CY8C32xxx, CY8C34xxx, CY8C36xxx, CY8C38xxx
CY8CTMA39x, CY8CTMA8xx, CY8CTMA6xx

PSoC® 3 Device Programming Specifications

Document # 001-62391 Rev. *L

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

Copyrights

Contents

1. Introduction

PSoC® 3 device programming refers to the programming of nonvolatile memory in PSoC 3 using an external host programmer. Nonvolatile memory, in the context of external host programmer, includes flash memory, EEPROM, device configuration nonvolatile latch (NVL), and write once NVL. PSoC 3 supports programming through the serial wire debug (SWD) interface or the Joint Test Action Group (JTAG) interface. The data to be programmed is stored in a hex file. This document explains the hardware connections, programming protocol, programming vectors, and the timing information to develop programming solutions for a PSoC 3 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. MiniProg3 programmer is used in the prototype stage of application development to program and debug PSoC 3 devices on the board. Third-party programmers are used for production programming of PSoC 3 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 external microcontroller can be used to perform in-system programming of PSoC 3 devices either for complete programming or partial firmware upgrade.

The host programmer programs the PSoC 3 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 documents, software, and a list of supported third-party programmers.

1.2 Hardware Connections

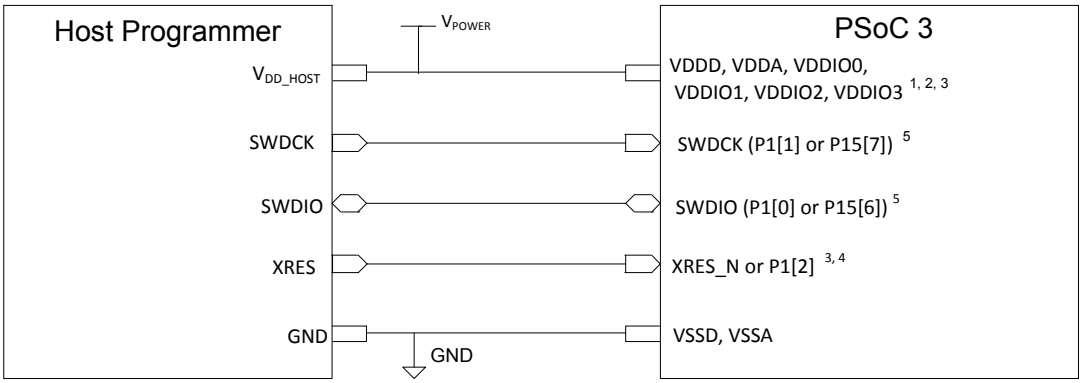
This section discusses hardware connections between the host programmer and the PSoC 3 device for programming through the SWD and JTAG interfaces. Only programming related connections are discussed. For a complete schematic of the PSoC 3 device for programming, including the PSoC 3 regulator output pins (VCCD, VCCA), see “Example Schematic” on page 78. The PSoC 3 device datasheet has information on device operating conditions, specifications, and pinouts for the different PSoC 3 packages.

1.2.1 SWD Interface

Figure 1-1 shows the hardware connections between the host programmer and the target PSoC 3 device to program through the SWD interface.

PSoC 3 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 3 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 3 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 3



Notes for Figure 1-1:

1. The voltage level of the host programmer and the supply voltage for PSoC 3 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 3 are powered by the VDDIO1 pin. USB SWD pins are powered by 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 3. The remaining PSoC 3 power supply pins (VDDD, VDDA, VDDIO0, VDDIO2, 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, VDDIO1 pins of PSoC 3. The remaining PSoC 3 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, VDDIOs) in PSoC 3.
3. For power cycle mode programming, XRES pin is not required. The VDDD, VDDA, VDDIO0, VDDIO1, VDDIO2, VDDIO3 pins of PSoC 3 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. 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 3 pins involved in SWD interface programming.

Table 1-1. Host Programmer Requirements for PSoC 3 Programming

Pin	Host Programmer Requirement	PSoC 3 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 3 changes between two drive modes for read and write operations on the SWDIO line using the Turnaround (TrN) phase of SWD protocol. Host must also change the drive mode of the SWDIO line during this TrN phase. When the host writes to SWDIO, PSoC 3 reads from SWDIO and vice-versa.

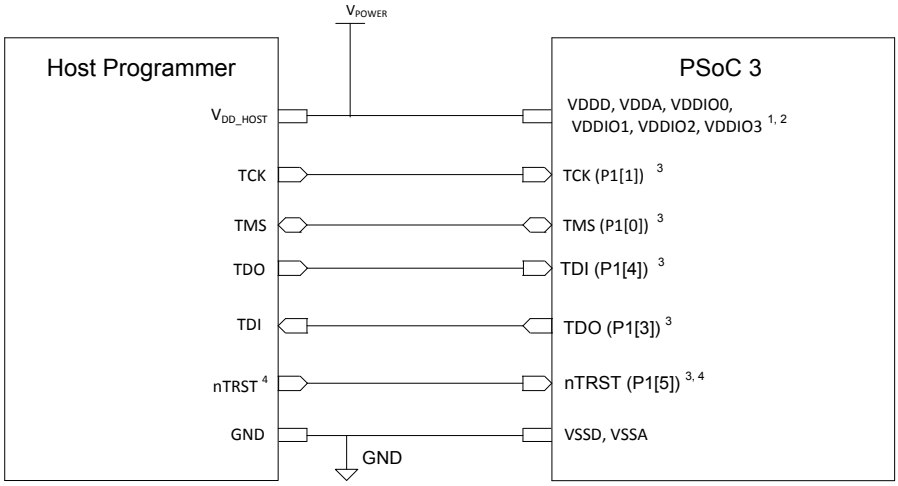
Pin	Host Programmer Requirement	PSoC 3 Function	Comment
XRES	Strong drive (CMOS drive) digital output	Digital input with internal 5.6 kΩ resistive pull-up to VDDIO1	The XRES pin or P1[2] as XRES in PSoC 3 is active low input and there is an internal 5.6 kΩ pull-up resistor to VDDIO1.
VDDA, VDDD, VDDIO	Positive voltage	Digital, analog, and I/O power supply	For power cycle mode, tie the VDDD, VDDA, and VDDIO pins of PSoC 3 to the same power supply. Toggle power to these pins to reset the device. See the PSoC 3 device data sheet 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)	

1.2.2 JTAG Interface

Figure 1-2 shows the hardware connections between the host programmer and PSoC 3 device to program through the JTAG interface.

There are fixed port pins to program PSoC 3 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 3 device programming, but can be enabled for debugging operations by programming the device configuration NVL with a 5-wire JTAG setting (default factory setting is 4-wire JTAG).

Figure 1-2. JTAG Programming Interface Connections between Host Programmer and PSoC 3



Notes for Figure 1-2:

1. The voltage level of the host programmer and the supply voltage for PSoC 3 I/O pins involved in programming should be the same. PSoC 3 JTAG pins are powered by VDDIO1. The host voltage level (V_{DD_HOST}) should be the same as VDDIO1 pin. The remaining PSoC 3 power supply pins (VDDD, VDDA, VDDIO0, VDDIO2, 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, VDDIOs) in PSoC 3.
3. PSoC 3 programming using third-party JTAG programmers is only possible if Debug Port Select (DPS) = “4- /5- wire JTAG”. Silicon revision 5 (TO6) or later added new bits in NVLs – Debug Enable, which if set, makes programming fully compliant with the JTAG standard. Therefore, revisions 2, 3, and 4 are programmable through JTAG pins but imposes some extra requirements to the JTAG master. Revisions 5 or later can be programmed in full compliance with the JTAG specification. The default NVL settings are DPS = “4-wire JTAG”, Debug Enable = “ON”. Normally, these settings must not be changed during programming. If they are modified, then the JTAG port can still be re-enabled by reprogramming NVLs using MiniProg3 in the SWD mode (see the connection in Figure 1-1). Note that if the JTAG master can switch the silicon between the SWD and JTAG modes, then it can still program the silicon if DPS= ”SWD”, but not when DPS = “Debug Port Disabled".

4. The nTRST pin is an optional connection for the JTAG interface. It is not functional during PSoC 3 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 3 pins involved in JTAG interface programming.

Table 1-2. Host Programmer Requirements for PSoC 3 JTAG Interface Programming

Pin	Host Programmer Requirement	PSoC 3 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 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 host is connected to TDO of PSoC 3 and vice-versa. TDI input in PSoC 3 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 host is connected to TDO of PSoC 3 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 3 has a pull-up resistor to ensure that the pin is in known state (logic high) when not driven by 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 JTAG interface. It is not functional during programming of PSoC 3 device. Use the TMS and TCK pins to do a reset of the JTAG TAP controller.
VDDA, VDDD, VDDIOs	Positive voltage	Digital, analog, I/O power supply	See the PSoC 3 device data sheet for specifications on power pins (VDDD, VDDA, VDDIO0, VDDIO1, VDDIO2, VDDIO3) and Ground pins (VSSD, 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: CY8C32xxx, CY8C34xxx, CY8C36xxx, CY8C38xxx, CY8CTMA39x, CY8CTMA8xx, CY8CTMA6xx PSoC® 3 Device Programming Specifications

Document Number: 001-62391

Revision	Issue Date	Origin of Change	Description of Change
**	06/29/2010	VVSK/ANDI	Initial version
*A	11/25/2010	VVSK/ANDI	Code changed in Step 1 - Appendix B
*B	04/04/2011	VVSK	Major rewrite of application note including addition of timing diagrams, programming specifications. Modified Step 1 of programming flow for both SWD, JTAG.
*C	04/18/2011	VVSK	Hyperlink issue fixed. Figure 8 and Figure 9 modified
*D	05/31/2011	VVSK	Changed title and abstract. Added Associated Part Family. Removed checking of "Revision ID" from Programming flow.
*E	09/20/2011	VVSK	Converted to TRM category
*F	04/10/2012	VVSK	Fixed typos. Corrected JTAG read procedure.
*G	05/17/2012	VVSK	Updated section 3.5 including Figure 3-12. Updated the code in section 5.5.
*H	05/24/2012	VVSK	Fixed a code error in section 5.5.
*I	12/05/2012	VVSK	Updated Figure 3-1, Figure 3-2, Figure 3-6, Figure 3-8. Added sections to include Step 12 Program EEPROM and Step 13 Verify EEPROM
*J	06/20/2013	ANDI	Updates for PSoC3 TO6 revision, described JTAG-compliant programming. Updated Sections 1.2.2, 3.1.2, 5.1.4 Added Section 5.3, A.2.4.3
*K	07/25/2016	MYKZ	Sunset review; no content updates
*L	04/25/2017	AESATMP8	Updated logo and Copyright.

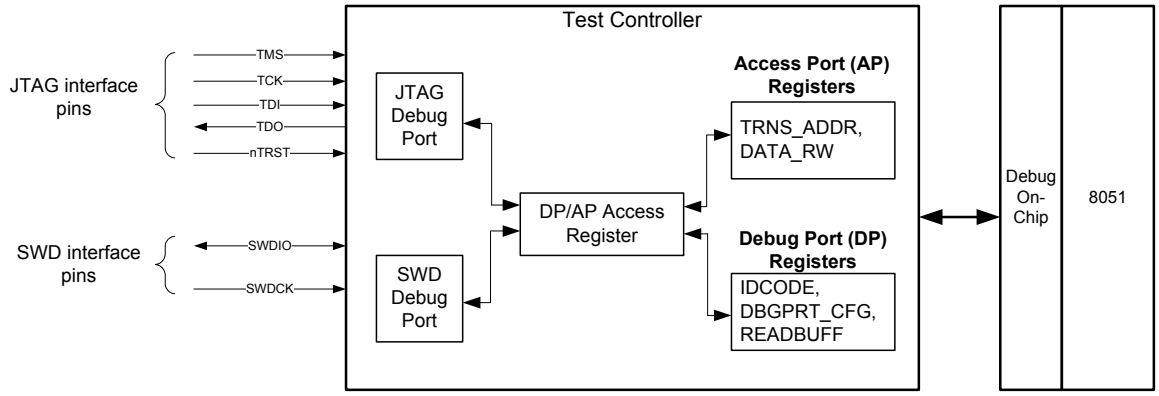
2. PSoC 3 Programming Interface

This section explains the programming interface in PSoC 3 and the registers used for programming. An overview of the SWD and JTAG interfaces is also provided. See "Nonvolatile Memory Organization in PSoC 3" on page 74 for details.

2.1 Test Controller Block

The host programmer communicates with PSoC 3 through the device's internal Test Controller (TC). The TC is the interface that provides access to the PSoC 3 Debug on Chip (DoC) module, which in turn provides access to the device memory and registers. Using TC and DoC, the host programmer writes to SRAM, sets internal registers, and programs the device's flash memory and nonvolatile latches (NVLs).

Figure 2-1. PSoC 3 Programming Interface



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

PSoC 3 test controller has a DP/AP Access register that is 35 bits wide. This register, which is part of the TC interface, is used to transfer data between the JTAG or SWD interface and the Debug Port and Access Port registers. The SWD interface enables direct reads and writes of the DP/AP Access register. The JTAG interface uses the DPACC and APACC instructions. The Access Port (AP) registers are used to read data from a specific address or write data to a specific address. The Debug Port Registers contain the Debug Port configuration such as byte size of AP register memory access and device JTAG ID.

2.1.1.1 Write to DP/AP Access Register

Figure 2-2 shows the structure in the JTAG Update-DR stage or when writing to the DP/AP Access register from the SWD interface.

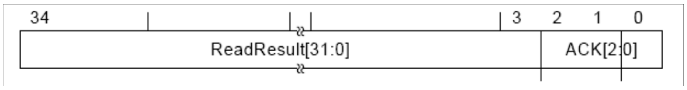
Figure 2-2. Writing to the DP/AP Access Register

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

2.1.1.2 Read from DP/AP Access Register

Figure 2-3 shows the structure of the 35-bit data register in the Capture-DR state of JTAG FSM (before start of shift out of data through TDO) or when reading the DP/AP Access register from the SWD interface.

Figure 2-3. Reading from the DP/AP Access Register



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

Table 2-1. ACK Response for SWD Transfers

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

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

The DP and AP registers listed in Table 2-2 are part of TC. JTAG has separate instructions (DPACC, APACC) to distinguish between AP and DP access; the SWD protocol uses the APnDP bit for this.

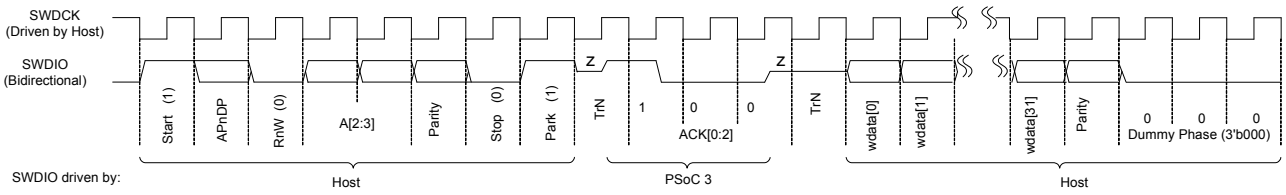
Table 2-2. Debug Port and Access Port Registers (PSoC 3)

Register Name	Register Type	Address (A[3:2])	Function
IDCODE	DP	00	32-bit Device JTAG IDCODE register. IDCODE register address bits are applicable only for SWD interface. JTAG interface has a separate IDCODE instruction to access IDCODE register directly.
DBGPRT_CFG	DP	01	8-bit Debug Port Configuration register - Bits [2:1] determine if the transfer size is 8, 16, or 32 bits. Bit [3] enables the auto address increment functionality. For 8-bit transfer size, write 2'b00 to Bits [2:1]; for 16-bit transfer size, write 2'b01 to Bits [2:1]; and for 32-bit transfer size, write 2'b10 to Bits [2:1]. To enable auto increment of address, write a 1'b1 to Bit [3]. Remaining bits are written '0'.
READBUFF	DP	11	Port Acquire key is written to this 32-bit register to acquire debug port for programming PSoC 3 through the SWD interface.
TRNS_ADDR	AP	01	24-bit Transfer Address register that holds the address that is used for PSoC 3 register access.
DATA_RW	AP	11	32-bit Data register that holds the data to be read from/written to the address specified by the TRNS_ADDR register.

2.2 SWD Interface

The SWD interface has two signals: data (SWDIO) and clock for data (SWDCK). The host programmer always drives the clock line, whereas either the programmer or PSoC 3 device drives the data line. The timing diagram for SWD protocol is given in Programming Specifications chapter on page 39. Host programmer and PSoC 3 communicate in packet format through the SWD interface. Write packet refers to the SWD packet transaction in which the host writes data to PSoC 3. Read packet refers to the SWD packet transaction in which the host reads data from PSoC 3. The Write packet and Read packet formats are illustrated in Figure 2-4 and Figure 2-5, respectively

Figure 2-4. SWD 'Write Packet' Timing Diagram



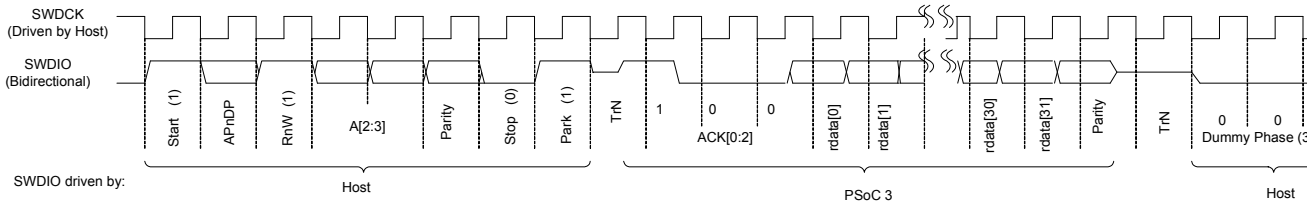
a.) Host Write Operation: Host sends data on the SWDIO line on the falling edge of SWDCK; PSoC 3 reads that data on the next SWDCK rising edge (Example: 8-bit header data, Write data(wdata[31:0]), Dummy phase (3'b000))

b.) Host Read Operation: PSoC 3 sends data on the SWDIO line on the rising edge of SWDCK; host reads that data on the next SWDCK falling edge (Example: 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 3 drives 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 3 and the host will not drive the line during the entire second TrN phase (indicated as 'z'). Host starts 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 3 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 2-5. SWD 'Read Packet' Timing Diagram



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

b.) Host Read Operation: PSoC 3 sends data on the SWDIO line on the rising edge of SWDCK; the Host reads that data on the next SWDCK falling edge (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 3 drives 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 3 and the host will not drive the line during the entire second TrN phase (indicated as 'z'). Host starts sending the Dummy phase (3'b000) on the next falling edge of SWDCK after the 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 3 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 2-4 and Figure 2-5). Each data transfer consists of three phases:

- **Packet request** – External host programmer issues a request to PSoC 3.
- **Acknowledge response** – PSoC 3 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 3 to host, following a read request – RDATA
- Host to PSoC 3, following a write request – WDATA

In Figure 2-4 and Figure 2-5, 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 PSoC 3, or ‘0’ for a write to PSoC 3.
4. The ADDR bits (A[3:2]) are register select bits for access port or debug port. See Table 2-2 for address bit definitions.
5. The parity bit has the parity of APnDP, RnW, and ADDR. This is even parity bit. If number of logical 1’s 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 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. Note that the ACK in the current SWD transfer reflects the status of the previous transfer. OK ACK means the previous packet was 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 the transaction is a read, the host ignores the data read in the data phase. PSoC 3 does not drive the line and the host must not check the parity bit as well.

   b. For a WAIT response, if the transaction is a write, PSoC 3 ignores the data phase. However, 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. A WAIT response indicates that the PSoC 3 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 retry.

   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 3 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 PSoC 3 to change the Drive modes on their respective SWDIO line. During the first TrN phase after packet request, PSoC 3 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. The second TrN phase is one-and-a-half cycle long. Neither the host nor PSoC 3 should drive SWDIO line during both phases as indicated by ‘z’ in Figure 2-4 and Figure 2-5.
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 2-4 and Figure 2-5, there is a “DUMMY” phase, which is three SWD clock cycles with SWDIO line held low. The dummy phase is not part of the SWD protocol. The three extra clocks with SWDIO low are required for the Test Controller in PSoC 3 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 50 or more cycles with SWDIO high. To return to the idle state, SWDIO must be clocked low once. The host programmer can begin a new SWD packet transaction from the idle state.

2.2.1 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. 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 LSB and the 'Park' bit is the most significant bit (MSB). Use Table 2-3 and vectors given in SWD and JTAG Vectors for Programming chapter on page 43 to implement PSoC 3 programming.

Table 2-3. SWD Transfer Request Data Packet for Test Controller DPACC and APACC Register Access

Pseudo Code	Register Name	Type of Operation	SWD Transfer Request Data (LSb bit sent first)	
			Binary	Hex
DPACC IDCODE Read	IDCODE	Read	8'b10100101	8'hA5
DPACC DP CONFIG Write	DPGPRT_CFG	Write	8'b10101001	8'hA9
DPACC READBUFF Write	READBUFF	Write	8'b10011001	8'h99
APACC ADDR Write	TRNS_ADDR	Write	8'b10001011	8'h8B
APACC DATA Read	DATA_RW	Read	8'b10011111	8'h9F
APACC DATA Write	DATA_RW	Write	8'b10111011	8'hBB

The TRNS_ADDR register holds the PSoC 3 memory address that needs to be accessed. To read or write PSoC 3 internal registers or SRAM, first write the address to the TRNS_ADDR register (pseudo code–APACC ADDR Write). For a write operation, write data to the DATA_RW register (pseudo code–APACC DATA Write). If it is a read operation, read the DATA_RW register twice (pseudo code–APACC DATA Read); the TC reads out data through the data line.

For example, to write 32'hB6 to the target device internal register at address 32'h4720, the following SWD transfers are necessary:

APACC ADDR WRITE [0x00004720]

APACC DATA WRITE [0x000000B6]

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

11010001 (TrN)(ACK)
(TrN)00000100111000100000000000000000(1)

11011101 (TrN) (ACK)
(TrN)01101110000000000000000000000000(1)

'(ACK)' indicates waiting for ACK from 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, write the address to the APACC ADDR register address. Then, read the DATA_RW register twice. The first read initiates the command to the DoC interface and the second read returns the requested value. For example, to read from address 32'h4720, the following transfers need to be done:

APACC ADDR Write [0x00004720]

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 2-4 and Figure 2-5, 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 and outputs data in binary format. An example is as follows. The SWD_packet function recognizes the SWD transfer, and puts the corresponding binary data into the outgoing data buffer for transmission.

SWD_packet(APACC_ADDR, 32'h4720)

SWD_packet(APACC_DATA_WRITE, 32'hB6)

Data= SWD_packet(APACC_DATA_READ)

2.3 JTAG Interface

PSoC 3 JTAG interface complies with the IEEE 1149.1-2001 specification and provides additional instructions. It has a 35-bit data register (called DP/AP access register) and 4-bit instruction register. Refer to "Test Controller" chapter of the PSoC 3 Architecture TRM for details on the instructions supported by JTAG interface and an explanation of 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 Programming Specifications chapter on page 39.

Table 2-4. PSoC 3 JTAG Instructions

Bit Code [3:0]	Instruction	PSoC 3 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 2-2 and Figure 2-3, respectively.

2.3.1 Register Access Using JTAG Interface

The following steps show how to access an address using the JTAG interface.

1. Assume that the address value to be 0xADD8E5 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 '0x00ADD8E5' (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 test controller initiates a write transfer request to the PSoC 3 DoC.
2. Assume that the data to read from register has address as 0xADD8E5.
   a. Shift the APACC instruction into the instruction register.
   b. Shift a '0' (write) followed by '01' (selecting TRNS_ADDR register) followed by '0x00ADD8E5' (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 test controller initiates a read transfer request to the PSoC 3 DoC; the data read from DATA_RW is invalid in this cycle.
   e. Wait at least 5 TCK clock cycles to avoid a WAIT response.
   f. Read the DATA_RW register again. The data is now valid.

2.4 Switching between JTAG and SWD Interfaces

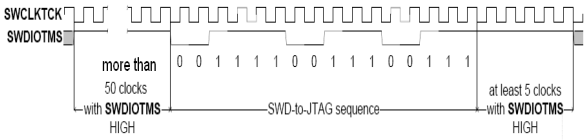
PSoC 3 supports programming through both the SWD and JTAG interfaces. It is also possible to switch from the SWD to JTAG protocol or vice-versa at any time. This switching is done by sending a specific key sequence on the SWDIO/TMS shared pin (referred to as SWDIOTMS) with clock on the TCK/SWDCK shared pin (referred to as SWCLKTCK). This switching is required for JTAG interface programming and is explained in "Step1: Enter Programming Mode" on page 20.

2.4.1 SWD to JTAG Switching

To switch programming interface from SWD to JTAG (4-wire) operation, the steps are as follows:

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 2-6. SWD to JTAG Switching Sequence



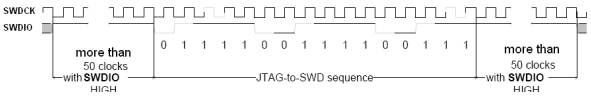
3. Send at least five **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that if programming interface is already in JTAG operation before sending the select sequence, the JTAG TAP enters the Test-Logic-Reset state.

2.4.2 JTAG to SWD Switching

To switch the programming interface from JTAG to SWD operation, the sequence is as follows:

1. Send 51 or more **SWCLKTCK** cycles with **SWDIOTMS** 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 **SWDIOTMS**. 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 2-7. First Three Steps of JTAG to SWD Switching

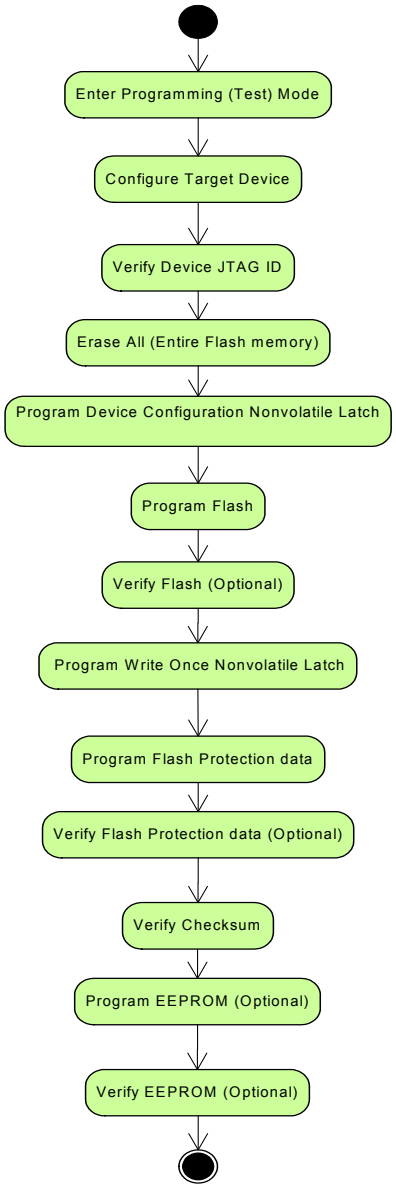


3. Send 51 or more **SWCLKTCK** cycles with **SWDIOTMS** HIGH. This ensures that if the programming spec is already in SWD operation before sending the select sequence, the SWD interface enters line reset state.
4. Send three or more **SWCLKTCK** cycles with **SWDIOTMS** low. This ensures that the SWD 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. There is no need to process the Device ID returned by the PSoC 3 device for this read packet. Ignore the Device ID returned by PSoC 3 in this step.

3. PSoC 3 Programming Flow

Figure 3-1 shows the steps to program a PSoC 3 device. Each step is discussed in detail in the later sections. All steps in the figure must be completed for a successful programming operation. The programming operation should be stopped if there is a failure in any step. The SWD/JTAG programming vectors are provided in SWD and JTAG Vectors for Programming chapter on page 43.

Figure 3-1. PSoC 3 Programming Flow



3.1 Step1: Enter Programming Mode

The first step in programming PSoC 3 is to enter the Programming mode (also known as Test mode) of the device. The host programmer must complete this step successfully for the remaining steps to be successful.

Two hardware dependent factors must be considered when entering programming mode. Select a method based on the hardware capabilities of the host programmer.

- Programming interface used: PSoC 3 supports programming through SWD or JTAG interfaces. The procedure to enter PSoC 3 programming mode depends on the programming interface used.
- Method of resetting PSoC 3 device: The first step to enter PSoC 3 programming mode is to reset the device and send the programming sequence. You can reset the PSoC device by three methods. The first two reset methods are used for SWD interface programming and the third one is used for JTAG interface programming.
  - Device reset (XRES) pin: The host programmer drives the PSoC 3 device reset pin (dedicated XRES pin or P1[2] pin configured as XRES) low.
  - Power cycle mode: The host programmer toggles power to the PSoC 3 power supply pins (Vddd, Vdda, Vddios) to do a device reset.
  - Software reset through JTAG interface: The device reset is done by writing to a specific register bit in PSoC 3 through the JTAG interface. This is a JTAG compliant method of programming PSoC 3 without using XRES pin or power cycle mode.

3.1.1 Enter Programming Mode through SWD Interface

Figure 3-3 shows the steps to enter programming mode (or test mode) of PSoC 3 using SWD interface; Figure 3-2 shows the corresponding timing diagram. See Table 4-3 on page 41 for specifications of timing parameters mentioned in the figures. Figure 3-2 and Figure 3-3 show both XRES method and power cycle mode of programming. Each methods is explained in later sections.

Figure 3-2. Timing Diagram to Enter Test Mode through SWD Interface

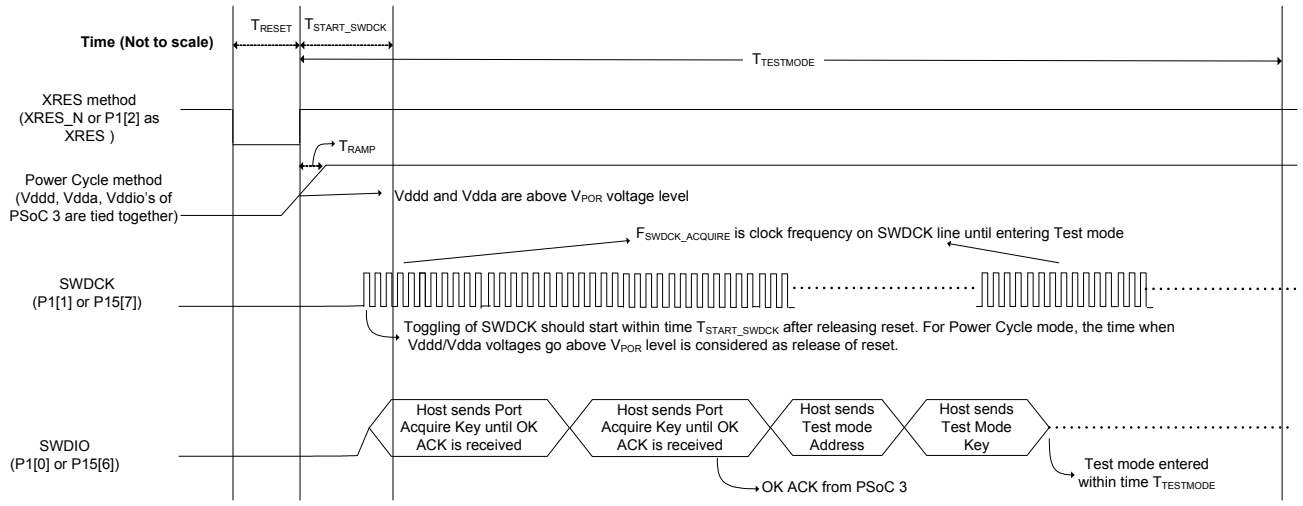
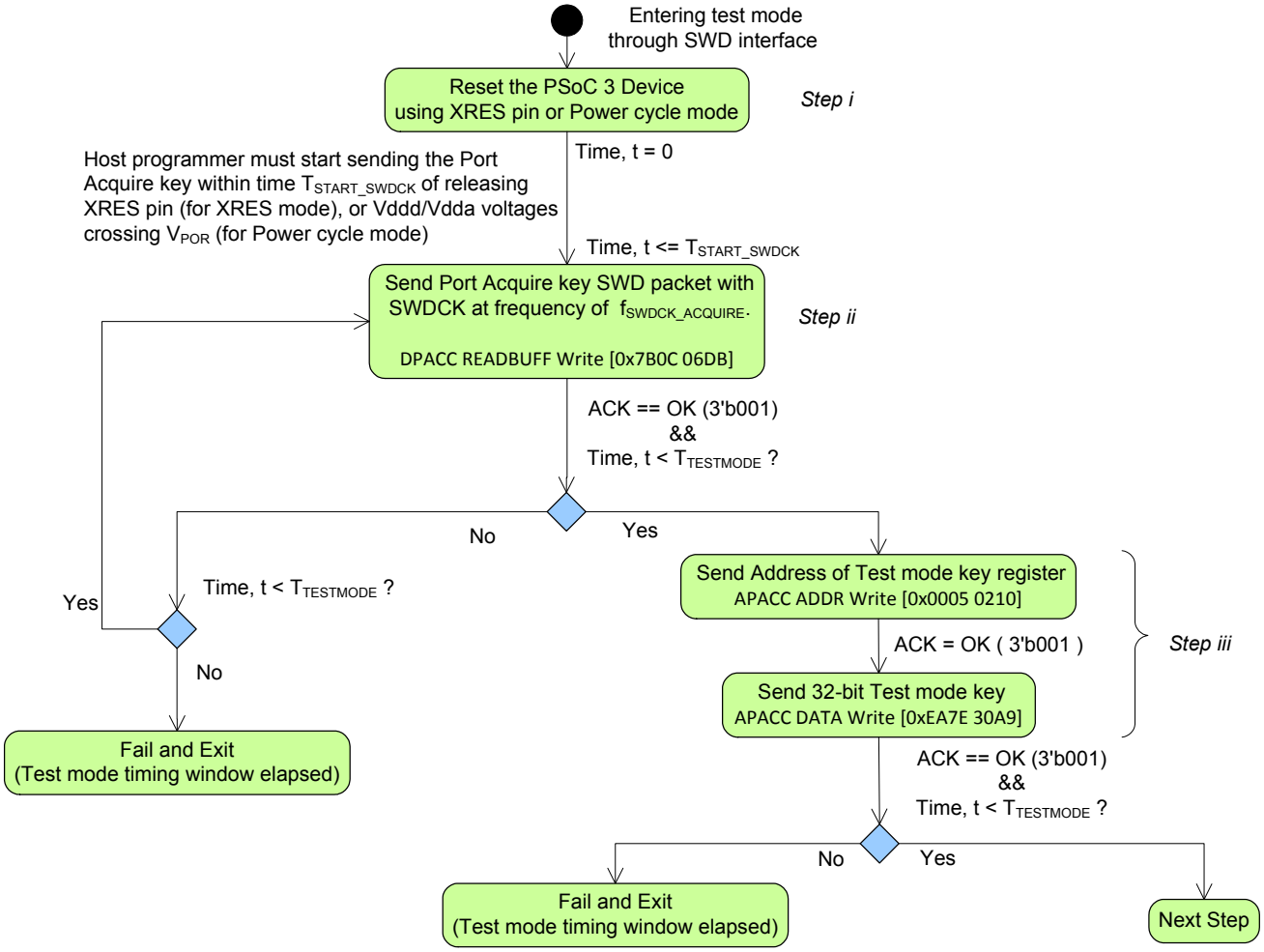


Figure 3-3. Enter Programming Mode through SWD



3.1.1.1 SWD Programming using XRES Pin

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

1. Host programmer drives the XRES pin of PSoC 3 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 the XRES signal, the host must start sending the Port Acquire key on SWDIO, SWDCK lines. The host must send this Port Acquire key continuously until an OK ACK is received from PSoC 3. 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 has been
  //received
} while (ACK != "OK" AND time_elapsed < TTESTMODE)

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

If the debug port is disabled, PSoC 3 ignores the first Port Acquire SWD packet sent after releasing reset. It does not return an OK ACK for first packet. PSoC 3 sends an OK ACK only during the second try of Port Acquire SWD packet. Therefore, the port acquire sequence must be sent continuously on the SWD interface until an OK ACK is received from PSoC 3. 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 and Figure 3-3, the SWDCK frequency during test mode entry is $f_{SWDCK\_ACQUIRE}$. The host programmer must meet this frequency specification to successfully enter PSoC 3 programming mode. After device reset is released, the internal test controller logic in PSoC 3 looks for 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 3 boot to complete after device reset is released. By ensuring that SWDCK line is always clocked at a frequency of $f_{SWDCK\_ACQUIRE}$, the host programmer can meet PSoC 3 test mode entry timing requirements. The host programmer must clock SWDCK line at this frequency with SWDIO held low even between sending of two SWD packets so that the $T_{ACQUIRE}$ time window is not missed. 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 later section.

3. After the host programmer receives an OK ACK for Port Acquire sequence, it must write the test mode key to the Test Mode Key register to enter PSoC 3 programming mode. This key must be written within time $T_{TESTMODE}$, as shown in Figure 3-2 and Figure 3-3. By ensuring that SWDCK is clocked at frequency of $f_{SWDCK\_ACQUIRE}$ during this step, the host programmer can enter PSoC 3 programming mode within time $T_{TESTMODE}$. The pseudo code for this step is given here.

```
APACC ADDR Write [0x0005 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
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 3-3 and Figure 3-2. The only difference is that, instead of driving XRES pin, the host programmer toggles power to the PSoC 3 power supply pins (Vddd, Vdda, Vddio0, Vddio1, Vddio2, and Vddio3) to cause a device reset.

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

- When the optional XRES pin (P1[2]) in 48-pin 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 3. This is because there is no dedicated XRES pin in 48-pin parts unlike the other pin count packages. Note that this condition of disabling P1[2] as XRES for 48-pin parts is done only by the user and not by Cypress. The 48-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. Power cycle method must be used in such a case.
- If it is required to program PSoC 3 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, VBUS power pin in USB interface powers the Vddd, Vdda, Vddio power supply pins in PSoC 3.

Ramp Rate Requirements for Power Cycle Mode Programming

The maximum power supply ramp rate is specified in the PSoC 3 device data sheet 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 power supply ramp duration (T_{RAMP}) from VPOR to final value is less than T_{START_SWDCK}.

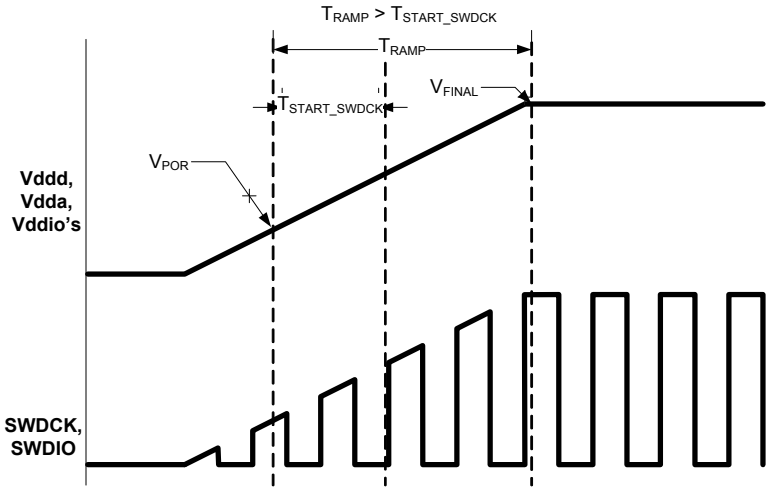
  Figure 3-2 shows that the host programmer must start sending the Port Acquire sequence within time duration T_{START_SWDCK} of Vddd and Vdda voltage levels crossing VPOR voltage level specification. If the time (T_{RAMP})

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, Vddio pins have reached final voltage value.

- When power supply ramp duration (T_{RAMP}) from VPOR to final value 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 3 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 SWDCK and SWDIO lines to match the instant value of 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 3's test controller is able to detect data (logic levels) on the SWDIO and SWDCK lines even when 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 entire SWD packet, as shown in Figure 3-2 and Figure 3-3. A modified method of entering PSoC 3 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 shows the modified steps to enter test mode of PSoC 3; Figure 3-6 shows the corresponding timing diagram. See Table 4-3 on page 41 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 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 and Figure 3-6 requires the programmer to toggle SWDCK alone at frequency of $f_{\mathbf{SWDCK\_ACQUIRE}}$ with SWDIO held low. This ensures that the host programmer meets the initial test mode timing requirements. After time T_{BOOT}, the programmer must send the port acquire and test mode key SWD packets. These SWD packets should be sent within time $T_{TESTMODE}$. An example C code that implements Figure 3-5 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

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

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

XRES_HIGH; /* Release XRES */

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

/* Initial Port Acquire window, 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 */

/* Now send Port Acquire key, Test mode
address, Test mode key SWD packets at fre-
quency of f_SWDCK_BITBANG to complete all steps
within time T_TESTMODE */
```

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

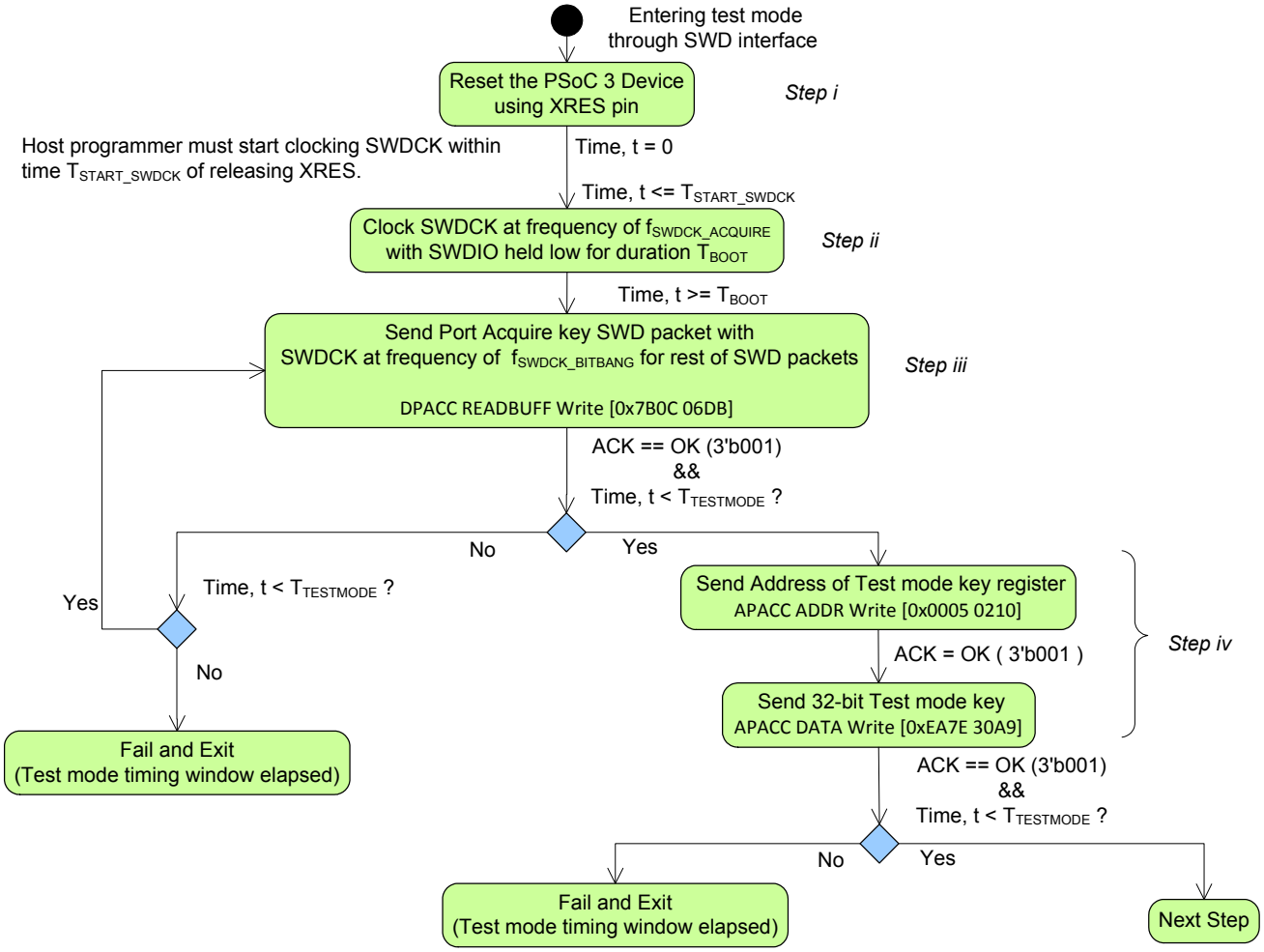
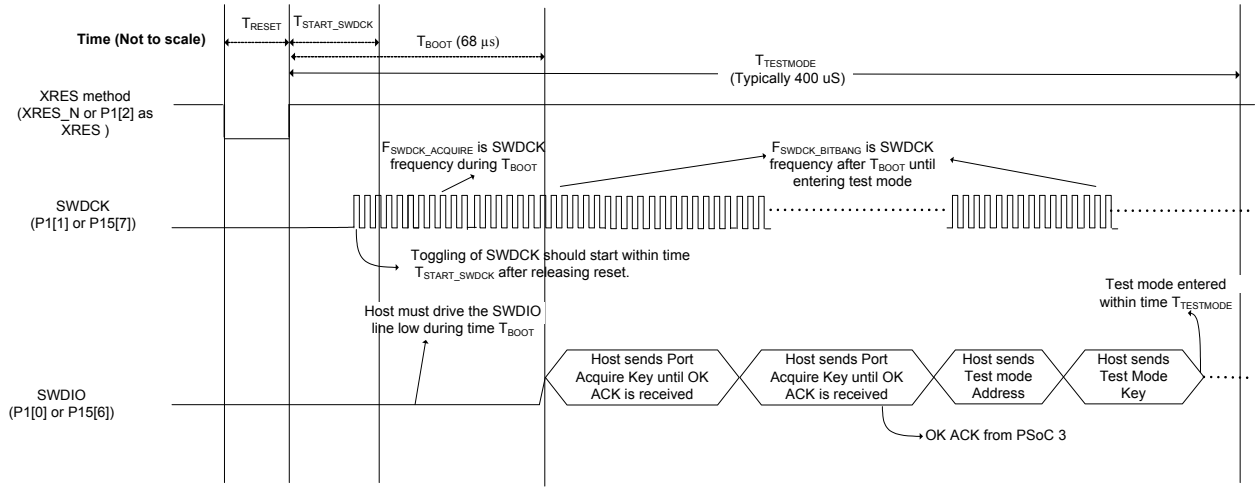


Figure 3-6. Timing Diagram to Enter Test Mode through SWD Interface (for bit banging programmers)



3.1.1.4 Determine $f_{SWDCK\_BITBANG}$:

In Figure 3-5, 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 3, 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}). Minimum value of $T_{TESTMODE}$ from Table 4-3 on page 41 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), and hence 196 SWDCK clock cycles are required for four SWD packets.

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

$f_{SWDCK\_BITBANG}$(no overhead) $\geq$ (1/1.6 µs) $\cong$ 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 SWDCK line during 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 Enter Programming Mode through JTAG Interface

The programming of PSoC3 silicon through JTAG interface can be implemented in one of the following two methods:

- Using JTAG interface (TCK, TMS, TDI, TDO) to communicate with the device, but not be in full compliance with the JTAG standard (IEEE1149.1).
- Full compliance with JTAG standard.

The most significant deviation from the standard of first method is frequency requirement to the JTAG master. It cannot enter PSoC 3 in the programming mode if running at a frequency lower than 1.5 MHz. Another deviation is that this method uses the SWD-to-JTAG switching sequence, which is not defined by the standard (but is ARM's invention).

The second method is implemented in full compliance with the JTAG standard: it is not variable with frequency (on the lower bound) and uses the standard JTAG transactions to go along JTAG's Finite State Machine. This programming method can be implemented by any standard JTAG master. Or if the JTAG tools support the SVF (or STAPL) script interpretation, then to program the device, the JTAG master can play the script representing the hex file content.

Note The second method works only for silicon revision 5 or later, while the first method supports all production revisions. The second method imposes extra requirements on the content of User NVLs: Debug Port Select = "4-/5-wire JTAG" and Debug Enable = "ON". These are default NVL settings from the factory. For the first method, the primary requirement is to have Debug Port Select = "4 -/5-wire JTAG or SWD".

In general, the first method is more universal and is recommended for implementation by JTAG programmers (if possible). It covers all silicon revisions and the JTAG/SWD setting of the Debug Port. To be closer to the standard, the SWD-to-JTAG sequence can be omitted, but in that case the device's NVLs must be configured only to the JTAG mode.

3.1.2.1 JTAG Acquisition Sequence

Figure 3-7 shows the steps to enter programming mode (or test mode) of PSoC 3 using JTAG interface; Figure 3-8 shows the corresponding timing diagram. See Table 4-3 on page 41 for specifications of timing parameters.

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

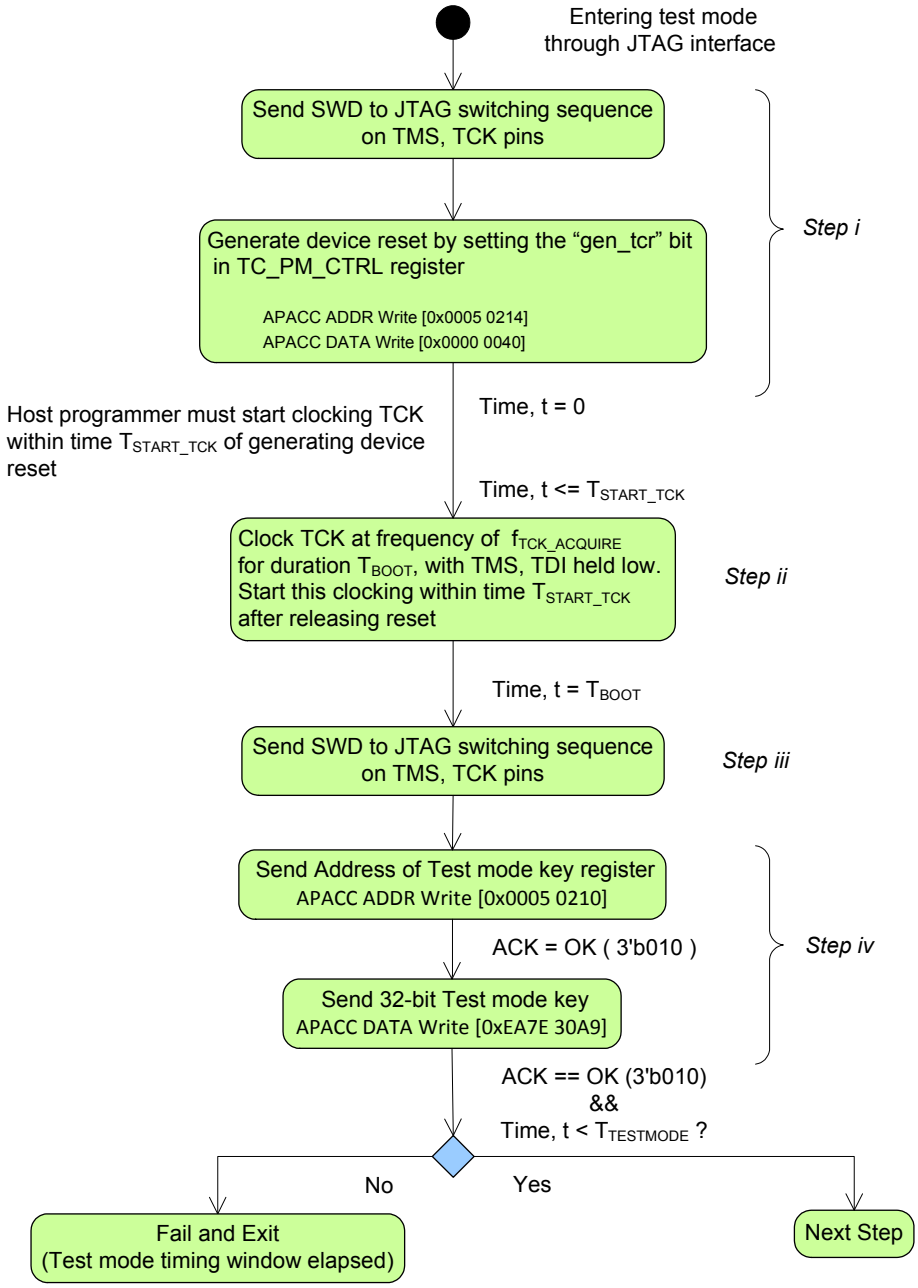
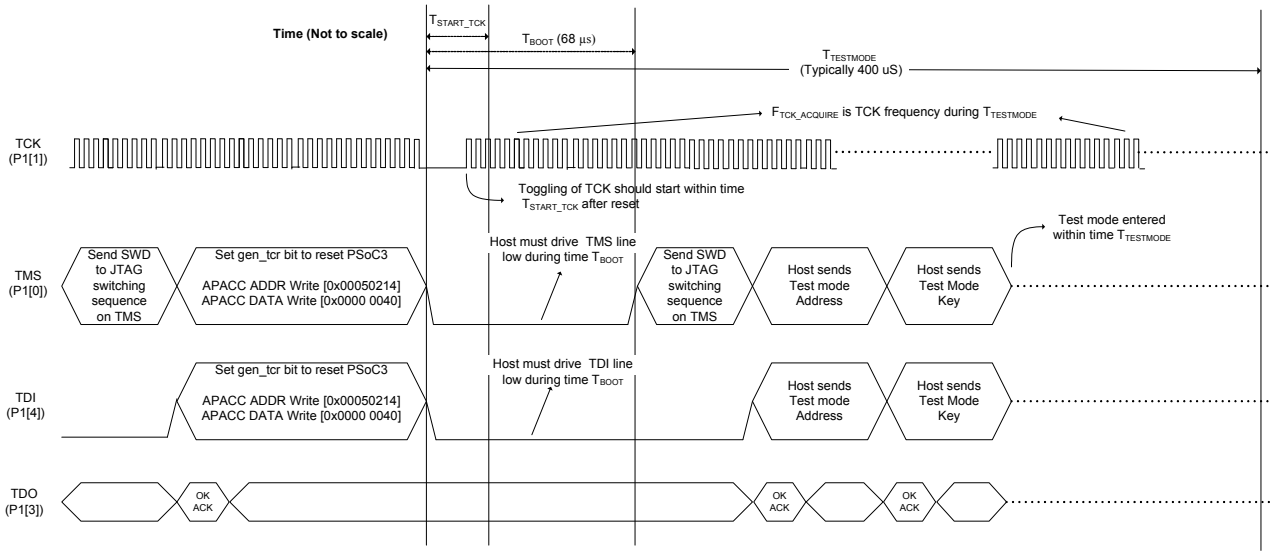


Figure 3-8. Timing Diagram to Enter Test Mode through JTAG Interface



The steps in Figure 3-7 to enter test mode using JTAG interface are as follows:

1. **Reset PSoC 3 device through JTAG interface:**

   For programming through the JTAG interface, PSoC 3 has to be reset through the JTAG interface pins. Using the XRES pin or Power cycle mode to do device reset is not a JTAG compliant programming method. By writing a specific value (0x40) to a register (TC_PM_CTRL), PSoC 3 can be reset through JTAG interface. Refer to the PSoC 3 technical reference manual for details of this register. Writing this value sets the "gen_tcr" bit (Bit 6) of the TC_PM_CTRL register which in turn triggers a device reset.

   Before writing to the TC_PM_CTRL register, a SWD to JTAG switching sequence is sent on TMS, TCK lines as shown in Figure 2-6 on page 17. This ensures that the 4-wire JTAG interface is selected as the debug port, even if DPS NVL setting is SWD interface or 5-wire JTAG. Refer to the description of the Device Configuration non-volatile latch in "Nonvolatile Memory Organization in PSoC 3" on page 74 for details on DPS setting. Devices coming out of factory are configured with DPS setting of "4-wire JTAG". So changing the DPS setting to SWD is a conscious choice made by user. Sending this sequence will help overcome that SWD setting impact on JTAG programming.

2. **Clock TCK line at frequency of $f_{TCK\_ACQUIRE}$ with TMS, TDI lines held low for duration T_{BOOT}:**

   After PSoC 3 is reset through the JTAG interface, the host starts clocking the TCK line at frequency of $f_{TCK\_ACQUIRE}$ for duration T_{BOOT} after reset. The host must start toggling the TCK line within time T_{START_TCK} of device reset. The TMS and TDI lines in PSoC 3 must be held low during this phase. This clocking ensures that the host meets initial timing requirement of "8 TCK clock cycles in time $T_{ACQUIRE}$". When 8 TCK cycles are detected within $T_{ACQUIRE}$ time window, the time window to enter test mode is extended to $T_{TESTMODE}$ as shown in Figure 3-8. The time window $T_{ACQUIRE}$ falls within time T_{BOOT}, the time it takes for PSoC 3 to complete the boot phase.

3. **Send SWD to JTAG switching sequence on TMS and TCK lines:**

   After completing clocking of TCK for time T_{BOOT}, the host must send the SWD to JTAG switching sequence on TMS and TCK lines. This changes the active debug port to 4-wire JTAG even if the DPS setting is SWD interface or 5-wire JTAG.

4. Write Test mode key to the Test mode register to enter programming mode (test mode):

   After sending the SWD to JTAG switching sequence, the host programmer must write the test mode key to test mode key register to enter PSoC 3 programming mode. This key must be written within time $T_{TESTMODE}$. By ensuring that TCK is clocked at frequency of $f_{TCK\_ACQUIRE}$ during this step, the host programmer can enter PSoC 3 programming mode. The pseudo code for this step is as follows.

```
/* Address of the Test mode key register */
APACC ADDR Write [0x0005 0210]

/* Write 32-bit test mode key */
APACC DATA Write [0xEA7E 30A9]

/* Exit on timeout or reception of FAULT
response which means the device did not
enter Programming mode within time T TESTMODE.
```

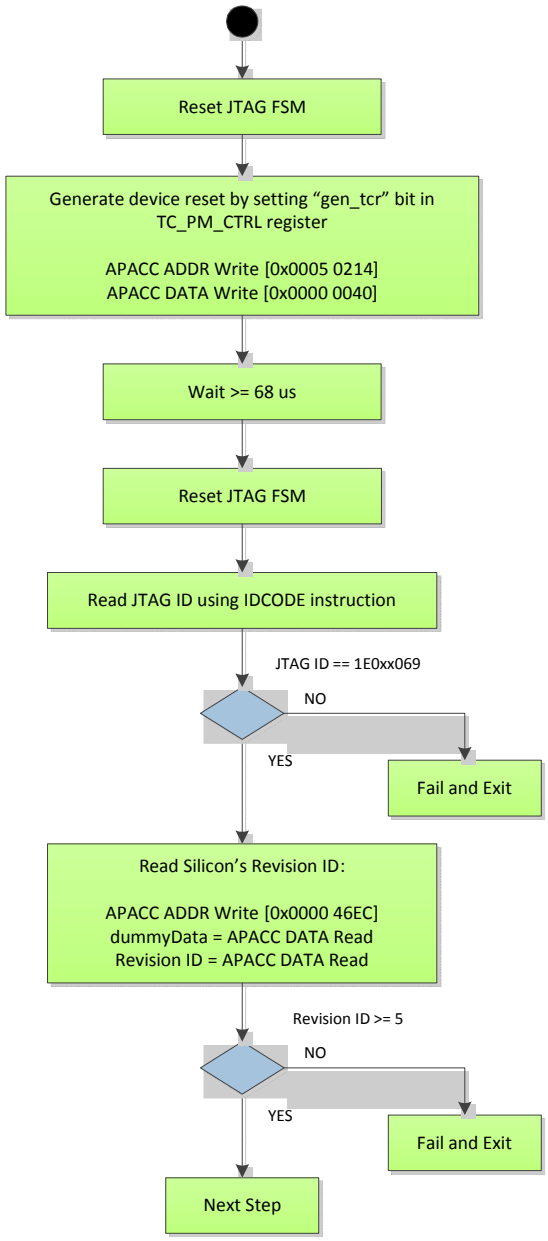
```
Retry again by doing reset and restarting.
*/
    if (ACK != "OK" OR time_elapsed > T_TESTMODE
    usec) then FAIL_EXIT
```

Note The programming method for JTAG interface that is shown in Figure 3-7 and Figure 3-8 will not work if the DPS setting is Debug Port Disabled. The third-party programmer implementing PSoC 3 JTAG programming should ensure that the DPS is never programmed with this setting. The location of DPS setting in hex file is in the Appendix chapter on page 71. This setting is not expected for JTAG interface programming. The programmer software can throw an error message and abort operation if a hex file with Debug Port Disabled setting attempts to program on PSoC 3. The default device factory setting for DPS is "4-wire JTAG".

3.1.2.2 *JTAG Compliant Entry in Programming Mode*

Figure 3-9 shows the steps to enter the programming mode using the JTAG compliant method. The verification of the JTAG ID and Revision ID ensures that the correct device is acquired for programming.

Figure 3-9. JTAG Compliant Entry in Programming Mode



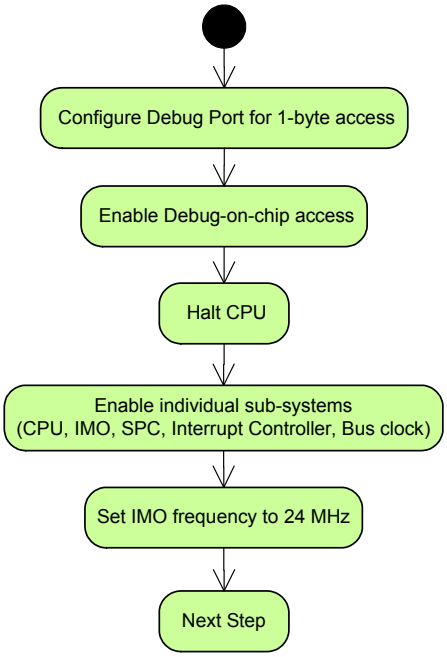
Notes for Figure 3-9:

1. **Reset JTAG FSM** – Moves Test Access Ports of all devices on the JTAG chain into the Reset state. This is a synchronization step.
2. **Device Reset** – This is the software reset of the PSoC 3 device (analog to toggling the XRES pin). This step is required if the ECC bit is changed in the NVLs during programming. However, in general Cypress recommends implementation of reset because it synchronizes the programmer and the target.
3. **Wait for ≥ 68 µs** – This is a mandatory step required by the silicon boot mechanism. It ensures that the boot is completed before accessing the target through the SWD/JTAG bus. It is recommended to have this delay in the millisecond range (for example, 1 ms).
4. **Verifying JTAG ID** – The IDCODE instruction must be set in the JTAG's instruction register. After that, the 32-bit JTAG ID must be shifted out of the data register. Ensure that the PSoC 3 device is a target before proceeding.
5. **Checking Revision ID** – Ensure that the current silicon's revision is ≥ 5. Previous revisions cannot be programmed in the JTAG compliant way.

3.2 Step 2: Configure Target Device

Figure 3-10 shows the sequence to configure the target PSoC 3 device before programming.

Figure 3-10. Configuring Target PSoC 3 Device

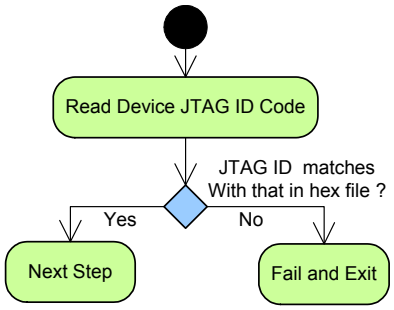


3.3 Step 3: Verify JTAG ID

To ensure that the target device corresponds to the device for which the hex file is meant, the JTAG ID of the target device must be compared against the JTAG ID information in the hex file. This ensures that hex file is completely compatible with device under test (DUT). If there is a mismatch in the JTAG IDs, the programming operation should be stopped. See "Intel Hex File Format" on page 71 for information on the location of JTAG ID in the hex file.

JTAG ID supports a separate instruction IDCODE to read the IDCODE register that contains the device JTAG ID. JTAG ID ca be read using the SWD interface with a packet request containing ADDR = 00, APnDP = 0, and RnW = 1.

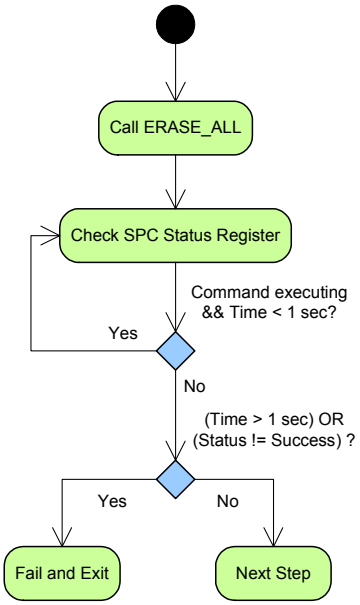
Figure 3-11. Verify Device ID of Target Device



3.4 Step 4: Erase Flash

Figure 3-12 demonstrates the Erase Flash process, which erases all flash data and configuration bytes, and all flash protection rows.

Figure 3-12. 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 address 32'h4720. 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 Organization in PSoC 3" on page 74. For more information on nonvolatile memory programming, refer to the PSoC 3 Architecture TRM.

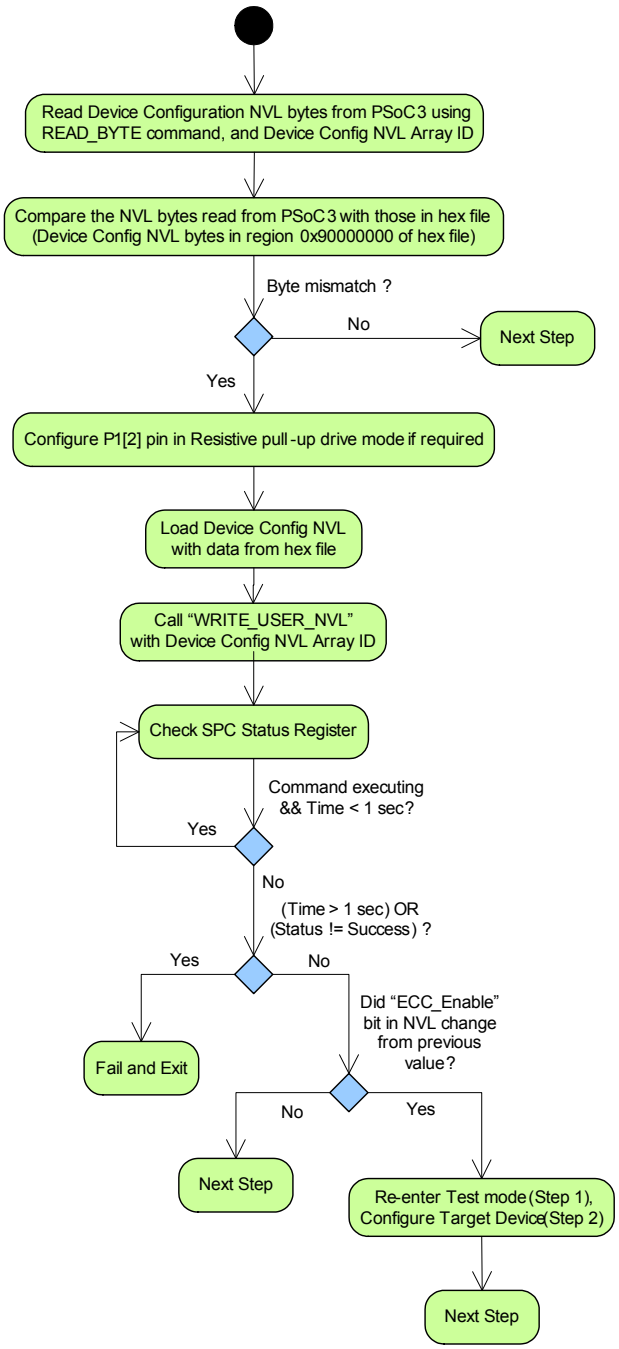
The ERASE_ALL command should not take longer than 1 second, otherwise an overtime error occurs.

3.5 Step 5: Program Device Configuration NVL

Figure 3-13 shows the Program Device Configuration Nonvolatile Latch (NVL) setup flow. This step writes the 4-byte Device Configuration NVL. 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 programming the device configuration nonvolatile latches, P1[2] pin must be in resistive pull-up drive mode if it has to function as an XRES pin. The P1[2] pin can either be a GPIO pin or an XRES pin as determined by the NVL settings. If the hex file NVL data has configuration that sets P1[2] as XRES, then the algorithm checks the current NVL port drive mode setting. If that setting is not in resistive pull-up mode, the P1[2] pin drive mode registers are set for resistive pull-up mode. This ensures that the active low P1[2] reset pin is held high by default.

Figure 3-13. Program Device Configuration NVL



The NV latches in PSoC 3 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 3 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 row is of size 256 bytes (if ECC is enabled) or 288 bytes (if ECC is disabled).

During the programming process, if the ECC feature is enabled, the row latch needs to be loaded with only 256 bytes of data. This data is in the region 0x00000000 of the hex file. The 32 ECC bytes are automatically calculated and loaded into the remaining 32 bytes of the row latch. This step needs to be done to program all flash rows.

During the programming process, if the ECC feature is disabled, the row latch needs to be loaded with all the 288 bytes. In this scenario, the 256 data bytes should be fetched from the main flash data region of the hex file at address 0x80000000. 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 two parameters to consider in the flash programming process.

- Number of rows (N) of flash memory: The value of 'N' depends on the flash memory size of the target device. For example, a 64 KB flash memory device has 256 rows [(64K/256) = 256 rows]. Note that the maximum size of flash memory in the PSoC 3 family is only 64 KB. Therefore, the entire flash memory is contained in a single flash array (a flash array is a physical organization of flash memory in a block. The maximum capacity of a single flash array is 64 KB). Also, the flash size parameter does not consider the size of ECC bytes. For example, a 64 KB flash size indicates that the main flash region capacity is 64 KB. It does not include the ECC bytes; ECC region is used only for configuration data and not for code space.

- 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

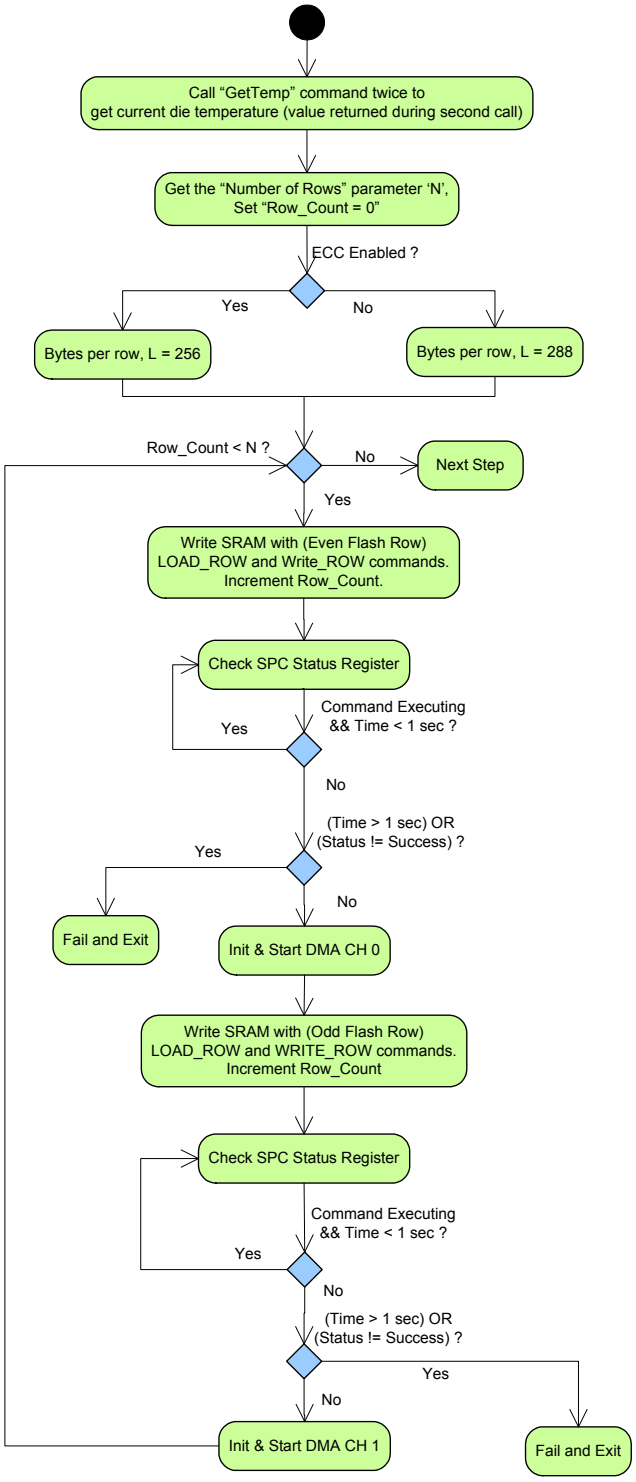
  L = 288 bytes, if ECC is disabled

  The ECC enabled/disabled setting is stored in bit 3 of byte 3 of device configuration NVL. This byte is stored in address 0x 90000003 of hex file.

Figure 3-14 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. See SWD and JTAG Vectors for Programming chapter on page 43 for more details on the command implementation.

The direct memory access (DMA) feature in PSoC 3 can speed up the flash programming 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 3-14. Program Flash



3.7 Step 7: Verify Flash (Optional)

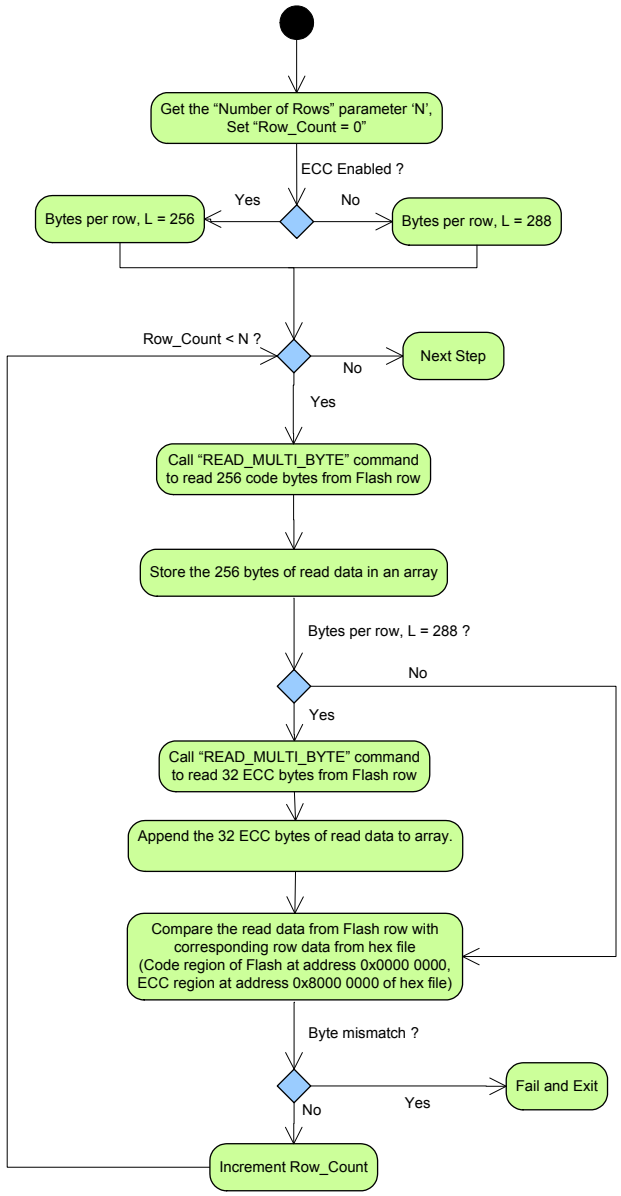
Figure 3-15 demonstrates the flash read process. This optional step allows reading back and verifying data programmed in the Program Flash step.

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 disabled, the 32 bytes of configuration data should also 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.

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 mismatch in even one of the bytes, the programming process should be aborted and restarted.

Note that in the hex file, the code region in flash row (256 bytes) starts at address 0x00000000. If ECC is disabled, the configuration bytes for flash rows start at the address 0x80000000 of the hex file. If ECC is disabled, 256 bytes from the main code data region (0x00000000) and 32 bytes from the ECC region (0x80000000) must be concatenated to form a flash row. If ECC is enabled, only 256 bytes from the main code data region (0x00000000) are needed to form a flash row.

Figure 3-15. Verify Flash Sequence



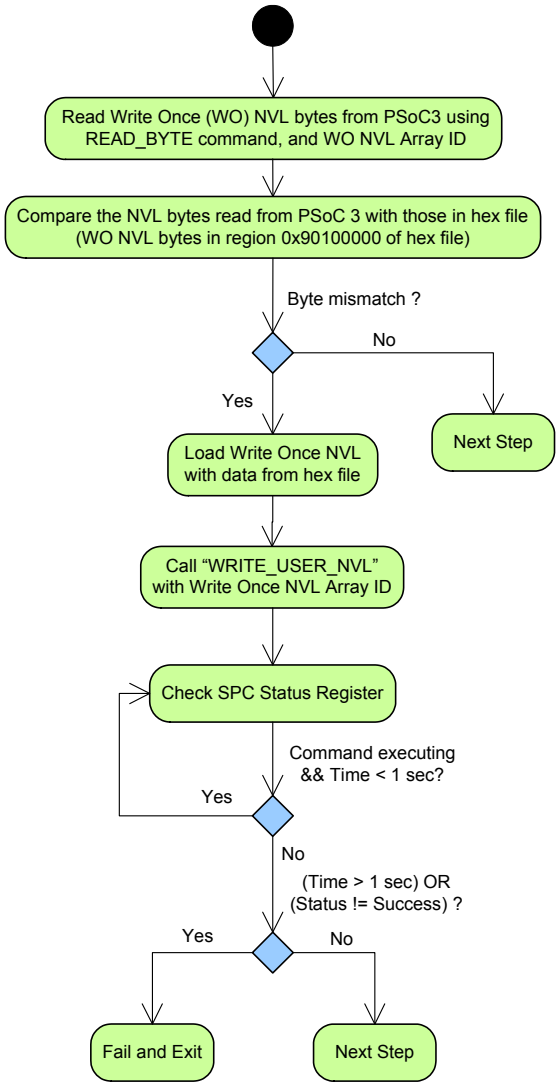
3.8 Step 8: Program WO NVL

Figure 3-16 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 checkbox; by default, it should be cleared. See Nonvolatile Memory Organization in PSoC

3 chapter on page 74 for details on the Device Security feature that is supported by 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 3-16. Program Write Once NVL

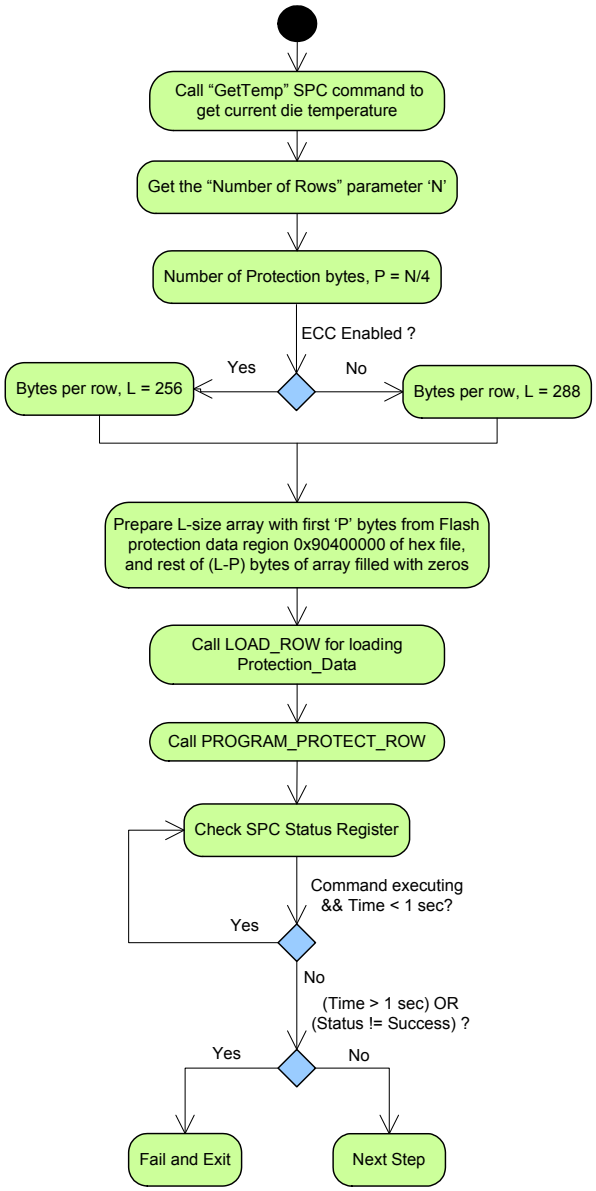


The NVLs in PSoC 3 have much lesser endurance compared to flash and EEPROM memory. Due to this, the write-once NVL is written only if new data needs to be programmed into the latch. This ensures that the latches are programmed only when there is change in the 4-byte security key in the hex file, which in turn maximizes the endurance time.

3.9 Step 9: Program Flash Protection

Figure 3-17 shows the sequence to program the protection rows in flash.

Figure 3-17. Program Flash Protection Sequence



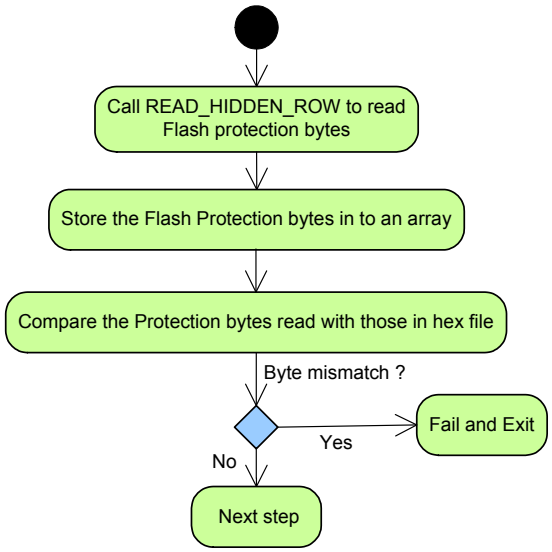
The protection rows start in address 32'h90400000 in the hex file, as shown in "Intel Hex File Format" on page 71. In this step, commands LOAD_ROW and PROGRAM_PROTECT_ROW are called to program flash protection data. Similar to "Step 6: Program Flash" on page 32, the Get Temp command is called initially 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. PSoC 3 can have a maximum of 256 flash rows and 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.

3.10 Step 10: Verify Flash Protection (Optional)

Figure 3-18 explains the flash protection data verification procedure. This step is optional and allows reading back and verifying the data programmed in the Program Flash Protection step.

Figure 3-18. Verify Flash Protection Sequence

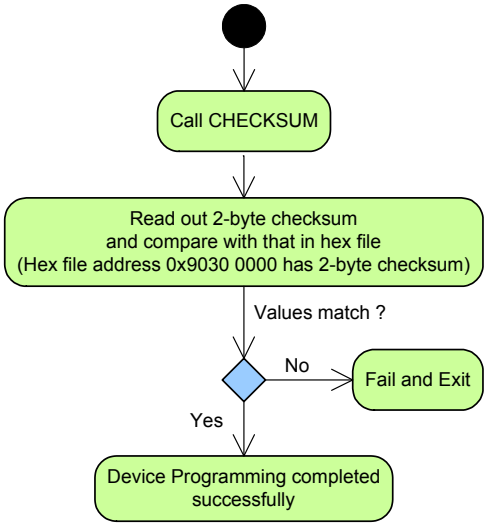


The READ_HIDDEN_ROW command is used to read out all bytes in the Flash Protection row. This command always returns 256 bytes irrespective of the ECC setting and the number of valid flash protection bytes. Each protection byte stores protection settings of four flash rows. PSoC 3 can have a maximum of 256 flash rows and a maximum of 64 flash protection bytes. The remaining bytes returned by the READ_HIDDEN_ROW command should be ignored during the verification step.

3.11 Step 11: Validate Checksum

Figure 3-19 demonstrates the checksum validation step. This step validates that the programming operation is successful. The CHECKSUM command is used to compute and return the checksum value, which can be read out through the data register at 32'h00004720. The checksum is a 4-byte value, so four SWD or JTAG 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 in the hex file and those read from the device mismatch, terminate the programming process. In the hex file, the 2-byte checksum of all flash rows (including the configuration bytes in each row if ECC is disabled) is stored at address 0x90300000 of the hex file (MSB first). This is explained in "Intel Hex File Format" on page 71.

Figure 3-19. Checksum Validation Sequence



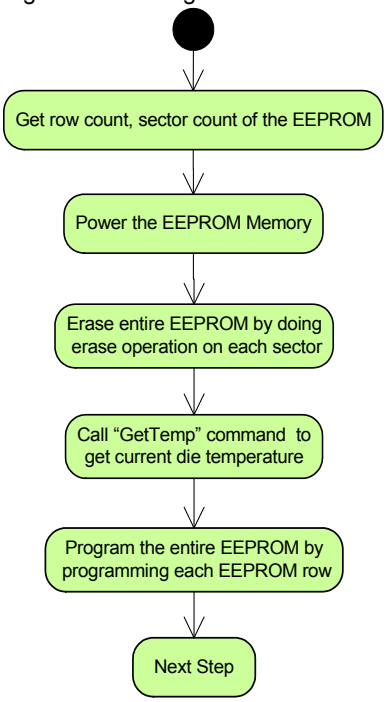
3.12 Step 12: Program EEPROM (Optional)

EEPROM nonvolatile memory in PSoC 3 is used to store constant data such as calibration data and look-up table. Some applications might require the EEPROM memory in PSoC 3 to be initialized as part of the device programming sequence. The programmer software can provide a configuration option to the end user to select whether or not 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.

Figure 3-20 shows the sequence to program the EEPROM memory in PSoC 3. First, the EEPROM memory needs to be powered to do any read/write operation on the EEPROM.

Then, the EEPROM memory is erased by doing a sector erase operation. A sector is a physical partition in EEPROM array and each sector can have up to 64 rows. The size of each row in EEPROM is 16 bytes. So a 2 KB EEPROM will have 128 rows organized as two sectors. The number of rows and sectors in the EEPROM memory of the PSoC 3 device can be calculated based on the EEPROM memory size in bytes given in the device datasheet. After erasing all the sectors in the EEPROM, the EEPROM is programmed row wise with the programming data coming from the EEPROM region of the hex file, as explained in A.1 Intel Hex File Format. Before programming EEPROM, 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.

Figure 3-20. Program EEPROM



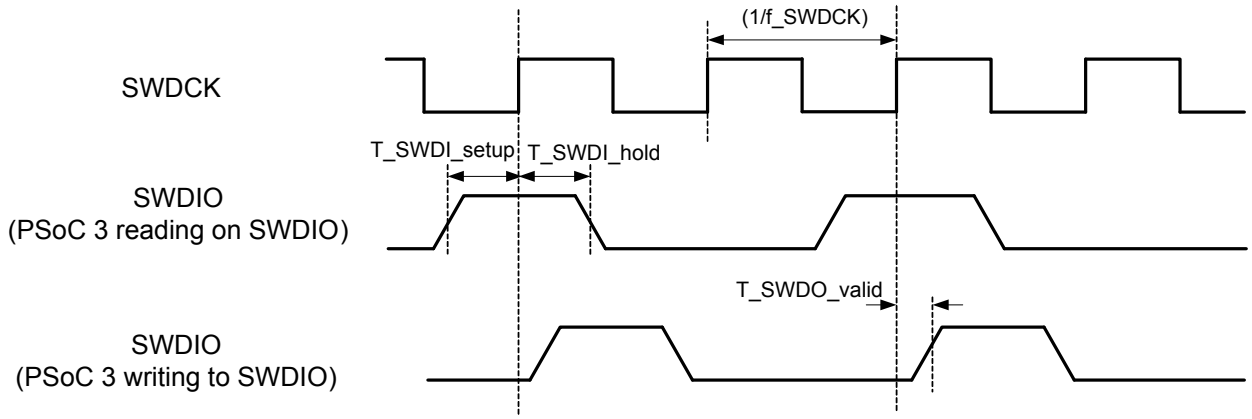
3.13 Step 13: Verify EEPROM (Optional)

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 On-Chip (DoC) interface. To speed up the verification process, the DoC is configured for 4-byte access so that one read operation fetches four bytes of EEPROM data. 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 SWDIO line on falling edge of SWDCK. PSoC 3 does the corresponding write or read operations on SWDIO on rising edge of SWDCK.

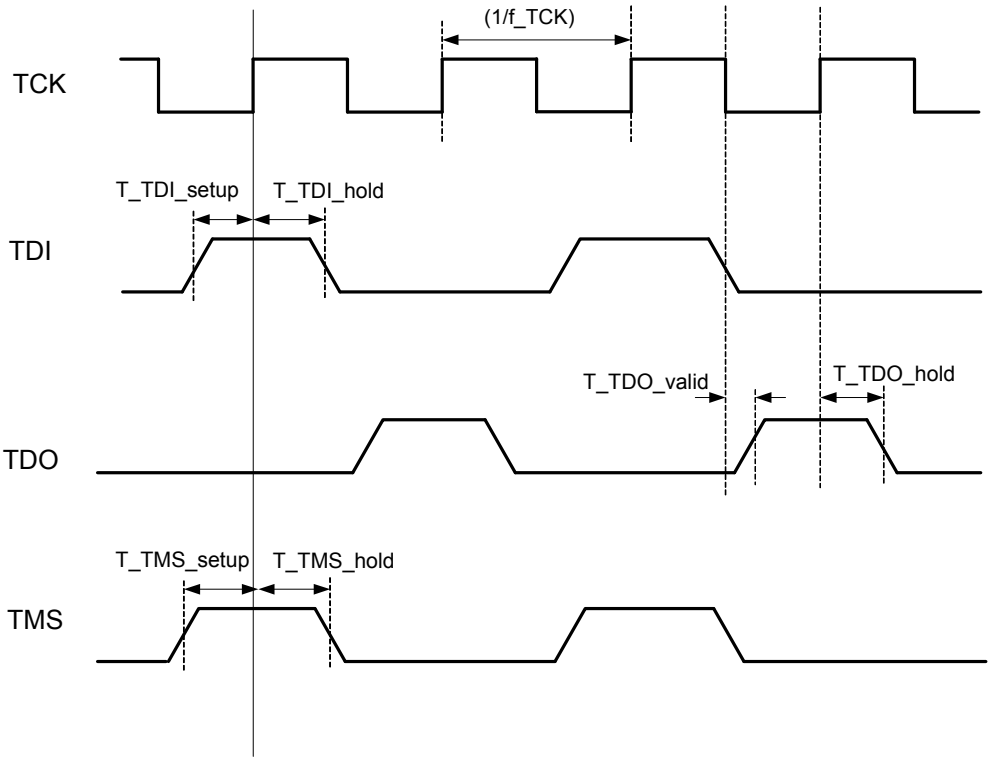
Table 4-1. SWD Interface AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
f_SWDCK	SWDCLK frequency	3.3 V ≤ VDDD ≤ 5 V	–	–	8[a]	MHz
		1.71 V ≤ VDDD < 3.3 V	–	–	7	MHz
		1.71 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 data sheet specification as the CPU clock frequency is configured for a fixed frequency of 24 MHz in the programming algorithm; 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 3 reads data on its TMS and TDI lines on the rising edge of TCK. The host should write to TMS and TDI pins of PSoC 3 on the falling edge of TCK. PSoC 3 writes to its TDO line on the falling edge of TCK. The host should read from the TDO line of PSoC 3 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, 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 data sheet specification as the CPU clock frequency is configured for fixed frequency of 24 MHz in 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 3 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}	Maximum time from release of device reset to start of TCK 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	1.4	–	f_SWDCK max	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. The minimum frequency is assuming no overhead or delay between SWD packets.	0.7	–	f_SWDCK max	MHz
$f_{TCK\_ACQUIRE}$	TCK clock frequency during Port Acquire, Test mode entry	f_TCK max is from Table 4-2	1.4	–	f_TCKmax	MHz
V_{POR}	Vddd, Vdda rising trip voltage		1.64	–	1.68	V

See the PSoC 3 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 programming procedure; the timing requirements are specified in Table 4.3 on page 41. Depending on the programming interface used, the appropriate method to enter PSoC 3's programming mode should be used from the following methods. A separate method is provided for bit banging programmers that need to program PSoC 3 through SWD interface. Detailed information on all these methods are provided in "Step1: Enter Programming Mode" on page 20.

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 3-3, Table 4-3 */
```

Step i.) Do device reset using XRES pin or 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 [0x0005 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

else NEXT_STEP /* Entered PSoC 3 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, Table 4-3 */
```

Step i.) Do device reset using XRES pin

```
time_elapsed = 0
```

Step ii.) Clock SWDCK at frequency of $f_{SWDCK\_ACQUIRE}$ for time T_{BOOT}. SWDIO pin of PSoC 3 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 = T_BOOT
```

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 < T_TESTMODE)//Check port acquire retry time

if (ACK != "OK" OR time_elapsed > T_TESTMODE) 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 [0x0005 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

else NEXT_STEP /* Entered PSoC 3 Programming mode */
```

5.1.3 Method C

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

Step i.) Do device reset through JTAG interface by sending SWD to JTAG switching sequence and then writing to a specific register

```
a.)Send SWD to JTAG switching sequence on TCK/TMS pins

b.)APACC ADDR Write [0x0005 0214] //Address of TC_PM_CTRL register
   APACC DATA Write [0x0000 0040] //Set the "gen_tcr" bit to generate reset
```

```
time_elapsed = 0
```

Step ii.) Clock TCK at frequency of $f_{TCK\_ACQUIRE}$ for time T_{BOOT}. TMS, TDI pins of PSoC 3 should be driven low by the host during time T_{BOOT}. Host should start clocking TCK within time T_{START_TCK} of doing device reset.

```
time_elapsed = T_BOOT
```

Step iii) Send the SWD to JTAG switching sequence on TMS, TCK pins after time T_{BOOT}. TCK frequency during this step should be $f_{TCK\_ACQUIRE}$.

Step iv) Send JTAG commands for entering test mode. TCK frequency during this step should be $f_{TCK\_ACQUIRE}$. This step should be completed within time $T_{TESTMODE}$ as given below.

```
APACC ADDR Write [0x0005 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

else NEXT_STEP /* Entered PSoC 3 Programming mode */
```

5.1.4 Method D

```
/* Entering Programming mode in JTAG compliant way */
/* Based on Test mode entry flowchart given in Figure 3-9 */
/* This method works only for silicon revision 5 (TO6) or later */
    a) Reset JTAG FSM

    b) Generate Software Reset for PSoC3

       APACC ADDR Write [0x0005 0214]   //Address of TC_PM_CTLR register
       APACC DATA Write [0x0000 0040]   //Set "gen_tcr" bit for reset

    c) Wait 1 ms //delay for at least 68 us

    d) Reset JTAG FSM

    e) Read and Verify JTAG ID of target

       JTAG ID = IDCODE Read                      //Read from IDCODE JTAG's register
       if (JTAG ID != 0x1E0xx069) then FAIL_EXIT; //check for PSoC3

    f) Read and Check Revision ID of target

       APACC ADDR Write [0x0000 46EC]        //Revision ID register
       dummyData = APACC DATA Read
       Revision ID = (APACC DATA Read) & 0xFF
       if (Revision ID < 5) then FAIL_EXIT; //check for revision
       else NEXT_STEP;                      //Correct device recognized
```

5.2 Step 2: Configure Target Device

```
/* Setting DP configuration register for one-byte access. */
DPACC DP CONFIG Write [0x0000 0000]

APACC ADDR Write [0x0005 0220]
APACC DATA Write [0x0000 00B3]     // Halt CPU, Enable Debug-on-chip (DoC) access

APACC ADDR Write [0x0000 46EA]
APACC DATA Write [0x0000 0001]    // Halt CPU

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

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

5.3 Step 3: Verify JTAG ID

```
/* Compare the 4-byte JTAG ID in Hex file(exp_idcode) at address 0x90500002 of hex file with
the JTAG ID read from device in this step. Abort programming operation if JTAG ID's mismatch.
4-byte JTAG ID in hex file is in Big-endian format. Refer Appendix-A for details */
//The following code shows how to read ID in SWD mode. For JTAG mode set JTAG's instruction
register to IDCODE and read ID from JTAG's data register.
if (DPACC IDCODE Read != exp_idcode) then FAIL_EXIT // Exit on JTAG ID mismatch
```

5.4 Step 4: Erase All (Entire Flash memory)

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

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

APACC ADDR Write [0x0000 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 [0x0000 4722]// SPC status register address
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
} while ((StatusReg != [0x02]) 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 [0x0000 4720]
    APACC DATA Write [0x0000 00B6] // First initiation key

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

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

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

    APACC ADDR Write [0x0000 4720]
    APACC DATA Write [ByteRead]    //Byte number of User NVL to be read

    // Poll status register bit till data is ready
    APACC ADDR Write [0x0000 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
    } 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 [0x0000 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 [0x0000 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
```

```
    } 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)
```

```
/* Pull-up is disabled on P1[2] now. Configure P1[2] for resistive pull-up drive mode,
output high state */
if (pullupEnable == 1)
{
     byte PinState
     APACC ADDR WRITE [0x0000 500A]    // Address of PRT1_PC2 register
     byte dummy = APACC DATA READ      //Dummy SWD Read
     PinState = APACC DATA READ          // Store the current pin configuration

     /* Configure the pin for resistive pull up mode, data high */
     PinState = (PinState & 0xF0) | (0x05)

     /* Write the modified value to the register */
     APACC DATA WRITE [PinState]
}
byte AddrCount = 0
while (AddrCount < 4)
{
     APACC ADDR Write [0x0000 4720]// Write to command data register
     APACC DATA Write [0x0000 00B6]// First initiation key

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

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

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

     APACC ADDR Write [0x0000 4720]
     APACC DATA Write [AddrCount]// Current address: 0 - 3

     APACC ADDR Write [0x0000 4720]
     APACC DATA Write [0x0000 00XX]     // Replace XX with data located in

                                              // (0x90000000 + AddrCount) of .hex file

   time_elapsed = 0
   APACC ADDR Write [0x0000 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
   } 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 [0x0000 4720]
APACC DATA Write [0x0000 00B6] // Call WRITE_USER_NVL command

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

```
    APACC ADDR Write [0x0000 4720]
    APACC DATA Write [0x0000 0006]// WRITE_USER_NVL opcode

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

    time_elapsed = 0
    APACC ADDR Write [0x0000 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
    } 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 0x00000000 and 0x80000000 of the hex file. See “Step 6: Program Flash” on page 32 for details on definition of the parameters ‘L’ and ‘N’ used in code as follows. N is the number of flash rows and L is the number of bytes per row.)

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

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

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

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

    //Wait until Temperature data is ready
    APACC ADDR Write [0x0000 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
    } while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

    if (time_elapsed > 1 sec) then FAIL_EXIT

    APACC ADDR Write [0x0000 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 [0x0000 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
        } 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 */
/*********************************************************************/

/* Setting DP configuration register for four-byte access to SRAM. */
DPACC DP CONFIG Write [0x0000 0004]

//Program Rows
int Row_Count = 0 //Variable to keep track of Row being programmed
int Byte_Count = 0 //Variable to keep track of byte number in a row

while (Row_Count < N)// See "Step 6:Program Flash" section for 'N'
{
    //-------------Programming EVEN ROW ----------------------

    //"B6" - SPC_KEY1, "D5" - SPC_KEY2, "02" - LOAD_ROW opcode, "00" - Flash ArrayID
     APACC ADDR Write [0x0000 0000]// SRAM address- 32'h00
     APACC DATA Write [0x0002 D5B6]// 4 byte data

     Byte_Count = 0

     //Send Row data to SRAM from HEX
     while (Byte_Count < L)    // Define L according to ECC settings
     {
        APACC ADDR Write [Byte_Count + 0x4]

        /* 4-bytes (d3d2d1d0) are from hex file starting at address (address of d0):
        i.) if Byte_Count < 256: Address of do = (0x00000000 + (Row_Count * 256) +
        Byte_Count)
        ii.) if 256 <= Byte_Count < 288: Address of do = (0x80000000 + (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]
        Byte_Count = Byte_Count + 4
     }

     //"00","00","00" - 3 NOPs for short delay, "B6" - SPC_KEY1
     APACC ADDR Write [(L - 1) + 0x05]
     APACC DATA Write [0xB600 0000]

    //"DA" - SPC_KEY1+SPC_PRG_ROW, "07" - SPC_PRG_ROW, "00" - Flash Array ID
    //"00" - High Byte of RowCount
     APACC ADDR Write [(L - 1) + 0x09]
     APACC DATA Write [0x0000 07DA]
```

```
//Low byte of row number, Temperature data
 APACC ADDR Write [(L - 1) + 0xD]
 APACC DATA Write [(0x00 << 24) | (Temp_Magnitude << 16) | (Temp_Sign << 8) |
                   (RowCount & 0xFF)]
 //DMA operations

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

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

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

 APACC ADDR Write [0x0000 7604]// PHUB_CFGMEM0_CFG1 register
 APACC DATA Write [0x0000 0000] // Sets upper 16-bit address of destination/source

 APACC ADDR Write [0x0000 7800] //PHUB_TDMEM0_ORIG_TD0 register
 APACC DATA Write [(0x01FF 0000) + L + 15] // Set TD transfer counts

 APACC ADDR Write [0x0000 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 [0x0000 4722]// Poll status register
 dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
 time_elapsed = 0
 do
 {
      StatusReg = (byte) APACC DATA Read // Save status register value
 } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)
 if (time_elapsed > 1 sec) then FAIL_EXIT

 APACC ADDR Write [0x0000 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, "00" - ArrayID
 APACC ADDR Write [0x0000 0200]// SRAM address 32'h200
 APACC DATA Write [0x0002 D5B6]// 4-byte data as commented above

//Send Row to SRAM from HEX

 Byte_Count = 0
 while (Byte_Count < L)//Define L according to ECC settings
 {
```

```
      APACC ADDR Write [Byte_Count + 0x204]

    /* 4-bytes (d3d2d1d0) are from hex file starting at address (address of d0):
    i.) if Byte_Count < 256: Address of do = (0x00000000 + (Row_Count * 256) +
    Byte_Count)
    ii.) if 256 <= Byte_Count < 288: Address of do = (0x80000000 + (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 [(L - 1) + 0x205]
 APACC DATA Write [0xB600 0000]

//"DA" - SPC_KEY1+SPC_PRG_ROW, "07" - SPC_PRG_ROW, "00" - Flash Array ID
//"00" - High Byte of RowCount
 APACC ADDR Write [(L - 1) + 0x209]
 APACC DATA Write [0x0000 07DA]// 0xDA = 0xD3 + 0x07 ( "WRITE_ROW" opcode)

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

//DMA operations

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

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

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

 APACC ADDR Write [0x0000 760C]// PHUB_CFGMEM1_CFG1 register
 APACC DATA Write [0x0000 0000]// Sets upper 16-bit address of destination/source

 APACC ADDR Write [0x0000 7808]//PHUB_TDMEM0_ORIG_TD0 register
 APACC DATA Write [(0x01FF 0000) + L + 15]  // Set TD transfer counts

 APACC ADDR Write [0x0000 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 [0x0000 4722]
 dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
 time_elapsed = 0
 do         // Poll status register
 {
      StatusReg = (byte) APACC DATA Read // Save status register value
 } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)
```

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

     APACC ADDR Write [0x0000 7024]  // PHUB_CH1_ACTION register
     APACC DATA Write [0x0000 0001] // This creates a direct DMA request Channel '1'

     // DMA will transfer data from SRAM, and call LOAD_ROW and then WRITE_ROW

     Row_Count = Row_Count + 1
}

//Make sure that last SPC request is completed
APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct
time_elapsed = 0
do               // Poll status register
{
  StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

//Resetting DP configuration register for one-byte access after Flash programming
DPACC DP CONFIG Write [0x0000 0000]
```

5.7 Step 7: Verify Flash (Optional)

See "Step 6: Program Flash" on page 32 for details on definition of the parameters 'L' and 'N' used in the following code. N is the number of flash rows and L is the number of bytes per row.

```
int RowCount = 0 //Variable to keep track of flash rows that have been read
int byte_index = 0 //Variable to keep track of number of bytes read in a Flash row
byte StatusReg //To store SPC_SR status register value
byte Data_Array[L] //Array of size 'L' bytes to store one row of data read from device
int32 address
// Iterate through all rows of flash
while (RowCount < N)
{
     address = RowCount * 256   //Starting address of Flash row

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

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

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

     APACC ADDR Write [0x0000 4720]
     APACC DATA Write [0x0000 0000]// Array ID

     APACC ADDR Write [0x0000 4720]
     APACC DATA Write [(address >> 16) & 0xFF]//MSB byte2 of 3-byte address

     APACC ADDR Write [0x0000 4720]
     APACC DATA Write [(address >> 8) & 0xFF]//Byte1 of 3-byte address
```

```
APACC ADDR Write [0x0000 4720]
APACC DATA Write [(address >> 0) & 0xFF]//LSB Byte0 of 3-byte address

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

//Wait until Data is ready

APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
     StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x0000 4720]
dummyByte = APACC DATA Read // Dummy SWD read

// Read 256 bytes of row data in to Data_Array
int ByteRead = 0
while (ByteRead <= 0x0000 00FF)
{
     Data_Array[byte_index] = APACC DATA Read// Save read data in to array
     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 [0x0000 4720]
     APACC DATA Write [0x0000 00B6]  //First initiation key

     APACC ADDR Write [0x00004720]
     APACC DATA Write [0x0000 00D7] //0xD7= (0xD3 + READ_MULTI_BYTE opcode)

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

     APACC ADDR Write [0x0000 4720]
     APACC DATA Write [0x0000 0000] // Array ID

     APACC ADDR Write [0x0000 4720]
      APACC DATA Write [(address >> 16) & 0xFF] //MSB Byte 2 of 3-byte address;

     APACC ADDR Write [0x0000 4720]
```

```
        APACC DATA Write [(address >> 8) & 0xFF] //Byte 1 of 3-byte address


        APACC ADDR Write [0x0000 4720]
        APACC DATA Write [(address >> 0) & 0xFF] //LSB Byte 0 of 3-byte address

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

        //Wait until Data is ready
        APACC ADDR Write [0x0000 4722]
        dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
        time_elapsed = 0
        do
        {
           StatusReg = (byte) APACC DATA Read // Save status register value
        } while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

        if (time_elapsed > 1 sec) then FAIL_EXIT


        APACC ADDR Write [0x0000 4720]
        dummyByte = APACC DATA Read // Dummy SWD read

        ByteRead = 0
        /* Number of ECC bytes per row is 32 */
        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 (+ECC data if applicable).
      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. */

   RowCount = RowCount + 1; // Next Flash row

}
```

5.8 Step 8: Program Write Once Nonvolatile Latch

Warning: Programming the device with correct security key is an irreversible process; perform this step only if all prior steps passed without errors. This 4-byte data is located in address 0x90100000 of the hex file.

```
/* The NV Latches have a lesser endurance, and hence 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 a byte mismatch */

byte ByteRead = 0  //Variable to track number of bytes that are read
byte StatusReg //To store SPC_SR status register value
byte Data_Array[4] //4-byte array to store the NVL data read from device
```

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

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

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

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

    APACC ADDR Write [0x0000 4720]
    APACC DATA Write [ByteRead]    //Byte number of Write Once NVL to be read

   // Poll status register bit till data is ready
    APACC ADDR Write [0x0000 4722]
   dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
   do
   {
      StatusReg = (byte) APACC DATA Read // Save status register value
   } 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 [0x0000 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
    time_elapsed = 0
    APACC ADDR Write [0x0000 4722]// SPC status register address
    dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status

    do
    {
         StatusReg = (byte) APACC DATA Read// Save status register value
    } 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 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 the following 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 [0x0000 4720]// Write to command data register
         APACC DATA Write [0x0000 00B6]// First initiation key

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

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

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

         APACC ADDR Write [0x0000 4720]
         APACC DATA Write [AddrCount]// Byte index in "Write Once NVL"

         APACC ADDR Write [0x0000 4720]
         APACC DATA Write [0x0000 00XX]        /* Replace XX with data located in
                                               (0x90100000 + AddrCount) of .hex file */
         time_elapsed = 0
         APACC ADDR Write [0x0000 4722]
         dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
         do           // Poll status register
         {
            StatusReg = (byte) APACC DATA Read // Save status register value
         } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

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

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

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

    APACC ADDR Write [0x0000 4720]
    APACC DATA Write [0x0000 00D9]// SPC_KEY2 + SPC_USER_NVL

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

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

    time_elapsed = 0
    APACC ADDR Write [0x0000 4722]
```

```
    dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
    do   // Poll status register
    {
         StatusReg = (byte) APACC DATA Read  // Save status register value
    } while ((StatusReg != [0x02]) 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. See "Step 6: Program Flash" on page 32 for details on definition of the parameters 'L' and 'N' used in the following code. N is the number of flash rows and L is the number of bytes per row.

```
byte protectionSize = N/4 //Each Flash protection byte stores protection data of 4 Flash rows

byte StatusReg //To store SPC_SR status register value

//Get the die temperature and store it in "Sign, Magnitude" bytes
/*********************************************************************/

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

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

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

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

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

//Wait until Temperature data is ready
APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
   StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x0000 4720]
dummyByte = 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 [0x0000 4722]// Poll status register
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
```

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


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

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 00D5] // Second initiation key: 0xD3 + 0x02

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

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0000]//Flash Array ID

int ByteCount = 0
while (ByteCount < L)
{
     APACC ADDR Write [0x0000 4720]

     if (ByteCount < protectionSize)
     {
       APACC DATA Write [0x0000 00XX]//Data at address (32'h90400000 + ByteCount) of 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 [0x0000 4720]
APACC DATA Write [0x0000 00B6] // First initiation key

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

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

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0000] //Flash array ID

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0000] //Row select value is always zero for protection data
```

```
APACC ADDR Write [0x0000 4720]
APACC DATA Write [Temp_Sign] //Send Sign byte of die temperature

APACC ADDR Write [0x0000 4720]
APACC DATA Write [Temp_Magnitude] //Send Magnitude byte of die temperature

APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do     // Poll status register
{
     StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

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

5.10 Step 10: Verify Flash Protection Data (Optional)

See “Step 6: Program Flash” on page 32 for details on definition of the parameter ‘N’ used in the following code. N is the number of rows in a flash array. PSoC 3 has only one flash array.

```
byte NumberOfProtectionBytes = N/4; //Each protection byte corresponds to 4 Flash rows
int byte_index = 0 //Variable to keep track of number of bytes read

/* Array to store the protection bytes read from PSoC3. Even though the maximum number of
protection bytes is only 64 for a 64 KB Flash memory, 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. Even
if ECC is disabled, only 256 bytes need to be read in case of reading protection rows */
byte Data_Array[256];
byte  StatusReg; //Variable to store the SPC status register value

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

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

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

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0000]// Flash Array ID

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

//Wait until Data is ready, and also the command status code is success
APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do
{
   StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

APACC ADDR Write [0x0000 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 64 KB flash memory, 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. Even if ECC is
disabled, only 256 bytes need to be read in case of reading protection rows */
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 "NumberOfProtectionBytes" 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 < NumberOfProtectionBytes)
{
   /* hexData[i] is from address (32'h90400000 + i) 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 */
```

5.11 Step 11: Validate 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.

```
byte  StatusReg; //Variable to store the SPC status register value
byte  b1, b2, b3, b4; //Variables to store the Checksum read from the device

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

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

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

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0000] //Flash array ID

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


APACC ADDR Write [0x0000 4720]
```

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

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0000] //Number of rows (higher byte)

APACC ADDR Write [0x0000 4720]
APACC DATA Write [N - 1]   //Number of rows (lower byte) minus one

APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do     // Poll status register
{
     StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

APACC ADDR Write [0x0000 4720]
dummyByte = APACC DATA Read// Dummy SWD read
b4 = (byte) APACC DATA Read       // Checksum byte 4 (MSB byte)
b3 = (byte) APACC DATA Read       // Checksum byte 3
b2 = (byte) APACC DATA Read       // Checksum byte 2
b1 = (byte) APACC DATA Read      // Checksum byte 1 (LSB byte)
read_checksum = [0xb2b1]// Save lower 2 bytes of checksum to a local variable

APACC ADDR Write [0x0000 4722]
dummyByte = APACC DATA Read   //Dummy SWD Read, Next Read gives correct status
time_elapsed = 0
do        // Poll status register till SPC is IDLE
{
     StatusReg = (byte) APACC DATA Read // Save status register value
} while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)

if (time_elapsed > 1 sec) then FAIL_EXIT

/* Compare with 2-byte checksum value in hex file (big endian format) at address 0x9030 0000.
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 (read_checksum != hexfile_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.

```
/* Get the number of rows, sectors in EEPROM based on the EEPROM memory size information in
the device datasheet. Each row has 16 bytes, and each sector can have a maximum of 64 rows */

byte NumofRows, NumofSectors

/* EEPROM_SIZE_IN_BYTES is given in the device datasheet */
NumofRows = EEPROM_SIZE_IN_BYTES / 16

if(NumofRows % 64 == 0)
{
       NumofSectors = NumofRows/64
}
```

```
else
{
       NumofSectors = (NumofRows/64) + 1
}

/* Power the EEPROM memory before doing any operations */
APACC ADDR Write [0x0000 43AC]
APACC DATA Write [0x0000 0011]

/* Erase the entire EEPROM before doing a program operation by using sector erase commands on
each sector */

byte SectorCount = 0

while(SectorCount < NumofSectors)
{
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [0x0000 00B6]/* First SPC Key */
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [0x0000 00DB]/* Second SPC Key = 0xD3 + 0x08 */
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [0x0000 0008]/* Erase Sector Opcode */
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [0x0000 0040]/* EEPROM Array ID */
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [SectorCount]/* EEPROM Sector Number */
       /* 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 [0x0000 4722]/* SPC status register address */
       /* Dummy SWD Read, Next Read gives correct status */
       byte dummy = APACC DATA Read
       byte StatusReg/* To store SPC_SR status register value */
       time_elapsed = 0
       do
       {
              StatusReg = (byte) APACC DATA Read /* Save status register value */
       } while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)
       if (time_elapsed > 1 sec) then FAIL_EXIT
       SectorCount = SectorCount + 1/* Next Sector */
}
/* Start of "Get_Temp" routine to get Die temperature. Used for EEPROM programming */
byte Temp_Sign, Temp_Magnitude /* To store the temperature data */

byte StatusReg/* To store SPC_SR status register value */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 00B6]/* SPC_KEY1 */
APACC ADDR Write [0x0000 4720]
```

```
APACC DATA Write [0x0000 00E1]/* SPC_KEY2 + SPC_GET_TEMP (0xD3+0x0E) */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 000E]/* SPC_ GET_TEMP opcode */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0003]/* Number of samples, valid values [1..5] */
/* Wait until Temperature data is ready */
APACC ADDR Write [0x0000 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 */
}while ((StatusReg != [0x01]) AND time_elapsed < 1 sec)
if (time_elapsed > 1 sec) then FAIL_EXIT
APACC ADDR Write [0x0000 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 is magnitude of temperature */
/* Wait for IDLE - just in case. Must be in idle state once data byte is read */
APACC ADDR Write [0x0000 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 */
} 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 temperature value received
is stored in Temp_Magnitude, Temp_Sign, and used in below programming step */
/* 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 [0x0000 4720]
       APACC DATA Write [0x0000 00B6]/* First SPC Key */
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [0x0000 00D5]/* Second SPC Key = 0xD3 + 0x02 */
       APACC ADDR Write [0x0000 4720]
       APACC DATA Write [0x0000 0002]  /* Load Row Opcode */
       APACC ADDR Write [0x0000 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 [0x0000 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 [0x0000 4722]/* SPC status register address */
byte dummy = APACC DATA Read/* Dummy SWD Read */
byte StatusReg/* To store SPC_SR status register value */
time_elapsed = 0
do
{
       StatusReg = (byte) APACC DATA Read /* Save status register value */
} while ((StatusReg != [0x02]) AND time_elapsed < 1 sec)
if (time_elapsed > 1 sec) then FAIL_EXIT
APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 00B6]/* First SPC Key */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 00DA]/* Second SPC Key = 0xD3 + 0x07 */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0007]/* Program Row Opcode */

APACC ADDR Write [0x0000 4720]
APACC DATA Write [0x0000 0040]/* EEPROM Array ID */
APACC ADDR Write [0x0000 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 [0x0000 4720]
APACC DATA Write [RowCount]/* LSB byte of the 2-byte row number */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [Temp_Sign] /* Temperature Sign byte */
APACC ADDR Write [0x0000 4720]
APACC DATA Write [Temperature_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 [0x0000 4722]/* SPC status register address */
byte dummy = APACC DATA Read/* Dummy SWD Read */
byte StatusReg/* To store SPC_SR status register value */
time_elapsed = 0
```

```
        do
        {
                StatusReg = (byte) APACC DATA Read/*Save status register */
        } while ((StatusReg != [0x02]) 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 */
/* Configure DAP for 4-byte access for faster verification */
DPACC DP CONFIG Write [0x0000 0004]

/* 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 3. 0x00008000 is EEPROM base address */
                read_address = 0x00008000 + (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 */
}
/* Resetting DP configuration register for one-byte access after verify operation */
DPACC DP CONFIG Write [0x0000 0000]
```

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 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, EEPROM data, flash protection 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 data sheet or the Device Selector menu in PSoC Creator to know the flash memory size of different part numbers.

0x8000 0000 – Flash Configuration Data: PSoC 3 devices have an error correcting code (ECC) feature, which is used to correct and detect bit error in main flash data. There is one ECC byte for every eight bytes of flash data. Thus, there are 32-bytes of ECC data for each row of flash. There is an option to use the ECC memory to store configuration data if you do not want the error correcting feature. The ECC enable bit in the device configuration NV latch (bit 3 of byte 3) can be checked to confirm if the ECC is enabled. This NV latch data byte is present at address 0x90000003. PSoC Creator generates this section of the hex file only if the ECC option is disabled. If this section is present in the hex file, the data needs to be appended with the flash code data during the flash programming step. For every 256 bytes in the code region of flash, 32 bytes from this section are appended. The last address of this section depends on the device flash memory capacity. A device with 64 KB flash memory has 8 KB of configuration memory. In this case, the last address is 0x80001FFF.

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 3" on page 74 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 WO NVL 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 3 Architecture TRM for details on this device security feature.

0x9020 0000 – EEPROM: PSoC 3 devices have on-chip EEPROM memory and the data to be programmed into the EEPROM is stored in this region. EEPROM is programmed row wise where each row contains 16 bytes. Because each record in the EEPROM region of the hex file contains 64 bytes of data, each record has the data corresponding to four 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 64 KB flash (with 256 rows) has 64 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	JTAG 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, you should 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.

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

Silicon Revision: This 1-byte value is for the different revisions of the silicon. For the same manufacturing part number, there are different revisions of the silicon such as ES1, ES2, and ES3. Production PSoC 3 devices also have the same revision number as ES3. For PSoC 3, the revision IDs are as follows:

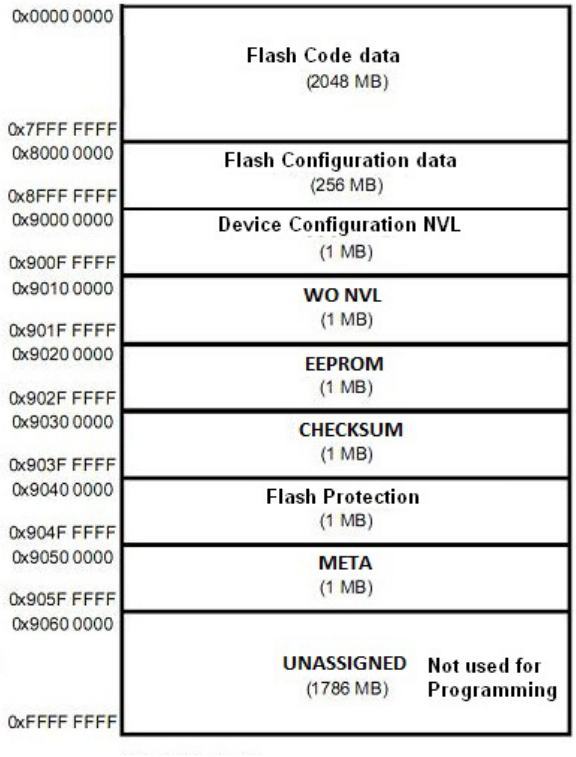
0 - ES1; 1 - ES2; 2 and above - ES3

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 the 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 3 Hex File Address Map



A.2 Nonvolatile Memory Organization in PSoC 3

PSoC 3 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. Note that programming EEPROM using external programmer is not defined in this document. Refer to the "Memory" section of the PSoC 3 Architecture 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 3 Architecture 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 that 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 3 flash memory has the following features:

- Organized in rows, where each row contains 256 code bytes plus 32 bytes for either error correcting codes (ECC) or configuration data storage. Flash memory can be programmed in resolution of rows.
- Organized as one block (Array) of 64, 128, or 256 rows depending on flash size. For a 64 KB flash memory size, the maximum number of rows in PSoC 3 is 256. Flash memory size refers only to the code space and not the configuration region size (ECC region used as configuration data).
- 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. PSoC 3 has a maximum of 64 protection bytes because the maximum flash memory size is 64 KB (64 KB = 256 rows).

A.2.4.2 EEPROM

PSoC 3 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 3 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 Table A-4.

Table A-4. Device Configuration NVL Register Map

User NVL byte	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					PRT15RDM[1:0]	
0x03	DIG_PHS_DLY[3:0]				ECCEN	DPS[1:0]		CFGSPEED

Note The DEBUG_EN and DPS[1:0] are highlighted in blue to emphasize their importance for the programming interface.

Table A-5 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 3 Architecture TRM for more details.

Table A-5. 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
DEBUG_EN	For external programmer it defines if access to Debug subsystem (DoC) is enabled or disabled.	0 – disabled 1 – enabled (default)
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.
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 3 device data sheet for NVL specifications.

There are four 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. Though not recommended, JTAG programming will work even if the DPS setting is SWD. 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.
- **Debug Enable setting:** This bit is only available in the PSoC 3 devices of revision 5 or later. Its value is crucial for compatibility with third-party tools and compliance with the JTAG standard. If it is set, then the external programmer can access all I/O registers of the silicon and thus execute the programming algorithm. In addition, ensure that DPS is set to SWD or JTAG; if not, access to the Debug Port will not be available and the Debug Enable setting will not have any effect.
- **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 3 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.

A.2.4.4 Write Once Nonvolatile Latches (WO NVL)

You 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 an OTP (One Time Programmable) 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 be exercised with extreme caution considering these effects.

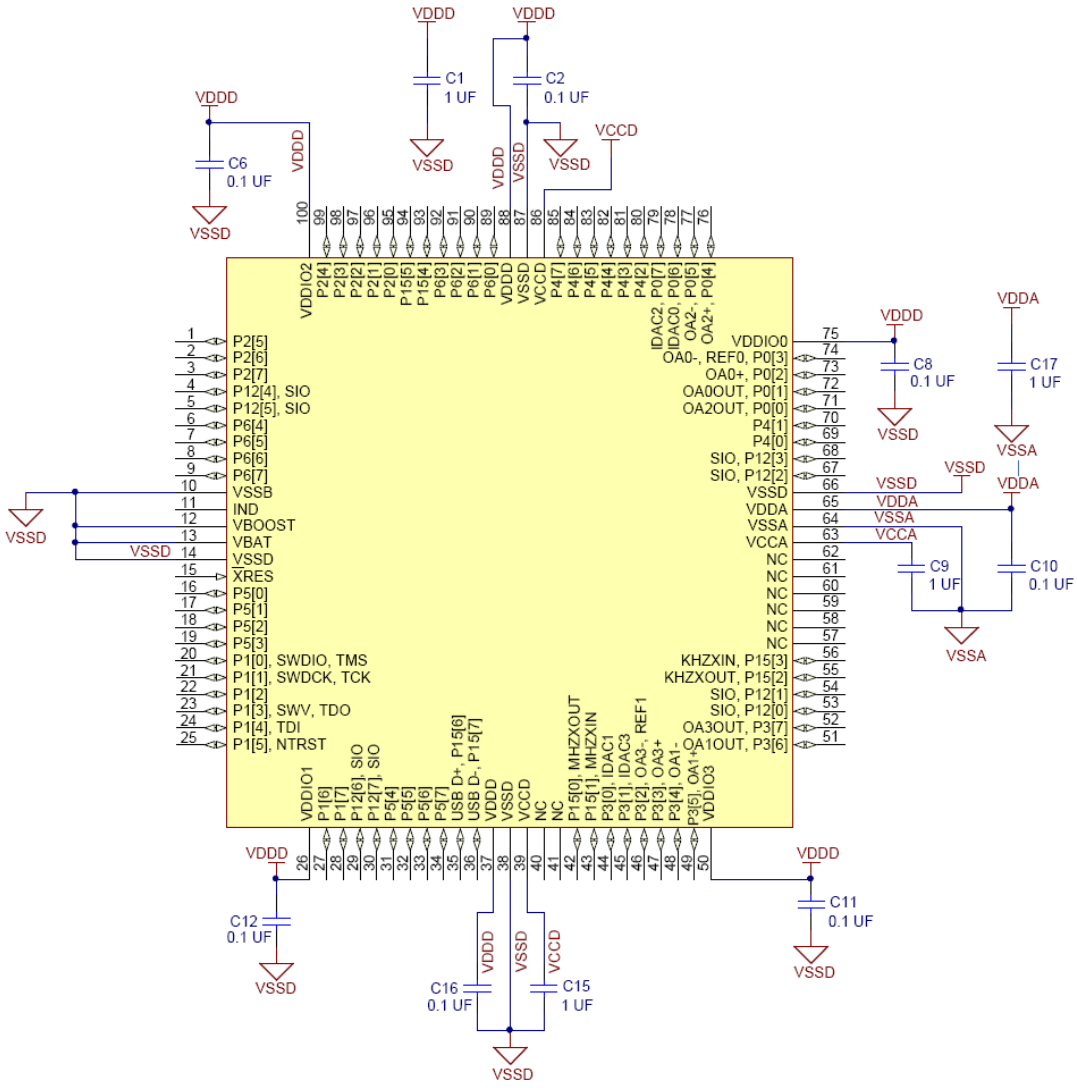
A.3 Example Schematic

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 3 packages; however, the pinout will vary for each package. See the PSoC 3 device data sheet for information on specific package pinout and for specifications on power supply pins. Note that Figure A-4 does not show the programming connections between the host programmer and PSoC 3. This is illustrated in Figure 1-1 on page 6.

Figure A-4 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-4. 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.