

XDPP1100 programming instruction

Scope and purpose

The Infineon XDPP1100 family of digital controllers contains a built-in microcontroller for feature enhancement or quick bug fixes. The microcontroller firmware (FW) can be updated or “patched” over the industry-standard I²C serial interface and saved in internal non-volatile memory (NVM). This document describes how to upload a FW patch over the I²C serial interface and also store a configuration to NVM.

Intended audience

XDPP1100 users.

Table of contents

Table of contents	1
1 Hardware and software requirements	3
2 Introduction of XDPP1100 OTP structure	6
3 I²C and PMBus communication protocol	7
3.1 I ² C communication protocol	7
3.2 PMBus communication protocol	7
4 Programming procedure using XDPP1100 GUI	9
4.1 Add device	9
4.1.1 Auto-populate device	9
4.1.2 Add a new device manually	9
4.1.3 Force I ² C connection	11
4.2 Multi Device Programmer tool	11
4.3 Check OTP partition	12
4.4 Store FW patch	13
4.5 Apply FW configuration	14
4.6 Load design file	14
4.7 Store configuration to OTP	14
5 Partial configuration and multi configuration	15
5.1 Partial configuration	15
5.1.1 Store PMBus partial configuration	15
5.1.2 Store register partial configuration	15
5.2 Multi configuration	16
5.2.1 Multi configuration setup	16
5.2.2 Store the first configuration	17
5.2.3 Store the second configuration	17
5.2.4 Check configuration in OTP	17
5.2.5 Verify the configuration with different XADDR resistor	17
6 Programming procedure using I²C protocol	18
6.1 Storing FW trim	18
6.2 Storing FW configuration	19
6.3 Patch file format	20
6.4 Storing a FW patch through I ² C protocol	20
6.5 Configuration file format	24
6.6 Storing full configuration using I ² C port	24

Hardware and software requirements

6.7	Store PMBus partial configuration	26
6.8	Store register partial configuration	26
6.9	Procedure to find remaining free memory in OTP	27
6.9.1	Procedure to check remaining space in data partition to store a new configuration.....	27
6.9.2	Procedure to find remaining space in patch partition to store a patch.....	28
7	Invalidating FW patch and configurations	30
7.1	Invalidating FW patch and configuration by GUI	30
7.2	Invalidating FW patch	30
7.3	Invalidating configurations (register, PMBus and partial configurations).....	31
	Revision history.....	33

Hardware and software requirements

1 Hardware and software requirements

The hardware requirements are listed below:

1. 3.3 V 20 mA DC power supply.
2. USB to I²C dongle USB007A (see [Figure 1](#)). The blue wire is not used.
3. USB Type-A male to Type-A female extension cable (optional) (see [Figure 2](#)).
4. XDPP1100 programming board.

The board could have IC solder down on the board to program the XDPP1100 in the application system. Or use a programmer board with IC socket. [Figure 3](#) is an example of the IC socket for the 4x4 QFN 24-pin version.

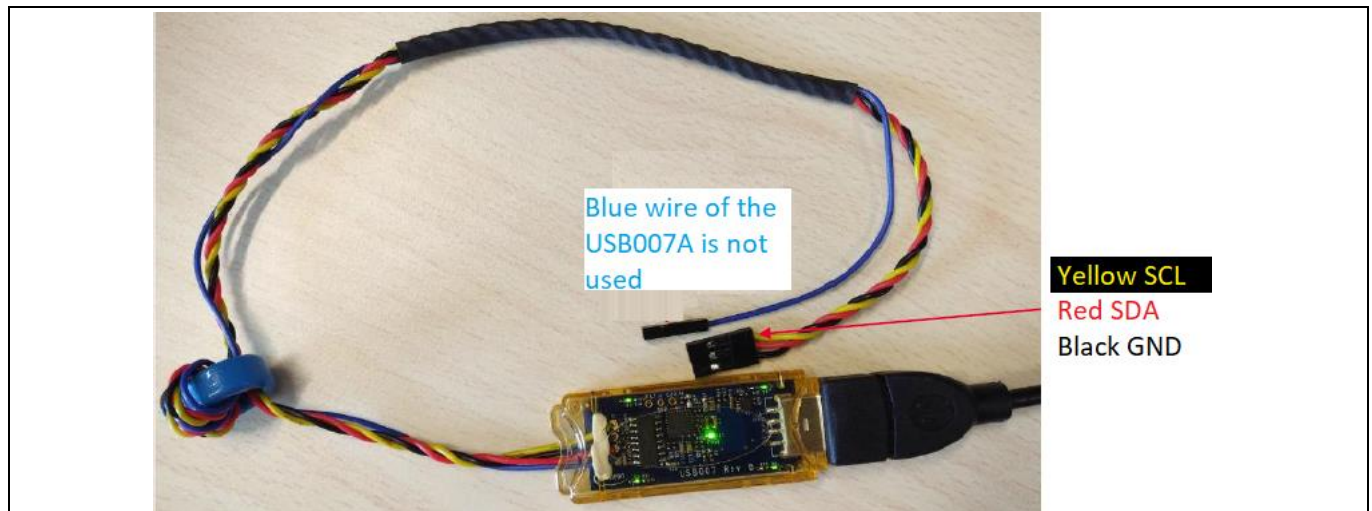


Figure 1 USB to I²C dongle



Figure 2 USB Type-A male to Type-A female extension cable

Hardware and software requirements

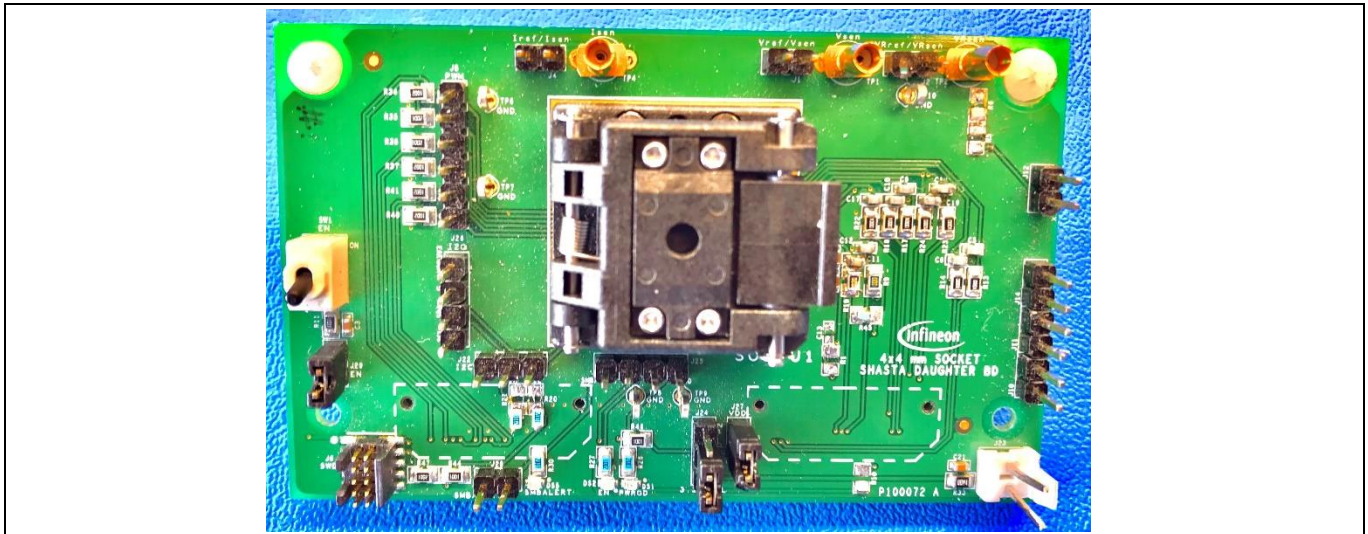


Figure 3 XDPP1100-Q024 4x4 QFN socket board

The board setup and the connection of the I²C to USB dongle is shown in **Figure 4**. The pull-up resistors of the SDA and SCL pins are optional because the USB007A has an internal pull-up resistor. If a pull-up resistor is used on the board, it should be in the 1 k Ω to 10 k Ω range. The connection of unused pins is not critical and they can be left floating. The system should be disabled if the controller is programmed on a system board.

The USB007A dongle mates with a 2.54 mm connector. The colors of the I²C header are defined as follows:

- Yellow: SCL
- Red: SDC
- Black: GND

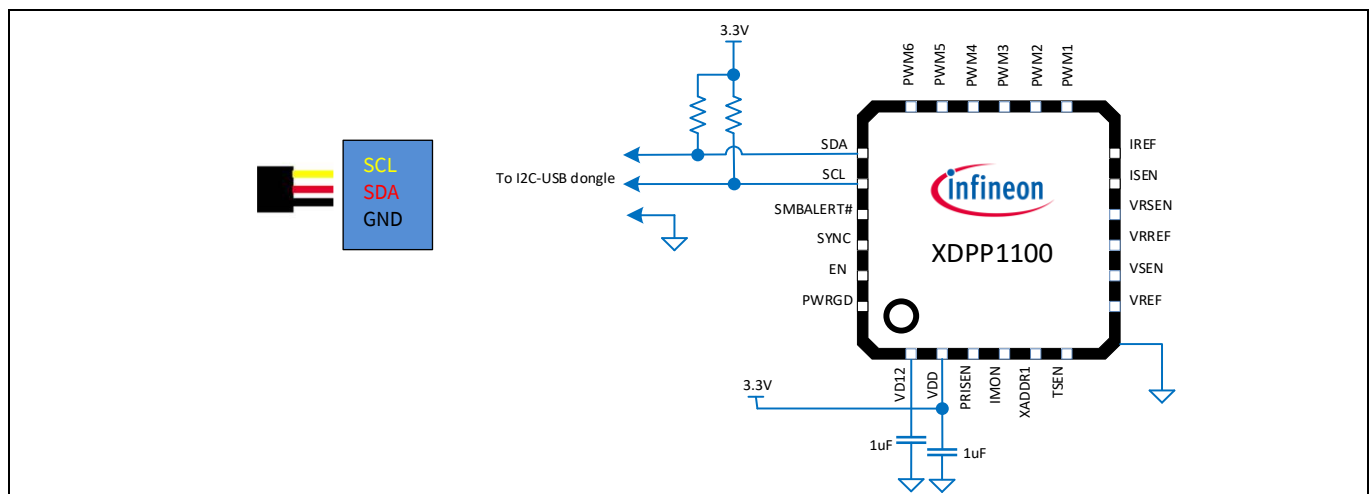


Figure 4 XDPP1100 I²C connection

Software requirements:

- XDPP1100 GUI, version build 7774

Hardware and software requirements

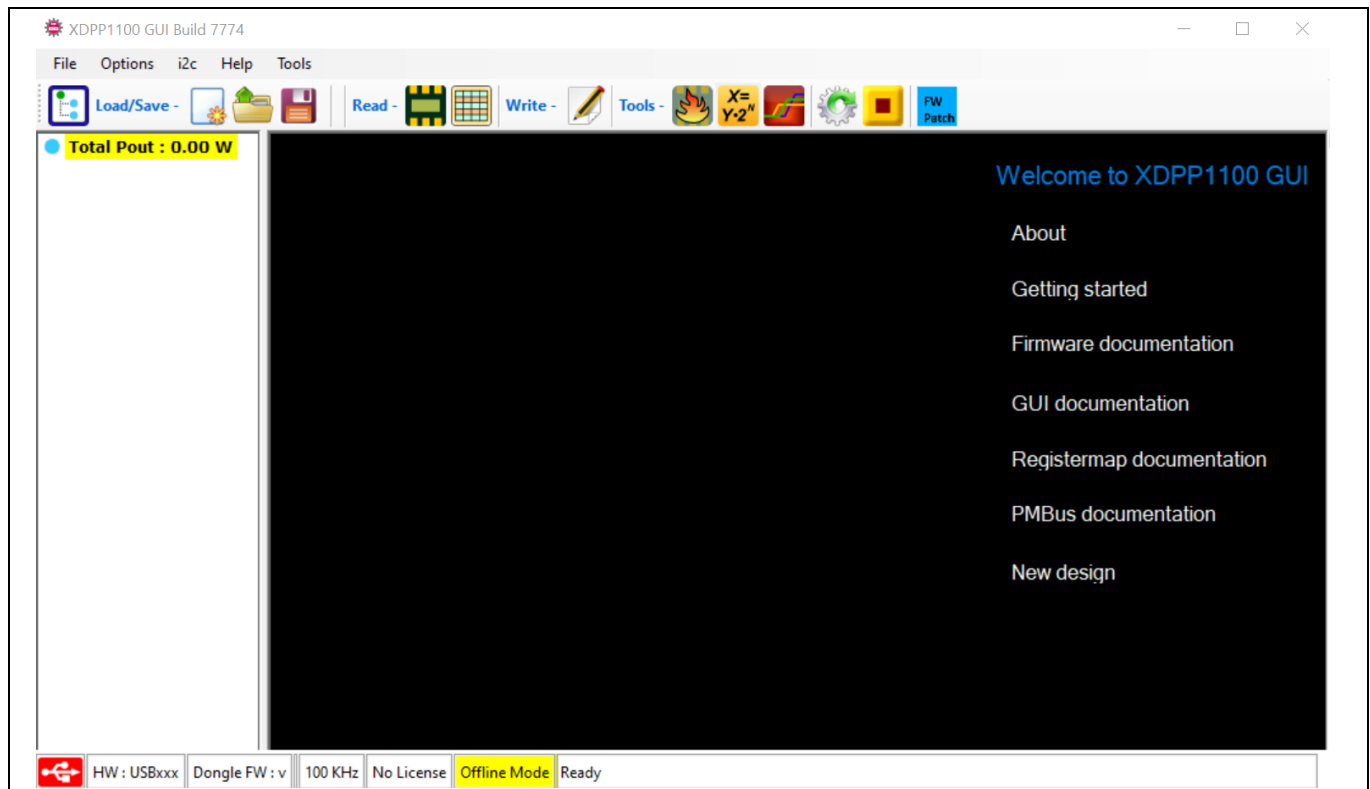


Figure 5 XDPP1100 GUI interface

Programming setup is shown in [Figure 6](#). The computer with XDPP1100 graphical user interface (GUI) connects to the USB GUI dongle USB007A via USB port; the USB dongle USB007A connects to the system board that has the XDPP1100 chip via I²C. The system board should have 3.3 V bias voltage, and XDPP1100 is powered up.

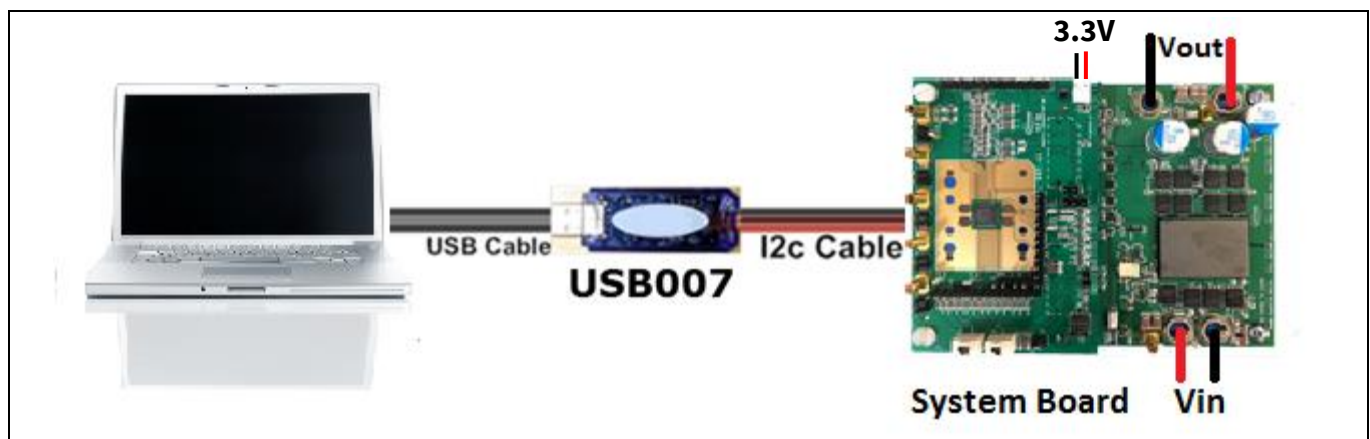


Figure 6 Connection of the programming board, USB dongle and computer

Introduction of XDPP1100 OTP structure

2 Introduction of XDPP1100 OTP structure

A total of 64 kB of one-time programmable (OTP) NVM space is available for the user to store trims, configurations, and patches. By default, 16 kB of memory is allocated for data partition to store trims and configurations. The remaining 48 kB is allocated for patch partitions to store the patches. These sizes of data and patch partitions are configurable and should be kept unchanged after a patch is stored in OTP.

Patch partitions can be further partitioned up to 16 times for storing multiple patches. In a given partition there can only be one active patch. So, to store a new patch in a partition, invalidate the patch before storing a new patch.

The procedure described in this document can be used to load a FW patch into devices which may or may not already contain a previously loaded patch. Care must be taken not to exceed the memory allowance when uploading a new patch.

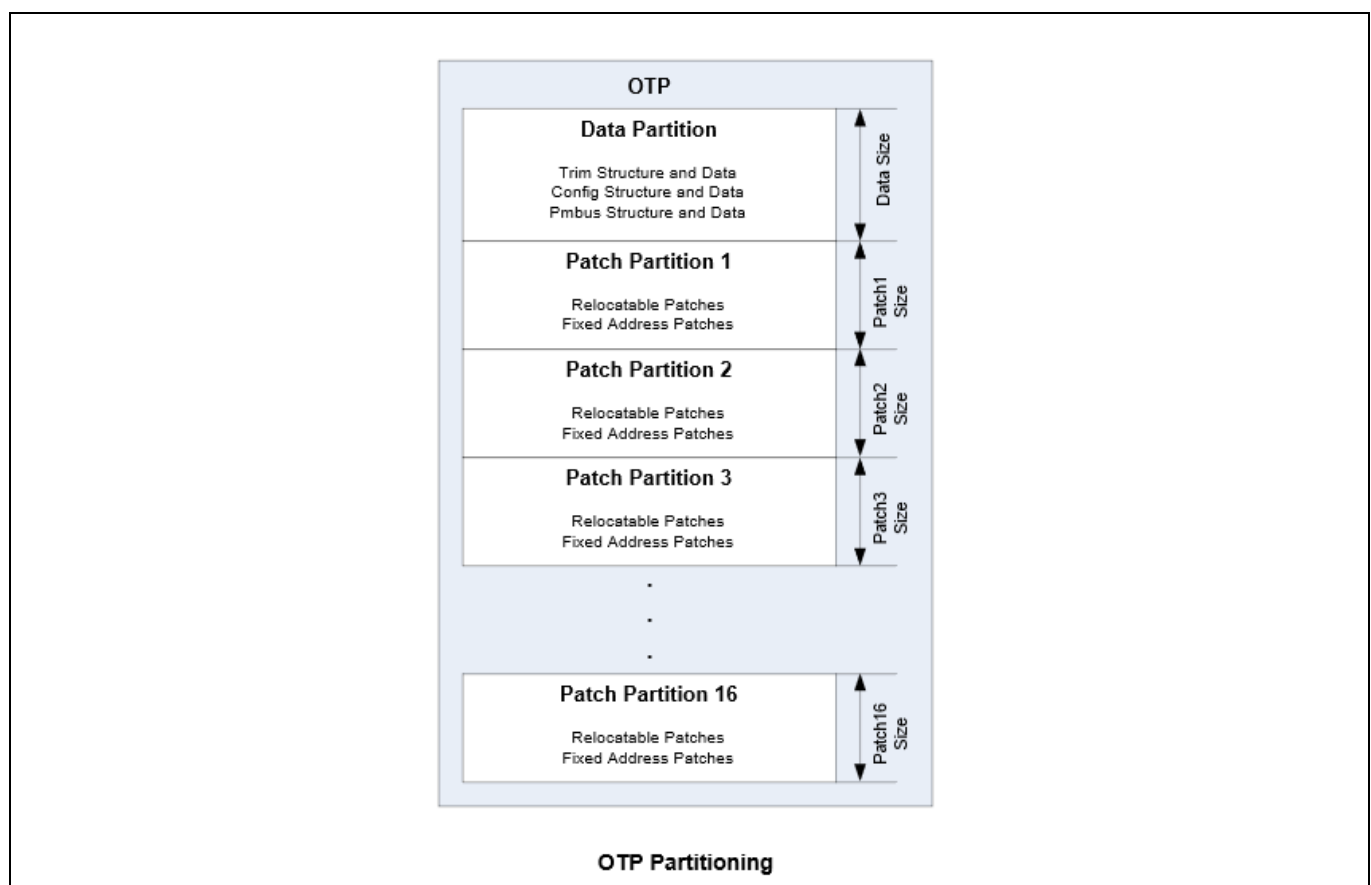


Figure 7 OTP partition

I2C and PMBus communication protocol

3 I²C and PMBus communication protocol

3.1 I²C communication protocol

- › I²C Write 4 Bytes
 - [I²C address] [regaddr0] [regaddr1] [regaddr2] [regaddr3] [data0] [data1] [data2] [data3]
 - Example: Writing to a register address 0x70000400, a value of 0x0000B409
 - [0x20] [0x00] [0x04] [0x00] [0x70] [0x09] [0xB4] [0x00] [0x00]
- › I²C Read 4 Bytes
 - [I²C address] [regaddr0] [regaddr1] [regaddr2] [regaddr3] [I²C address+1] [data0] [data1] [data2] [data3]
 - Example: reading from a register address 0x70000400. The read value is 0x0000B409
 - [0x20] [0x00] [0x04] [0x00] [0x70] [0x21] [0x09] [0xB4] [0x00] [0x00]

3.2 PMBus communication protocol

- › PMBus command Write a Byte
 - [PMBus address] [command] [dataByte]
 - Example: Writing PMBus command VOUT_MODE (0x20) to value of 0x18
 - [0x80] [0x20] [0x18]
- › PMBus command Read a Byte
 - [PMBus address] [command] [PMBus address+1] [data0]
 - Example: Reading a PMBus command VOUT_MODE (0x20). The read value is 0x18.
 - [0x80] [0x20] [0x81] [0x18]
- › PMBus command Write a Word
 - [PMBus address] [command] [dataByte0] [dataByte1]
 - Example: Writing PMBus command TON_RISE (0x61) to value of 0x0050
 - [0x80] [0x61] [0x50] [0x00]
- › PMBus command Read a Word
 - [PMBus address] [command] [PMBus address+1] [dataByte0] [dataByte1]
 - Example: Reading a PMBus command TON_RISE (0x61). The read value is 0x0050.
 - [0x80] [0x61] [0x81] [0x50] [0x00]
- › PMBus Write a Block command
 - [PMBus address] [command] [number of bytes] [dataByte0] [dataByte1] ... [dataByteN]
 - Example: Writing PMBus command MFR_PWM_DEADTIME (0xCF) to value of [0x00] [0x40] [0x00] [0x40] [0x00] [0x60] [0x00] [0x60] [0x00]

I2C and PMBus communication protocol

- [0x80] [0xCF] [0x18] [0x00] [0x40] [0x00] [0x40] [0x00] [0x60] [0x00] [0x60] [0x00]
- › PMBus Read a Block command
 - [PMBus address] [command] [PMBus address+1] [number of bytes] [dataByte0] [dataByte1] ... [dataByteN]
 - Example: Reading a PMBus command MFR_PWM_DEADTIME (0xCF). The read value is [0x00] [0x40] [0x00] [0x40] [0x00] [0x60] [0x00] [0x60] [0x00]
 - [0x80] [0xCF] [0x81] [0x18] [0x00] [0x40] [0x00] [0x40] [0x00] [0x60] [0x00] [0x60] [0x00]

Note: XDPP1100 uses 7-bit addresses for I²C and PMBus. A 7-bit address can define 128 addresses. But some addresses are reserved for SMBus standard and should not be used. See pages 81 to 82 of the SMBus specification.

Note: The I²C protocol's 8-bit address, with bit 0, defines write (0) or read (1). So, when programming the IC, we need to left-shift by one bit, i.e. multiply the address by two. Hence the above examples show the address as 0x20 for I²C and 0x80 for PMBus.

Programming procedure using XDPP1100 GUI

4 Programming procedure using XDPP1100 GUI

4.1 Add device

The XDPP1100 can be added in the GUI by auto-populate or manually adding the device.

4.1.1 Auto-populate device

Use the auto-populate function to detect the device that is active (with 3.3 V bias). Click the “Auto Populate” icon shown in the red block in [Figure 8](#), and a device will be automatically added into the device window.

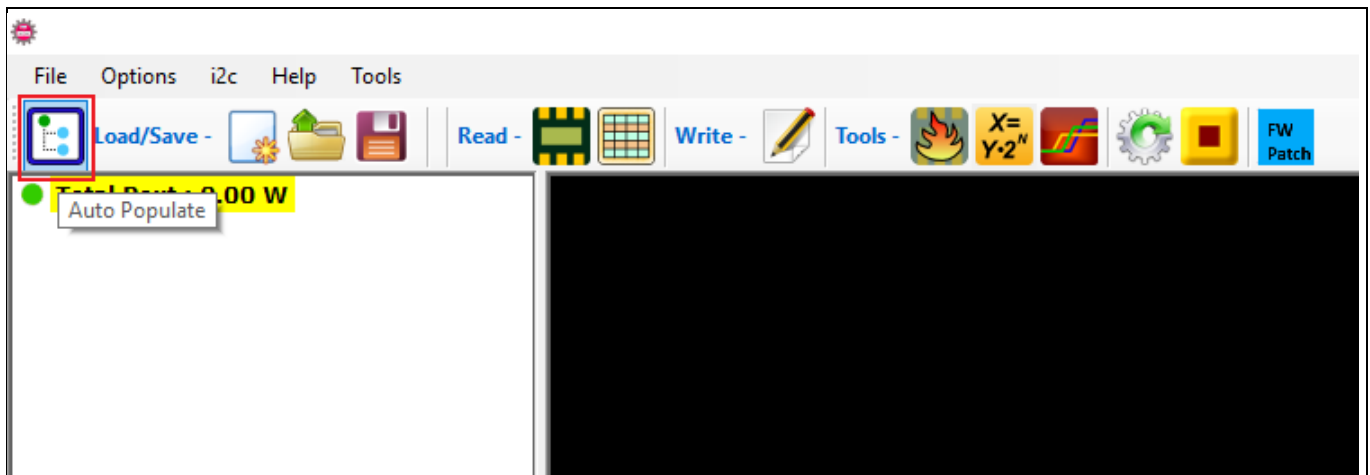


Figure 8 Auto-populate function

4.1.2 Add a new device manually

The user can also manually add a device to the design. Right-click the Device Manager which is highlighted in yellow in the left window ([Figure 9](#)), choose Add Device and follow the arrow to the right to pick a device to use.

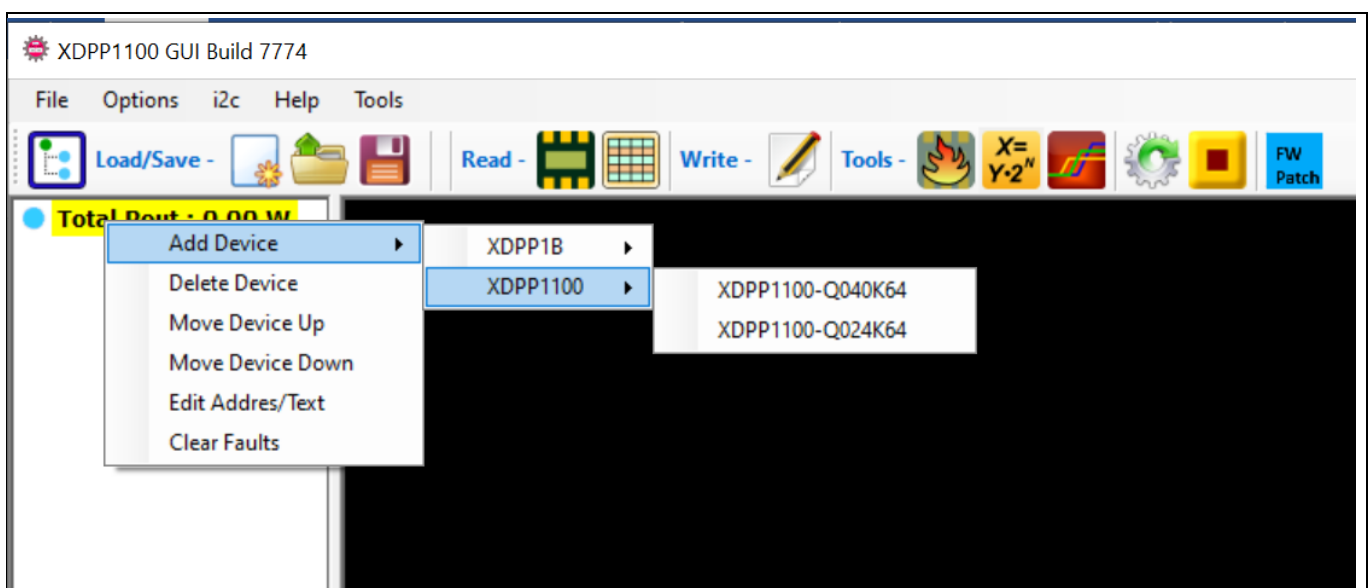


Figure 9 Adding a device manually

Programming procedure using XDPP1100 GUI

If the device is added manually, the default I²C address is 0x10, and the default PMBus address is 0x40, as shown in **Figure 10**. With correct I²C and PMBus addresses, the dot in front of the device will turn blue, indicating the device is ready for I²C communication. If the dot is gray that means the IC is not communicating through I²C; the address may be wrong.

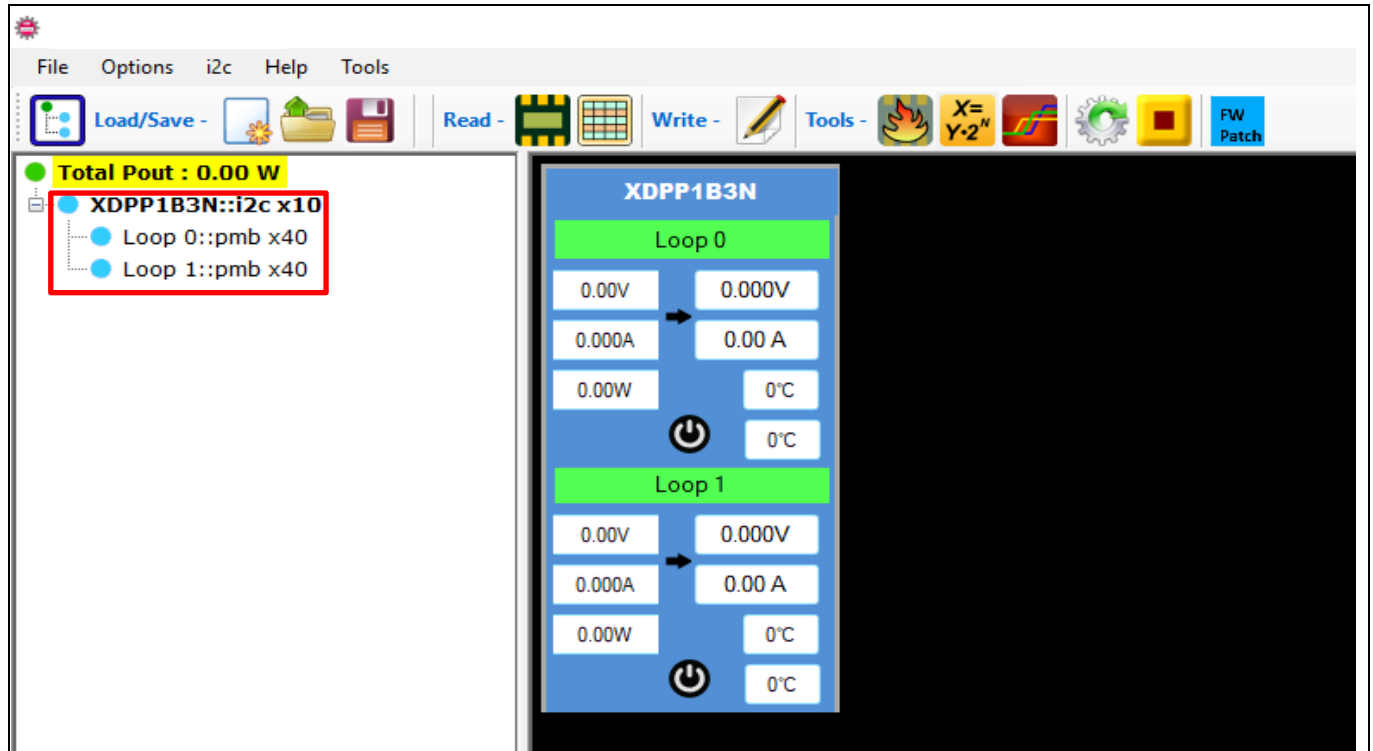


Figure 10 Default I²C and PMBus address

To verify that the assigned address is correct, use the I²C tool to scan the I²C address (**Figure 11**). If the scan result does not match with default values, manually change the I²C and PMBus addresses by right-clicking the device and selecting “Edit Address/Text” (**Figure 12**). Change the I²C and PMBus addresses according to the scanned result. This step is not required with the auto-populate function, as this will scan device addresses automatically.

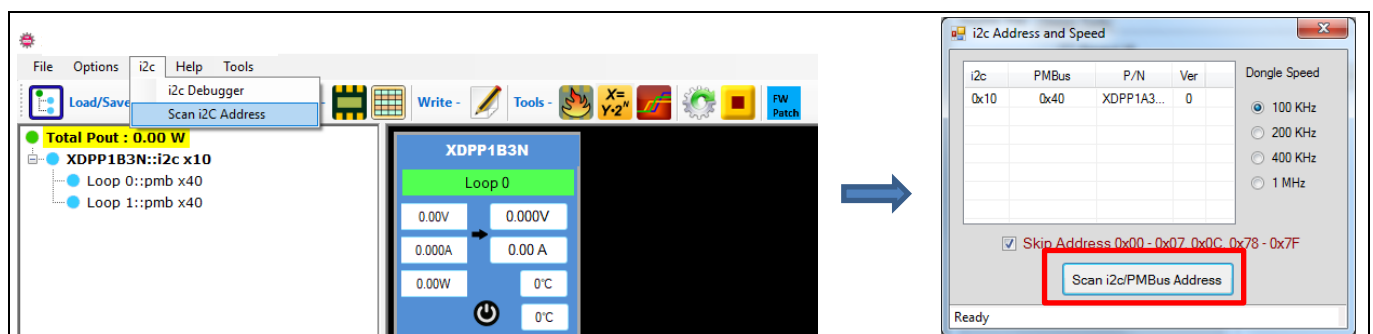


Figure 11 Scan I²C address

Programming procedure using XDPP1100 GUI

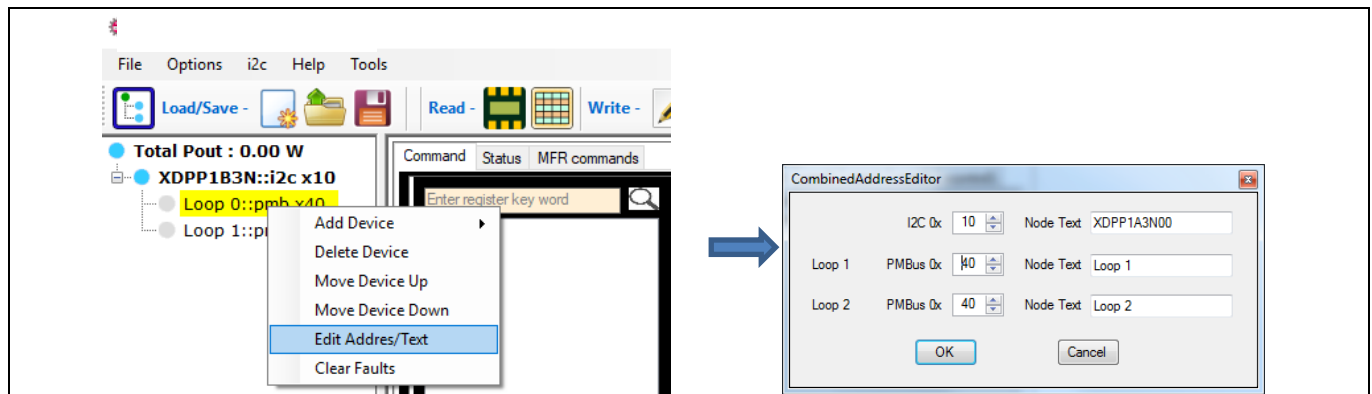


Figure 12 Manually change device address

4.1.3 Force I²C connection

The I²C communication can also be enabled by checking “Force i2c/PMBus OK” (Figure 13). If the USB dongle is connected to a computer, the USB sign in the lower corner will turn green. The I²C status button on the register map page indicates “In Sync” in green.

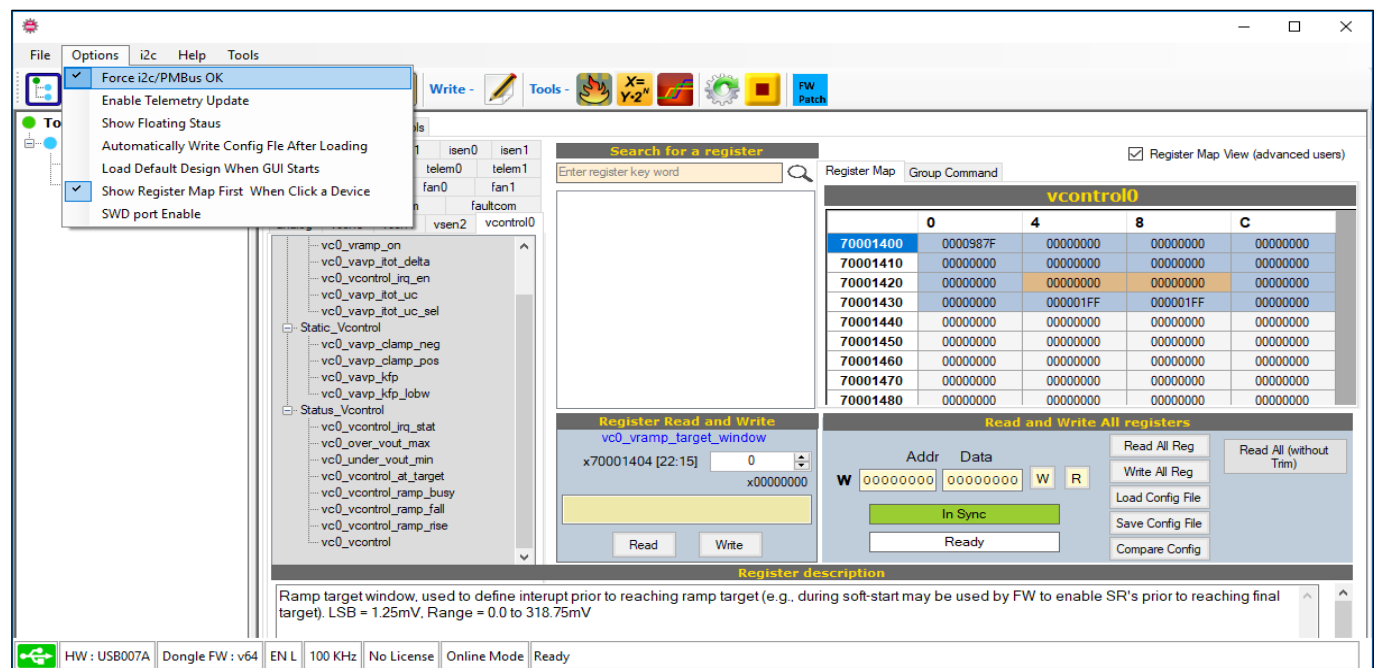


Figure 13 Force I²C/PMBus connection

4.2 Multi Device Programmer tool

Use the GUI “Multi Device Programmer” tool to store the FW patch (when applicable), register configuration and PMBus configuration to OTP.

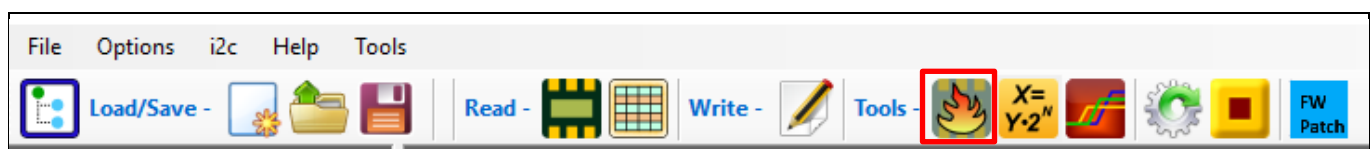


Figure 14 Multi Device Programmer tool

Programming procedure using XDPP1100 GUI

4.3 Check OTP partition

Before storing a patch or configuration, the user should check if the OTP memory has been properly partitioned. Click the “OTP Data” button (Figure 15), and a pop-up window shows all the OTP-related information (Figure 16). Verify the OTP partition and size in the “FW Trim” section by clicking the “Read FW Trim” button.

The data partition can be set up to 0x7C00 or 31 kB. The default data partition is 0x4000 or 16 kB. If a different OTP partition is desired, change the partition size and click the “Store Trim” button. The new partition will be saved in the OTP trim structure.

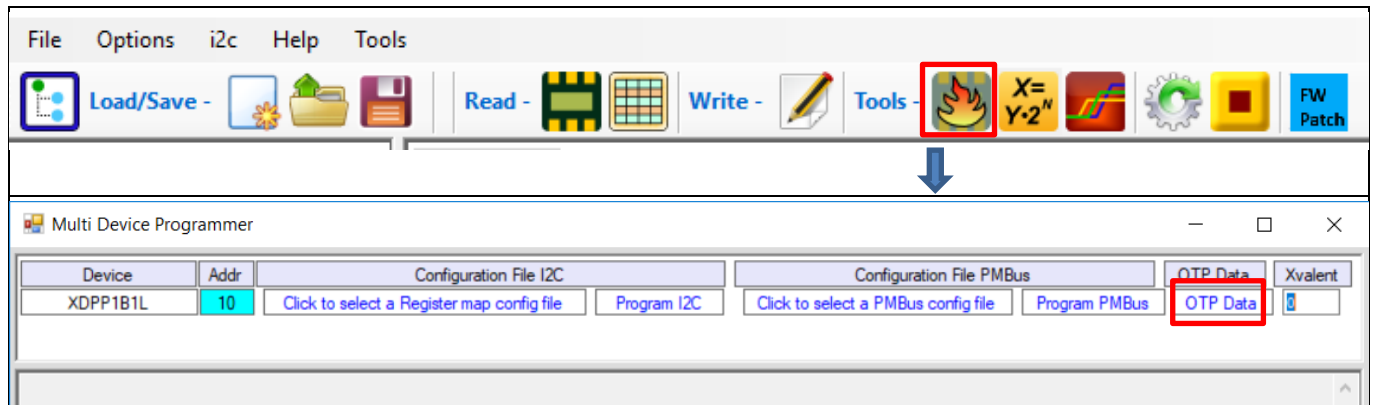


Figure 15 Multi Device Programmer tool – OTP data

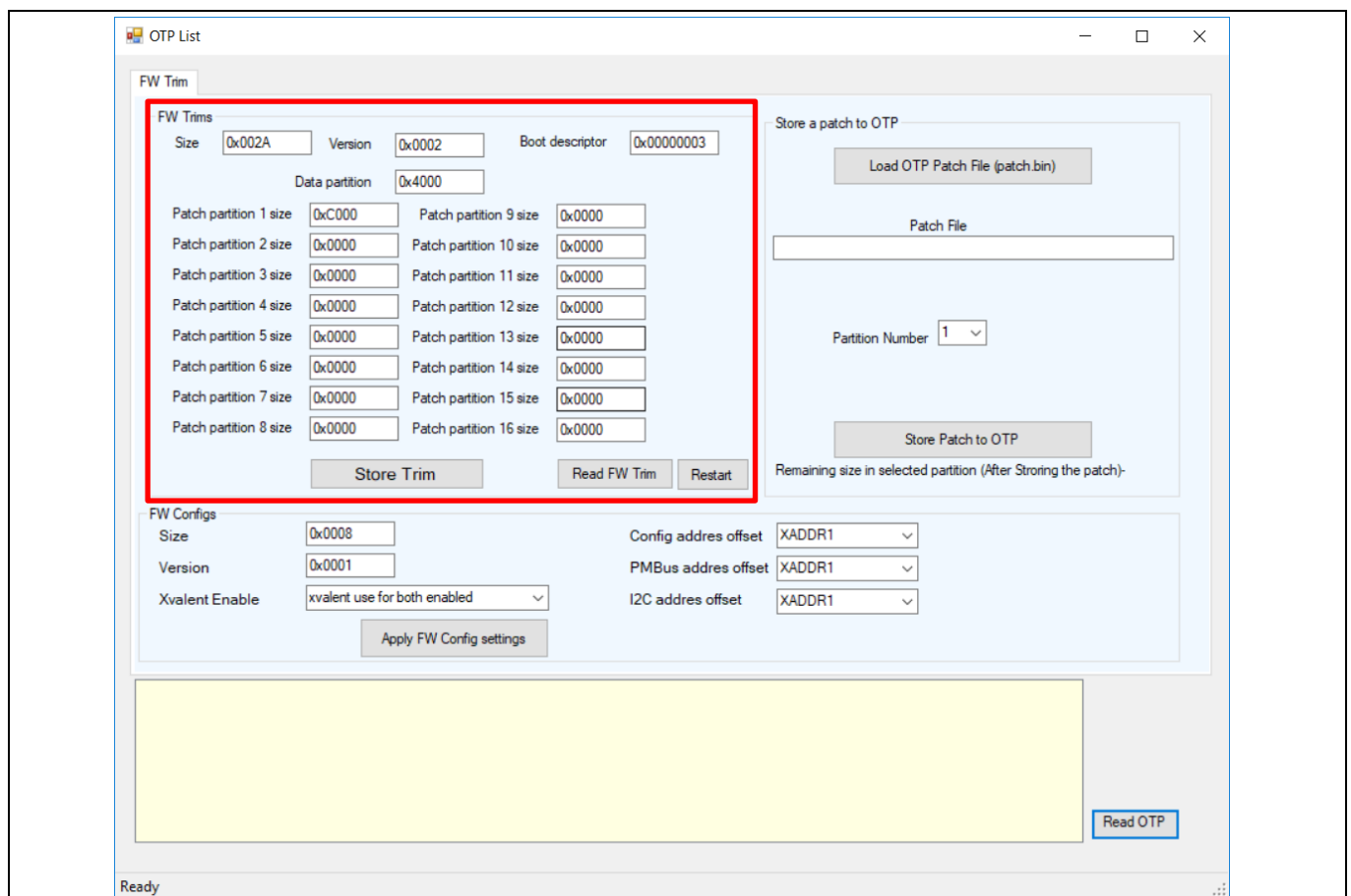


Figure 16 Multi Device Programmer tool – OTP data, FW trims

Programming procedure using XDPP1100 GUI

The number of FW patch rewrites depends on the size of the patch code. The old patch must be invalidated before storing a new patch to the same partition. The facility to invalidate the existing patch is provided in the GUI “FW patch” tool.

4.4 Store FW patch

If a FW patch applies, use the following steps to store the patch to OTP.

- Click on the button “Load OTP Patch File (patch.bin)” (**Figure 17**) to browse for the patch.bin file; select the file and click on OK.
- Select the patch partition (from 1 to 15). By default, patch partition is 1.
- Click on the button “Store Patch to OTP”.
- Optional steps:
 - To verify if a patch is stored in OTP or not, check the data present in the OTP area from starting address to size of the patch file. The last word of the patch file is data CRC.
 - The starting address of the patch partition is 0x1002400, if the data partition size is 16 kB = 0x4000.

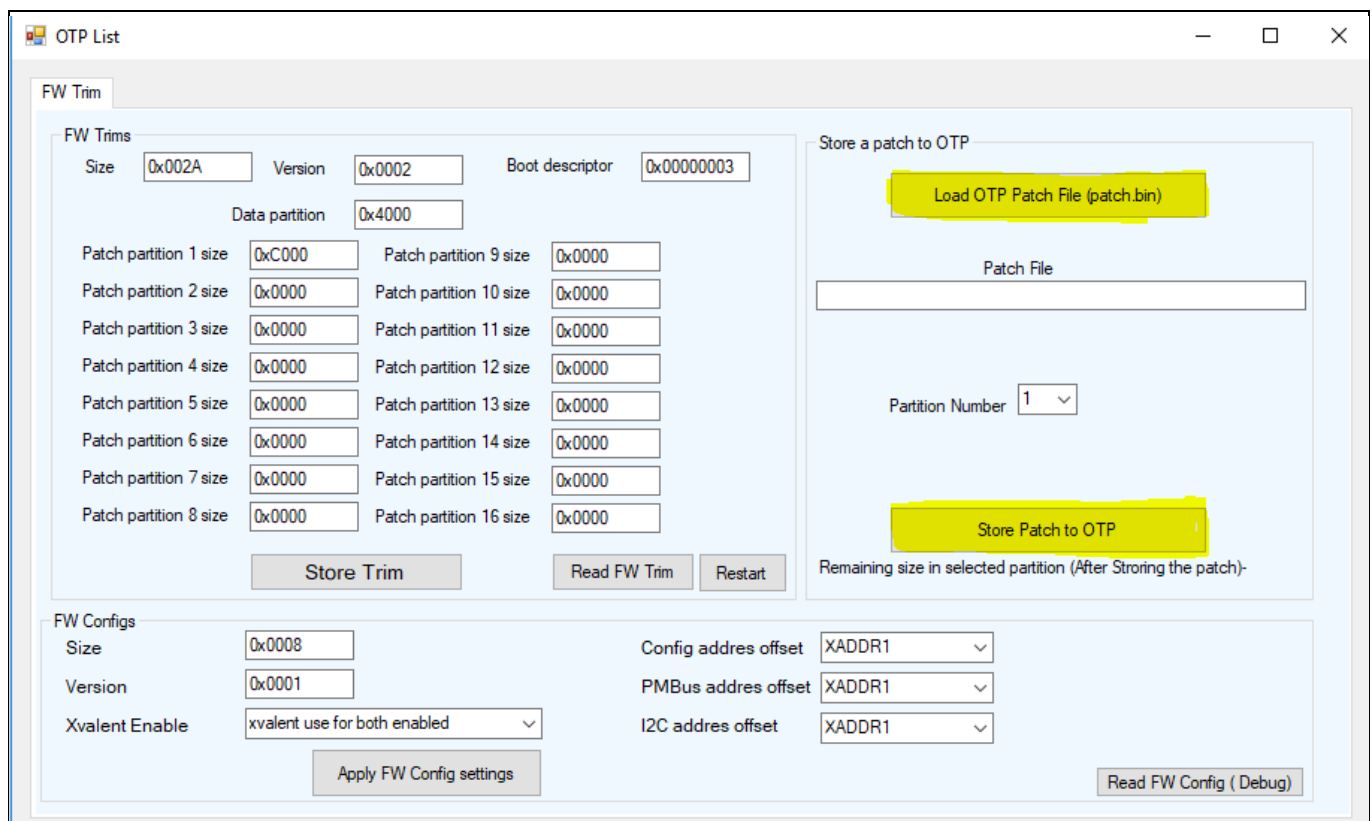


Figure 17 Multi Device Programmer tool – OTP data, store patch to OTP

Note: Size, Version and Boot descriptor are constants – changing these values is not recommended.

Programming procedure using XDPP1100 GUI

4.5 Apply FW configuration

In the “FW Config” section of **Figure 17**, the user can select which XADDR pin is going to be used to define the address offset, and which XADDR pin will program the multi configuration offset (configuration address offset).

For the 40-pin version XDPP1100-Q040, two XADDR pins are available: XADDR1 and XADDR2. It is possible to set XADDR2 as the configuration address offset, making it independent from the PMBus and I²C address offset. For the 24-pin version XDPP1100-Q024, only XADDR1 is available and all of the three offsets should be configured by XADDR1.

4.6 Load design file

The design file should be loaded into RAM before storing in OTP. If this is not yet done, close the Multi Device Programmer and open the design file; click the “Write All (I²C and PMBus)” shortcut button to write all registers and PMBus commands to RAM.

4.7 Store configuration to OTP

Open the Multi Device Programmer, and set the Xvalent number if you have multiple configurations to be stored in OTP. For a single configuration, use the default Xvalent=0. Click the “Program Configuration to OTP” button to store both I²C and PMBus configurations to OTP.

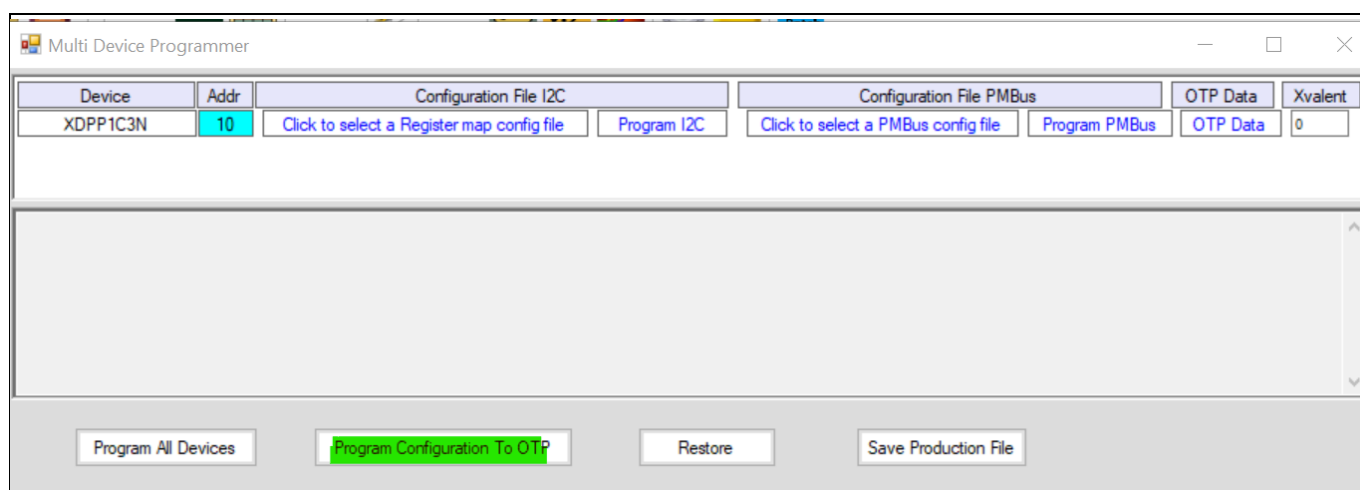


Figure 18 Program configuration to OTP

Partial configuration and multi configuration

5 Partial configuration and multi configuration

5.1 Partial configuration

If only a couple of parameters were modified after the whole configuration was stored in OTP, partial configuration could be used to update these parameters without the need to rewrite the whole configuration. This could save OTP space and allow many more updates. Partial configurations are stored to OTP in the data partition area. Two types of partial configurations are available.

- Partial PMBus configuration
- Partial register configuration

PMBus partial configuration has a limit of 32 partial configurations per Xvalent. Register partial configuration has no limit on the number of stores as long as OTP space is available.

5.1.1 Store PMBus partial configuration

Use PMBus command 0x17 STORE_USER_CODE to write individual PMBus commands to the OTP – for example, if the command 0x55 VIN_OV_FAULT_LIMIT was updated to a new value. Write the new threshold to the 0x55 command. Then write 85 (hex 0x55 equals decimal value 85) to 0x17 STORE_USER_CODE. The new VIN_OV_FAULT_LIMIT will be saved to OTP.

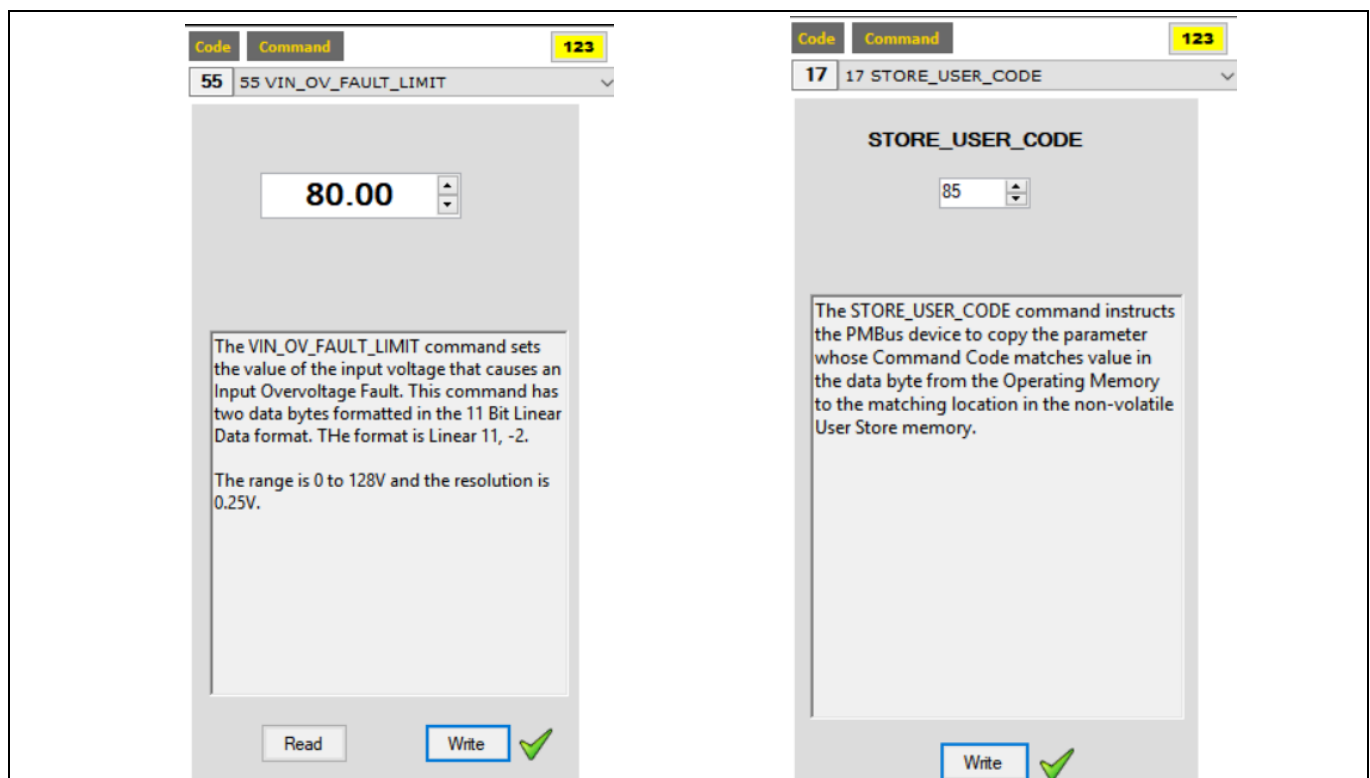


Figure 19 STORE_USER_CODE

5.1.2 Store register partial configuration

To store a register partial configuration to OTP, go to the register map and click the “Group Command” tab (Figure 20). Choose the target register and double-click the register. It will be brought to the “Group Command” table. Go to the ValHex column and modify the register value, then click the “Write Partials” button.

Partial configuration and multi configuration

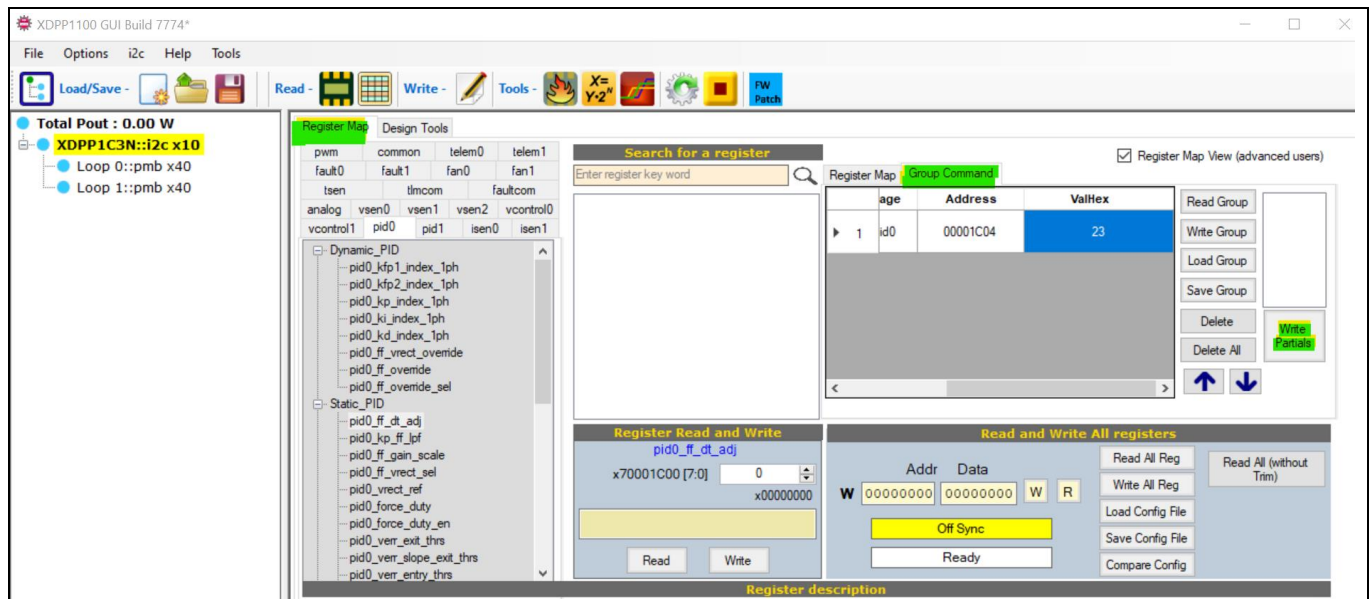


Figure 20 Store register partial configuration using GUI

5.2 Multi configuration

This section uses an example to show how to store multi configurations to OTP and use the XADDR2 pin to select which configuration should be loaded at power-up.

5.2.1 Multi configuration setup

Enable the I²C and PMBus address offset by XADDR resistor, and select the XADDR1 pin for I²C and PMBus offset, or select the XADDR2 pin for multi configuration offset.

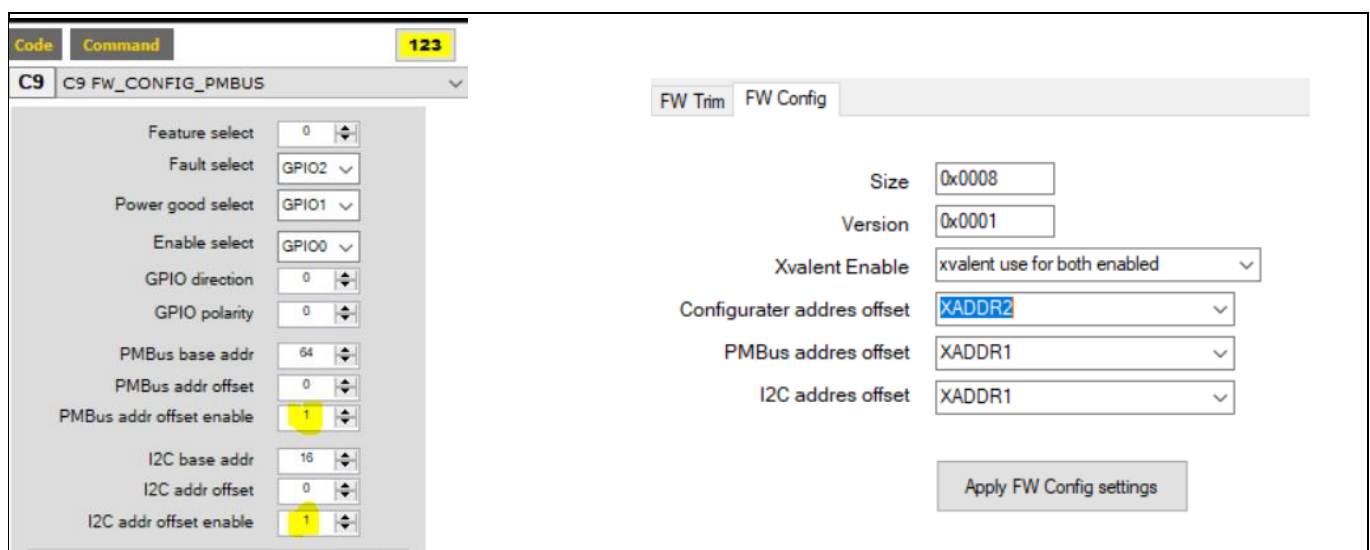


Figure 21 Address offset configuration

Partial configuration and multi configuration

5.2.2 Store the first configuration

Load the first design file, click “Write All” and go to the Multi Device Programmer to store the first configuration with Xvalent=0.

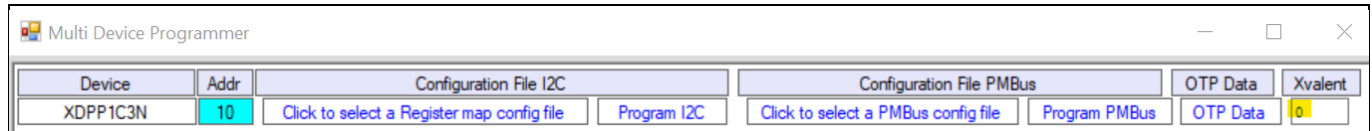


Figure 22 Store configuration to Xvalent=0

5.2.3 Store the second configuration

Load the second design file, click “Write All” and go to the Multi Device Programmer to store the second configuration with Xvalent=1.

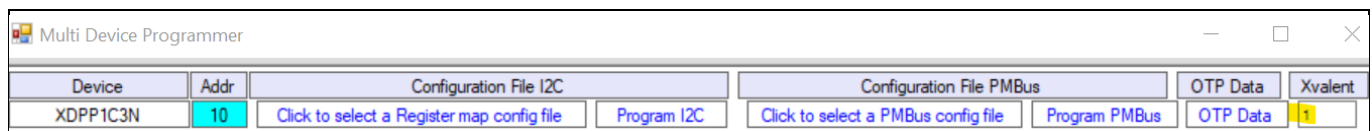


Figure 23 Store configuration to Xvalent=1

5.2.4 Check configuration in OTP

In the FW Patch tool, check “Trim and Config”. Find two configurations with Xvalent=0 and Xvalent=1.

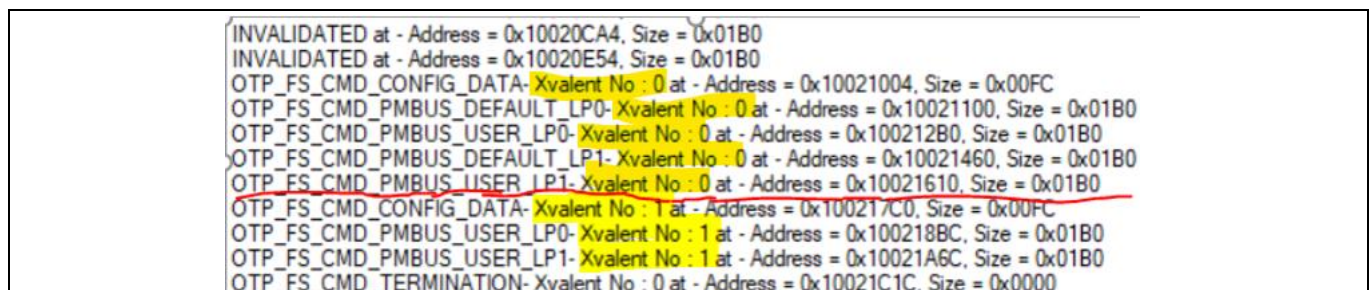


Figure 24 Find configurations in OTP

5.2.5 Verify the configuration with different XADDR resistor

Power cycle 3.3 V V_{DD}, the configuration is selected by the remainder of the config_offset/number_of_configuration.

Power cycle 3.3 V V_{DD}, with 47k address resistor at XADDR2 (offset=0), remainder of config_offset/number_of_configuration = remainder of (0/2)=0, the first configuration was selected.

Power cycle 3.3 V V_{DD} and put 6.8k resistor to XADDR2 (offset=3), remainder of config_offset/number_of_configuration = remainder of (3/2)=1, the second configuration was selected.

Programming procedure using I2C protocol

6 Programming procedure using I²C protocol

6.1 Storing FW trim

The section below provides the details on storing FW trims to OTP for OTP partitioning for data partition, and patch partitions. The below example is for 4 partitions:

- Data partition = 0x4000,
- Patch partition 1 = 0x1000,
- Patch partition 2 = 0x5800,
- Patch partition 3 = 0x5800

The same partition is shown using the GUI in [Figure 25](#) below.



Figure 25 OTP partitioning

- Write data partition of size 0x4000 and patch partition 1 of size 0x2000 at address 0x2005c510
 - Byte0 = 0x00 (data partition size, low byte)
 - Byte1 = 0x40 (data partition size, high byte)
 - Byte2 = 0x00 (patch partition 1 size, low byte)
 - Byte3 = 0x20 (patch partition 1 size, high byte)
- The following data bytes appear on the I²C bus:
[0x20] [0x10] [0xC5] [0x05] [0x20] [0x00] [0x40] [0x00] [0x20]

Programming procedure using I2C protocol

- Write patch partition 2 of size 0xA000 and patch partition 3 of size 0x0000 at address 0x2005c514
 - Byte0 = 0x00 (patch partition 2, low byte)
 - Byte1 = 0xA0 (patch partition 2, high byte)
 - Byte2 = 0x00 (patch partition 3 size, low byte)
 - Byte3 = 0x00 (patch partition 3 size, high byte)
- The following data bytes appear on the I²C bus:
[0x20] [0x14] [0xC5] [0x05] [0x20] [0x00] [0xA0] [0x00] [0x00]
- Write MFR_FIRMWARE_COMMAND PMBus command with the data byte 0x03 to store the trim (including the new partition sizes) to OTP.
 - MFR_FIRMWARE_COMMAND = 0xFE
 - Store trim = 0x03
- The following data bytes appear on the I²C bus:
[0x80] [0x00] [0x00] [0x80] [0xFE] [0x03]

6.2 Storing FW configuration

The section below provides the details on storing FW configuration to OTP to select I²C address offset, PMBus address offset, config offset and Xvalent enabled/disabled.

- Write I²C Address offset, PMBus address offset, Config address offset, and Xvalent enabled at address 0x2005c500
 - Byte0 = 0x00 (Xvalent enabled/disabled)
 - Byte1 = 0x00 (Config address offset)
 - Byte2 = 0x00 (PMBus address offset)
 - Byte3 = 0x00 (I²C Address offset)
- The following data bytes appear on the I²C bus:
[0x20] [0x00] [0xC5] [0x05] [0x20] [0x00] [0x00] [0x00] [0x00]
- Write MFR_FIRMWARE_COMMAND PMBus command with the data byte 0x04 to store configuration to OTP
 - MFR_FIRMWARE_COMMAND = 0xFE
 - Store Config = 0x04
- The following data bytes appear on the I²C bus:
[0x80] [0x00] [0x00] [0x80] [0xFE] [0x04]

Programming procedure using I2C protocol

6.3 Patch file format

Patches are released in standard .bin format. The data format of the patch file is shown below:

Byte 15	Byte 14	Byte 13	Byte 12	Byte 11	Byte 10	Byte 9	Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Location of patch				CRC header				Size redundant		Size		Option		Reser ved	cmd
data1				Version new				Version base				Execute			
...			Data3...		Data2...	
...															
						EOF		CRC data		dataN		

Figure 26 Patch file format

6.4 Storing a FW patch through I²C protocol

- Parse the patch.bin file to read out header information and actual data information.
- The header has the important information to store the patch:
 - Command and size
 - A command value of 0x10 indicates an active patch
 - A size of 0x1000 indicates 4 kB of active patch
 - Header CRC
 - Location of patch will determine the patch starting address; make sure the patch is prepared for the correct partition
 - Example: A patch is prepared for partition 1, and partition 1 starting address is 0x10024000. Location of patch should be 0x10024000.
 - Version base and version new
 - This will be coming from the partial patch_ config file in the patch project
 - Actual data to be stored to OTP
 - Data CRC
 - Example: Below is the patch.bin file

Figure 27 shows the header information of a patch.

Header Info			
CMD	16	Option	0000
Size	1000	Size Redundant	1000
Header CRC	5BFD1D7F	Data CRC	36937D5E
Location of Patch	10024000	Execution of Patch	100245FD
Version Base	5C843EFA	Version new	5971BBBB

Figure 27 Patch header information

Programming procedure using I2C protocol

Address	1	2	3
00000000	00000010	10001000	5bfd1d7f 10024000
00000010	100245fd	5c843efa	5971bbbb 4604b510
00000020	f0004620	4903fa87	460a4620 f84ef000
00000030	0000bd10	00030d40	4806b510 28006800
00000040	6901d002	47882004	49032001 60080440
00000050	0000bd10	20063a04	e000e280 4806b510
00000060	28006840	6901d002	47882005 49032001
00000070	60080480	0000bd10	20063a04 e000e280
00000080	2800b538	4c0ed10e	28006820 68c1d00b
00000090	47882004	90006960	68856820 34082004
000000a0	47a8cc0e	4807bd38	20006020 480460a0
000000b0	30082301	60417003	49036082 e7e960c1
000000c0	20063a04	100054cc	10024039 2800b538
000000d0	4c0fd10f	28006860	68c1d00c 47882005
000000e0	6a40480b	68609000	20056885 cc0e3418
000000f0	bd3847a8	60604807	61a02000 23014804
00000100	70033018	60826041	60c14903 0000e7e8

Figure 28 patch.bin file

- Read the patch file data into chunks of 1 kB (0x400) and loop through the size of the patch (from patch.bin).
- Follow the below steps for first 1 kB of data:
 - Store the header and data to the scratch pad area, starting at the address of 0x20061800.
 - Write the first 1 kB (0x400) values to the address from 0x20061800 to 0x20061BFC.
 - Example: From the above patch.bin file:
 - [0x20] [0x00] [0x18] [0x06] [0x20] [0x10] [0x00] [0x00] [0x00]
 - [0x20] [0x04] [0x18] [0x06] [0x20] [0x00] [0x10] [0x00] [0x10]
 - [0x20] [0x08] [0x18] [0x06] [0x20] [0x7F] [0x1D] [0xFD] [0x5B]
 - [0x20] [0x0C] [0x18] [0x06] [0x20] [0xBB] [0xBB] [0x71] [0x59]
 -
 - [0x20] [0xFC] [0x1B] [0x06] [0x20] [0x5E] [0x7D] [0x93] [0x36]
 - After this step, you will have the patch.bin file's first 1 kB of data placed on the scratch pad.
 - Scratch pad has the below data:

	0	4	8	C
20061800	00000010	10001000	5BFD1D7F	10024000
20061810	100245FD	5C843EFA	5971BBBB	4604B510
20061820	F0004620	4903FA87	460A4620	F84EF000
20061830	0000BD10	00030D40	4806B510	28006800
20061840	6901D002	47882004	49032001	60080440
20061850	0000BD10	20063A04	E000E280	4806B510
20061860	28006840	6901D002	47882005	49032001
20061870	60080480	0000BD10	20063A04	E000E280
20061880	2800B538	4C0ED10E	28006820	68C1D00B
20061890	47882004	90006960	68856820	34082004
200618A0	47A8CC0E	4807BD38	20006020	480460A0
200618B0	30082301	60417003	49036082	E7E960C1
200618C0	20063A04	100054CC	10024039	2800B538

Figure 29 Scratch pad data

Programming procedure using I2C protocol

- Write **MFR_FIRMWARE_COMMAND_DATA** PMBus command with the data options as below.
Byte 0 and Byte 1 includes the size of bytes ($1\text{ kB} = 1024 = 4 * 256 + 0$). Byte 2 will provide information included or not on the header. Byte 3 will provide information on the partition number.
- PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
- MFR_FIRMWARE_COMMAND_DATA = 0xFD
- writeData[4]
 - writeData[0] = 0
 - writeData[1] = 4 // $1024 = 256 * 4 + 0$ -> Byte1(MSB) = 4 and Byte0(LSB) = 0
 - writeData[2] = 1 // header is included
 - writeData[3] = 1 // partition number = 1
- Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x04] [0x01] [0x01]

Note: Description of the data bytes:

[0x80] [PMBus address]

[0x00] [PMBus page command code]

[0x00] [page number: 0 = loop 0, 1 = loop 1]

[0x80] [PMBus address]

[0xFD] [PMBus command code]

[0x04] [length (n): 4 = 4 bytes data]

[0x00] [0x04] [0x01] [0x01] [byte0] [byte1] [byte2][Byten-1]

- Write **MFR_FIRMWARE_COMMAND** PMBus command with the data byte as 0x06 to store the patch to OTP.
- PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 6)
- MFR_FIRMWARE_COMMAND = 0xFE
- Store register configuration = an argument of 0x6
- Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x06]
- Follow the below steps for the next 1 kB of data.
 - For example, if patch.bin file has a size of 0x1000 (4 kB). This loop will run three times, each time storing 1 kB of data to the scratch pad and then moving the data to store to OTP.
 - Store the next 1 kB of data to the scratch pad area, starting at the address of 0x20061800.
 - Write **MFR_FIRMWARE_COMMAND_DATA** PMBus command with the data options as below.
Byte 0 and Byte 1 includes the size of bytes ($1\text{ kB} = 1024 = 4 * 256 + 0$). Byte 2 will provide

Programming procedure using I2C protocol

information included or not on the header. Byte 3 will provide information on the partition number.

- PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
- MFR_FIRMWARE_COMMAND_DATA = 0xFD
- writeData[4]
 - writeData[0] = 0
 - writeData[1] = 4 // $1024 = 256 * 4 + 0$ -> Byte1(MSB) = 4 and Byte0(LSB) = 0
 - writeData[2] = 0 // no header is included
 - writeData[3] = 1 // partition number = 1
- Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x04] [0x00] [0x01]
- Write MFR_FIRMWARE_COMMAND PMBus command with the data byte as 0x06 to store the patch to OTP
- PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 6)
- MFR_FIRMWARE_COMMAND = 0xFE
- Store register configuration = an argument of 0x6
- Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x06]
- Restart to make sure the changes of patch will take effect, by writing MFR_FIRMWARE_COMMAND PMBus command with the data byte as 0xE
 - PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 0xE)
 - MFR_FIRMWARE_COMMAND = 0xFE
 - Store register configuration = an argument of 0xE
 - Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x0E]

Note: It is advisable to store a patch before storing the configuration to OTP, as the stored patch is adding MFR PMBus commands. This is changing the total number of supported commands and hence storing a configuration later will include both standard commands as well as MFR commands to OTP. The sequence of storing patch and configuration is important if there are newly added MFR commands in the patch.

Note: Storing a patch to an IC that already has a configuration stored: Storing a patch with added MFR commands will create a conflict in terms of total number of supported commands and size of the configuration, as the previously stored configuration does not include MFR commands and the patch is supporting new commands. In this case, it is advisable to invalidate the previous configuration and store a patch first and then store the PMBus configurations.

Programming procedure using I2C protocol

Note: After storing a patch, invalidate the patch and store a new patch. The latest active patch is always remapped to the beginning of the partition by FW. So reading the OTP always shows the active patch at the beginning of the patch partition.

6.5 Configuration file format

The data format of the configuration file is shown in **Figure 30**.

Byte 15	Byte 14	Byte 13	Byte 12	Byte 11	Byte 10	Byte 9	Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
data1				CRC header				Size redundant		Size		Option		Xvalue nt	cmd
data5				data4				data3				data2			
...			data7...		data6...	
...															
						EOF		CRC data		dataN		

Figure 30 Configuration file format

6.6 Storing full configuration using I²C port

- Write the configuration to RAM – both register configuration and PMBus commands.
 - Write register buffer values into the respective address by following the I²C protocol.
 - Write all PMBus commands for both the loops (based on the selected part).
- Follow the below steps to write the register configuration from RAM to OTP.
 - Write **MFR_FIRMWARE_COMMAND_DATA** PMBus command with the data options as below. Byte 0 mentions the Xvalent number and other bytes can be ignored or simply written as zeroes.
 - PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
 - MFR_FIRMWARE_COMMAND_DATA = 0xFD
 - writeData[4]
 - writeData[0] = 0 //Xvalent number
 - writeData[1] = 0
 - writeData[2] = 0
 - writeData[3] = 0
 - Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x00] [0x00] [0x00]
 - Write to register address FW_CONFIG (0x2005c500) the below data bytes:
 - Byte0 = 8 //Config size LSB
 - Byte1 = 0 //MSB config size MSB
 - Byte2 = 1 //Config version LSB

Programming procedure using I2C protocol

- Byte3 = 0 //Config version
- Following data bytes appear on the I²C bus:
 - [0x20] [0x00] [0xC5] [0x05] [0x20] [0x08] [0x00] [0x01] [0x00]
- Write to register address fw_config_xv_enable (0x2005c504) the below data bytes:
 - Byte0 = 1 //Config Xv enabled (0 – Xvalent use disabled, 1 – Xvalent use enabled)
 - Byte1 = 0 //Configurator address (0 = XADDR1, 1 = XADDR2)
 - Byte2 = 0 //PMBus offset (0 = XADDR1, 1 = XADDR2)
 - Byte3 = 0 //I²C offset (0 = XADDR1, 1 = XADDR2)
- Following data bytes appear on the I²C bus:
 - [0x20] [0x04] [0xC5] [0x05] [0x20] [0x01] [0x00] [0x00] [0x00]
- Write **MFR_FIRMWARE_COMMAND** PMBus command with the data byte as 0x04 to store register configuration.
- PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 4)
- MFR_FIRMWARE_COMMAND = 0xFE
- Store register configuration = an argument of 0x4
- Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x04]
- Follow the below steps to write the PMBus configuration from RAM to OTP:
 - Write **MFR_FIRMWARE_COMMAND_DATA** PMBus command with the data options as below. Byte 0 mentions the Xvalent number and other bytes can be ignored or simply written as zeroes.
 - PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
 - MFR_FIRMWARE_COMMAND_DATA = 0xFD
 - writeData[4]
 - writeData[0] = 0 //Xvalent number
 - writeData[1] = 0
 - writeData[2] = 0
 - writeData[3] = 0
 - Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x00] [0x00] [0x00]
 - Write PMBus command STORE_DEFAULT_ALL
 - PMBus command (STORE_DEFAULT_ALL)
 - Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0x11]
 - Write PMBus commands STORE_USER_ALL

Programming procedure using I2C protocol

- PMBus command (STORE_USER_ALL)
- Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0x15]

6.7 Store PMBus partial configuration

Below are the steps for writing partial PMBus configuration to OTP. PMBus partial configuration has a limit of 32 partial configurations per Xvalent.

- Write the PMBus configuration (PMBus commands) to the RAM.
- Example: Writing PMBus command TON_RISE (0x61) to value of 0x0050
- [0x80] [0x00] [0x00] [0x80] [0x61] [0x50] [0x00]
- Write STORE_USER_CODE PMBus command with data byte as command code to store partial configuration to OTP.
- Example: Writing PMBus command STORE_USER_CODE (0x17) to value of 0x61
- [0x80] [0x00] [0x00] [0x80] [0x17] [0x61]

6.8 Store register partial configuration

Below are the steps for writing partial register configuration to OTP. Register partial configuration has no limit on the number of stores (as long as OTP space is available).

- Write the register address and data to the scratch pad area, starting at the address of 0x20061800. (RAM_SCRATCHPAD_ADDRESS) and 0x20061804 (RAM_SCRATCHPAD_ADDRESS + 4).
- Example: To store register “pid0_kp_index_1ph” – address “0x70001C04” a value of “0x00C32DF7”
- [0x20] [0x00] [0x18] [0x06] [0x20] [0x04] [0x1C] [0x00] [0x70]
- [0x20] [0x04] [0x18] [0x06] [0x20] [0xF7] [0x2D] [0xC3] [0x00]
- After this step, register address and data are placed on the scratch pad.
- Scratch pad has the below data:

	0	4	8	C
20061800	70001C04	00C32DF7	0000000A	00180018
20061810	E3542488	70001C04	00C32DF7	7337B8C6
20061820	75C6FC98	A10C63C1	F77AF16F	70257214
20061830	982C670F	8C3A8411	2C1C0FA1	FD4A190B
20061840	783B5772	AF244835	56035F8E	E80580C2
20061850	7A3CAC7D	6855D667	DDC3F3E7	A76E8068
20061860	A5ABE7B6	1E7EF6BC	75D0CF3E	4D0B1000

Figure 31 Scratch pad data

- Write PMBus command 0xFD **MFR_FIRMWARE_COMMAND_DATA** with the data options as below. Byte 0 is for Xvalent number. Byte 1 and Byte 2 include the size and total number of bytes to move to OTP. Byte 3 is not used.
- PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)

Programming procedure using I2C protocol

- MFR_FIRMWARE_COMMAND_DATA = 0xFD
 - writeData[0] = 0 //Xvalent number
 - writeData[1] = 8 // for one register (4 bytes for address and 4 bytes of data = 8 bytes)
 - writeData[2] = 0
 - writeData[3] = 0
- Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x08] [0x00] [0x00]
- Write PMBus command 0xFE **MFR_FIRMWARE_COMMAND** with the data byte 0x14 to store partial configuration OTP.
- PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 0x14)
- MFR_FIRMWARE_COMMAND = 0xFE
- Store register configuration = an argument of 0x14
- Following data bytes appear on the I²C bus (see Note: for the description of the data byte):
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x14]

6.9 Procedure to find remaining free memory in OTP

The section explains how to find the remaining size of the OTP, so the programmer knows if the IC has enough space to store a new configuration or a patch.

6.9.1 Procedure to check remaining space in data partition to store a new configuration

- Write **MFR_FIRMWARE_COMMAND_DATA** PMBus command with the data options as below. Byte 3 will provide information on the partition number.
- PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
- MFR_FIRMWARE_COMMAND_DATA = 0xFD
- writeData[4]
 - writeData[0] = 0 // not used
 - writeData[1] = 0 // not used
 - writeData[2] = 0 // not used
 - writeData[3] = 0 // partition number = 0
- Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x00] [0x00] [0x00]

Note: Description of the data bytes:

[0x80] [PMBus address]

Programming procedure using I2C protocol

[0x00] [PMBus page command code]

[0x00] [page number: 0 = loop 0, 1 = loop 1]

[0x80] [PMBus address]

[0xFD] [PMBus command code]

[0x04] [length (n): 4 = 4 bytes data]

[0x00] [0x00] [0x00] [0x00] [byte0] [byte1] [byte2][Byten-1]

- Write **MFR_FIRMWARE_COMMAND** PMBus command with the data byte as 0x10 to find the remaining size in the selected partition in OTP.
 - PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 0x10)
 - MFR_FIRMWARE_COMMAND = 0xFE
 - Store register configuration = an argument of 0x10
 - Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x10]
- Read **MFR_FIRMWARE_COMMAND_DATA** PMBus command. The return value provides the remaining space in number of bytes.
 - PMBus Read Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
 - Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x81] [0x04] [0x00] [0x20] [0x00] [0x00]
 - In the above example, the return value is 0x00004000. So, the free space in bytes = 0x2000 = 8 kB.

6.9.2 Procedure to find remaining space in patch partition to store a patch

- Write **MFR_FIRMWARE_COMMAND_DATA** PMBus command with the data options as below. Byte 3 will provide information on the partition number.
 - PMBus Write Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
 - MFR_FIRMWARE_COMMAND_DATA = 0xFD
 - writeData[4]
 - writeData[0] = 0 // not used
 - writeData[1] = 0 // not used
 - writeData[2] = 0 // not used
 - writeData[3] = 1 // partition number = 1
 - Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x04] [0x00] [0x00] [0x00] [0x01]

Note: Description of the data bytes:

Programming procedure using I2C protocol

[0x80] [PMBus address]

[0x00] [PMBus page command code]

[0x00] [page number: 0 = loop 0, 1 = loop 1]

[0x80] [PMBus address]

[0xFD] [PMBus command code]

[0x04] [length (n): 4 = 4 bytes data]

[0x00] [0x00] [0x00] [0x01] [byte0] [byte1] [byte2][Byten-1]

- Write **MFR_FIRMWARE_COMMAND** PMBus command with the data byte as 0x10 to find the remaining size in the selected partition in OTP.
 - PMBus command Write Byte (MFR_FIRMWARE_COMMAND, 0x10)
 - MFR_FIRMWARE_COMMAND = 0xFE
 - Store register configuration = an argument of 0x10
 - Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFE] [0x10]
- Read **MFR_FIRMWARE_COMMAND_DATA** PMBus command. The return value provides the remaining space in number of bytes.
 - PMBus Read Block command (MFR_FIRMWARE_COMMAND_DATA, writeData)
 - Following data bytes appear on the I²C bus:
 - [0x80] [0x00] [0x00] [0x80] [0xFD] [0x81] [0x04] [0x00] [0x40] [0x00] [0x00]
 - In the above example, the return value is 0x00004000. So, the free space in bytes = 0x4000 = 16 kB.

Invalidating FW patch and configurations

7 Invalidating FW patch and configurations

7.1 Invalidating FW patch and configuration by GUI

In GUI FW Patch tool, choose the OTP partition number and click the “Invalidate a patch in a partition” to invalidate the patch in that partition. There is an option to invalidate the configuration right after invalidating the patch. This is recommended if there are MFR commands defined in the patch. Both register and PMBus configuration would be invalidated when this option is checked. To keep the configuration in OTP, uncheck “Invalidate configuration”.

The tool also provides multiple options to invalidate register configuration, invalidate PMBus configuration, invalidate register partial configuration (“Invalidate Config Partial”) and invalidate PMBus partial configuration.

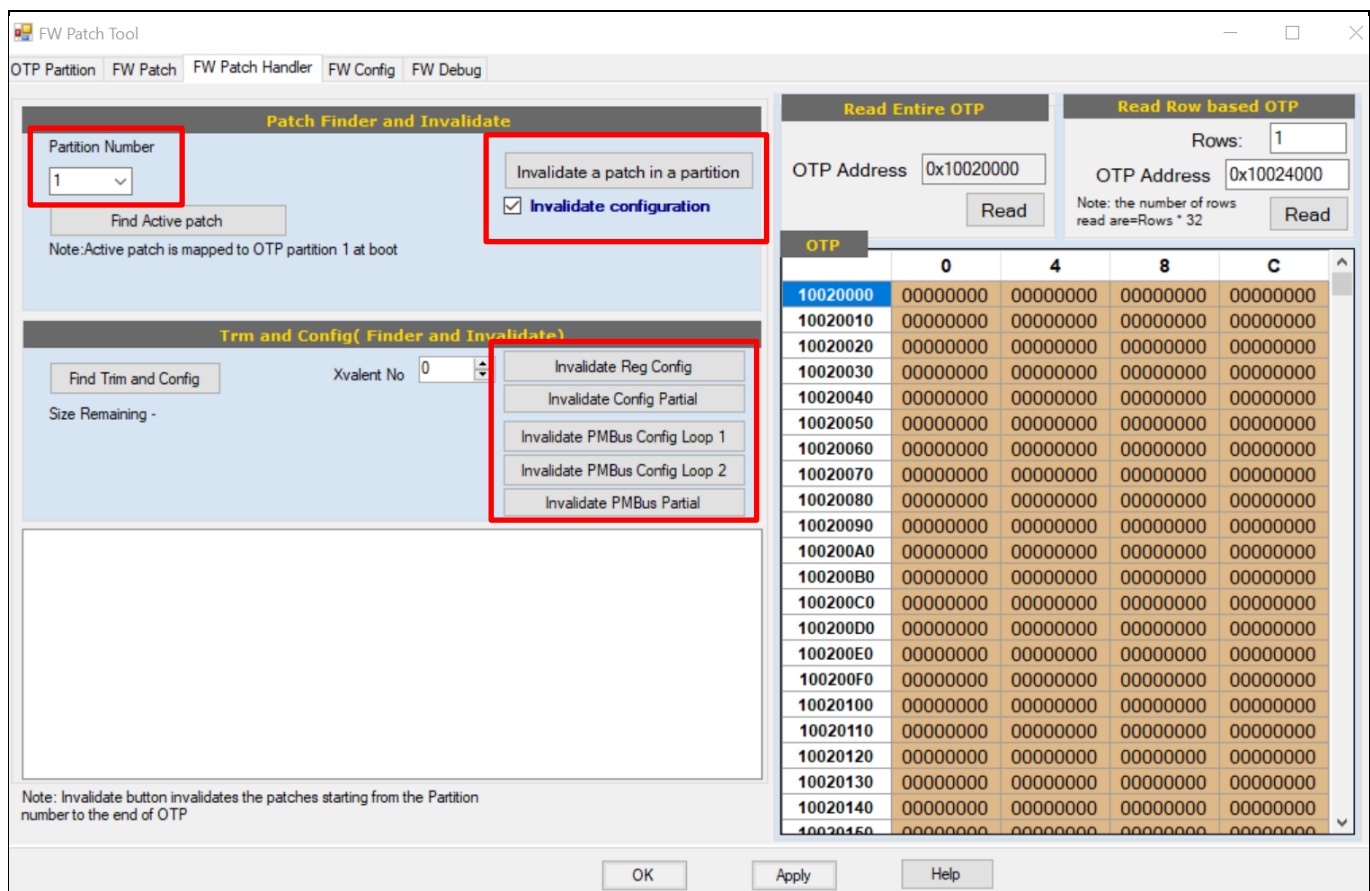


Figure 32 FW patch tool – invalidate patch and configuration

7.2 Invalidating FW patch

Code Listing 1

```
Try
// 1. Disable the MMU
// Store the boot descriptor into temp variable, to load back again at the end of the function
// boot_descriptor_addr = "0x2005C544"
i2c Read 4Bytes (boot_descriptor_addr, data2)
// Disable MMU by writing 0xF into boot descriptor register
i2c Write 4Bytes("0000000F", boot_descriptor_addr)
```

Invalidating FW patch and configurations

Code Listing 1

```

// 2.Reset after disabling the MMU
    PMBus Write Byte command (MFR_FIRMWARE_COMMAND, 0xE)

// 3. Pass the parameters to FIRMWARE COMMAND DATA – PMBus command to invalidate the patch
    writeData(0) = &H10           // Patch file command type (0x10)
    writeData(1) = 0
    writeData(2) = 0
    writeData(3) = partition_number // Partition number – to invalidate partition – “1”
    PMBus Write Block(MFR_FIRMWARE_COMMAND_DATA, writeData)

// 4 .Execute the PMBus command (FIRMWARE COMMAND) to invalidate the patch – pass the argument as 0x12 to invalidate
    PMBus Write Byte command (MFR_FIRMWARE_COMMAND, &H12)

// 5. Read back the status to check if invalidate is successful or failed. If status returned is 0 then success, else failed.
    data = PMBus Read Block command (MFR_FIRMWARE_COMMAND_DATA, 4)
    If (data(0) <> 0) Then
        MessageBox.Show("Invalidating the patch failed")
    End If
    Catch ex As Exception
        MessageBox.Show("Invalidating patch failed")
    Finally

// 6. Enable the MMU by restoring the boot descriptor – which is saved in the first step into data2
    i2c Write 4Bytes(data2, boot_descriptor_addr, 0x10)

// 7. Reset after enabling the MMU
    PMBus Write Byte(MFR_FIRMWARE_COMMAND, 0xE)

End Try

```

7.3 Invalidating configurations (register, PMBus and partial configurations)

The code listing shows the procedure for invalidating register configuration, PMBus configurations for loop 0, loop 1 and also partial configurations.

The command type determines the type of configuration to be invalidated.

- To invalidate register configuration set command = 0x04
- To invalidate PMBUS_DEFAULT_LP0 set command = 0x06
- To invalidate PMBUS_USER_LP0 set command = 0x07
- To invalidate PMBUS_DEFAULT_LP1 set command = 0x08
- To invalidate PMBUS_USER_LP1 set command = 0x09
- To invalidate CONF_PARTIAL set command = 0x0A
- To invalidate PMBUS_PARTIAL set command = 0x0B

Invalidating FW patch and configurations

Code Listing 2

```

Try
// 1. Disable the MMU
    // Store the boot descriptor into temp variable, to load back again at the end of the function
    // boot_descriptor_addr = "0x2005C544"
    i2c Read 4Bytes (boot_descriptor_addr, data2)
    // Disable MMU by writing 0xF into boot descriptor register
    i2c Write 4Bytes("0000000F", boot_descriptor_addr)

// 2. Reset after disabling the MMU
    PMBus Write Byte command (MFR_FIRMWARE_COMMAND, 0xE)

// 3. Pass the parameters to FIRMWARE COMMAND DATA – PMBus command to invalidate the patch
    writeData(0) = command    // Command value is based on above in the description
    writeData(1) = 0
    writeData(2) = 0
    writeData(3) = 0          // Data partition – "0"
    PMBus Write Block(MFR_FIRMWARE_COMMAND_DATA, writeData)

// 4. Execute the PMBus command (FIRMWARE COMMAND) to invalidate the patch – pass the argument as
// 0x12 to invalidate
    PMBus Write Byte command (MFR_FIRMWARE_COMMAND, &H12)

// 5. Read back the status to check if invalidate is successful or failed. If status returned is 0 then success,
// else failed.
    data = PMBus Read Block command (MFR_FIRMWARE_COMMAND_DATA, 4)
    If (data(0) <> 0) Then
        MessageBox.Show("Invalidating the patch failed")
    End If
Catch ex As Exception
    MessageBox.Show("Invalidating patch failed")
Finally

// 6. Enable the MMU by restoring the boot descriptor – which is saved in the first step into data2
    i2c Write 4Bytes(data2, boot_descriptor_addr, 0x10)

// 7. Reset after enabling the MMU
    PMBus Write Byte(MFR_FIRMWARE_COMMAND, 0xE)

End Try

```

Revision history

Revision history

Document version	Date of release	Description of changes
V 1.0	11-06-2021	First release
V 1.1	13-09-2021	Modified the order of block length and PMBus read address in PMBus Block Read command in 3.2, 6.7.1 and 6.7.2 sections
V 1.2	27-10-2021	Added FW store trim and FW store config sections in the document

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-10-27

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2021 Infineon Technologies AG.

All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

AN_2106_PL88_2106_165408

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.