteCount)  */
              APACC ADDR Write [0x4000 4720]
              APACC DATA Write [EEPROMByteData]
       }
       /* Read SPC status register to check the status of SPC command. If "Command Success"
          statusis not received within 1 second, then exit the programming operation */
       APACC ADDR Write [0x4000 4722]/* SPC status register address */
       int32 dummy = APACC DATA Read/* Dummy SWD Read */
       int32 StatusReg/* To store SPC_SR status register value */
       time_elapsed = 0
       do
       {
              StatusReg = APACC DATA Read /* Save status register value */
              StatusReg = (StatusReg >> 16) & 0xFF /* status code is in 3rd byte */
```

```
    } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)
    if (time_elapsed > 1 sec) then FAIL_EXIT
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 00B6]/* First SPC Key */
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 00D8]/* Second SPC Key = 0xD3 + 0x05 */
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 0005]/* Write Row Opcode */
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [0x0000 0040]/* EEPROM Array ID */
    APACC ADDR Write [0x4000 4720]
    /* MSB byte of the 2-byte row number. Always zero for EEPROM since maximum number of
       rows can only be 128 */
    APACC DATA Write [0x0000 0000]
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [RowCount]/* LSB byte of the 2-byte row number */
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [Temp_Sign]   /* Temperature Sign byte */
    APACC ADDR Write [0x4000 4720]
    APACC DATA Write [Temp_Magnitude]/* Temperature Magnitude byte */
    /* Read SPC status register to check the status of SPC command. If "Command Success"
       status is not received within 1 second, then exit the programming operation */
    APACC ADDR Write [0x4000 4722]/* SPC status register address */
    int32 dummy = APACC DATA Read/* Dummy SWD Read */
    int32 StatusReg/* To store SPC_SR status register value */
    time_elapsed = 0
    do
    {
          StatusReg = APACC DATA Read /* Save status register value */
          StatusReg = (StatusReg >> 16) & 0xFF /* status code is in 3rd byte */
    } while ((StatusReg != [0x0000 0002]) AND time_elapsed < 1 sec)
    if (time_elapsed > 1 sec) then FAIL_EXIT
    RowCount = RowCount + 1 /* Next EEPROM row to be programmed */
}
```

5.13 Step 13: Verify EEPROM (Optional)

```
/* Get the number of rows in EEPROM based on the EEPROM memory size information in the device
datasheet. Each row has 16 bytes */
byte NumofRows
/* EEPROM_SIZE_IN_BYTES is given in the device datasheet */
NumofRows = EEPROM_SIZE_IN_BYTES / 16
int read_address/* Location of EEPROM address to be read */
int read_data/* 4-byte data read from EEPROM */
byte ByteRead = 0  /* Variable to track number of bytes that have been read */
byte Data_Array[16] /* Array to store the EEPROM row data read from the device */
/* Verify the data programmed in to EEPROM, one row at a time */
while(RowCount < NumOfRows)
{
       ByteRead = 0
       /* Read the EEPROM row data from the device in 4-byte chunks and store in the array */
       while(ByteRead < 16)
       {
              /* Address of EEPROM in PSoC 5. 0x40008000 is EEPROM base address */
              read_address = 0x40008000 + (RowCount * 16) + ByteRead
              APACC ADDR Write [read_address]
              dummyByte = APACC DATA Read/* Dummy SWD read */
              read_data = APACC DATA Read/* Actual 4-byte EEPROM data */
              /* Store the 4-byte data in the array */
              Data_Array[ByteRead] = (byte) (read_data)
              Data_Array[ByteRead + 1] = (byte) (read_data >> 8)
              Data_Array[ByteRead + 2] = (byte) (read_data >> 16)
              Data_Array[ByteRead + 3] = (byte) (read_data >> 24)
              ByteRead = ByteRead + 4 /* Read the next 4-bytes */
       }
       /* Verify the row data read from the device against the hex file data */
       for(ByteRead = 0; ByteRead < 16; ByteRead++)
       {
              /* Replace XX below with byte data from the hex file at address (0x90200000 +
                 (RowCount * 16) + ByteRead). Verify operation is a failure if there is a byte
                 mismatch  */
              if(Data_Array[ByteRead] != XX) then FAIL_EXIT
       }
```

```
        RowCount = RowCount + 1 /* Next row */
}
```

A. Appendix

A.1 Intel Hex File Format

Intel hex file records are a text representation of hexadecimal coded binary data. Only ASCII characters are used; the format is portable across most computer platforms.

Each line (record) of the Intel hex file consists of six parts, as shown in Figure A-1.

Figure A-1. Hex File Record Structure

Start code	Byte count	Address	Record type	Data	Checksum
(Colon character)	(1 byte)	(2 bytes)	(1 byte)	(N bytes)	(1 byte)

- **Start code:** one character - an ASCII colon ':'
- **Byte count:** two hex digits (1 byte) - specifies the number of bytes in the data field
- **Address:** four hex digits (2 bytes) - a 16-bit address at the beginning of the memory position for the data
- **Record type:** two hex digits (00 to 05) - defines the type of data field. The record types used in the hex file generated by PSoC Creator are:
  - 00 - Data record, which contains data and 16-bit address
  - 01 - End of file record, which is a file termination record and has no data. This must be the last line of the file; only one is allowed for every file
  - 04 - Extended linear address record, which allows full 32-bit addressing. The address field is 0000, the byte count is 02. The two data bytes represent the upper 16 bits of the 32 bit address, when combined with the lower 16-bit address of the 00 type record
- **Data:** a sequence of ‘n’ bytes of the data, represented by 2n hex digits
- **Checksum:** two hex digits (1 byte), which is the least significant byte of the two's complement of the sum of the values of all fields except fields 1 and 6 (Start code ‘:’ byte and two hex digits of the Checksum)

Examples for the different record types used in the hex file generated by PSoC Creator are as follows.

Consider that these three records are placed in consecutive lines of the hex file.

:0200000490006A

:0420000000000005F7

:00000001FF

The first record (*:0200000490006A*) is an extended linear address record as indicated by the value in the Record Type field (04). The address field is 0000, the byte count is 02. This means that there are two data bytes in this record. These data bytes (9000) specify the upper 16-bits address of the 32-bit address of data bytes. In this case, all the data records that follow this record are assumed to have their upper 16-bit address as 0x9000 (in other words, the base address is 0x90000000). 6A is the checksum byte for this record.

The next record (*:0420000000000005F7*) is a data record, as indicated by the value in the Record Type field (00). The byte count is 04 indicating that there are four data bytes in this record (00000005). The 32-bit starting address for these data bytes is at address 90002000. The upper 16-bit address (9000) is derived from the extended linear address record in the first line;

the lower 16-bit address is specified in the address field of this record as 2000. F7 is the checksum byte for this record.

The last record (*:00000001FF*) is the end of file record, as indicated by the value in the Record Type field (01). This is the last record of the hex file.

Note The data records of the following multi-bytes region in the hex file are in big-endian format (MSB in lower address): Checksum data at address 0x9030 0000 of hex file; metadata at address 0x9050 0000. The data records of the rest of the multi-byte regions in hex file are all in little-endian format (LSB in lower address).

A.1.1 Organization of Hex File Data

The hex file generated by PSoC Creator contains different types of data, which includes the flash code data, flash configuration data, flash protection data, EEPROM data, customer nonvolatile latch, and write once latch data. Apart from this, the hex file also contains metadata. Metadata is information that is not used for programming the device memory. It is used to maintain data integrity of the hex file and store silicon revision and device ID information. All information including metadata are stored at specific addresses. This allows the programmer to identify which data is meant for what purpose. The address map is explained here and summarized in Figure A-2.

0x0000 0000 – Flash Code Region Data: The flash code data starts at address 0x0000 0000 of the hex file. Each record in the hex file contains 64 bytes of actual data; arrange these into rows of 256 bytes. This is because each flash row of device is of length 256 code bytes (not including the 32 configuration bytes, which are stored in another region). The last address of this section depends on the flash memory size of the device for which the hex file is intended. As an example, for a device with a flash memory capacity of 256 KB, the end address is 0x0003FFFF. See the respective device datasheet or the Device Selector menu in PSoC Creator to know the flash memory size of different part numbers.

0x8000 0000 – Flash Configuration Data: There are 32 bytes of configuration data for each row of flash. This data needs to be appended with the main flash data during the flash programming step. For every 256 code bytes in Program Flash, 32 bytes from this section are appended. The last address of this section depends on the device flash memory capacity. A device with 256 KB flash memory has 32 KB of configuration memory. So in this case, the last address is 0x80007FFF.

0x9000 0000 – Device Configuration NV Latch Data: A 4-byte device configuration nonvolatile latch is used to configure the device even before the reset is released. These four bytes are stored in the addresses starting from 0x90000000. One important bit in this NV latch data is the ECC enable bit (bit 3 of byte 3 located at address 0x90000003). This bit determines the number of bytes to be written during a flash row write process. See "Nonvolatile Memory Organization in PSoC 5LP" on page 80 for details of these four NVL bytes.

0x9010 0000 – Secured Device Mode Configuration Data: This section contains four bytes of the write-once nonvolatile latch data that is used to enable device security. **Warning:** Programming the write-once NV latch with the correct 32-bit key locks the device; perform this step only if all previous steps are passed without errors. PSoC Creator generates all four bytes as zero if the device security feature has not been enabled to ensure that there is no accidental programming of the latch with correct key. Failure analysis support may be lost on units after this step is performed with correct key. Refer to Appendix B of the PSoC 5LP TRM for details on this device security feature.

0x9020 0000 – EEPROM: PSoC 5LP devices have on-chip EEPROM memory and the data to be programmed in to the EEPROM is stored in this region. EEPROM is programmed row wise where each row contains 16 bytes. Since each record in the EEPROM region of the hex file contains 64 bytes of data, each record has the data corresponding to 4 contiguous EEPROM rows.

0x9030 0000 – Checksum Data: This 2-byte checksum data is the checksum computed from the entire flash memory of the device (main code and configuration data). This 2-byte checksum is compared with the checksum value read from the device to check if correct data has been programmed. Though the CHECKSUM command sent to the device returns a 4-byte value, only the lower two bytes of the returned value are compared with the checksum data in the hex file. The 2-byte checksum in the data record is in Big-endian format (MSB byte is first byte).

0x9040 0000 – Flash Protection Data: This section contains data to be programmed to configure the protection settings of flash memory. Arrange data in this section in a single row to match the internal flash memory architecture. Because there are two bits of protection data for each main flash row, a 256 KB flash (with 1024 rows, 256 rows in each of four 64K Flash arrays) has 256 bytes of protection data.

0x9050 0000 – Metadata: The data in this section of the hex file is not programmed into the target device. It is used to check the data integrity of hex file, silicon revision for which the hex file is intended, and so on. The different data in this section is tabulated as follows.

Table A-1. Metadata Organization in Hex FIle

Starting Address	Data Type	Number of Bytes
0x9050 0000	Hex file version	2 (big-endian)
0x9050 0002	Device ID	4(big-endian)
0x9050 0006	Silicon revision	1
0x9050 0007	Debug Enable	1
0x9050 0008	Internal use	4

Hex File Version: This 2-byte data (big-endian format) is used to differentiate between different hex file versions. For example, if new metadata information or EEPROM data is added to the hex file generated by PSoC Creator, there is a need to distinguish between the different versions of hex files. By reading these two bytes you can ascertain which version of the hex file is going to be programmed. At present, PSoC Creator generates only one type of hex file and this field always has a constant value of 0x0001. The only value that this field accepts is 0x0001 because there is only one version of the hex file.

Device ID: This field has the 4-byte device ID (big-endian format), which is unique to each part number. Compare the device ID read from the device with the device ID present in this field to make sure the correct device for which the hex file is intended is programmed. See the device datasheet for information on the device IDs of different part numbers.

Silicon Revision: This 1-byte value is for different revisions of the silicon. This data is not used anywhere in the PSoC 5LP programming sequence. For PSoC 5LP, the revision IDs are as follows:

1 – ES1 (TM)

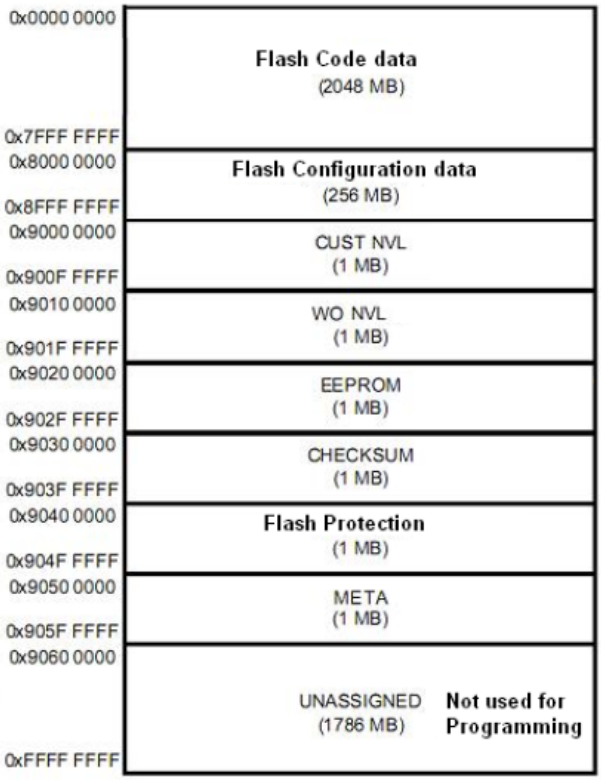
2 – ES2 (LP)

Debug Enable: This 1-byte data stores a Boolean value indicating if debugging is enabled for the program code. This is also not used in programming. The possible values for this byte are:

0 – Debugging Disabled, 1 – Debugging Enabled

Internal Use: The 4-byte data is used internally by PSoC Programmer software. It is not related to actual device programming and need not be used by programmers of third party vendors.

Figure A-2. PSoC 5LP Hex File Address Map



A.2 Nonvolatile Memory Organization in PSoC 5LP

PSoC 3 and PSoC 5LP devices have three types of nonvolatile memory: Flash, Electronically Erasable Programmable Read Only Memory (EEPROM), and Nonvolatile Latch (NVL). This section gives a quick overview of the interface used to program the nonvolatile memory. It also discusses nonvolatile memory organization. EEPROM memory is not explained in this section because programming of EEPROM using external programmer is not defined in the device programming specification currently. Refer to the “Memory” section of the PSoC 5LP TRM for detailed information on these topics.

A.2.1 Nonvolatile Memory Programming

All nonvolatile memory programming operations are done through a simple command/status register interface summarized in Table A-2.

Table A-2. SPC Command and Status Registers

Register	Size (Bits)	Description
SPC_CPU_DATA	8	Data to/from the CPU
SPC_DMA_DATA	8	Data to/from the DMAC
SPC_SR	8	Status – ready, data available, status code

Commands and data are sent as a series of bytes to either SPC_CPU_DATA or SPC_DMA_DATA, depending on the source of the command. The programming procedure in this document always uses the SPC_CPU_DATA register. Response data is read via the same register to which the command is sent. The status register, SPC_SR, indicates whether a new command can be accepted, when data is available for the most recent command, and a success/failure response for the most recent command.

A.2.2 Commands

Before sending a command to the SPC_CPU_DATA or SPC_DMA_DATA register, the SPC_Idle bit in SPC_SR[1] must be ‘1’. SPC_Idle will go to ‘0’ when the first byte of a command (0xB6) is written to a data register, and go back to ‘1’ when command execution is complete or an error is detected. Commands sent to either data register while SPC_Idle is ‘0’ are ignored. All commands must adhere to the following format:

- Key byte #1 – always 0xB6
- Key byte #2 – 0xD3 plus the command code (ignore overflow)
- Command code byte
- Command parameter bytes
- Command data bytes

Refer to the “Nonvolatile Memory Programming” chapter in the PSoC 5LP TRM for a list of command codes and the explanation, parameters, and return values for each command.

A.2.3 Command Status

The status register, SPC_SR, indicates whether a new command can be accepted, when data is available for the most recent command, and a success/failure response for the most recent command. The bit-field definitions of the SPC_SR register is given in Figure A-3.

Figure A-3. SPC_SR Status Register Bit Field Definitions

Bits	7	6	5	4	3	2	1	0
SW Access:Reset	R:000000						R:1	R:0
HW Access	R/W						R/W	R/W
Name	Status_Code						SPC_Idle	Data_Ready

Data_Ready bit: This bit (Bit [0] of SPC_SR) indicates whether the SPC has data that is ready to be read from the SPC CPU or DMA Data Register.

SPC_Idle bit: This bit (Bit [1] of SPC_SR) indicates whether the SPC is currently executing an instruction. The bit transitions low as soon as the first byte of the 2-byte command key (0xB6) is written into the SPC CPU or DMA Data Register. The bit transitions high as soon as an instruction completes or if the second byte of the command key is invalid.

Status_code (5-bit status code): The Status Code (Bits [7:2] of SPC_SR) represents the exit status of the last executed SPC instruction. The values of this field are given in Table A-3.

Table A-3. Status Codes for an SPC Command

SPC Status Code (Bits[7:2] in SPC_SR register)	Meaning
0x00	Operation successful
0x01	Invalid array ID for given command
0x02	Invalid 2-byte key
0x03	Addressed nonvolatile memory array is asleep
0x04	External access failure (SPC is not in external access mode)
0x05	Invalid 'N' value for given command
0x06	Test mode failure (SPC is not in programming mode)
0x07	Smart write algorithm checksum failure
0x08	Smart write parameter checksum failure
0x09	Protection check failure: Flash protection settings are in a state which prevents the given command from executing
0x0A	Invalid address parameter for the given command
0x0B	Invalid command code
0x0C	Invalid row ID parameter for given command
0x0D	Invalid input value for Get Temp and Get ADC commands
0x0E	Tempsensor Vbe is currently driven to an external device
0x0F	Invalid SPC state
0x10 – 0x3F	Smart write return codes (only when using Smart Write algorithm)
0x20	PEP program failure (only when using PEP algorithm): Data verification failure (row latch checksum!= programmed row checksum)

A.2.4 Nonvolatile Memory Organization

A.2.4.1 Flash Program Memory

PSoC 5LP flash memory has the following features:

- Organized in rows, where each row contains 256 main flash code bytes plus 32 bytes for configuration data storage. The size of each flash row is 288 bytes. Flash memory can be programmed in resolution of rows.
- Organized as either one array of 128 or 256 rows, or as multiple arrays of 256 rows each.
- For each flash row, protection bits control whether the flash can be read or written by external debug devices and whether it can be reprogrammed by a boot loader. For each flash array, flash protection bits are stored in a hidden row in that array. In the hidden row, two protection bits per row are packed into a byte, so each byte in the hidden row has protection settings for four flash rows of that array.

A.2.4.2 EEPROM

PSoC 5LP EEPROM has the following features:

- Organized in rows, where each row contains 16 bytes.
- Organized as one block (array) of 32, 64, or 128 rows, depending on the size of EEPROM memory.

A.2.4.3 Device Configuration NVLs

PSoC 5LP has a 4-byte array of device configuration NVLs that are used to configure the device at reset. The NVL register map is shown in Figure A-4.

Figure A-4. Device Configuration NVL Register Map

Register Address	7	6	5	4	3	2	1	0
0x00	PRT3RDM[1:0]		PRT2RDM[1:0]		PRT1RDM[1:0]		PRT0RDM[1:0]	
0x01	PRT12RDM[1:0]		PRT6RDM[1:0]		PRT5RDM[1:0]		PRT4RDM[1:0]	
0x02	XRESMEN	DEBUG_EN	Reserved				PRT15RDM[1:0]	
0x03	DIG_PHS_DLY[3:0]				ECCEN	DPS[1:0]		CFGSPEED

Table A-4 shows the details for individual fields and their factory default settings that are relevant to device programming. Refer to the "Nonvolatile Latch" chapter of the PSoC 5LP Architecture TRM for more details.

Table A-4. Device Configuration NVL Register Description, Default Values

Field	Description	Settings
XRESMEN	Controls whether pin P1[2] is configured as a GPIO pin or as an XRES pin.	0 (default value for devices with dedicated XRES) - GPIO pin 1 (default value for devices without dedicated XRES) - XRES pin
DPS[1:0]	Controls the usage of various Port 1 pins as a debug/Programming port.	00b - 5-wire JTAG 01b (default) - 4-wire JTAG 10b - SWD 11b - debug ports disabled.
DEBUG_EN	This bit allows access to be granted to the Cortex-M3's DAP, which enables firmware debug and programming when either in JTAG or SWD mode.	0 - Debug Disabled (no DAP access) 1 (default) - Debug Enabled (DAP access)
ECCEN	Controls whether ECC flash is used for ECC or for general configuration and data storage.	0 (default) - ECC disabled 1- ECC enabled

PSoC Creator enables modifying the device configuration NVLs. However, the number of NVL erase/write cycles is limited. See the PSoC 5LP device datasheet for NVL specifications.

There are three settings in NVL that are relevant to the programming flow.

- **Debug Port Select (DPS) setting:** This 2-bit value determines the default protocol that is used to program or debug the device through the Port 1 pins without sending the Port Acquire key. Entering programming mode through JTAG interface is dependent on DPS setting.

  Note The DPS setting is relevant only for JTAG interface programming. The only recommended DPS settings for JTAG programming are 4-wire JTAG and 5-wire JTAG. Programmers that support JTAG interface programming should not allow a hex file with "Debug Ports Disabled" setting to be programmed to the device, as this prevents further programming of the device through the JTAG interface.
- **XRESMEN setting:** P1[2] pin may be configured either as an external reset (XRES_N) pin or as a GPIO pin. The configuration of that pin is controlled with this NVL bit.
  - 0 - P1[2] is a GPIO pin. This is the default factory setting for non 48-pin devices that already have a dedicated XRES pin.
  - P1[2] is configured as a XRES_N. This is the default factory setting for 48-pin devices that do not have a dedicated XRES pin.

  To program 48-pin devices, which do not have a dedicated XRES pin, the P1[2] pin can be used as an XRES. To facilitate this, 48-pin devices that come out of the factory have default value of XRESMEN = 1. Take care not to program the device with NVL setting of “XRESMEN = 0". Otherwise, It is not possible to program the device further using XRES pin as P1[2]

is now configured as a GPIO pin. Power cycle mode programming is the only available option if P1[2] is disabled as XRES pin for 48-pin devices.

To program non 48-pin devices, which have a dedicated XRES pin, the P1[2] pin cannot be directly used as an XRES pin. This is because the devices with dedicated XRES pin that come out of the factory have default value of XRESMEN = 0. The reason for this feature is that there is a dedicated XRES pin already available; only in rare cases P1[2] is also used as an XRES pin.

- **ECCEN setting:** Flash memory in PSoC 5LP is organized in rows, where each row contains 256 code bytes plus 32 bytes for either error correcting codes (ECC) or configuration data storage. The ECCEN bit determines whether these 32 bytes are used for error correction or data storage.
  - 0 (default) - ECC feature is disabled
  - 1 - ECC feature is enabled

  If the ECC feature is disabled, then during the Programming Flash step, 288 (255 + 32) bytes need to be loaded while programming each flash row. If ECC is enabled, only 256 bytes need to be loaded.
- **DEBUG_EN:** This bit allows access to be granted to the Cortex-M3's DAP, which enables firmware debug and programming when either in JTAG or SWD mode. JTAG or SWD can be enabled by the Debug Port Select (DPS) bits. When DEBUG_EN is not set, it is required to enter test mode to gain DAP access and enable device programming.
  - 0 - Debug Disabled (no DAP access)
  - 1 (default) - Debug Enabled (DAP access)

A.2.4.4 Write Once Nonvolatile Latches (WO NVL)

The user can write the key in WOL to lock out external access only if no flash protection is set. In the programming flow, programming of WOL is done before the flash protection bytes.

Note that when the WO NVL is programmed with the correct 32-bit key (0x50536F43) and the device is reset after programming, the part cannot be programmed further, and becomes a One Time Programmable (OTP) device. The WO NVL locks the part out of Debug and Test modes; it also permanently gates off the ability to erase or alter the contents of the latch. This step should hence be exercised with extreme caution considering these effects.

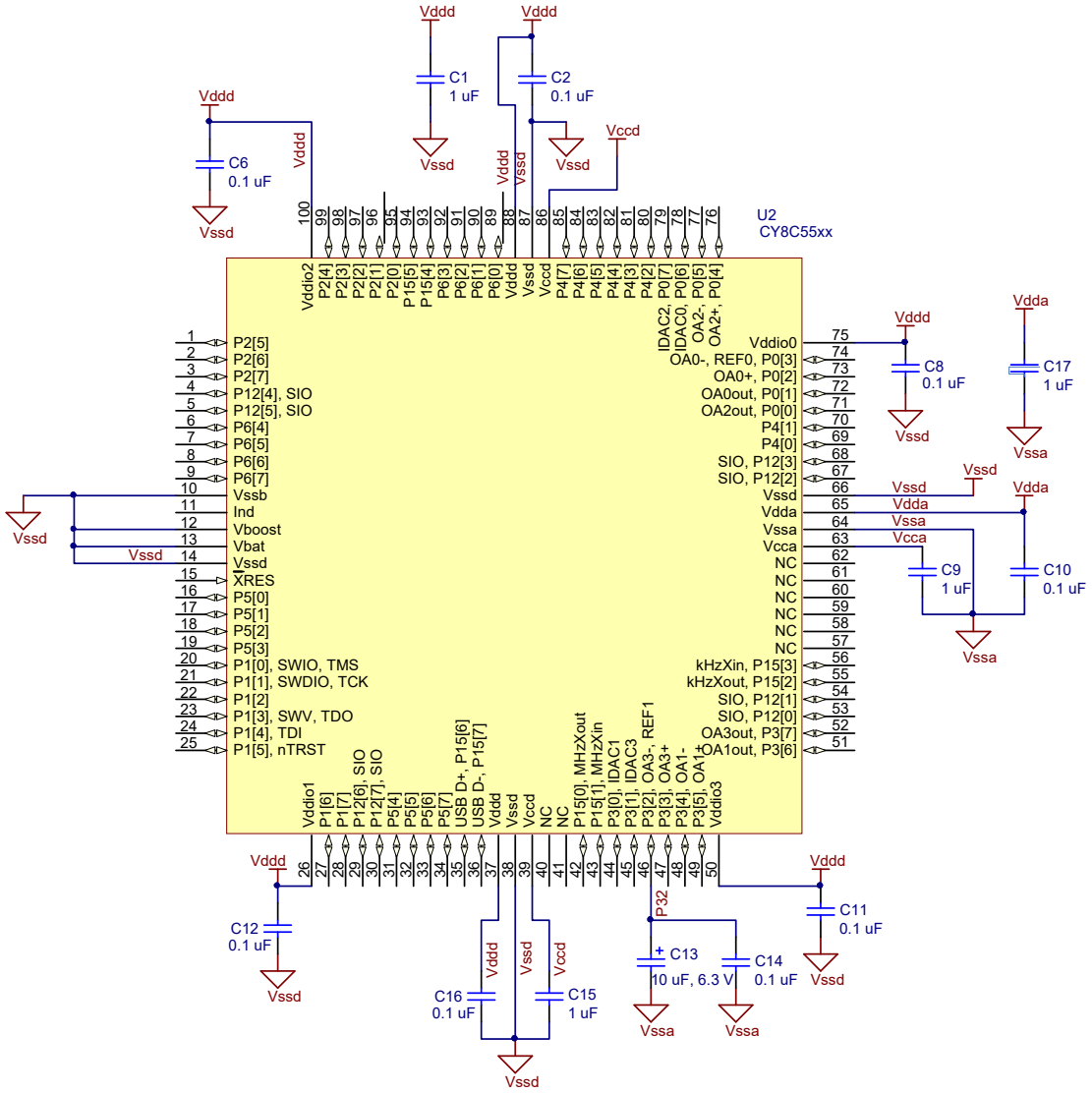
A.3 Example Schematic

The following figure shows an example reference schematic for the 100-pin TQFP part with the power connections. This can also be used for the other PSoC 5LP packages; however, the pinout will vary for each package. See the PSoC 5LP device datasheet for information on specific package pinout and for specifications on power supply pins. Note that Figure A-5 does not show the programming connections between the host programmer and PSoC 5LP. This is illustrated in Figure 1-1 on page 6.

Figure A-5 shows that:

- The two pins labeled Vddd must be connected together.
- The two pins labeled Vccd must be connected together, with capacitance added. The trace between the two Vccd pins should be as short as possible.
- The two pins labeled Vssd must be connected together.

Figure A-5. 100-pin TQFP Part with Power Connections



Note The two Vccd pins must be connected together with as short a trace as possible. A trace under the device is recommended.