Application Note

Using an Embedded Controller (EC) to Load a Patch Bundle Directly to the TPS25751 or TPS26750

Chris Sterzik, Roy Chou

ABSTRACT

The TPS25751 and TPS26750 are configurable USB Type-C® and Power Delivery (PD) controllers optimized for applications supporting power. In general, these PD controllers use I2Cc to load a binary image from an external EEPROM during boot. If an embedded controller (EC) is available, then the EC can be used in place of an EEPROM to load the image to the PD through the I2Ct interface. The technical reference manuals (1 and 2) demonstrate the flow for writing the image to the PD over the I2Ct Bus to multiple PD controllers at the same time. This application note describes this flow in greater detail and focuses on writing the image to one PD controller. A step-by-step explanation for the PTCH mode to APP mode transition by using an I2C command through I2Ct is provided along with example code.

Table of Contents

Trademarks

1 Introduction

In some applications, an embedded controller (EC) is required to interact with the PD controller during the application run time (APP mode) or potentially there is an EC in the system performing house-keeping activities. To reduce cost, the EEPROM can be removed and the function of storing and updating the PD with the binary image can be moved to the EC. This application note focuses on using an EC to load an image and move the PD controller from PTCH mode to APP mode with four character code (4CC) commands. Before discussing the EC, a brief description of the dead battery configuration is provided in the context of an application without an EEPROM.

2 ADCINX Setting

The ADCIN1 input pin determines the PD controller target address while the ADCIN2 input pin defines the dead battery configuration. A complete description can be found in the device data sheet; see 3 and 4. The term dead battery is taken from PC notebook applications when the battery is dead and the requirement is that the PD controller enables the power path without any interaction (or power) to charge the dead battery from VBUS. The PD controllers offer two different dead battery configurations[1]: AlwaysEnableSink and SafeMode. SafeMode provides USB Type-C sink capability, but does not enable the sink path. In SafeMode, the PD state machine is effectively disabled until the binary image is loaded. AlwaysEnableSink enables the power path when a USB Type-C connection is made (the port partner must be a source) and the USB PD functionality remains disabled until the configuration is loaded.

The steps described in this application note are independent of the dead battery configuration. The handling of the dead battery flag and operation of the system by the EC are beyond the scope of this document.

[1] NegotiateHighVoltage is omitted from this discussion.

3 Unique Address Interface Protocol

The patch burst mode (PBM) feature uses both the SMBUS protocol and a simple I2C write. The SMBUS protocol is described in the data sheet (see 3and 4) and applicable to all register accesses. Register write and read examples used in PBM are listed in SMBUS Register Write Example and SMBUS Register Read Example. After the PBMs command is issued using the SMBUS protocol, the I2C writes are to the target address established in the PBMs command. I2C Patch Burst Mode Write shows the simple I2C writes for sending the image to the PD controller.

Table 3-1. SMBUS Register Write Example

TYPE	ACK	ADDRESS	READ	DATA	DESCRIPTION
start					
address	TRUE	0x21	FALSE		PD I2C Address for register access
data	TRUE			0x08	Register Address
data	TRUE			0x04	Number of bytes (sent to target)
data	TRUE			0x50	P
data	TRUE			0x42	B
data	TRUE			0x4D	M
data	TRUE			0x73	S
stop					

Table 3-2. SMBUS Register Read Example

TYPE	ACK	ADDRESS	READ	DATA	DESCRIPTION
start					
address	TRUE	0x21	FALSE		PD I2C Address for register access
data	TRUE			0x09	Register Number
start					Repeated Start to change from Write to Read
address	TRUE	0x21	TRUE		PD I2C Address for register access
data	TRUE			0x40	Number of Bytes[2]
data	TRUE			0x00	
data	TRUE			0x00	
data	TRUE			0x00	
data	TRUE			0x00	
	TRUE			0x30	PBM Address
	FALSE			0x31	Timeout. Controller NACKs the last byte to be read.
stop					

[2] The controller can choose to read all of the bytes or only a selection of bytes. In this example the controller only received 6 bytes, nacking, the last byte. In the final read of the DATA1 regsiter, 0x09, all 0x40 bytes are read.

Table 3-3. I2C Patch Burst Mode Write

TYPE	ACK	ADDRESS	READ	DATA	DESCRIPTION
start					
address	TRUE	0x30	FALSE		PBM I2C Address
data	TRUE			0x01	Image Byte 0
data	TRUE			0x00	Image Byte 1
data	TRUE			0xE0	Image Byte 2
data	TRUE			0xAC	Image Byte 3
Bytes 4 through 4093					
data	TRUE				Image Byte 4094
stop[3]					
start					
address	TRUE	0x30	FALSE		PBM I2C Address
data					Image Byte 4095
Bytes 4096 through 11390					
data	TRUE			0x00	Image Byte 11391
stop					

[3] The EC is limited to a transmit size of 4095 bytes. The PD controller auto-increments the PBM address pointer and is not reset by the I2C start or stop. The PBM pointer can be reset by issuing a PBMs command.

4 PTCH Mode to APP Mode

The patch bundle is a binary image which includes (bundles) both the device configuration and the firmware updates (patch). Section 4.2 shows how to generate the patch bundle, also referred to as the low region binary. Section 4.1 steps through the Patch Burst Mode (PBM) process and the resulting transition from *PTCH* to *APP* mode. A general description of the PBM can be found in the TRMs of the PD controllers and this example follows the same description with the exception of using interrupts. The steps include how to set up and maintain interrupts during the PBM process.

The flow chart in Pushing a Patch Bundle Over the I2Ct Bus assumes that there is no EEPROM present and the START is a cold boot or hard reset of the PD controller.

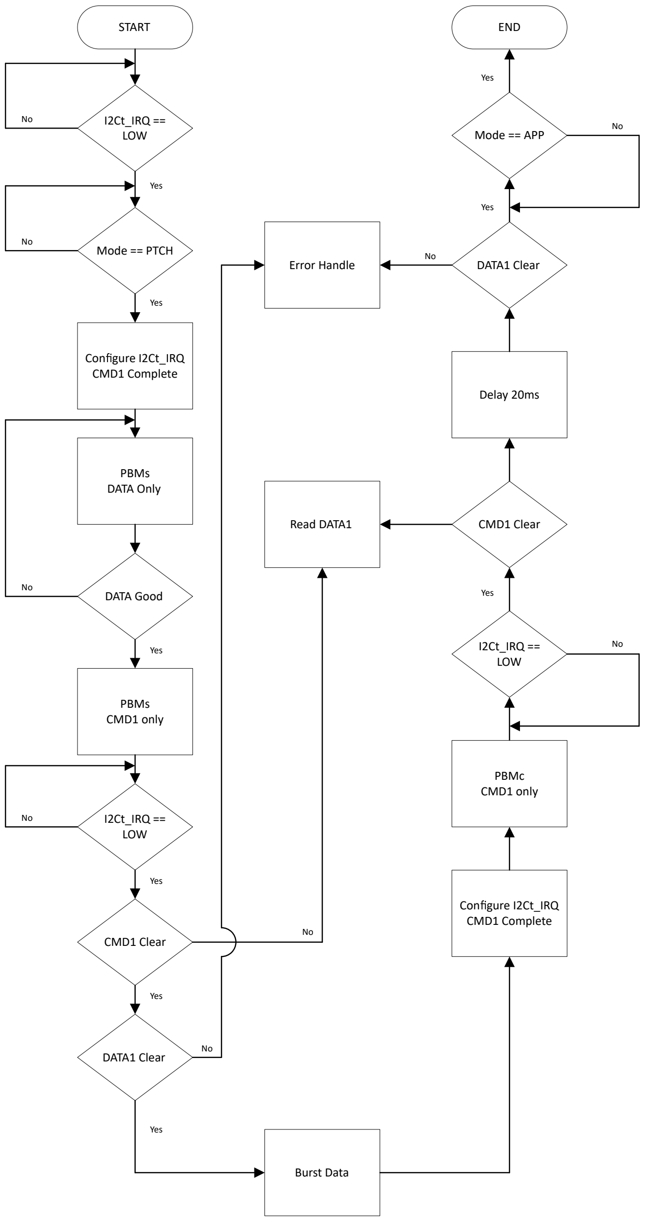

Figure 4-1. Pushing a Patch Bundle Over the I2Ct Bus

4.1 Step of PTCH Mode to APP Mode

1. **I2Ct_IRQ == Low:**

   Upon a cold boot (power cycle or GAID), the PD controller enters *PTCH* mode and only the *Ready for Patch[81]* interrupt is enabled automatically. The interrupt registers can be updated while in *PTCH* mode. This PBM implementation uses the *Ready for Patch* and CMD1 complete interrupts. The primary reason for using interrupts instead of polling the registers is to reduce the PD controller CPU loading to just the activities associated with the commands.

   The *Ready for Patch* interrupt is used at the beginning of the PBM process to indicate that the PD controller is ready. The CMD1 complete interrupt is used to alert the EC that the PBMs and PBMc commands are complete. Polling of the MODE register, 0x03, is still included in this example to account for the time when the patch is loaded but the PD controller is not yet transitioned to the application mode: *APP*.
2. **Mode == PTCH:**

   PTCH and *APP* modes are described in the PD controller TRMs. The EEPROM on the PD controller EVM is disabled (SDA disconnected) and therefore the PD controller transitions to and stays in *PTCH* mode. The check for the *PTCH* mode at the beginning of the process is not necessary but included for completeness. The following is the example for the command and the logic analyzer capture is shown in Read *PTCH* Mode.

   [0x21] + ACK (Unique Address/Wr/A)

   0x03 + ACK (Register Number/A)

   [0x21] + ACK (Unique Address/R/A)

   0x04 (Byte Count)

   0x50 0x54 0x43 0x48 (*PTCH* in 4ASCII characters)

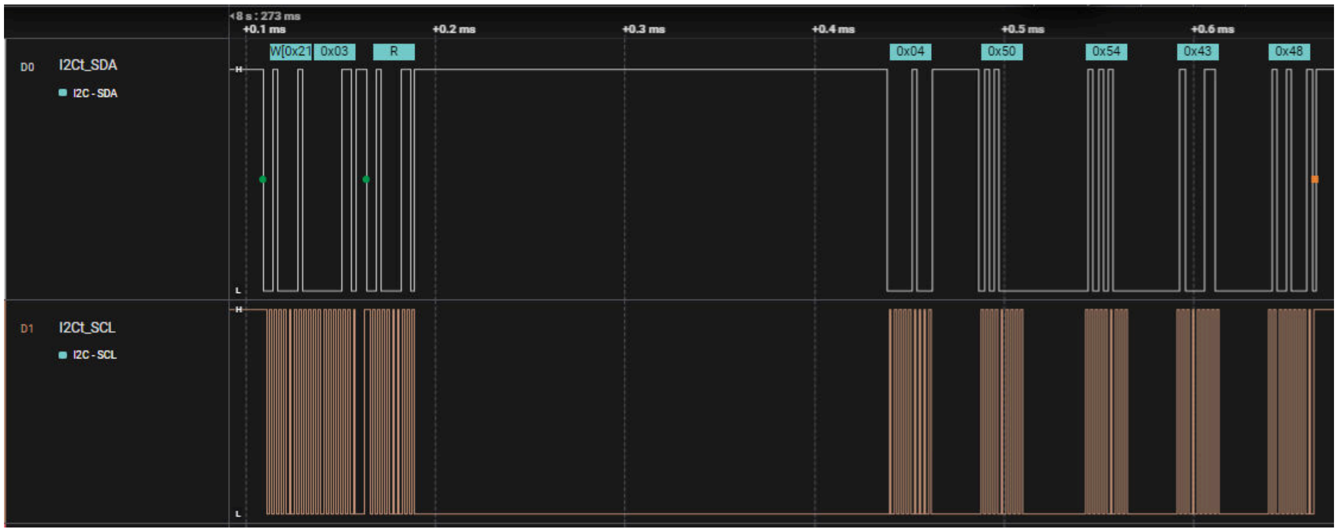

Figure 4-2. Read *PTCH* Mode

3. **Configure I2Ct_IRQ, CMD1 Complete:**

   The CMD1 complete interrupt is used to alert the EC that the PBMs command is complete. Setting the Interrupt Mask and clearing the interrupts are accomplished with registers 0x16 and 0x18, respectively. See 1 and 2

   [0x21] + ACK (Unique Address/Wr/A)

   0x16 + ACK (Register Number/A)

   0x0B (Byte Count)

   0x00 0x00 0x00 0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x01 (MSB)

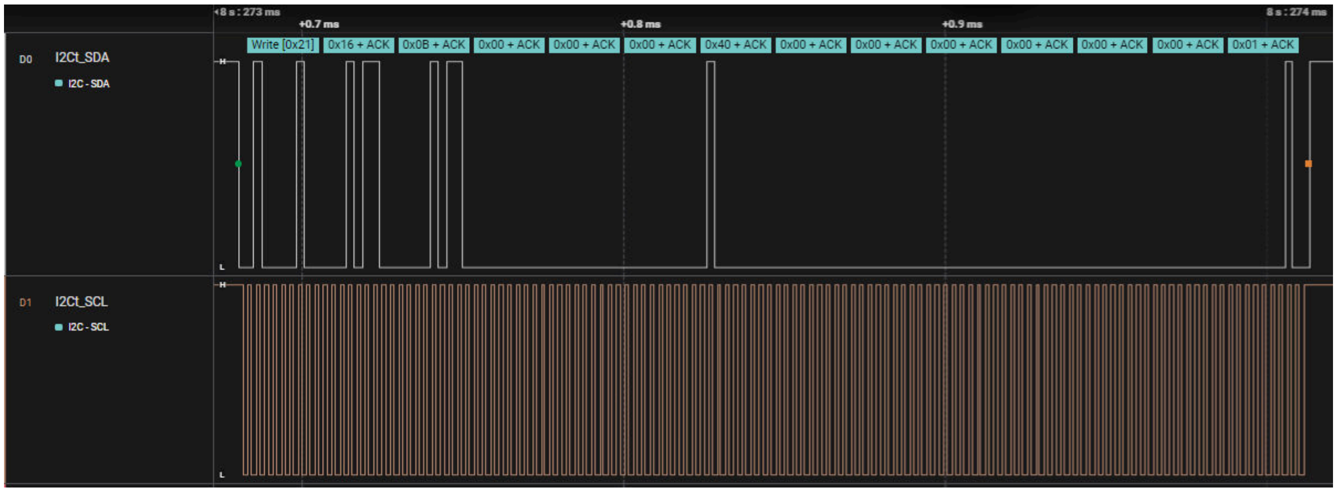

Figure 4-3. Configuring the Interrupt Mask Register, 0x16

   [0x21] + ACK (Unique Address/Wr/A)

   0x18 + ACK (Register Number/A)

   0x0B (Byte Count)

   0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF (MSB)

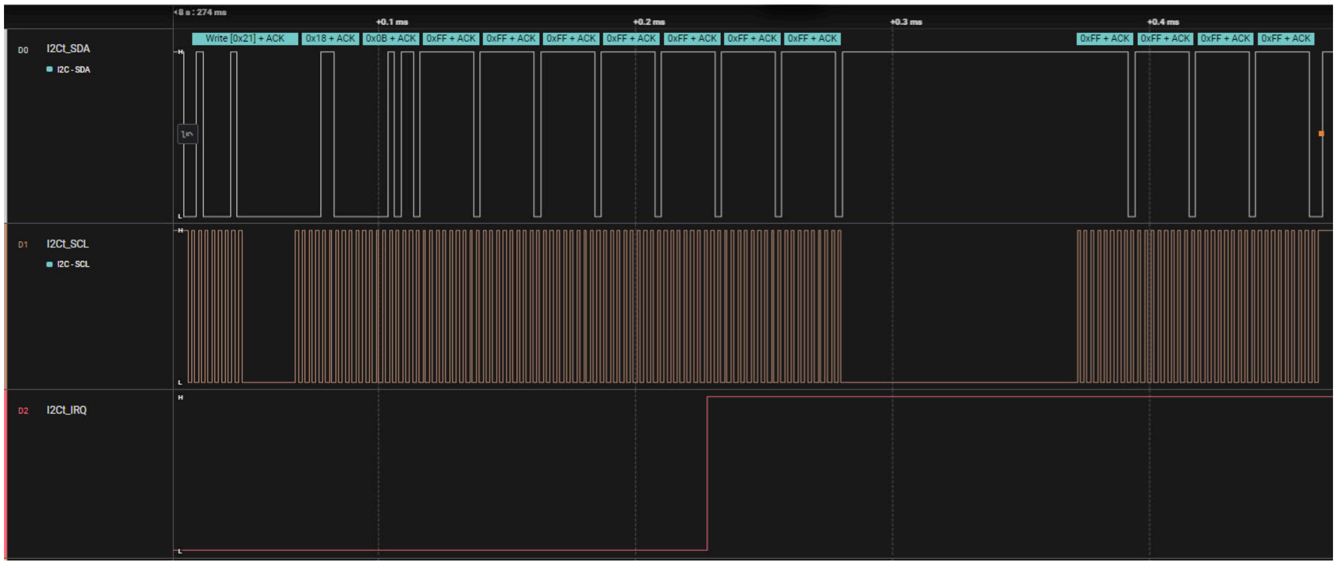

Figure 4-4. Interrupt Clear Register, 0x18

4. ***PBMs* Data Only:**

   The PBMs command is defined in the TRM reference. For this example, the parameters for PBMs are listed in Table 4-1.

Table 4-1. PBMs Configuration: DATA1 Register

Description	Value	Comment
Bundle Size	0x00002C80	See Section 5
I2C Burst Data Target Address	0x30	0x30, See reference 1.
Timeout	0x31	3.1 seconds; see reference 1

   [0x21] + ACK (Unique Address/Wr/A)

   0x09 + ACK (Register Number/A)

   0x06 (Byte Count)

   0x80 0x2C 0x00 0x00 0x30 0x32 (bundle size, I2C target address, timeout value)

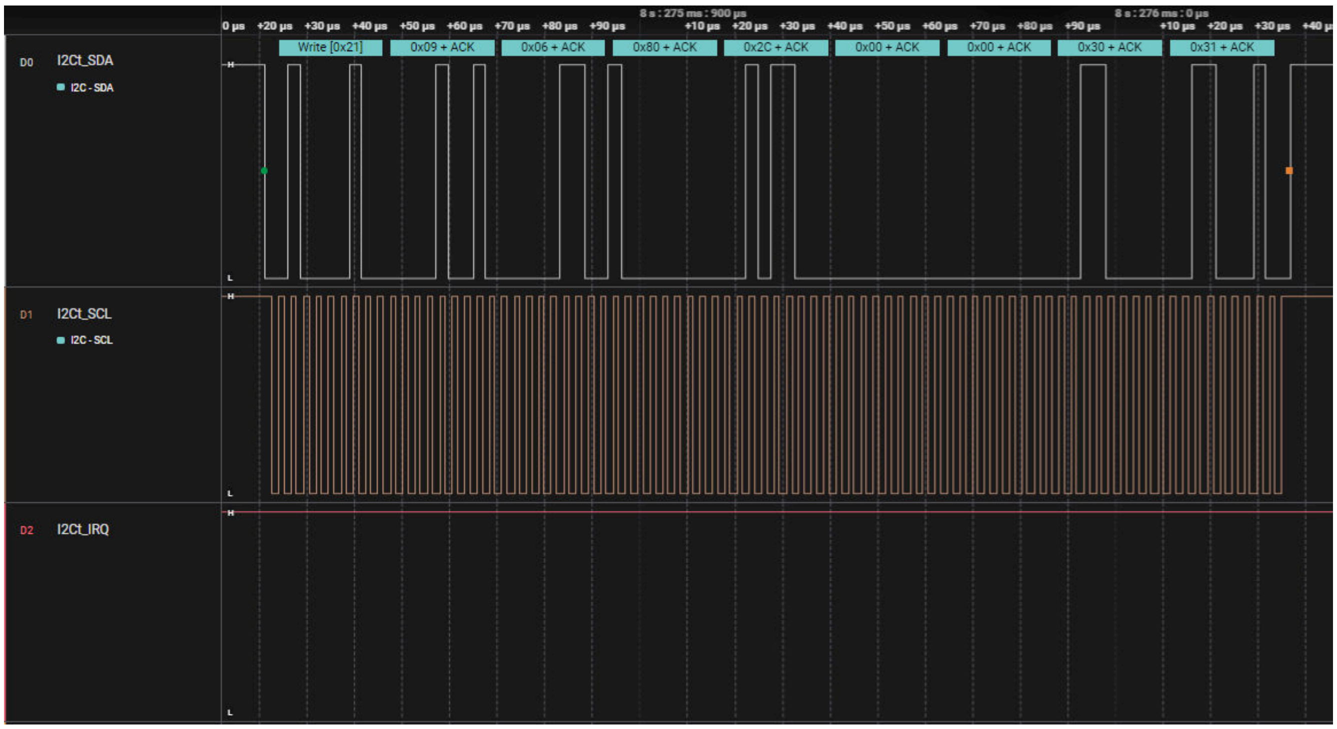

Figure 4-5. DATA1 Register, 0x09, for PBMs

5. **Data Good:**

   Sending the PBMs command requires writing to the DATA register multiple times. In the example, the values of 0x09 are confirmed before writing the PBMs command in register 0x08. There is a delay of 500us between the writing to and reading from register 0x09.

   [0x21] + ACK (Unique Address/Wr/A)

   0x09 + ACK (Register Number/A)

   [0x21] + ACK (Unique Address/R/A)

   0x40 (Byte Count)

   0x00 0x00 0x00 0x00 0x00 0x00 (incorrect, rewrite DATA1)

   0x80 0x2C 0x00 0x00 0x30 0x32 (correct, proceed to writing CMD1)

6. **PBMs CMD1:**

   After confirming DATA1, then write CMD1 = *PBMs*. The I2Ct_IRQ asserts low as shown in Figure 4-6.

   [0x20] + ACK (Unique Address/Wr/A)

   0x08 + ACK (Register Number/A)

   0x04 (Byte Count)

   0x50 0x42 0x4D 0x73 (*PBMs* in 4ASCII characters)

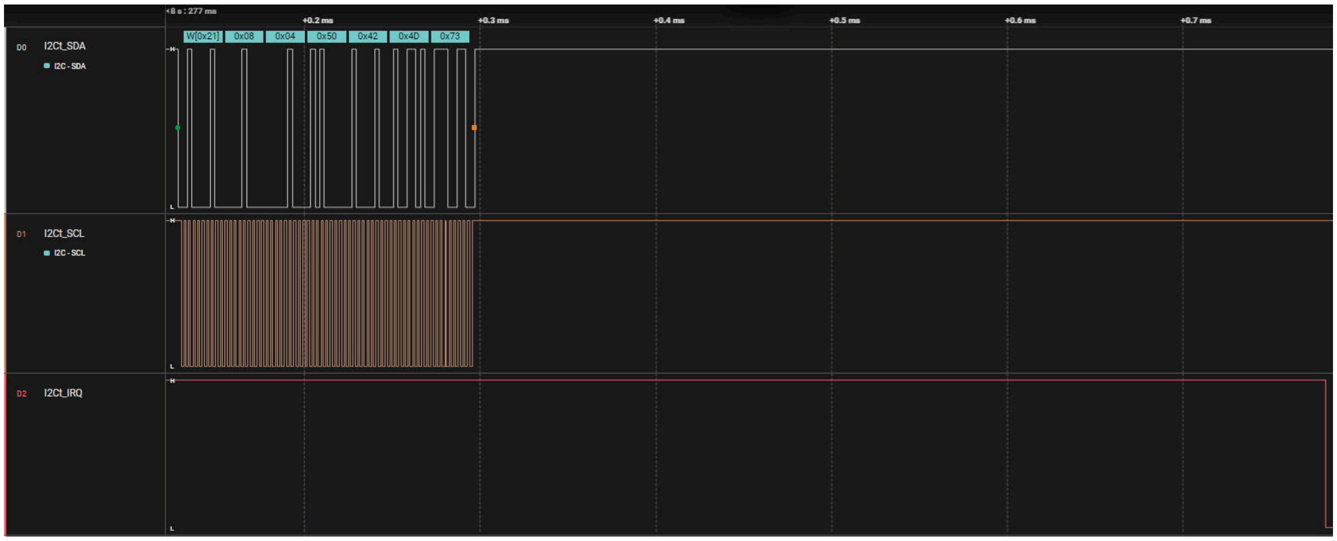

Figure 4-6. Write PBMs to CMD1 Register, 0x08

7. **I2Ct_IRQ == Low**

   The IRQ signal represents when the CMD1 Complete event has occurred and the CMD1 and DATA registers can be read to confirm the outcome of the PBMs command. The expected results are shown in steps 8 and 9.

8. **CMD1 Clear (PBMs)**

   The command register, 0x08, indicates that the command is successfully completed when the content of the register is cleared. For simplicity the example only checks the first bit to confirm that the contents are not "!CMD" which indicates that the PBMs command was corrupted or the DATA register was loaded with an illegal value[4].

   [0x21] + ACK (Unique Address/Wr/A)

   0x08 + ACK (Register Number/A)

   [0x21] + ACK (Unique Address/R/A)

   0x04 (Byte Count)

   0x00 0x00 0x00 0x00

[4] If the IRQ for CMD1 complete was not used then the CMD1 register can be polled until the contents transition from [0x50, 0x42, 0x4D, 0x73] to either [0x00, 0x00, 0x00, 0x00] or [0x21, 0x43, 0x4D, 0x44]. CMD1[0]=0x21 indicates that the command completed unsuccessfully and CMD1[0]=0x00 indicates successful completion.

9. **DATA1 Clear (PBMs)**

   The data register, 0x09, indicates a successful patch when the first byte, PatchStartStatus, is cleared. Non-zero values of PatchStartStatue, 0x04, 0x05, and 0x06, indicate invalid bundle size, target address, or timeout value, respectively. See 1.

   [0x21] + ACK (Unique Address/Wr/A)

   0x09 + ACK (Register Number/A)

   [0x21] + ACK (Unique Address/R/A)

   0x40 (Byte Count)

   0x00 0x00 0x00 0x00 0x30 0x31
10. **Burst Data**

   In this step the PMBUS format is not used and the contents of the binary image are written directly to the I2C Burst Data Target Address, specified in the PBMs command, table reference. The burst format is impacted by the MCU architecture. In this case, the burst size is limited to 4KB so three successive bursts (4095bytes, 4095bytes, and 3202bytes) are sent to the PD with a 500us delay between each burst. An additional 500us delay is added to delay when the PBMc command is sent relative to the end of the final burst.

   [0x30] + ACK (Unique Address/Wr/A)

   lowRegion_i2c_array[0], lowRegion_i2c_array[1]..., lowRegion_i2c_array[4094]

   [0x30] + ACK (Unique Address/Wr/A)

   lowRegion_i2c_array[4095], lowRegion_i2c_array[4096]..., lowRegion_i2c_array[8189]

   [0x30] + ACK (Unique Address/Wr/A)

   lowRegion_i2c_array[8190], lowRegion_i2c_array[8191]..., lowRegion_i2c_array[11391]
11. **Configure I2Ct_IRQ, CMD1 Complete**

   The CMD1 complete interrupt is used to alert the EC that the PBMc command is complete. Setting the Interrupt Mask and clearing the interrupts are accomplished with registers 0x16 and 0x18, respectively. The interrupt mask was already set in step 3. Repeat clearing the interrupt as shown in Figure 4-4 and Figure 4-7.
12. **PBMc CMD1 Only**

   The PBMc command does not include input data so the command only is sent.

   [0x21] + ACK (Unique Address/Wr/A)

   0x08 + ACK (Register Number/A)

   [0x21] + ACK (Unique Address/R/A)

   0x04 (Byte Count)

   0x50 0x42 0x4D 0x63 (‘PBMc’ in 4ASCII characters)

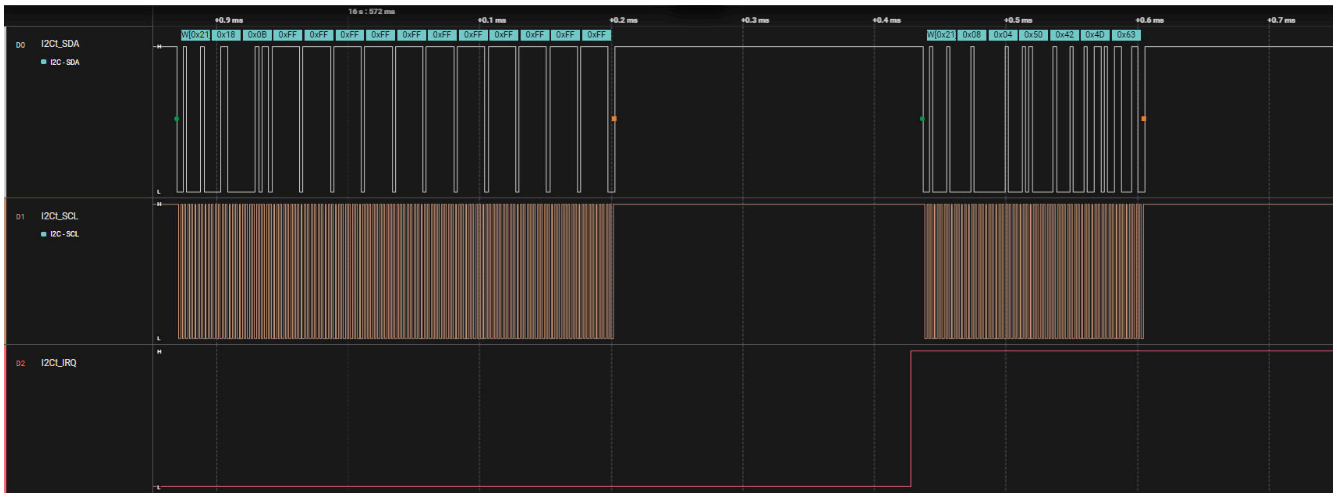

Figure 4-7. Clear Interrupts and Write PBMc to CMD1 Register

13. **I2Ct_IRQ == Low**

    The IRQ signal represents when the CMD1 Complete event has occurred and the CMD1 register can be read to confirm the outcome of the PBMc command.

14. **CMD1 Clear (PBMc)**

    Similar to the PBMs CMD1 clear the CMD1 register is read and confirmed to be cleared of the original command and not equal to *!CMD*.

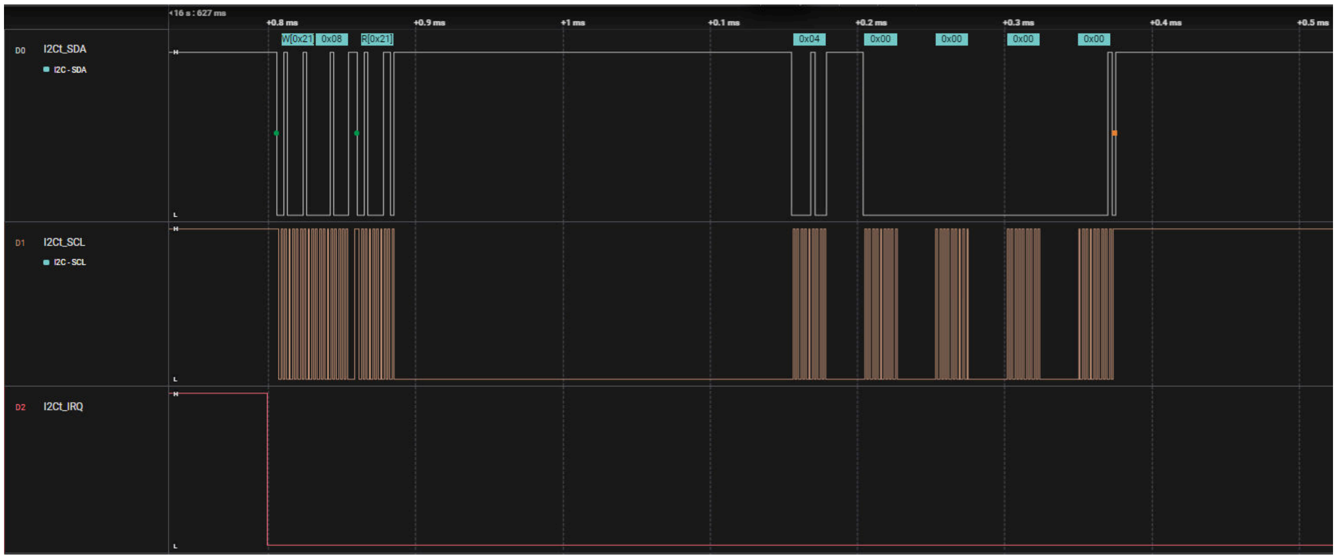

Figure 4-8. Read CMD1 After I2Ct_IRQ Asserted (PBMc)

15. **Delay 20ms** [5]

    The 20ms delay allows the PD controller to load and apply the image. Once the delay elapses, the DATA1 and Mode registers are read to confirm the success.

[5] The 20ms delay can be replaced with the Patch Loaded interrupt. No reduction in time was achieved using this interrupt in place of the delay. Therefore the 20ms delay described in the device TRMs is maintained in this document. See 1 and 2

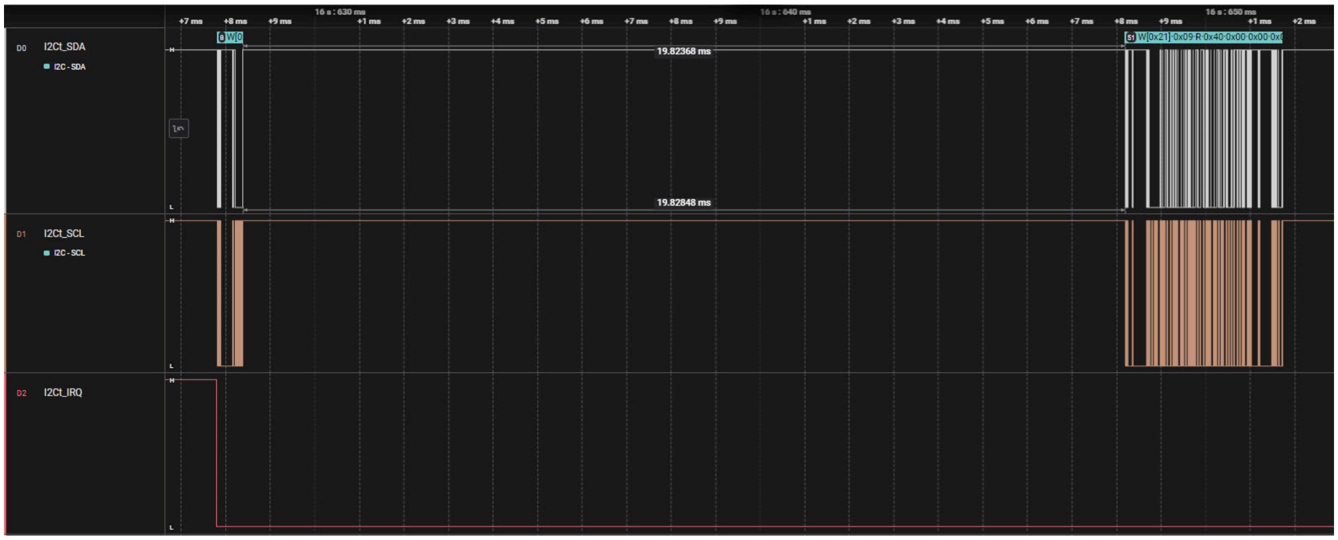

Figure 4-9. Delay Between CMD1 and DATA1 register reads

16. **DATA1 Clear (PBMc)**

    In this instance, 40 bytes are read from the DATA1 register.

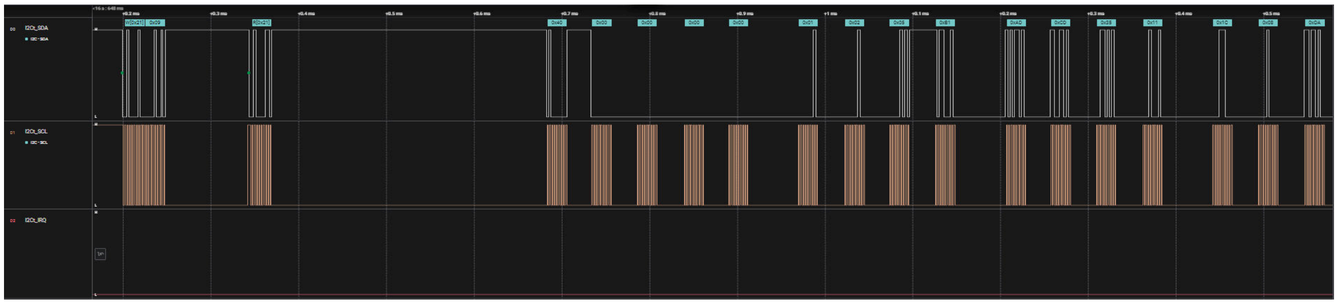

Figure 4-10. Read DATA1 Register after PBMc complete

17. **Mode == APP**

    The final step is to verify that the PD has transitioned to the APP mode. Once in APP mode, the PD controller is now operational with the custom configuration applied.

    [0x21] + ACK (Unique Address/Wr/A)

    0x03 + ACK (Register Number/A)

    [0x21] + ACK (Unique Address/R/A)

    0x04 (Byte Count)

    0x41 0x50 0x50 0x20 ('APP ' in 4 ASCII characters)

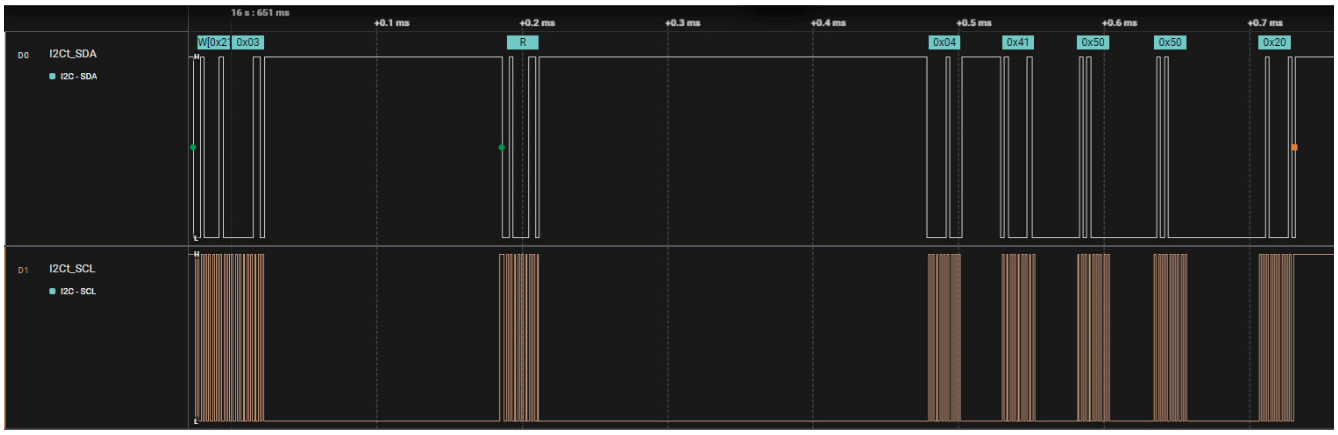

Figure 4-11. APP Mode

4.2 Step of Generating Low Region Binary

The USBC PD Application Customization Tool(GUI) utilizes a number of questions to generate a custom PD controller configuration. Once the configuration is complete, a patch bundle, containing configuration and firmware updates, can be generated. The full flash binary generates an image in the format the PD controller expects when reading from an EEPROM over I2Cc. The low region binary, see Figure 4-12, is a smaller format and intended for writing to the PD controller over I2Ct. This low region binary is what is sent in step 10.

1. Answer Questionnare in the GUI.
2. If necessary configure any additional settings using the Advanced Configuration switch.
3. Generate low region binary by selecting from the Export drop down menu.
4. Select the approriate format and the desired file name.

As shown in Figure 4-13, the GUI provides options to export the low region file as either a binary or a c file. For this application note the c file format was chosen and included in the device project for building the EC code. Modification of the source file generated from the GUI and the associated definitions are found in Section 5.

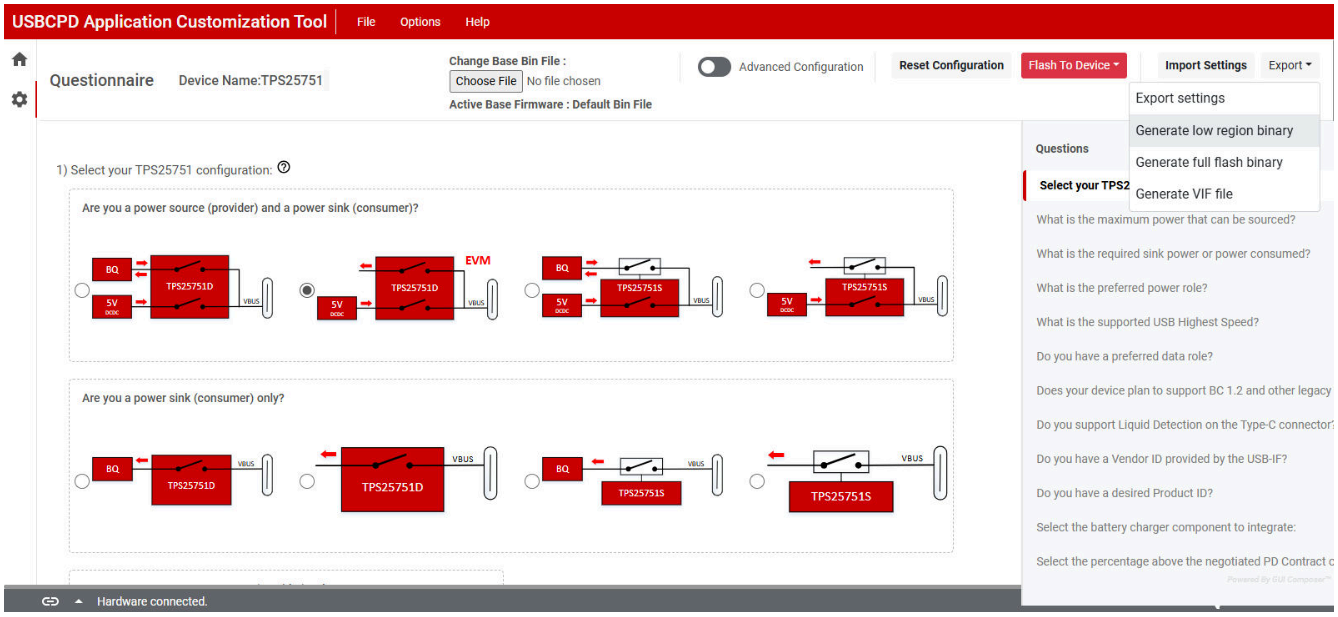
USBCPD Application Customization Tool
File
Options
Help
Questionnaire
Device Name:TPS25751
Change Base Bin File :
Choose File
No file chosen
Active Base Firmware : Default Bin File
Advanced Configuration
Reset Configuration
Flash To Device
Import Settings
Export
Export settings
Generate low region binary
Generate full flash binary
Generate VIF file
Questions
Select your TPS2
1) Select your TPS25751 configuration:
Are you a power source (provider) and a power sink (consumer)?
Are you a power sink (consumer) only?
BQ
5V
TPS25751D
TPS25751S
VBUS
EVM
What is the maximum power that can be sourced?
What is the required sink power or power consumed?
What is the preferred power role?
What is the supported USB Highest Speed?
Do you have a preferred data role?
Does your device plan to support BC 1.2 and other legacy
Do you support Liquid Detection on the Type-C connector
Do you have a Vendor ID provided by the USB-IF?
Do you have a desired Product ID?
Select the battery charger component to integrate:
Select the percentage above the negotiated PD Contract c
Hardware connected.

Figure 4-12. Generating image from USBCPD Application Customization Tool

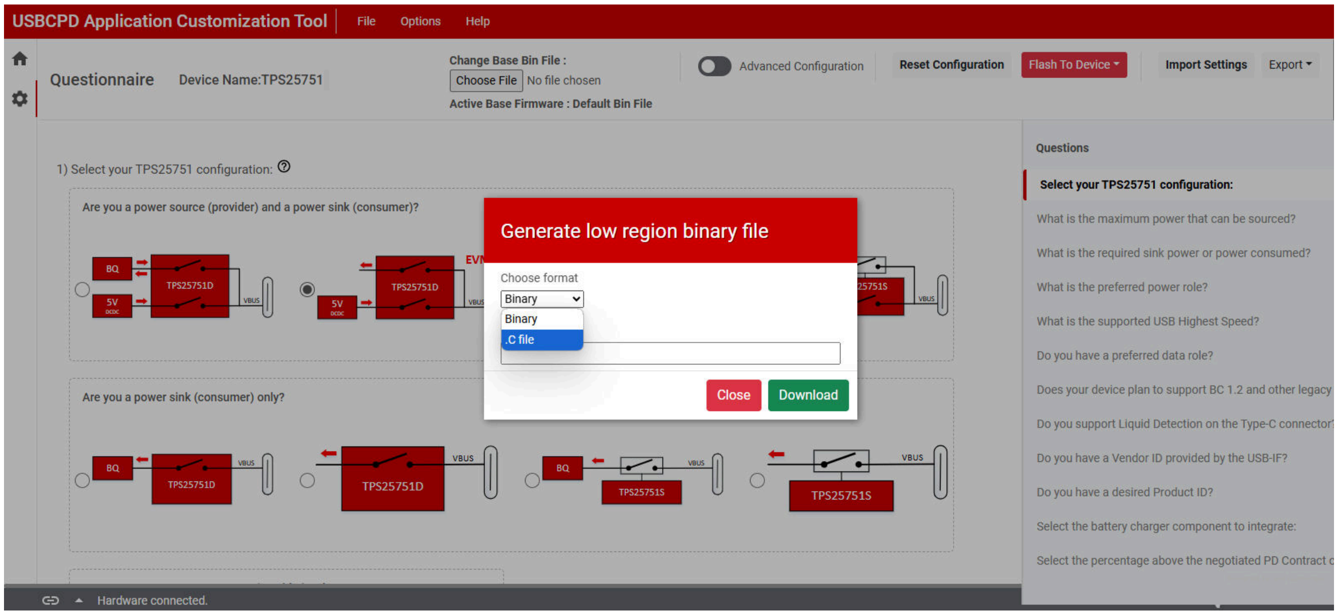
USBCPD Application Customization Tool
File
Options
Help
Questionnaire
Device Name:TPS25751
Change Base Bin File :
Choose File
No file chosen
Active Base Firmware : Default Bin File
Advanced Configuration
Reset Configuration
Flash To Device
Import Settings
Export
1) Select your TPS25751 configuration:
Are you a power source (provider) and a power sink (consumer)?
Are you a power sink (consumer) only?
Generate low region binary file
Choose format
Binary
Binary
.C file
Close
Download
Questions
Select your TPS25751 configuration:
What is the maximum power that can be sourced?
What is the required sink power or power consumed?
What is the preferred power role?
What is the supported USB Highest Speed?
Do you have a preferred data role?
Does your device plan to support BC 1.2 and other legacy
Do you support Liquid Detection on the Type-C connector
Do you have a Vendor ID provided by the USB-IF?
Do you have a desired Product ID?
Select the battery charger component to integrate:
Select the percentage above the negotiated PD Contract c
Hardware connected.

Figure 4-13. Selecting image output type

5 Example Code

The image file generated from the GUI was slightly modified as shown in the following code.

```
#include "ti_msp_dl_config.h"
#include "lrb.h"

const uint8_t tps25751_lowRegion_i2c_array[SIZEOFLRB] = {
0x01, 0x00, 0xe0, 0xac, 0xfe, 0xff, 0xff, 0xff, 0x80, 0x06, 0x00, 0x00, 0x00, 0x26, 0x00, 0x00,
0x71, 0x3e, 0x9c, 0xb8, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
...
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
```

The file, *lrb.h*, is shown in the following code for context.

```
#define SIZEOFLRB  0x2C80   // 11392
/* Maximum size of TX packet, MCU 0xFFF */
#define I2C_TX_MAX_PACKET_SIZE
/* SIZEOFLRB/I2C_TX_MAX_PACKET_SIZE +1 */
#define LRB_NUMBER_OF_PACKETS  (0x0003)
/* SIZEOFLRB % I2C_TX_MAX_PACKET_SIZE */
#define LRB_REMAINDER         (0x0C82)
extern const uint8_t tps25751_lowRegion_i2c_array[SIZEOFLRB];
```

The following example code is based upon the MSPM0C110x Driver Library[6] and the steps found in Section 4.1.

```
    /*
     *  1. Wait for Ready for Patch interrupt
     */
    debounce = 1;
    /* Skip glitch at boot */
    while(debounce)
    {
        while(DL_GPIO_readPins(
                GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN));
        delay_cycles(DELAY_500us);
        if(!DL_GPIO_readPins(
                GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN))
        {
             debounce = 0;
        }
    }
    /*
     *  2. Mode == PTCH
     */
    readRegister(MODE_REGISTER, MODE_BYTE_CNT, gRxPacket);
    if(gRxPacket[1] != 0x50)
    {
        delay_cycles(DELAY_10ms);
        readRegister(MODE_REGISTER, MODE_BYTE_CNT, gRxPacket);
    }
    /*
     *  3a. Configure I2Ct_IRQ for CMD1 Complete
     */                                                   */
    gTxPacket = (uint8_t*)maskIntReg;
    writeRegister(INT_BYTE_CNT, (uint8_t*)gTxPacket);
    waitingForPD = 1;
    while(waitingForPD)
    {
        /*
         *  3b. Clear Interrupts
         */
        gTxPacket = (uint8_t*)clrIntReg;
        writeRegister(INT_BYTE_CNT, (uint8_t*)gTxPacket);
        //  Wait for interrupt to clear
        while(DL_GPIO_readPins(GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN)==0);
        waitingForGoodData = 1;
        while(waitingForGoodData)
```

[6] MSPM0 SDK (2.05.00.05)/Examples/Development Tools/LP_MSPM0C1104 LaunchPad/DriverLib/i2c_controller_rw_multiple_fifo_interrupts

```
        {
            /*
             *  4. Send PBMs Data
             */
            gTxPacket = (uint8_t*)PBMsData;
            writeRegister(DATA_BYTE_CNT, gTxPacket);
            delay_cycles(DELAY_500us);
            /*
             * 5. Data Good
             */
            readRegister(DATA_REG, (DATA_BYTE_CNT-1), gRxPacket);
            if(gRxPacket[1] == 0x80)
            {
                waitingForGoodData = 0;
            }
        }
        /*
         *  6. Send PBMs to CMD1
         */
        gTxPacket = (uint8_t*)PBMsCmd;
        writeRegister(CMD_BYTE_CNT, gTxPacket);
        /*
         *  7. Wait for I2Ct_IRQ
         */
        while(DL_GPIO_readPins(
                    GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN));
        /*
         *  8. CMD1 Clear
         */
        readRegister(CMD_REG, (CMD_BYTE_CNT-1), gRxPacket);
        /*
         *  9. DATA1 Clear
         */
        readRegister(DATA_REG, (DATA_BYTE_CNT-1), gRxPacket);
        if(gRxPacket[1] == 0x0)
        {
            waitingForPD = 0;
        }
    }
    /*
     *  10. Burst Data
     */
    ii = LRB_NUMBER_OF_PACKETS;
    while(ii)
    {
        if(ii == 1)
        {
            gTxLen = LRB_REMAINDER;
        }
        else
        {
            gTxLen = I2C_TX_MAX_PACKET_SIZE;
        }
        /* The FIFO is 8-bytes deep, and this function returns number of bytes written to FIFO */
        gTxPacket = (uint8_t*)&tps25751_lowRegion_i2c_array[(LRB_NUMBER_OF_PACKETS-
ii)*I2C_TX_MAX_PACKET_SIZE];
        ii = ii-1;
        gTxCount = DL_I2C_fillControllerTXFIFO(I2C_INST, gTxPacket, gTxLen);
        DL_I2C_enableInterrupt(
                I2C_INST, DL_I2C_INTERRUPT_CONTROLLER_TXFIFO_TRIGGER);
        gI2cControllerStatus = I2C_STATUS_TX_STARTED;
        while (!(
            DL_I2C_getControllerStatus(I2C_INST) & DL_I2C_CONTROLLER_STATUS_IDLE))
            ;
        DL_I2C_startControllerTransfer(
            I2C_INST, I2C_TGT_BURST_ADDRESS, DL_I2C_CONTROLLER_DIRECTION_TX, gTxLen);
        while ((gI2cControllerStatus != I2C_STATUS_TX_COMPLETE) &&
            (gI2cControllerStatus != I2C_STATUS_ERROR)) {
            __WFE();
        }
        while (DL_I2C_getControllerStatus(I2C_INST) &
            DL_I2C_CONTROLLER_STATUS_BUSY_BUS)
            ;
        /* Trap if there was an error */
        if (DL_I2C_getControllerStatus(I2C_INST) &
            DL_I2C_CONTROLLER_STATUS_ERROR) {
            __BKPT(0);
        }
        while(!(
            DL_I2C_getControllerStatus(I2C_INST) & DL_I2C_CONTROLLER_STATUS_IDLE))
            ;
        delay_cycles(DELAY_500us);
    }
    /*
     *  11. Clear Interrupts
     */
    gTxPacket = (uint8_t*)clrIntReg;
    writeRegister(INT_BYTE_CNT, (uint8_t*)gTxPacket);
    //  Wait for interrupt to clear
    while(DL_GPIO_readPins(
        GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN)==0)
        ;
    /*
     *  12. Send PBMc to CMD1 Register
     */
    gTxPacket = (uint8_t*)PBMcCmd;
    writeRegister(CMD_BYTE_CNT,gTxPacket);
    /*
     *  13. I2Ct_IRQ == Low, CMD1 complete
     */
    while(DL_GPIO_readPins(
        GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN));
    /*
     *  14. Read CMD1
     */
    readRegister(CMD_REG, (CMD_BYTE_CNT-1), gRxPacket);
    /*
     *  15. Delay 20ms
     */
    delay_cycles(DELAY_20ms);
    /*
     *  16. Read Data1
     */
    readRegister(DATA_REG, 0x28, gRxPacket);
    /*
     *  17. MODE === APP
     */
    waitingForPD = 1;
    while(waitingForPD)
    {
        readRegister(MODE_REGISTER, MODE_BYTE_CNT, gRxPacket);
        if(gRxPacket[1] == 0x41)
        {
            waitingForPD = 0;
        }
        else
        {
            delay_cycles(DELAY_10ms);
        }
    }
```

6 References

1. Texas Instruments, *TPS25751 Technical Reference Manual*, technical reference manual.
2. Texas Instruments, *TPS26750 Technical Reference Manual*, technical reference manual.
3. Texas Instruments, *TPS25751 USB Type-C® and USB PD Controller with Integrated Power Switches*, data sheet.
4. Texas Instruments, *TPS26750 USB Type-C® and USB PD Controller With Integrated Power Switches Optimized for Power Applications*, data sheet.
5. Texas Instruments, *MSPM0 software development kit (SDK)*, software
6. Texas Instruments, *PTCH to APP*, E2E™ design support forum.
7. Texas Instruments, *Salae code*, E2E™ design support forum.
8. Texas Instruments, *Mask off*, E2E™ design support forum.

7 Revision History

Changes from Revision * (July 2024) to Revision A (June 2025)	Page
• Added TPS26750 to document	1
• Updated the numbering format for tables, figures, and cross-references throughout the document	1
• Deleted specific references to the TPS25751 and make generic	2
• Deleted specific references to the TPS25751	2
• Changed scope to Patch Burst Mode	3