XDPP1100 programming instruction

Scope and purpose

The Infineon XDP™ digital power controller **XDPP1100 family** 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

This document is intended for 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 I²C communication protocol 7
3.2 PMBus communication protocol 8
4 Programming procedure using XDPP1100 GUI **10**
4.1 Add device 10
4.1.1 Auto-populate device 10
4.1.2 Add a new device manually 10
4.1.3 Force I²C connection 12
4.2 Multi Device Programmer tool 12
4.3 Check OTP partition 13
4.4 Store FW patch 14
4.5 Apply FW configuration 15
4.6 Load design file 15
4.7 Store configuration to OTP 15
5 Partial configuration and multi configuration using XDP1100 GUI **16**
5.1 Partial configuration 16
5.1.1 Store PMBus partial configuration 16
5.1.2 Store register partial configuration 17
5.2 Multi configuration 17
5.2.1 Multi configuration setup 17
5.2.2 Store the first configuration 18
5.2.3 Store the second configuration 18
5.2.4 Check configuration in OTP 18
5.2.5 Verify the configuration with different XADDR resistor 18
5.3 Invalidating FW patch and configuration 19
6 Programming procedure using I²C protocol **20**
6.1 Storing FW trim for OTP partitioning 20
6.2 Storing FW configuration 21
6.3 Storing a firmware patch 22
6.3.1 Patch file format 22
6.3.2 Storing a firmware patch to OTP 22

6.3.3 Storing a firmware patch in RAM 25
6.4 Storing configuration directly to OTP via scratchpad 27
6.4.1 Configuration file format 27
6.4.2 Configuration section format 29
6.4.2.1 Configuration section header 29
6.4.2.2 Configuration row number 30
6.4.2.3 First DWORD in configuration data header 30
6.4.2.4 Configuration size and redundant size 30
6.4.2.5 Configuration header CRC 30
6.4.2.6 Configuration data 30
6.4.2.7 Configuration data CRC 30
6.4.3 Steps for programming the configuration 31
6.4.3.1 Parsing section header and header code 32
6.4.3.2 Invalidating existing OTP data 32
6.4.3.3 Programming the sections of the configuration file 32
6.5 Storing configuration to OTP via volatile registers 34
6.5.1 Production file (.xcf) format 34
6.5.2 Storing register configuration to OTP 35
6.5.3 Storing PMBus configuration to OTP 36
6.5.4 Storing partial PMBus configuration to OTP 37
6.5.5 Storing partial register configuration to OTP 38
6.6 Invalidating firmware patches and configurations 39
6.6.1 Invalidating a firmware patch 39
6.6.2 Invalidating configurations (register, PMBus and partial configurations) 40
6.7 Configurator status check 41
6.8 Procedure to find remaining free memory in OTP 41
6.8.1 Procedure to find remaining space in data partition to store a new configuration 41
6.8.2 Procedure to find remaining space in patch partition to store a patch 41
Revision history 43

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.

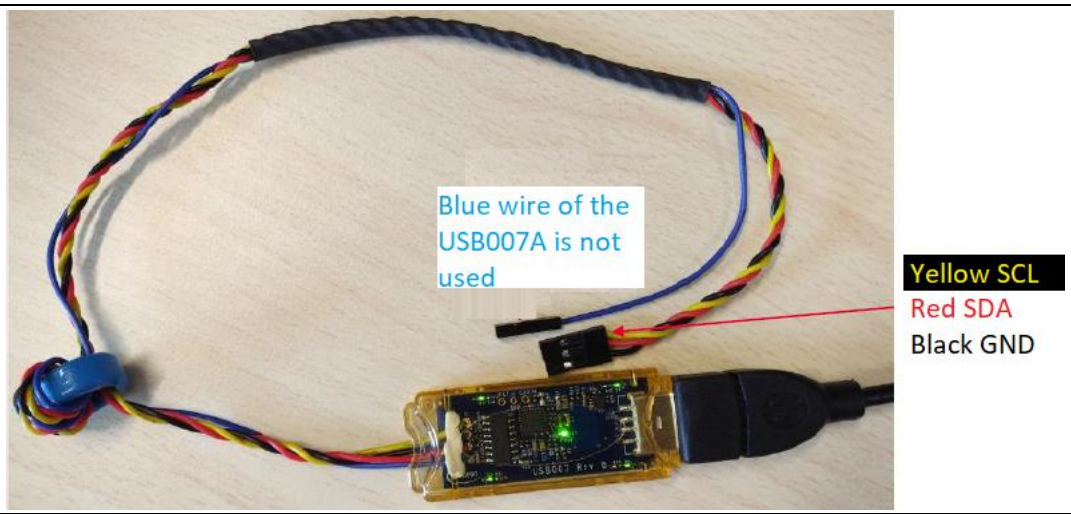


Figure 1 USB to I²C dongle



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

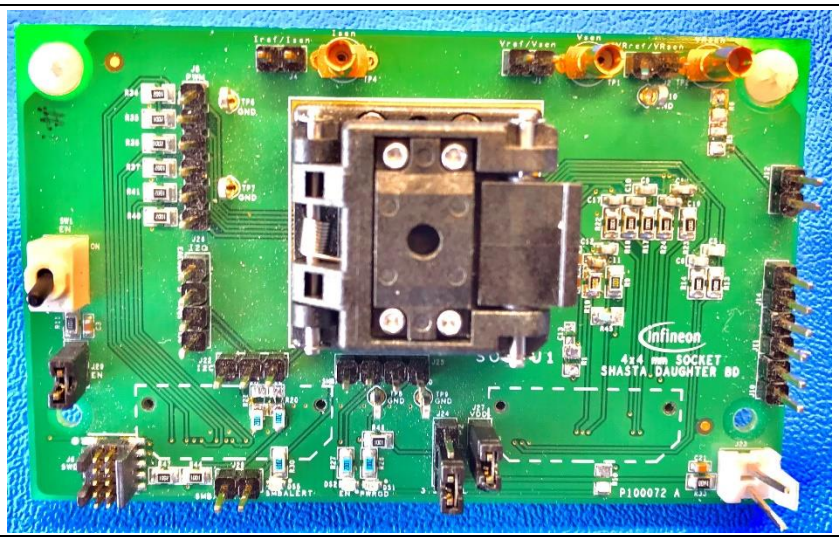


Figure 3 XDPP1100-Q024 4x4 QFN socket board

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 kΩ to 10 kΩ 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

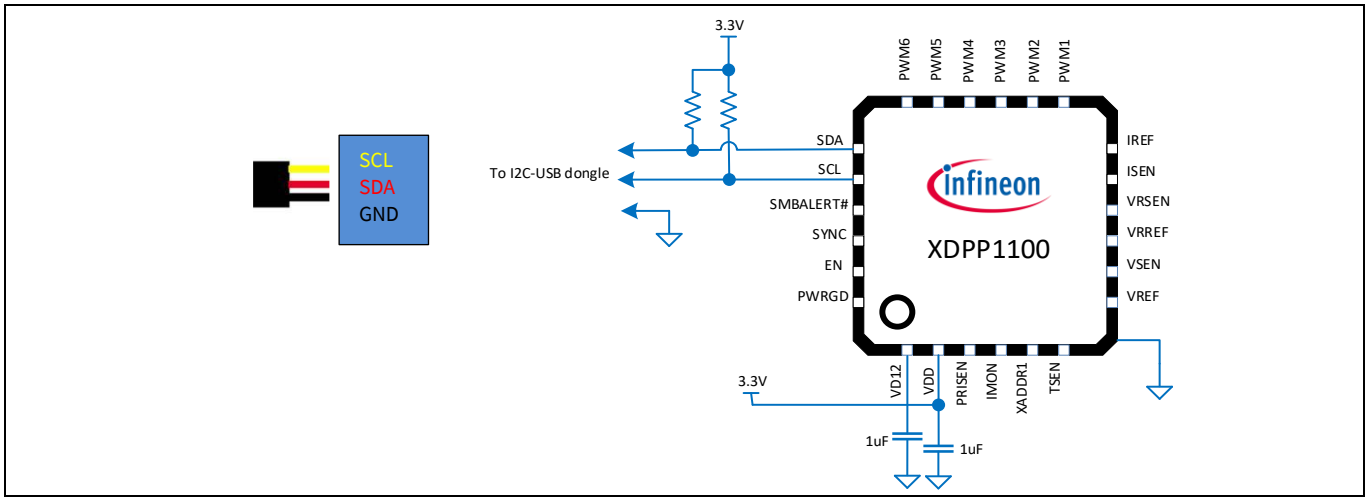


Figure 4 XDPP1100 I²C connection

Software requirements:

- XDPP1100 GUI, version build 7774

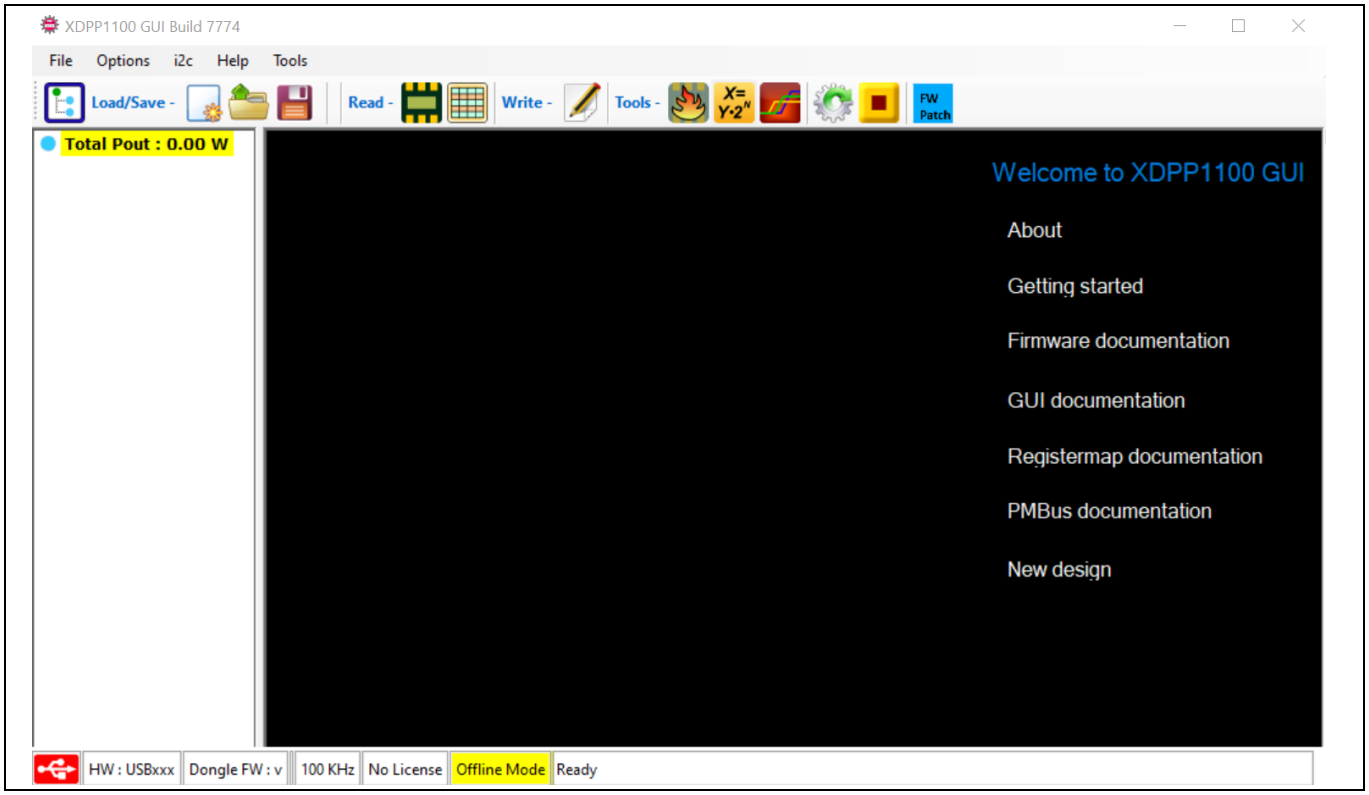


Figure 5 XDPP1100 GUI interface

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.

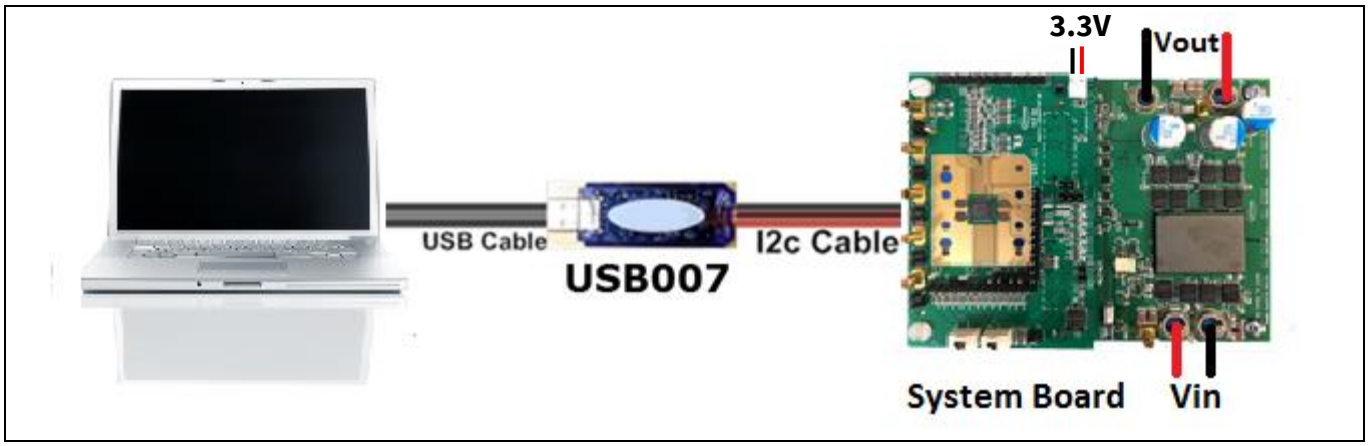


Figure 6 Connection of the programming board, USB dongle and computer

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 partitions) 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.

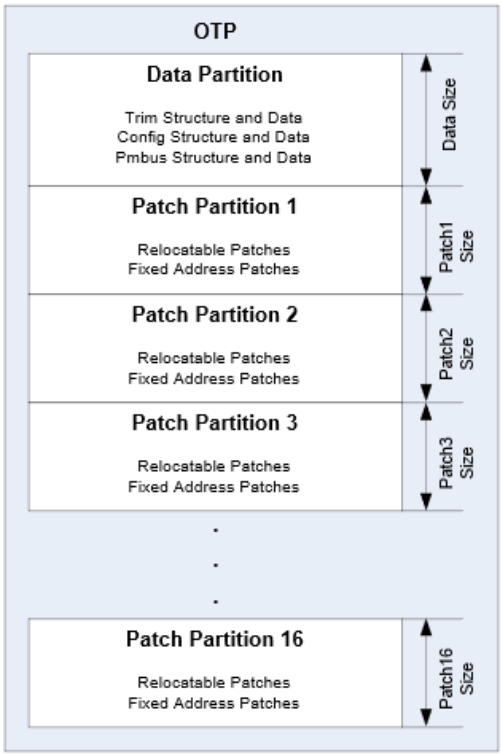


Figure 7 OTP partition

3 I²C and PMBus communication protocol

3.1 I²C communication protocol

› **I²C Write 4 Bytes**

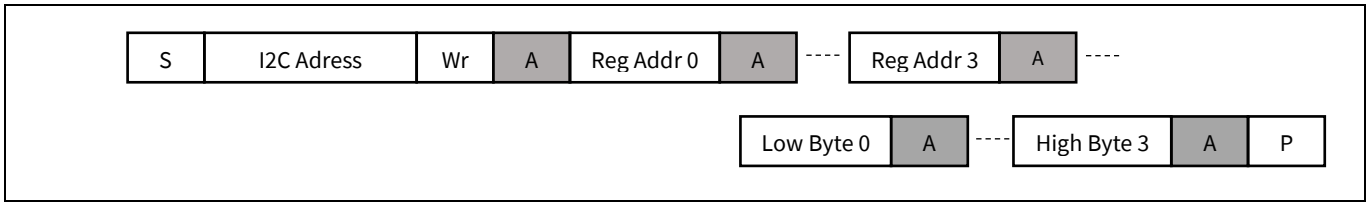


Figure 8 I2C Write 4 bytes

- [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**

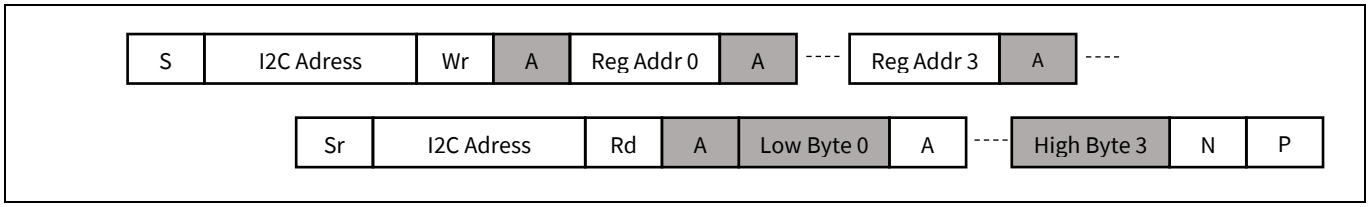


Figure 9 I2C Read 4 bytes

- [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 Send Byte command**

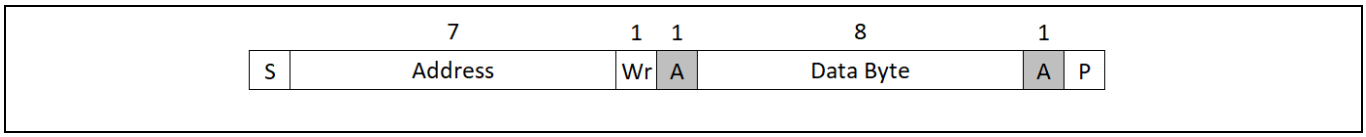


Figure 10 PMBus Send byte

- [PMBus address] [command code]
- Example: Writing PMBus command – Store user all (0x15)
- [0x80] [0x15]

› **PMBus Write Byte command**

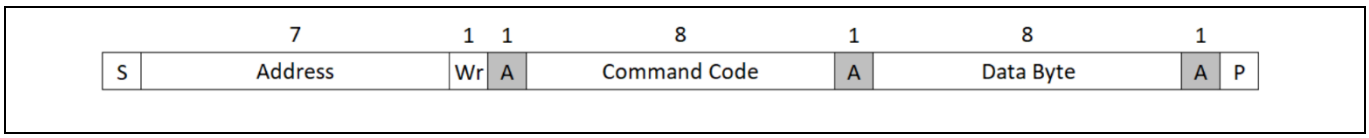


Figure 11 PMBus Write byte

- [PMBus address] [command] [dataByte]
- Example: Writing PMBus command VOUT_MODE (0x20) to value of 0x18
- [0x80] [0x20] [0x18]

› **PMBus Read Byte command**

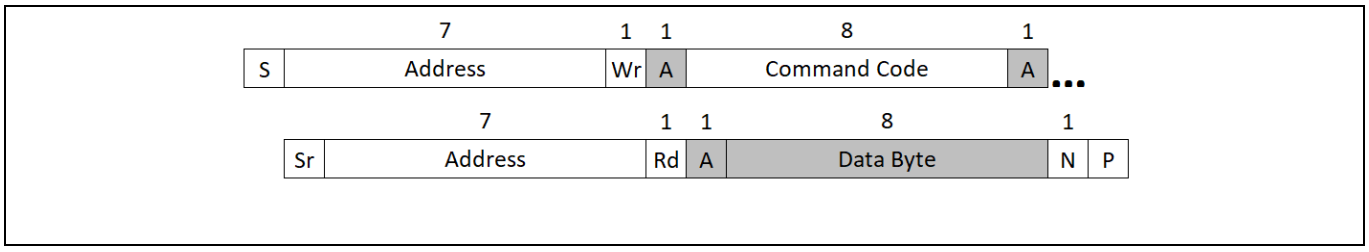


Figure 12 PMBus Read Byte

- [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 Block Write command

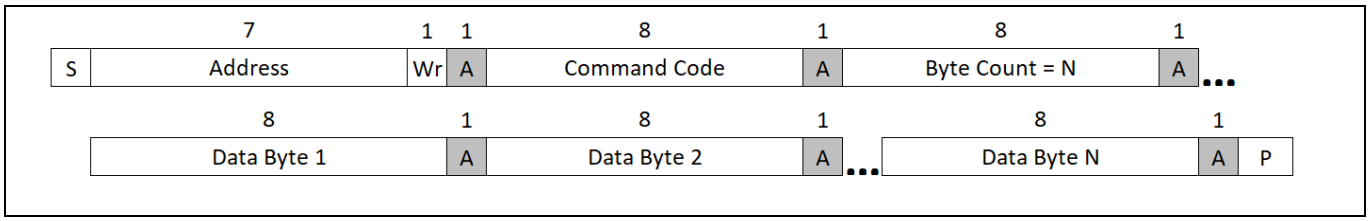


Figure 13 PMBus Block Write

- [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]
- [0x80] [0xCF] [0x18] [0x00] [0x40] [0x00] [0x40] [0x00] [0x60] [0x00] [0x60] ……. [0x00]

› PMBus Block Read command

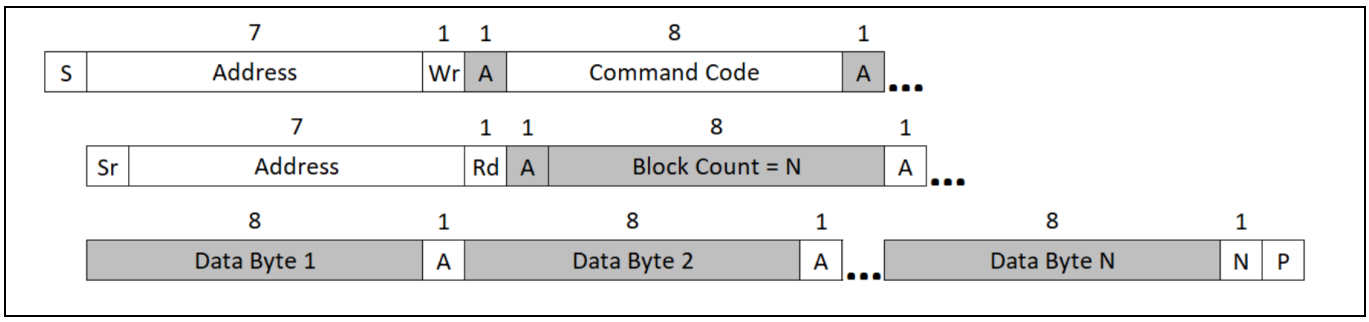


Figure 14 PMBus Block Read

- [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 I^2C 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 I^2C 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 I^2C and 0x80 for PMBus.

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 15**, and a device will be automatically added into the device window.

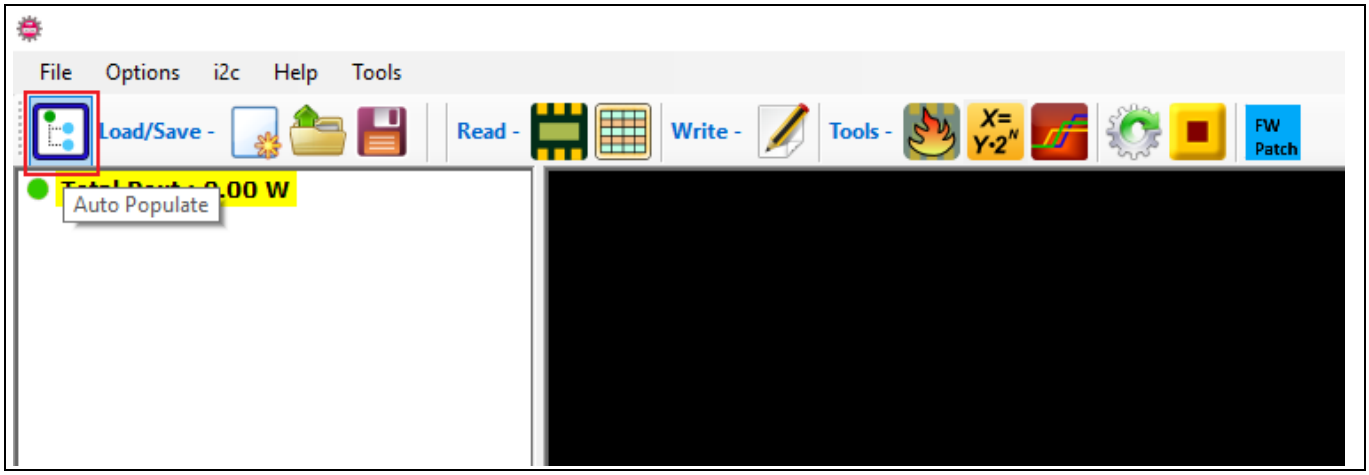


Figure 15 Auto-populate function

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 16**), choose Add Device and follow the arrow to the right to pick a device to use.

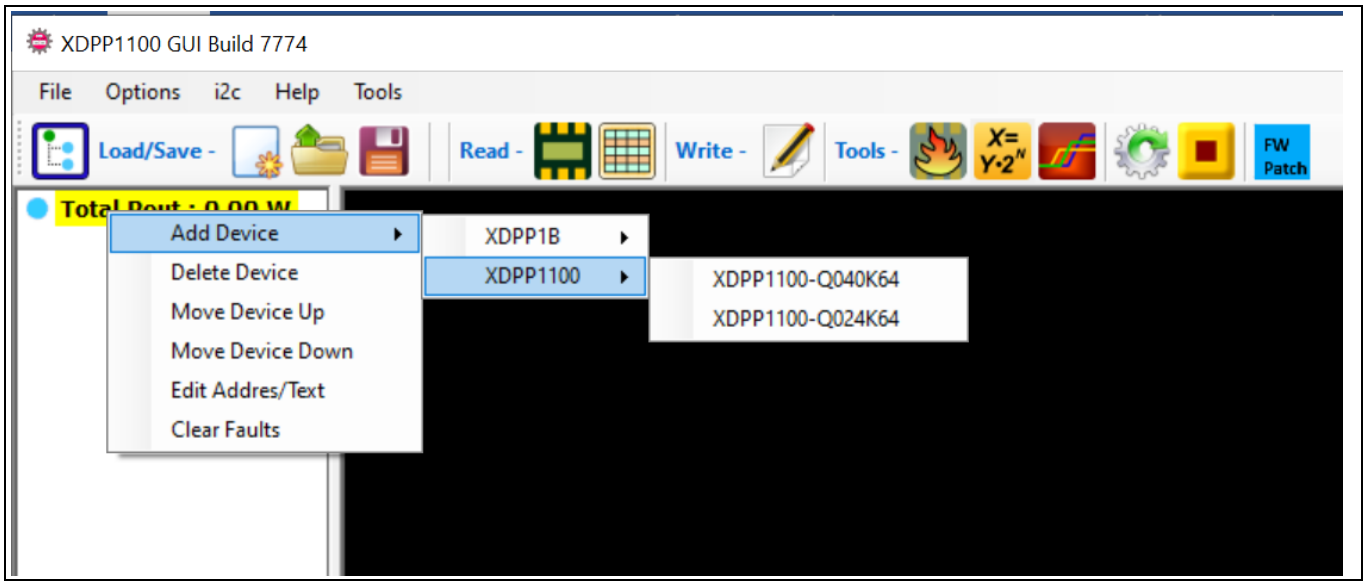


Figure 16 Adding a device manually

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 17**. 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.

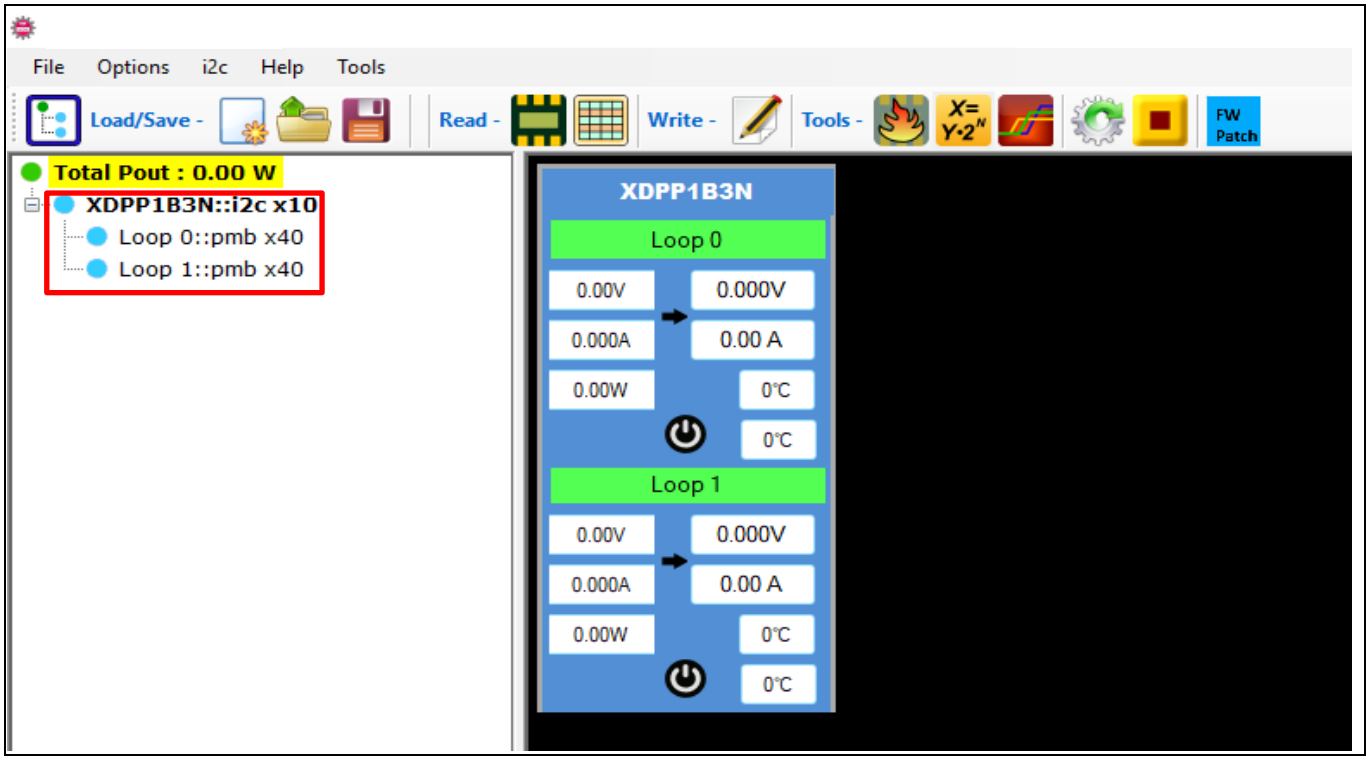


Figure 17 Default I²C and PMBus address

To verify that the assigned address is correct, use the I²C tool to scan the I²C address (**Figure 18**). 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 19**). 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.

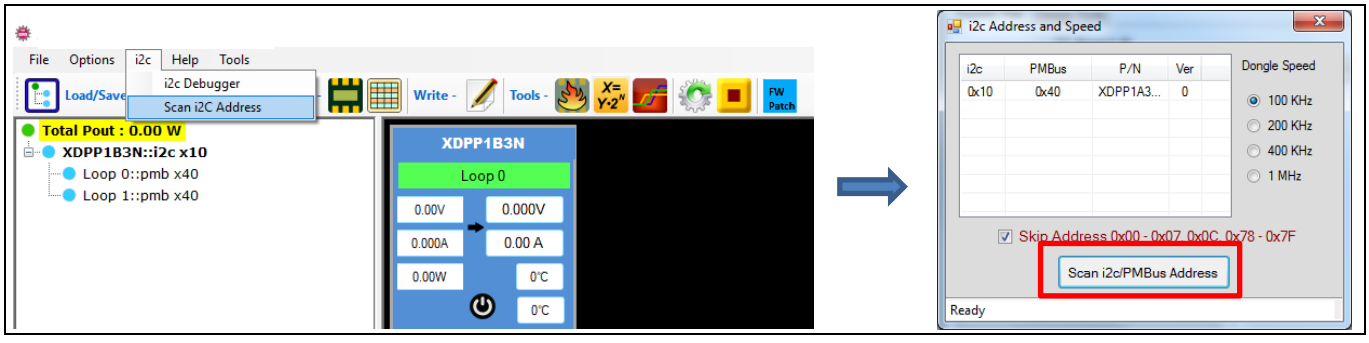


Figure 18 Scan I²C address

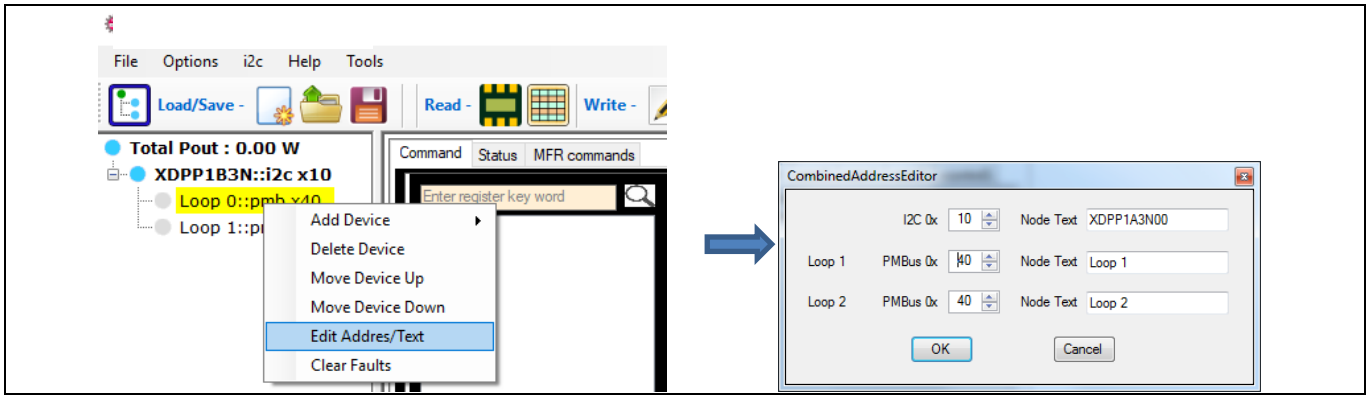


Figure 19 Manually change device address

4.1.3 Force I²C connection

The I²C communication can also be enabled by checking “Force i2c/PMBus OK” (Figure 20). 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.

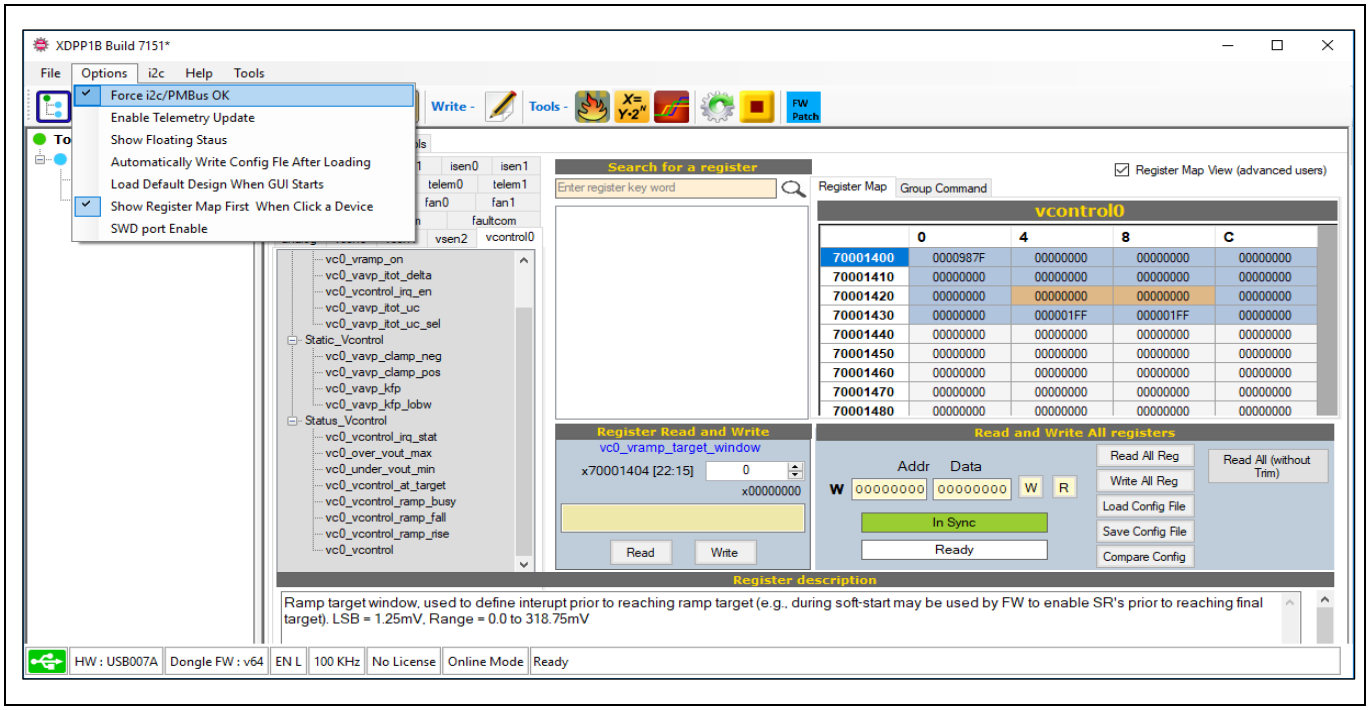


Figure 20 Force I²C/PMBus connection

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.

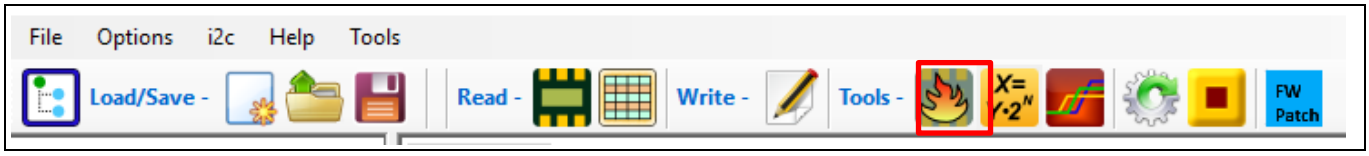


Figure 21 Multi Device Programmer tool

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 22**), and a pop-up window shows all the OTP-related information (**Figure 23**). 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.

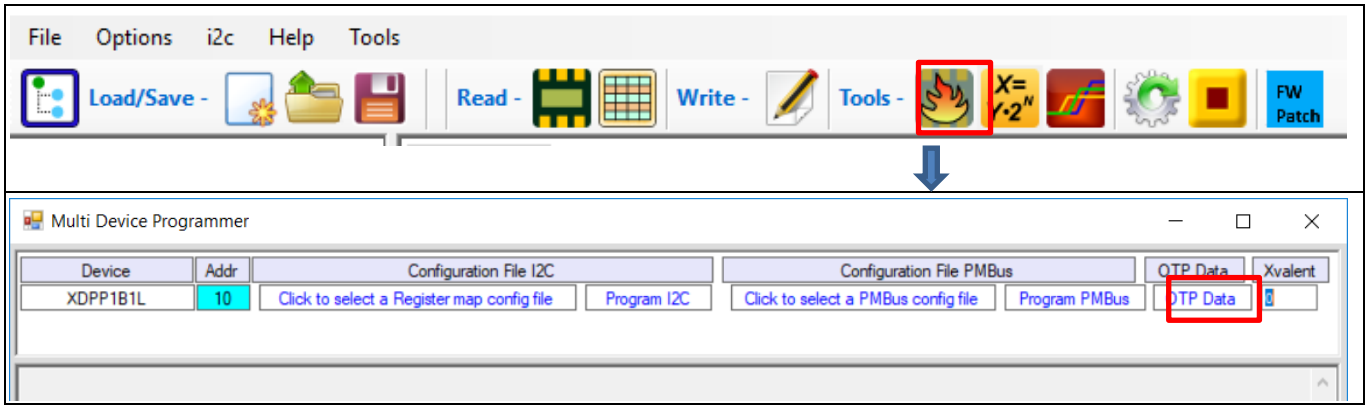


Figure 22 Multi Device Programmer tool – OTP data

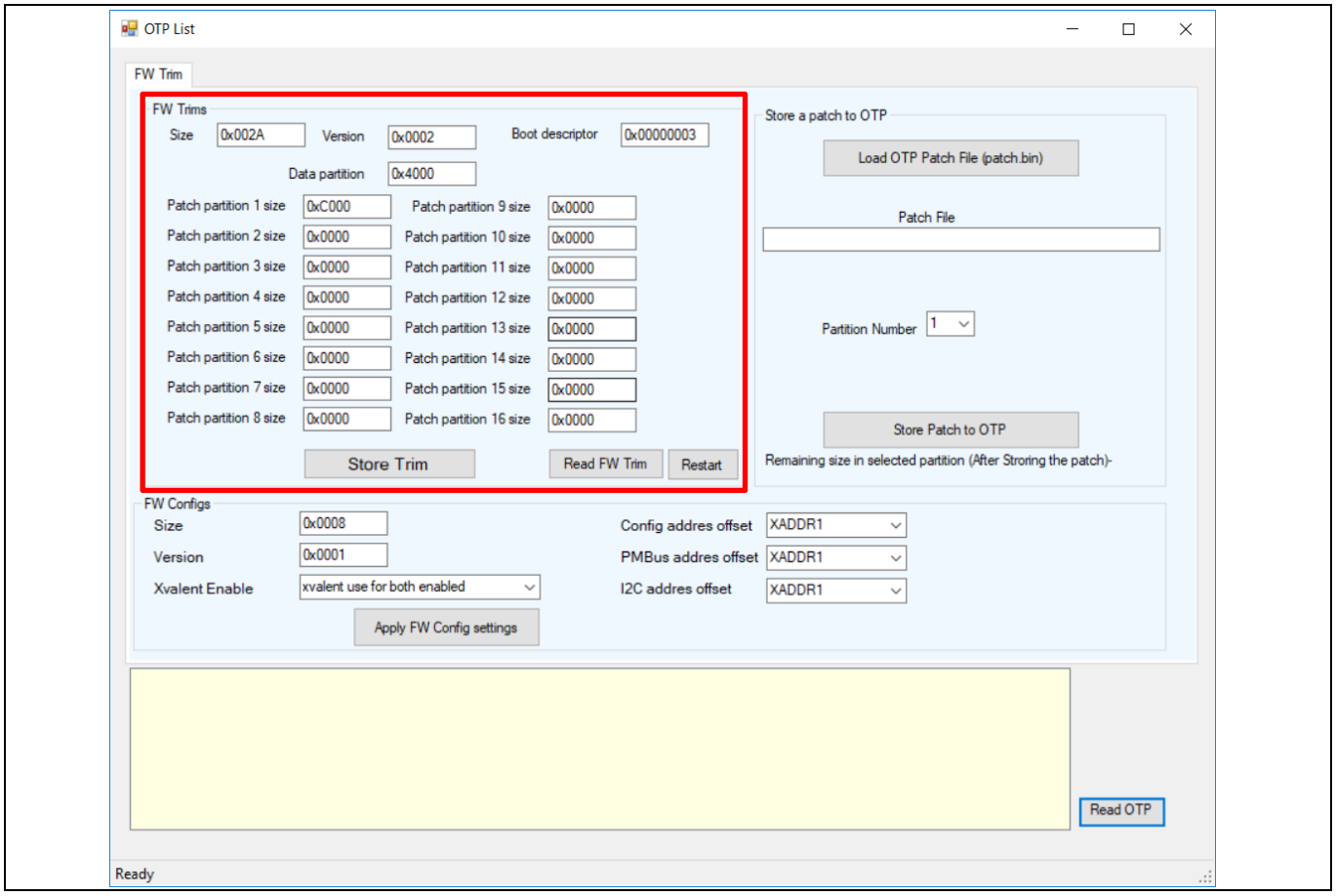


Figure 23 Multi Device Programmer tool – OTP data, FW trims

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 24**) 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.

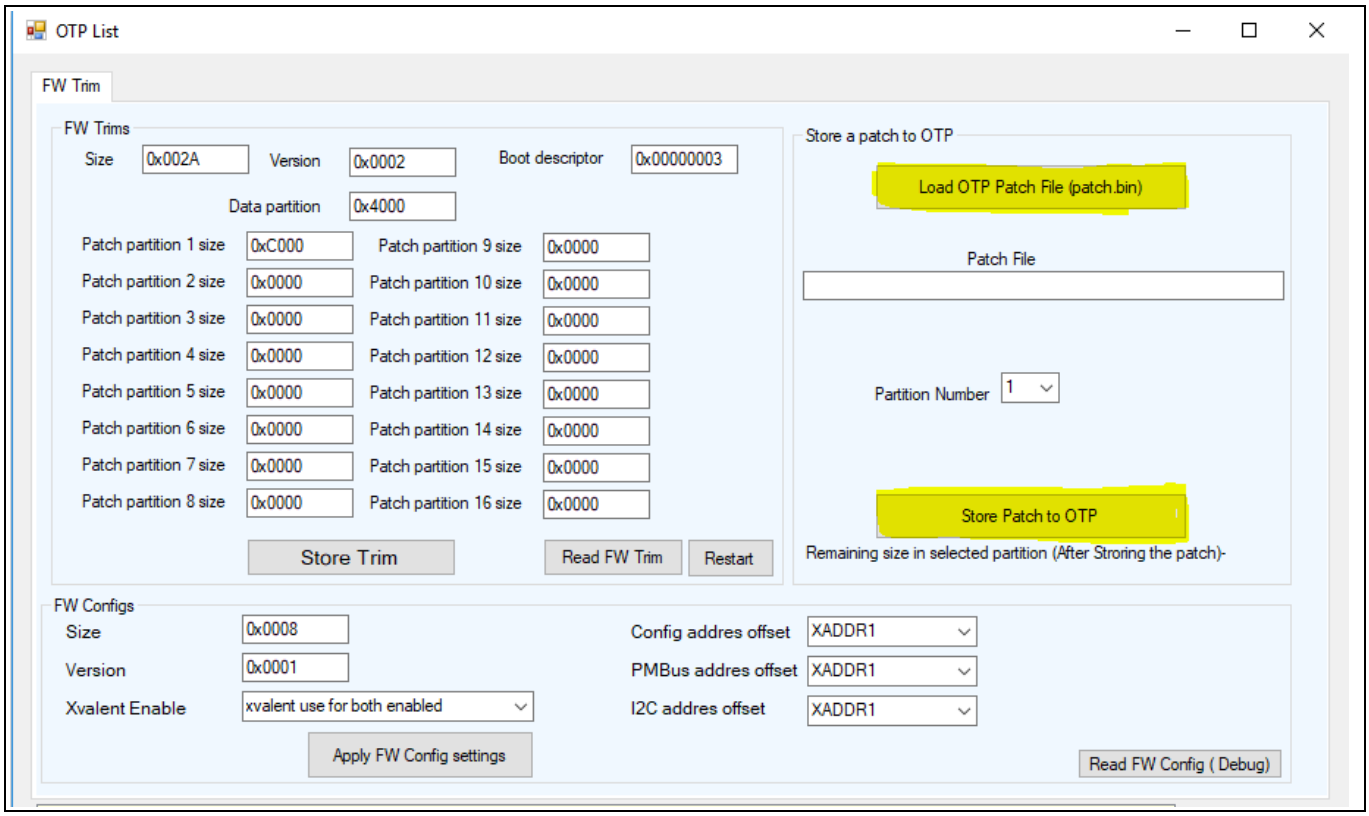


Figure 24 Multi Device Programmer tool – OTP data, store patch to OTP

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

4.5 Apply FW configuration

In the “FW Config” section of **Figure 24**, 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.

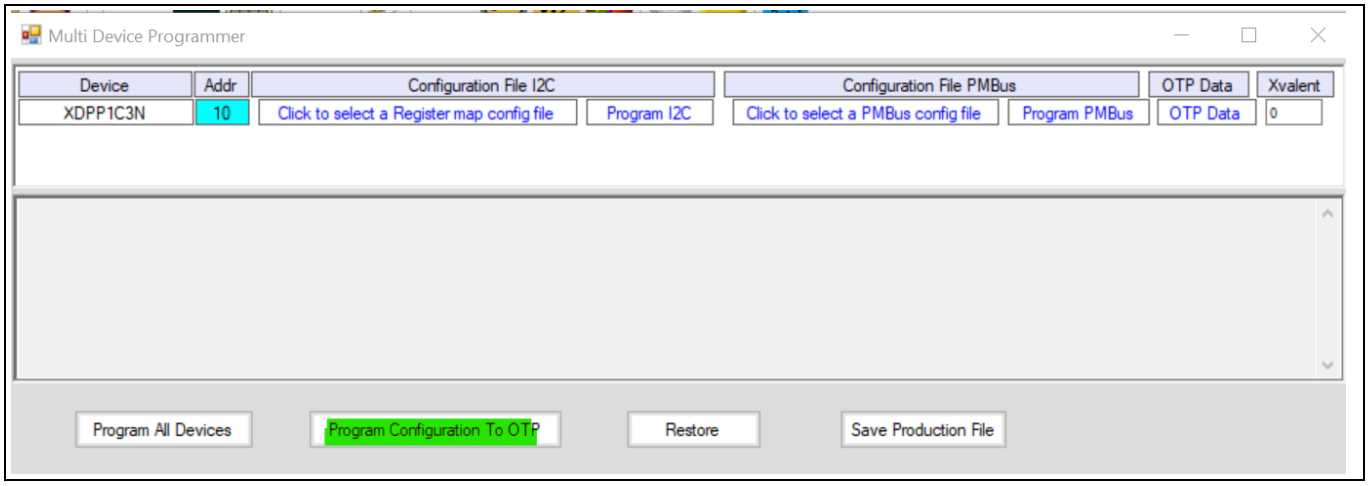


Figure 25 Program configuration to OTP

5 Partial configuration and multi configuration using XDP1100 GUI

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.

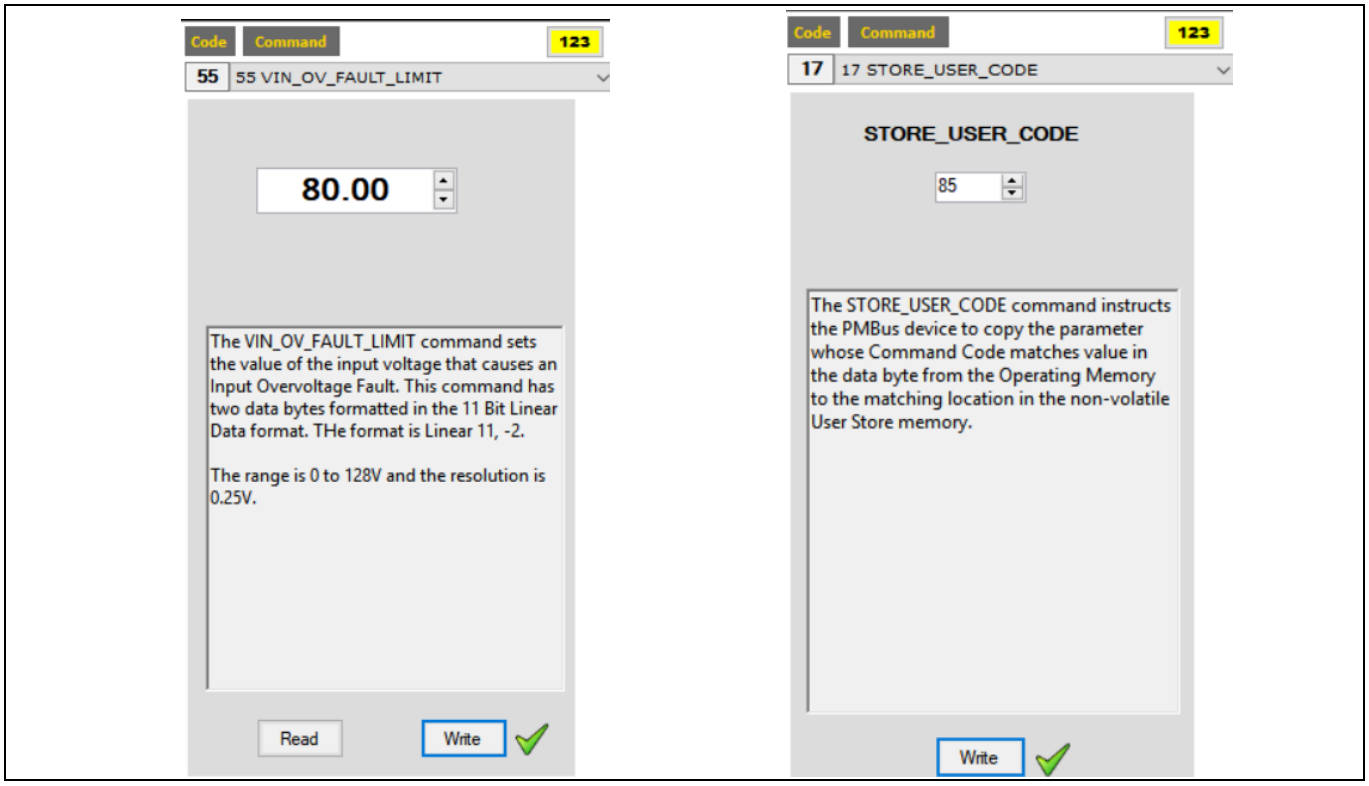


Figure 26 STORE_USER_CODE

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 27**). 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.

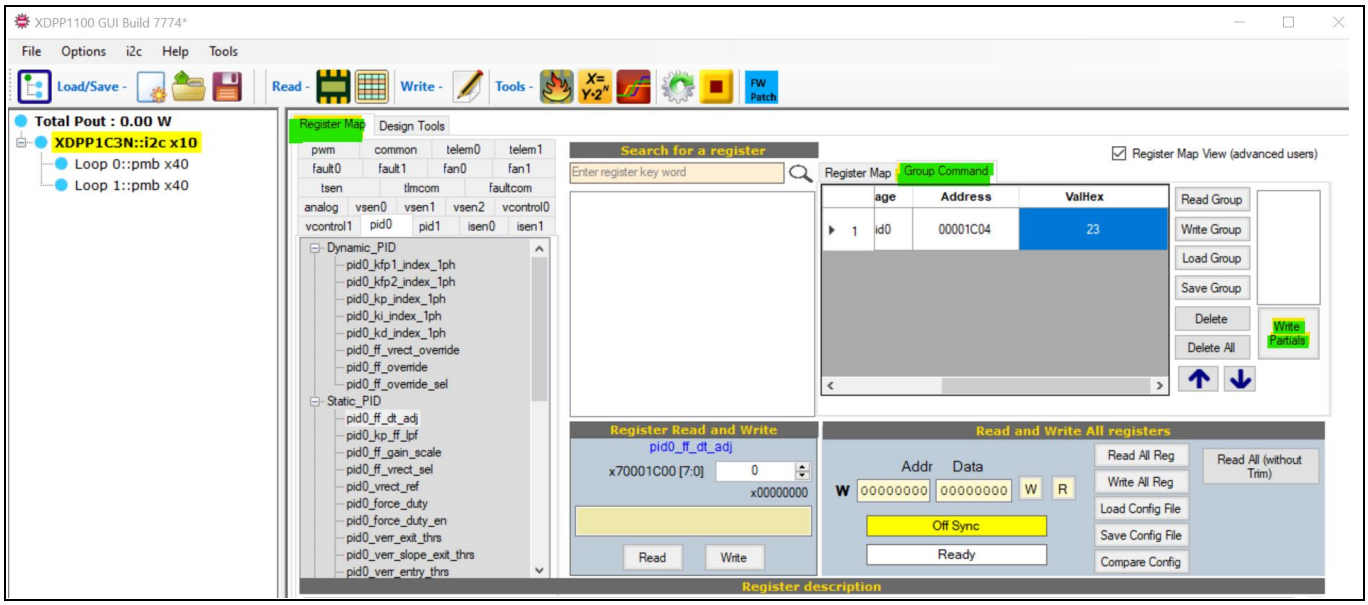


Figure 27 Store register partial configuration using GUI

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.

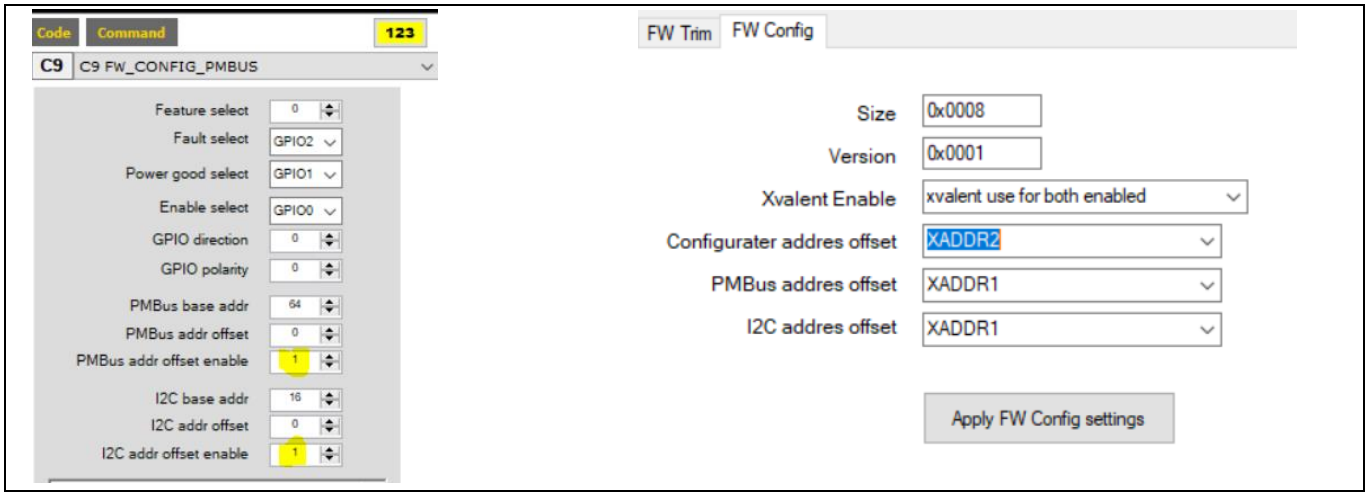


Figure 28 Address offset configuration

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.

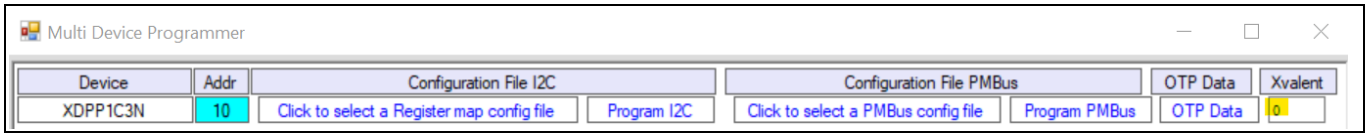


Figure 29 Store configuration to Xvalent=0

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.

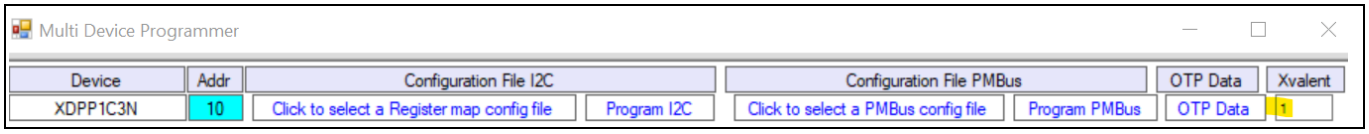


Figure 30 Store configuration to Xvalent=1

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.

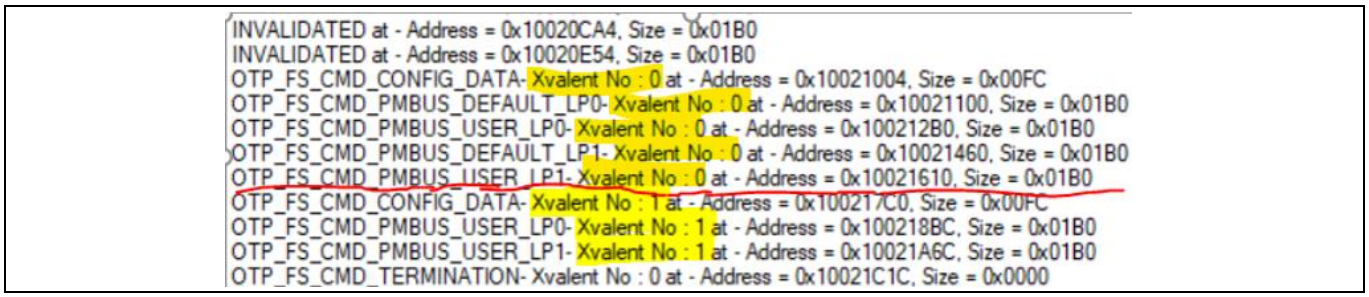


Figure 31 Find configurations in OTP

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.

5.3 Invalidating FW patch and configuration

In the GUI FW Patch tool, choose the OTP partition number and click “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. Ticking this option is recommended if there are MFR commands defined in the patch. Both would 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.

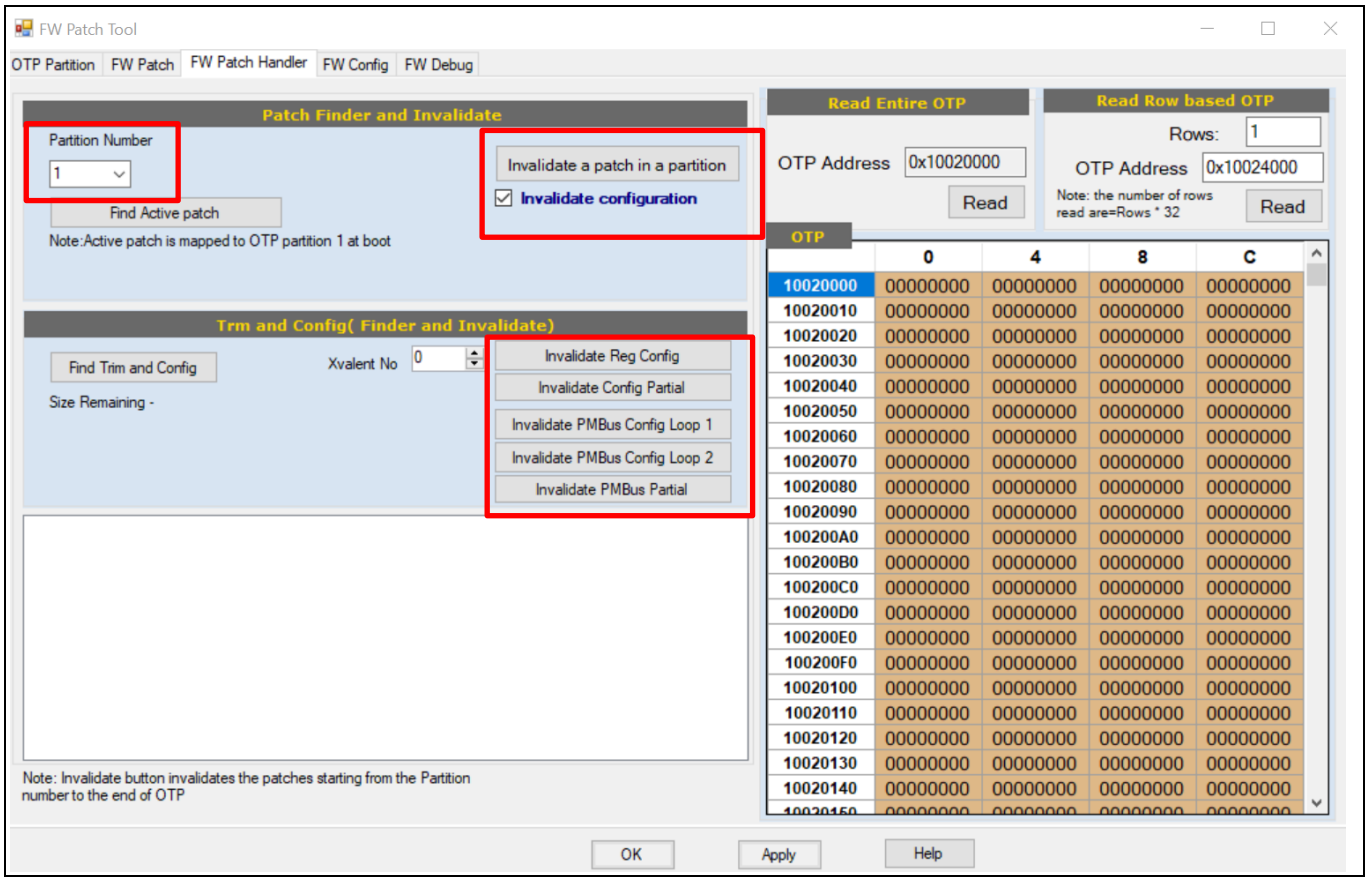


Figure 32 FW patch tool – invalidate patch and configuration

6 Programming procedure using I²C protocol

6.1 Storing FW trim for OTP partitioning

For most standard applications, the OTP partition sizes don't have to be changed. Please refer to **Section 2** for more information regarding the OTP structure.
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 (=16kB),
- Patch partition 1 = 0x2000 (=8kB),
- Patch partition 2 = 0x8000 (=32kB),
- Patch partition 3 = 0x2000 (=8kB).

Code Listing 1

```
// 1. Write data partition of size 0x4000 and patch partition 1 of size 0x2000 at address 0x2005C510
     writeData(0) = 0x00   //0x00 (data partition size, low byte)
     writeData(1) = 0x40   //0x40 (data partition size, high byte)
     writeData(2) = 0x00   //0x00 (patch partition 1 size, low byte)
     writeData(3) = 0x20   //0x20 (patch partition 1 size, high byte)
     i2c Write 4 Bytes (0x2005C510, writeData)

// 2. Write patch partition 2 of size 0x8000 and patch partition 3 of size 0x2000 at address 0x2005C514
     writeData(0) = 0x00   //0x00 (patch partition 2  size, low byte)
     writeData(1) = 0x80   //0x80 (patch partition 2 size, high byte)
     writeData(2) = 0x00   //0x00 (patch partition 3 size, low byte)
     writeData(3) = 0x20   //0x20 (patch partition 3 size, high byte)
     i2c Write 4 Bytes (0x2005C514, writeData)

// 3.  Write MFR_FIRMWARE_COMMAND with the data byte 0x03 to store the trim (including the new
//      partition sizes) to OTP.
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x03)

// 4.  Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While   // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

Note: OTP partitioning supports up to 16 partitions. To perform more partitions, write to the next address 0x2005C518 for partition 4 and 5. Write to the next address 0x2005C51C for partition 6 and 7. Write to the next address 0x2005C520 for partition 8 and 9. Write to the next address 0x2005C524 for partition 10 and 11. Write to the next address 0x2005C528 for partition 12 and 13. Write to the next address 0x2005C52C for partition 14 and 15. Write to the next address 0x2005C530 for partition 16.

6.2 Storing FW configuration

The section below provides the details on storing FW configuration to OTP to enable or disable the XValent (multi-config) and the I²C / PMBus address offset functionality, and to select the source (XADDR1 or XADDR2 pin) for I2C address offset, PMBus address offset and config XValent number. This setting can also be written in the same step as writing the register config (see **Section 6.5.1**), as it will be stored in the same file in OTP.

Code Listing 2

```
// 1. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0   // Xvalent number (0 for the basic configuration)
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 0
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 2. Write to register address fw_config_xv_enable (0x2005C504) the following data bytes
     writeData(0) = 1   // XValent and address offsets enabled (0 – disabled, 1 – enabled)
     writeData(1) = 0   // Source of Config XValent number (0 = XADDR1, 1 = XADDR2)
     writeData(2) = 0   // Source of PMBus address offset (0 = XADDR1, 1 = XADDR2)
     writeData(3) = 0   // Source of I2C address offset (0 = XADDR1, 1 = XADDR2)
     i2c Write 4 Bytes (0x2005C504, writeData)

// 3.  Write MFR_FIRMWARE_COMMAND with the data byte 0x04 to store configuration to OTP
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x04)

// 4.  Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While   // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

6.3 Storing a firmware patch

6.3.1 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		Reserved	cmd
data1				Version new				Version base				Execute			
•••		•••		•••		•••		•••		•••		Data3...		Data2...	
•••															
						EOF		CRC data		dataN		•••		•••	

Figure 33 Patch file format

6.3.2 Storing a firmware patch to OTP

- There can only be one active patch in a patch partition. If a patch already stored in the patch partition needs to be updated, invalidate the active patch (see **Section 6.6.1**) and store the new patch.
- Parse the patch.bin file to read out header information and actual data information.
- The header includes important information to store the patch:
    - Command and size
        - The command value for an active patch is always 0x10
        - The size is given as hex value in bytes. For example, 0x1000 = 4096 = 4 kB.
    - Header CRC
    - Location of the 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
        - The patch version is set in the file partial_patch.cfg in the patch project and can be used to identify the patch.
    - Actual data to be stored to OTP
    - Data CRC

Example: **Figure 34** shows the header information of a patch, **Figure 35** shows the beginning of a patch file.

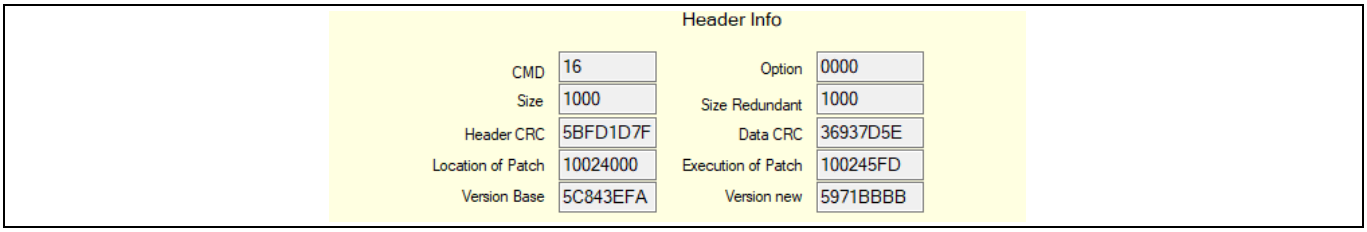
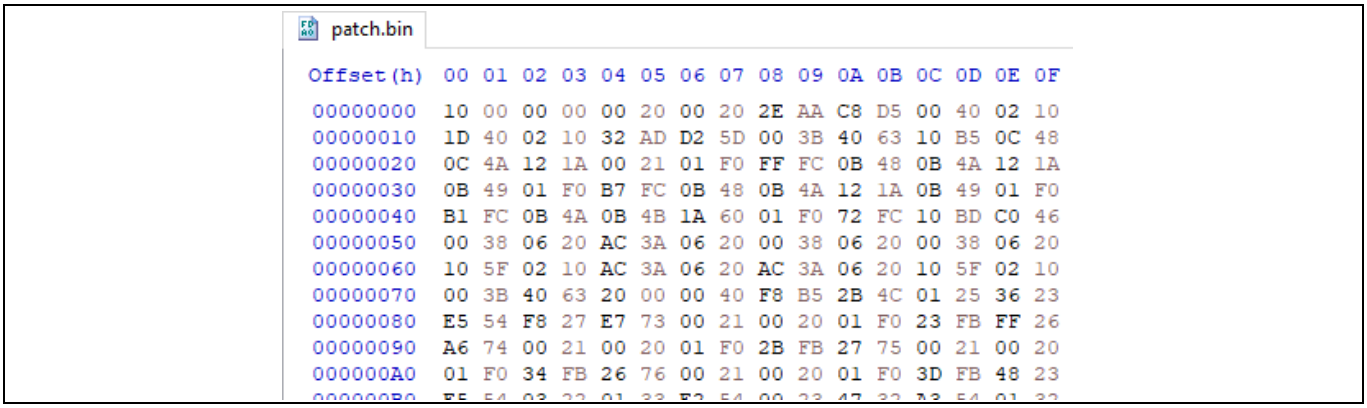


Figure 34 Patch header information


patch.bin

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	10	00	00	00	00	20	00	20	2E	AA	C8	D5	00	40	02	10
00000010	1D	40	02	10	32	AD	D2	5D	00	3B	40	63	10	B5	0C	48
00000020	0C	4A	12	1A	00	21	01	F0	FF	FC	0B	48	0B	4A	12	1A
00000030	0B	49	01	F0	B7	FC	0B	48	0B	4A	12	1A	0B	49	01	F0
00000040	B1	FC	0B	4A	0B	4B	1A	60	01	F0	72	FC	10	BD	C0	46
00000050	00	38	06	20	AC	3A	06	20	00	38	06	20	00	38	06	20
00000060	10	5F	02	10	AC	3A	06	20	AC	3A	06	20	10	5F	02	10
00000070	00	3B	40	63	20	00	00	40	F8	B5	2B	4C	01	25	36	23
00000080	E5	54	F8	27	E7	73	00	21	00	20	01	F0	23	FB	FF	26
00000090	A6	74	00	21	00	20	01	F0	2B	FB	27	75	00	21	00	20
000000A0	01	F0	34	FB	26	76	00	21	00	20	01	F0	3D	FB	48	23

Figure 35 patch.bin file

Code Listing 3

```
// 1. Parse the patch.bin file and data into patch_array
     For i = 0 To EOF patch.bin
          File read (patch_array(i))
     Next
     // The patch has to be written to the scratchpad (starting at 0x20061800) and copied from there
     // to OTP in chunks of 1 kB (0x400)

// 2. Follow the below steps to write the first 1 kB of data to the scratchpad
     file_pointer = 0
     For ptr = 0 To 0x400 Step 4
          writeData(0) = patch_array(file_pointer++)
          writeData(1) = patch_array(file_pointer++)
          writeData(2) = patch_array(file_pointer++)
          writeData(3) = patch_array(file_pointer++)
          i2c Write 4 Bytes (0x20061800 + ptr, writeData)
     Next
     // After this step, you will have the first 1 kB of the patch.bin file placed on the scratchpad
```

	0	4	8	C
20061800	00000010	20002000	D5C8AA2E	10024000
20061810	1002401D	5DD2AD32	63403B00	480CB510
20061820	1A124A0C	F0012100	480BFCFF	1A124A0B
20061830	F001490B	480BFCB7	1A124A0B	F001490B
20061840	4A0BFCB1	601A4B0B	FC72F001	46C0BD10
20061850	20063800	20063AAC	20063800	20063800
20061860	10025F10	20063AAC	20063AAC	10025F10
20061870	63403B00	40000020	4C2BB5F8	23362501
20061880	27F854E5	210073E7	F0012000	26FFFB23
20061890	210074A6	F0012000	7527FB2B	20002100
200618A0	FB34F001	21007626	F0012000	2348FB3D

```
     // Follow the steps below to move data from scratchpad to OTP

// 3. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0   // Size LSB = 0
     writeData(1) = 4   // Size MSB (4 * 256 + 0 )= 1024 = 1 kB
     writeData(2) = 1   // header is included
     writeData(3) = 1   // partition number = 1
```

Code Listing 3

```
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 4. Write MFR_FIRMWARE_COMMAND with the data byte 0x06 to store the 1kB of patch to OTP
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x06)

// 5. Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While   // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While

// 6. Follow the below steps for the remaining chunks of 1 kB of data until end of patch.bin file
     For i = 2 To (patch_size/0x400)
          // Write the 1 kB (0x400) values to the address from 0x20061800 to 0x20061BFC
          For (ptr = 0 To 0x400 Step 4)
               writeData(0) = patch_array(file_pointer++)
               writeData(1) = patch_array(file_pointer++)
               writeData(2) = patch_array(file_pointer++)
               writeData(3) = patch_array(file_pointer++)
               i2c Write 4 Bytes (0x20061800 + ptr, writeData)
          Next
          // After this step, you will have the next 1 kB of the patch.bin placed on the scratchpad
          // Follow the below steps to move data from scratchpad to OTP

// 7. Write MFR_FIRMWARE_COMMAND_DATA with the following data
          writeData(0) = 0   // Size LSB = 0
          writeData(1) = 4   // Size MSB (4 * 256 + 0 )= 1024 = 1 kB
          writeData(2) = 0   // header is not included
          writeData(3) = 1   // partition number = 1
          PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 8. Write MFR_FIRMWARE_COMMAND with the data byte 0x06 to store the 1kB of patch to OTP
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x06)

// 9. Wait until writing to OTP is completed
          data(0) = 1
          i = 0
          While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
               If i >= 10 Then
                    Exit While   // Exit if writing was not successful after 1s – do some fault handling
               End If
               i++
               Wait(100ms)
               PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
               PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
          End While
     Next
```

Code Listing 3

```
// 10. Restart to make sure the changes of patch will take effect, by writing MFR_FIRMWARE_COMMAND
//      PMBus command with the data byte 0x0E
      PMBus Write Byte (MFR_FIRMWARE_COMMAND, 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 changes 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 the 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, store a patch first, and then store the PMBus configurations.

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.

6.3.3 Storing a firmware patch in RAM

It is possible to store a firmware patch in RAM and execute it directly. This can be very useful for firmware development and testing, as the OTP space and, therefore, the number of patch rewrites is limited. The patch partition usually has 48 kB, meaning a 6 kB firmware patch can be written to OTP only 8 times.

There is a size limit of 8 kB for storing a patch in RAM. If the patch is bigger, one option might be to disable some of the patch functionalities to bring down the size for testing. Once the new functionality works as expected, enable everything again and store the full patch to OTP for additional testing.

Before storing a patch in RAM, if there is any active patch in the patch partition, it has to be invalidated (see **Section 6.6.1**).

Register config data stored in OTP will be loaded to RAM when the device is powered up before storing the patch to RAM, and it will still be in RAM afterwards. In general, the same applies to PMBus config data stored in OTP. However, if the patch includes additional MFR PMBus commands, there are two options:

- The additional MFR commands are included in the data in OTP. When the device is powered up, the PMBus config is invalid as the patch is not yet active, and therefore the PMBus config is not loaded. After storing the patch in RAM, the PMBus config data can be restored from OTP by sending the RESTORE_USER_ALL PMBus command.
- The additional MFR commands are not included in the data in OTP. When the device is powered up, the PMBus config is loaded. After storing the patch, only the data for the additional MFR PMBus commands are missing. The MFR PMBus command with the correct values must be written.

It is also possible not to store any config in OTP during firmware development and testing but to write all the config registers and all the PMBus commands to RAM each time after storing the patch in RAM.

Code Listing 4

```
// 1. Parse the patch.bin file and data into patch_array.
     For i = 0 To EOF patch.bin
          File read (patch_array(i))
     Next
     // The patch has to be written to the scratchpad (starting at 0x20061800)
     // Maximum size is 8 kB (0x2000).

// 2. Write the data from patch_array to the scratchpad
     file_pointer = 0
     For ptr = 0 To patch_size Step 4
          writeData(0) = patch_array(file_pointer++)
          writeData(1) = patch_array(file_pointer++)
          writeData(2) = patch_array(file_pointer++)
          writeData(3) = patch_array(file_pointer++)
          i2c Write 4 Bytes (0x20061800 + ptr, writeData)
     Next
     // After this step, you will have the complete patch.bin file placed on the scratchpad
```

	0	4	8	C
20061800	00000010	20002000	D5C8AA2E	10024000
20061810	1002401D	5DD2AD32	63403B00	480CB510
20061820	1A124A0C	F0012100	480BFCFF	1A124A0B
20061830	F001490B	480BFCB7	1A124A0B	F001490B
20061840	4A0BFCB1	601A4B0B	FC72F001	46C0BD10
20061850	20063800	20063AAC	20063800	20063800
20061860	10025F10	20063AAC	20063AAC	10025F10
20061870	63403B00	40000020	4C2BB5F8	23362501
20061880	27F854E5	210073E7	F0012000	26FFFB23
20061890	210074A6	F0012000	7527FB2B	20002100
200618A0	FB34F001	21007626	F0012000	2348FB3D

```
// 3. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 1   // Partition for which the patch was compiled
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 0
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 4.  Write MFR_FIRMWARE_COMMAND with the data byte 0x16 to execute the patch from RAM
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x16)

// 5. Read back MFR_FIRMWARE_COMMAND_DATA and check if it is 0 (= successful)
     Wait(100ms)
     PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     If data <> 0 Then
          // command was not successful – do some fault handling
     End If

// 6. Restore or write config registers and / or PMBus commands, if necessary (see description
//     above code listing for details)
```

6.4 Storing configuration directly to OTP via scratchpad

The configuration data can be stored directly to OTP (via scratchpad) for production programming. This option has the benefit that the data will be copied 1:1 from the configuration file to OTP, which makes the process easy to handle and allows verification by reading back the OTP content (or CRC values from OTP).

6.4.1 Configuration file format

The configuration file used for storing the configuration directly to OTP can be generated in the XDPP1100 GUI. First, open the board design in XDPP1100 GUI (if you are using a firmware patch with MFR commands, make sure the correct PMBus spreadsheet is loaded before), go to the Multi-Device-Programmer and click on the “XDP GUI file” button at the bottom right (if you are using a firmware patch with MFR commands, make sure to tick the “Patched commands” box before clicking the button).

The configuration file is a text file, containing the different configuration sections. Typically, XDPP1100 has an XV0 Config, an XV0 PMBus Loop0 Default, an XV0 PMBus Loop0 User, an XV0 PMBus Loop1 Default and an XV0 PMBus Loop1 User section. For XDPP1100-Q024, only loop 0 is used and the configuration file therefore does not include the XV0 PMBus Loop1 User section (see **Figure 36**).

At power-up, the device automatically loads the Config section and the PMBus User sections. The PMBus Default sections are only loaded, if the PMBus User sections are missing or if the PMBus command RESTORE_DEFAULT_ALL is sent. However, it is recommended to program the PMBus Default sections, even if they are not used, because otherwise, the device will show a memory fault after power-up. For updating the configuration of a previously programmed device, it is usually not necessary to reprogram the PMBus Default sections. Some OTP memory can be saved by only reprogramming the Config section and PMBus User sections.

The length of the PMBus sections depends on the number of patched MFR commands which might be added by using a firmware patch.

```
//*********************************
// Do not edit any line below CRC32
//*********************************
System Design File CRC32 : 0x9E1CACA0
System Design File Ver : 1.1
Culture : en-GB.
Created Date : 2/28/2023 4:11:27 PM
GUI Version : 1.0
Created by : BergerChrist
Company : Infineon Technologies
i2c Address Format : 8-Bit
Number of Devices : 1
//*****************************
//Device 0
//*****************************
Part Number : XDPP1100-Q024
i2c Address : 0x20
PMBus Address : 0x80
Configuration Checksum : 0xA4288335
[User Properties]
[End User Properties]
[Configuration Data]
//XV0 Config
000 00000004 01000100 97AD8B32 326BA109
010 93580040 9AC00001 0000000C 7F00987F
020 C2980098 8840C1D2 00000009 00000000
030 000C0000 10000000 24644180 00951469
040 00000000 00000000 000000C0 29020000
050 40020900 80101000 DFC00000 0000400F
060 60011040 00000010 00000000 0003F800
070 D0000008 89499873 004D29A5 00000000
080 00002AA8 01FFF800 06248100 410C3061
090 1861A630 86186186 E6181861 23028503
0A0 8007600C 01002006 00000002 00000000
0B0 18528064 0C360620 00C00113 18603031
0C0 21154000 40855448 A2104090 AA24108A
0D0 20482042 00000108 00000000 70005700
0E0 F1EF0000 3280201C 18618002 00000006
0F0 00000010 00010008 00000001 03EB4853
//XV0 PMBus Loop0 Default
000 00000006 01B001B0 2C3F6C6D 000001F6
010 D0001D00 00000000 00000000 C8001600
020 00000000 0000E800 E8100000 06660000
030 F17C0000 8C03008C 00F088F0 00E80000
040 E80080F0 38004000 C0001600 00120000
050 7D000780 005A8000 0FEC0FEE 80F13000
060 F08CF128 70B8F088 F06480F0 7800A000
070 F1900000 0000F3FC FFF0C800 FF13FFF0
080 00003313 00000000 00000000 00000000
090 00000000 00000000 00000000 00000000
0A0 00000000 00000000 00000000 00000000
0B0 00000000 00000000 00000000 00000000
0C0 00000000 00000000 00000000 00000000
0D0 00000000 00000000 00000000 00000000
0E0 00000000 00000000 00000000 00000000
0F0 00000000 00000000 00000000 000C0033
100 0000C000 00000000 00000000 00080000
110 00028002 C6178000 00ECCD15 00000000
120 00000000 00000000 00000000 00000000
130 00000000 00000000 00000000 40001000
140 00880000 C0000000 A03C0000 7038B3FF
150 70387038 9C287038 00009C28 00000000
160 00000000 00180000 00000000 00000000
170 00000000 00000000 00000000 00000000
180 52000001 4C9410B8 00024112 00000000
190 00000000 00000000 00000000 00000000
1A0 00000000 00000000 00000000 53EBD717
//XV0 PMBus Loop0 User
000 00000007 01B001B0 E0956CF3 000001F6
010 D0001D00 00000000 00000000 C8001600
020 00000000 0000E800 E8100000 06660000
030 F17C0000 8C03008C 00F088F0 00E80000
040 E80080F0 38004000 C0001600 00120000
050 7D000780 005A8000 0FEC0FEE 80F13000
060 F08CF128 70B8F088 F06480F0 7800A000
070 F1900000 0000F3FC FFF0C800 FF13FFF0
080 00003313 00000000 00000000 00000000
090 00000000 00000000 00000000 00000000
0A0 00000000 00000000 00000000 00000000
0B0 00000000 00000000 00000000 00000000
0C0 00000000 00000000 00000000 00000000
0D0 00000000 00000000 00000000 00000000
0E0 00000000 00000000 00000000 00000000
0F0 00000000 00000000 00000000 000C0033
100 0000C000 00000000 00000000 00080000
110 00028002 C6178000 00ECCD15 00000000
120 00000000 00000000 00000000 00000000
130 00000000 00000000 00000000 40001000
140 00880000 00000000 B3FF0000 7038B3FF
150 70387038 9C287038 00009C28 00000000
160 00000000 00FF0000 00000000 00000000
170 00000000 00000000 00000000 00000000
180 00000000 4C941000 00024112 00000000
190 00000000 00000000 00000000 00000000
1A0 00000000 00000000 00000000 1A2DA9C3
[End Configuration Data]
[End]
```

Figure 36 Configuration file example for XDPP1100-Q024

6.4.2 Configuration section format

All configuration sections follow the same format. A section header describes the type of section. Generally, each row contains a 3-digit running byte count followed by 16 bytes of data arranged in 4 DWORDs (DW0...DW3).

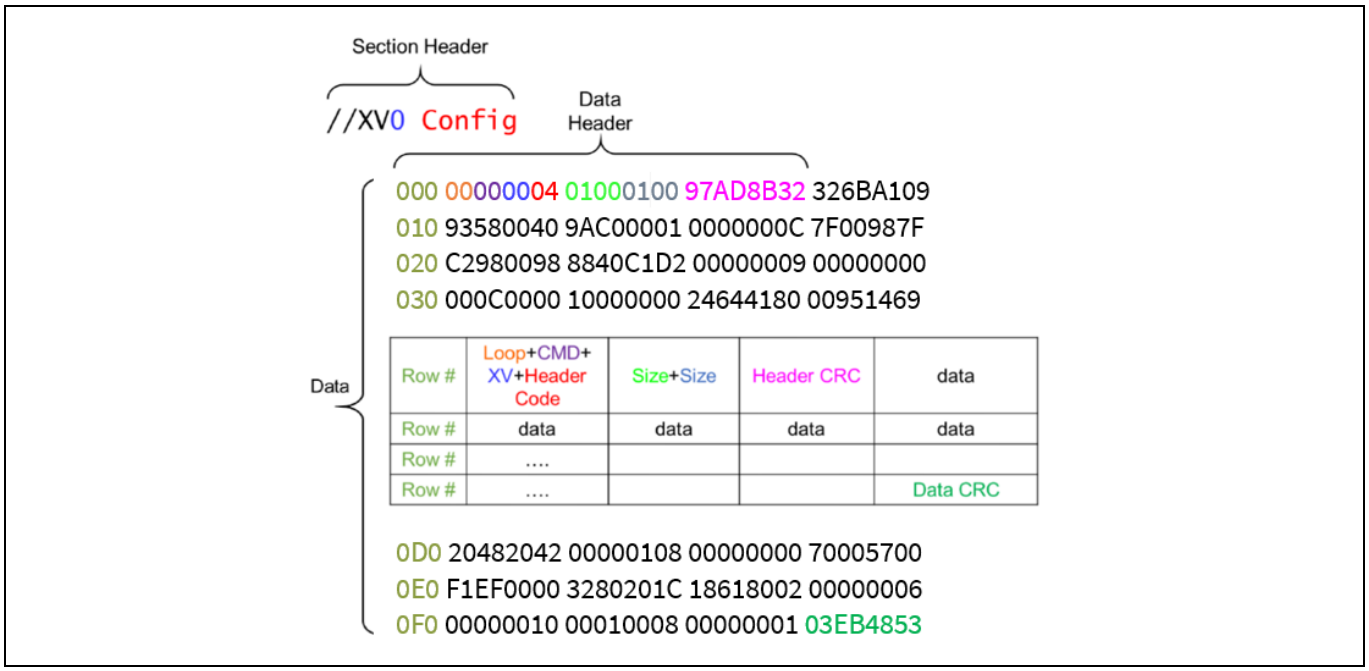


Figure 37 Configuration section example

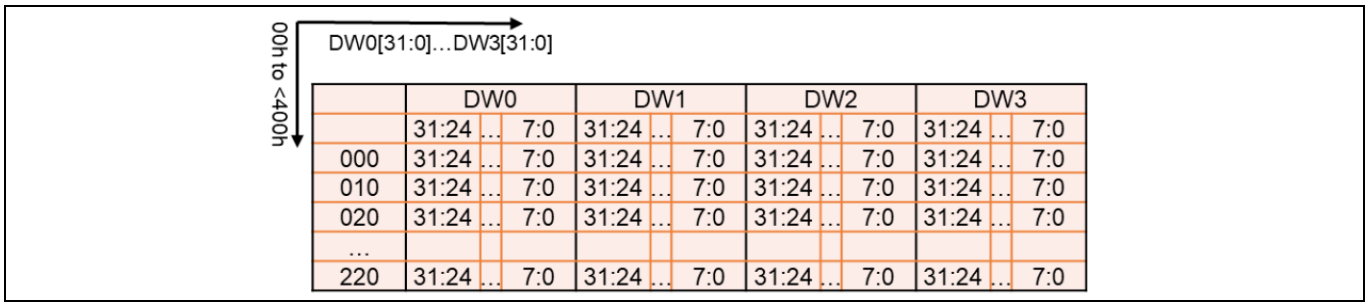


Figure 38 Data arrangement

6.4.2.1 Configuration section header

The section header separates different sections of the data. It always begins with "//XV", followed by a number. This number is the XVcode. The header also displays a short-word description of the section type. In the example above, the XVcode is "0" and the section type is "Config".
During power-up of the XDPP1100 controller, the value of the resistor connected to the XADDR pin would be measured. Based on the resistance, the controller would decode the XV resistance value into one of 16 codes ranging from 0 to 15. The controller would then download the configuration data from a data section matching that XVcode. If the configuration data with that particular XVcode is not in OTP, then the XV0 data would be used. If a partial configuration exists in OTP for that XVcode, the controller would download those values and override the existing values.

The section type describes how the data is used.

6.4.2.2 Configuration row number

Each succeeding row begins with a 3-digit running byte count (in hexadecimal base). It can be used as a row number designation for that particular section. For example, if the 3 digits are “020”, the byte count would be 32 since 0x20 equals 32 in decimal. Additionally, since each row contains a maximum of 16 bytes, if we drop the trailing 0, it can be viewed as row #2. The first row should be considered as row #0.

6.4.2.3 First DWORD in configuration data header

The first DWORD in a data section also contains the XVcode and the section type information. The header code encodes the section type so it can be recorded in the OTP. Once again, for the example above, the XVcode is “0”, and the section type is encoded to be “04”. In **Section 6.6.2**, there is a list of all the header codes of XDPP1100 configuration sections and their meanings.

In case the section type is PMBus partial (header code 0x0B), the most significant byte defines the associated loop (or page), and the 2nd most significant byte of the DWORD is the command code for the partial data. Those two bytes should be set to 0 for any other header codes.

6.4.2.4 Configuration size and redundant size

The second DWORD in the configuration data header specifies the size information of the section. The size of the section is recorded twice for data integrity purposes. In the example above, the size of the section is “0x0100” in hexadecimal units, which is 256 bytes of data. The header checksum value verifies this.

6.4.2.5 Configuration header CRC

The third DWORD in the configuration header is the header CRC. The header CRC checks the validity of the first two DWORDs. Above, the header CRC is “0x97AD8B32”.

6.4.2.6 Configuration data

Following the header CRC, the Register data information is all packed together in a data block. The block is divided into DWORDs. DWORDs are used because that is the most efficient way to program the device based on the native 32-bit architecture of the communication interface to internal registers. The DWORD ordering in the configuration file is DWORD0 on the left, followed by DWORD1, DWORD2, and DWORD3 at the furthermost right location on each row. Note that the byte ordering within a DWORD is such that the most significant byte (bits [31:24]) is located on the left, and the least significant byte (bits [7:0]) is on the right. When programming the device, the entire DWORD would be programmed at the same time.

6.4.2.7 Configuration data CRC

The final DWORD in a data section is always the data CRC. Its location could be DWORD0, DWORD1, DWORD2, or DWORD3 on the last row, depending on the data size. The CRC is calculated over the data over the section, beginning with the first byte after the data header and ending with the last byte of data (obviously not including the data CRC itself). In the example above, the first byte after the data header is located at DWORD3, bits [7:0], which is 0x80. The next byte in the CRC calculation is 0x3f, followed by 0x4b and 0x2b. For the next byte, the row number is incremented. Therefore, it is located at row 010, DWORD0, bits [7:0] which is 0x9b and so on. Finally, the example shows the value of the data CRC to be “0x03EB4853”.

6.4.3 Steps for programming the configuration

The configuration data is stored in the OTP using the scratchpad without changing any operational registers. The commands used to upload any section types are common. The overview steps for programming XDPP1100 are shown below. The steps provided assume that the user has already communicated with the device over PMBus. Each data section must be programmed individually to ensure no buffer overrun inside the device.

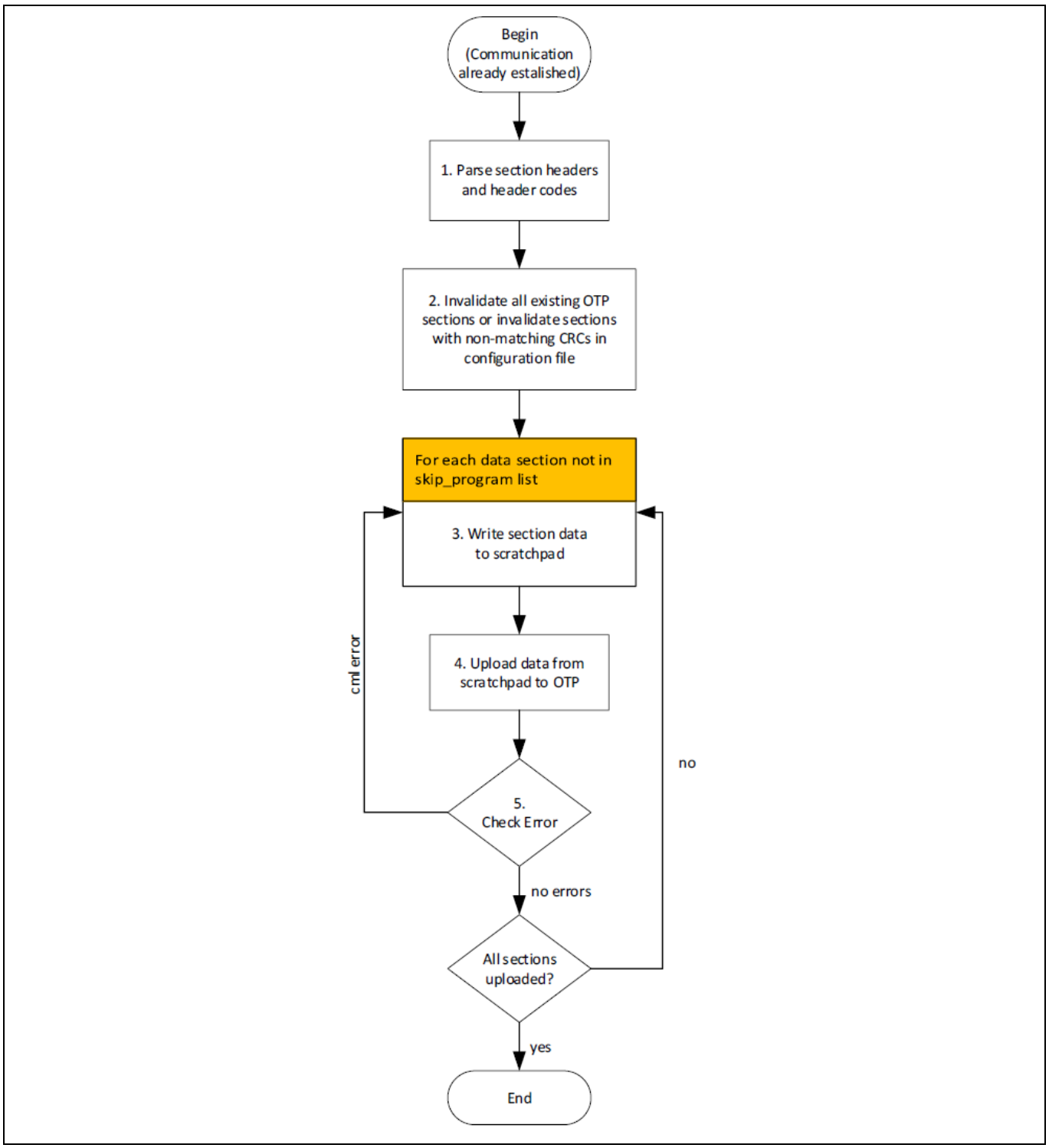


Figure 39 Data arrangement

6.4.3.1 Parsing section header and header code

The programmer needs to extract the XVcode and the header code from the header. Recall that this information can be obtained easily by first recognizing the section header that begins with "//XV". Once this is determined, the data header DWORD can readily be identified. The header code is the least significant byte (byte0), and the XVcode is the 2nd least significant byte (byte1). The XVcode is usually 0 (unless the multi-config feature is used).

The size of the section (bytes [1:0] or bytes [3:2]) of the 2nd DWORD should also be recorded to validate the number of bytes written. For the example given above, the 2nd DWORD is 02280228. Therefore, the size of 0x228 or decimal 552 bytes should be recorded.

6.4.3.2 Invalidating existing OTP data

The old data must be "invalidated" before writing new data to the OTP. Note that this is a non-reversible process. Once the data has been invalidated, the device can no longer access the data. To invalidate the old data, please refer to **Section 6.6.2**.

6.4.3.3 Programming the sections of the configuration file

The different sections of the configuration file are programmed one after another. First, the data is written to a scratchpad in the device's RAM, and afterwards, a command is executed to store the data from the scratchpad to OTP. A full configuration typically consists of the register configuration and three (XDPP1100-Q024) or four (XDPP1100-Q040) PMBus loop configurations, so the following code listing has to be executed four or five times for programming a new device.

For updating the configuration of a previously programmed device, it is usually not necessary to reprogram the PMBUS Default sections, because the device will always load the User sections at power-up. In that case, only the register configuration and one (XDPP1100-Q024) or two (XDPP1100-Q040) PMBus loop configurations have to be programmed.

Code Listing 5

```
// 1. Read the data from the section you want to program to an array. Please note that this is a text file using
//    ASCII encoding. As the data is stored as DWORDs in the text file, it is best to use a DWORD array.
//    If you use a Byte array, make sure that the Bytes are written in the correct order (see Section 6.4.2).
     data-dword-array = Read_Section_From_Config_File()
     config-size = data-dword-array.GetUpperBound(0)
     config-bytes = 4 * config-size

//    The data has to be written to the scratchpad (starting at 0x20061800) and copied from there to OTP.

// 2. Follow the below steps to write the data to the scratchpad.
     For ptr = 0 To config-size
          i2c Write 4 Bytes (0x20061800 + 4*ptr, data-dword-array(ptr) )
     Next
     // After this step, you will have the complete data section placed on the scratchpad.
```

Code Listing 5

	0	4	8	C
20061800	00000004	01000100	97AD8B32	326BA109
20061810	93580040	9AC00001	0000000C	7F00987F
20061820	C2980098	8840C1D2	00000009	00000000
20061830	000C0000	10000000	24644180	00951469
20061840	00000000	00000000	000000C0	29020000
20061850	40020900	80101000	DFC00000	0000400F
20061860	60011040	00000010	00000000	0003F800
20061870	D0000008	89499873	004D29A5	00000000
20061880	00002AA8	01FFF800	06248100	410C3061
20061890	1861A630	86186186	E6181861	23028503
200618A0	8007600C	01002006	00000002	00000000

```
// The total amount of data written should match the value extracted in Section 6.4.3.1.
// Follow the steps below to move data from scratchpad to OTP.

// 3. Write MFR_FIRMWARE_COMMAND_DATA with the following data
//    The number of bytes to copy to OTP has to be entered. If it is e.g. 256 bytes (=0x100), the MSB will be
//    0x01, the LSB will be 0x00. The number can be split up using modulo and integer division.
    writeData(0) = config-bytes Mod 256   // Size LSB (256 modulo 256 = 0 = 0x00)
    writeData(1) = config-bytes \ 256     // Size MSB (256 integer division 256 = 1 = 0x01)
    writeData(2) = 0
    writeData(3) = 0                      // partition number = 0
    PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 4. Write MFR_FIRMWARE_COMMAND with the data byte 0x11 to store the data to OTP
    PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x11)

// 5. Wait until writing to OTP is completed
    data(0) = 1
    i = 0
    While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
        If i >= 10 Then
            Exit While   // Exit if writing was not successful after 1s – do some fault handling
        End If
        i++
        Wait(100ms)
        PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
        PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
    End While


// 6. The new config is only loaded to the volatile memory and becomes effective after restarting the
//    device. You can either do this by cycling the supply voltage or by writing MFR_FIRMWARE_COMMAND
//    PMBus command with the data byte 0x0E. This is only needed after writing the last data section.
    PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x0E)
```

6.5 Storing configuration to OTP via volatile registers

With this approach, the configuration is first written to the volatile registers in RAM. Therefore, it will be lost after cycling the input voltage. To permanently store the configuration, it has to be written to OTP. Once this is done, the device will load the configuration from OTP to RAM automatically when it is powered up.

It is possible to store a full configuration using this method by writing all the registers and PMBus commands, as described in the following Sections. However, the direct method via scratchpad (**Section 6.4**) is easier to use in most cases for storing a full configuration, e.g., during production.

The main benefit of storing the configuration via volatile registers is that updating an existing configuration can be easily done. A device which has been previously programmed will load the configuration to RAM when it is powered up. Specific registers or PMBus commands can be easily changed and stored in the changed configuration back to OTP. All the registers and PMBus commands do not have to be written, only those which are to be changed, followed by the commands to copy the configuration from RAM to OTP.

6.5.1 Production file (.xcf) format

For storing a full configuration via volatile registers, it is possible to export a production file (.xcf) from XDPP1100 GUI, which includes all the register values and PMBus commands. Open the board design in XDPP1100 GUI (if a firmware patch with MFR commands is used, make sure the correct PMBus spread sheet is loaded before), go to the Multi-Device-Programmer and click on the “Save Production File” button at the bottom.

This file can be parsed to extract all the I2C and PMBus commands to be sent to the controller. Below is a short description of this .xcf file format. For more details, please refer to the separate XDPP1100 Production File Application Note.

- All numbers in the .xcf file are hex values.
- Lines with valid data always start with %.
- The number of characters in the line (excluding % and spaces) is given as hex value.
- Type 6h stands for data, all lines in the .xcf file are type 6.
- The CRC calculation is the sum of all hex values in the line (except CRC). For register data, the CRC is always 2 characters, for PMBus configuration it is always 3 characters.
    - Example for register data in **Table 1**: CRC = 1+6+6+8+7+4+10+9 = 51 = 33h
    - Example for PMbus configuragion in **Table 2**: CRC = 2+2+6+1+8+9+9 = 37 = 025h
- Data Length is the number of data characters as hex value.
- Register Address (8 characters) is the register address to write the data to.
  Command Code (2 characters) is the PMBus command to write the data to.
- Data is the register data or PMBus data to be written.

Register Configuration:

Table 1 shows an example of a line containing register data “% 16 6 33 8 70000400 0000A009”.

Table 1 Register Configuration

Valid Data	Number of characters in line	Type	CRC	Data Length	Register Address	Data
%	16h = 22 characters	6	33	8h = 8 characters	70000400	0000A009

PMBus configuration:

Table 2 shows an example of a line containing PMBus data "% 22 6 025 18 99 000000000000000000000000".

Table 2 PMBus Configuration

Valid Data	Number of characters in line	Type	CRC	Data Length	Command Code	Data
%	22h = 34 characters	6	025	18h = 24 characters	99	000000000000 000000000000

6.5.2 Storing register configuration to OTP

Code Listing 6

```
// 1. Write the register configuration to RAM
     // Write register values into the respective address by following the I2C protocol
     For register = start to end
          i2c Write 4 Bytes (register.address, register.data)
     Next
     // Follow the below steps to write the register configuration from RAM to OTP

// 2. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0   // Xvalent number (usually 0 if multi-config is not used)
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 0
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 3. Optional – Only needed if you want to change the settings
//    Write to register address fw_config_xv_enable (0x2005C504) the following data byes
     writeData(0) = 1   // XValent and address offsets enabled (0 – disabled, 1 – enabled)
     writeData(1) = 0   // Source of Config XValent number (0 = XADDR1, 1 = XADDR2)
     writeData(2) = 0   // Source of PMBus address offset (0 = XADDR1, 1 = XADDR2)
     writeData(3) = 0   // Source of I2C address offset (0 = XADDR1, 1 = XADDR2)
     i2c Write 4 Bytes (0x2005C504, writeData)


// 4. Write MFR_FIRMWARE_COMMAND with the data byte 0x04 to store register configuration to OTP
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x04)

// 5. Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While   // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

6.5.3 Storing PMBus configuration to OTP

There are two PMBus pages (0 and 1) for the two control loops of the device, which have to be stored in OTP separately.

XDPP1100-Q040 uses both loops. Therefore, the configuration of both loops has to be stored in OTP as PMBus Default section and as PMBus User section.
XDPP1100-Q024 uses loop 0 only. Therefore, the configuration of loop 0 has to be stored in OTP as PMBus Default section and PMBus User section, but the configuration of loop 1 should only be stored in OTP as PMBus Default section.

For updating the configuration of a previously programmed device, it is usually not necessary to reprogram the PMBus Default sections. Some OTP memory can be saved by only reprogramming the PMBus User sections.

Code Listing 7

```
// There are two PMBus pages (0 and 1) for the two Control Loops of the device
     For loop = 0 to 1
// 1. Write the PMBus configuration (PMBus commands) to RAM
          // Write page command to select the loop
          // Write all PMBus commands for the selected loop
          PMBus Write Byte (PAGE, loop)  // select loop 0 or 1
          For command code = start to end
               Write Pmbus command (command code, data bytes)
          Next
          // Follow the below steps to write the PMBus configuration from RAM to OTP

// 2.Write MFR_FIRMWARE_COMMAND_DATA with the following data
          writeData(0) = 0  // Xvalent number  (usually 0 if multi-config is not used)
          writeData(1) = 0
          writeData(2) = 0
          writeData(3) = 0
          PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 3. Write PMBus commands STORE_DEFAULT_ALL and STORE_USER_ALL to store PMBus
//     configurations to OTP
          // Store loop 0 and loop 1 Default sections – can be omitted when reprogramming
          PMBus Send Byte (STORE_DEFAULT_ALL)
          // Store loop 0 User for both device types – Store loop 1 User only for XDPP1100-Q040
          If ( (loop = 0) OR (Device = XDPP1100-Q040) ) Then
               PMBus Send Byte (STORE_USER_ALL)
          End If

// 4. Wait until writing to OTP is completed
          data(0) = 1
          i = 0
          While data(0) <> 0  // As long as data(0) is not 0, writing to OTP is ongoing
               If i >= 10 Then
                    Exit While  // Exit if writing was not successful after 1s – do some fault handling
               End If
               i++
               Wait(100ms)
               PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)  // 0x05 = Get Configurator Status
               PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
          End While
     Next
```

6.5.4 Storing partial PMBus configuration to OTP

Below are the steps for writing a partial PMBus configuration to OTP. This can be used to store single changed PMBus commands to OTP instead of storing the complete PMBus section. Like this, some OTP memory can be saved. PMBus partial configuration has a limit of 32 partial configurations per Xvalent.

Code Listing 8

```
// 1. Write the PMBus configuration (PMBus commands) to RAM
//    Example: Writing PMBus command TON_RISE (0x61) to value of 0x0050 for loop 0
     PMBus Write Byte (PAGE, 0)  // select loop 0
     PMBus Write Word (TON_RISE, 0x0050)

// 2. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0  // Xvalent number  (usually 0 if multi-config is not used)
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 0
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 3. Write PMBus command STORE_USER_CODE with the command code to be stored to OTP in
//    the data byte
     // Example: Writing PMBus command STORE_USER_CODE (0x17) with value of 0x61 (TON_RISE)
     PMBus Write Byte (STORE_USER_CODE, 0x61)

// 4.  Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0  // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While  // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)  // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

6.5.5 Storing partial register configuration to OTP

Below are the steps for writing a partial register configuration to OTP. This stores single or multiple changed config registers to OTP instead of storing the complete configuration section. Like this, some OTP memory can be saved. Register partial configuration has no limit on the number of stores (as long as OTP space is available).

Code Listing 9

```
// 1. Write the register address and data to the scratchpad, at the address 0x20061800
//    (RAM_SCRATCHPAD_ADDRESS) and 0x20061804 (RAM_SCRATCHPAD_ADDRESS + 4)
     i2c Write 4 Bytes (0x20061800, "Reg address")
     i2c Write 4 Bytes (0x20061804, "Reg data")
     // After this step, register address and data are placed on the scratchpad

// 2. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0   // Xvalent number  (usually 0 if multi-config is not used)
     writeData(1) = 8   // Size – for one register: 4 bytes for address and 4 bytes of data = 8 bytes
     writeData(2) = 0
     writeData(3) = 0
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 3. Write MFR_FIRMWARE_COMMAND with the data byte 0x14 to store the partial configuration to OTP
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x14)

// 4. Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While   // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

6.6 Invalidating firmware patches and configurations

6.6.1 Invalidating a firmware patch

The MMU (memory management unit) maps the active patch to the beginning of the patch partition so that it can be executed correctly. To read/write the OTP's real (unmapped) content, the MMU needs to be disabled.

While the MMU is disabled, the patch is not executed any more. If there are patched PMBus commands in the configuration stored in OTP, the PMBus configuration will not be valid any more and, therefore, not be loaded. All settings are set to reset values. This also affects the I2C/PMBus address of the device. If the I2C/PMBus is changed via the PMBus setting or offset resistor, it will be reset to 0x10 / 0x40 while the patch is not executed. After storing a new patch with the same patched PMBus commands as the previous one, the PMBus config will be valid again and loaded correctly. The address change can be avoided by storing the PMBus command FW_CONFIG_PMBUS (0xC9), which includes the address settings, as a partial PMBus config before starting with the invalidation. The partial PMBus configuration will be loaded in any case so that the I2C / PMBus address stays the same during the invalidation and storing of the new patch. Please refer to **Section 6.5.4** for instructions on how to store a partial PMBus configuration.

Code Listing 10

```
// 1. Disable the MMU
     // Store the boot descriptor (address 0x2005C544) into temp, to restore it at the end of the function
     i2c Read 4 Bytes (0x2005C544, temp)
     // Disable MMU by writing 0xF into boot descriptor register
     i2c Write 4 Bytes (0x2005C544, 0x0000000F)

// 2.Reset after disabling the MMU
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x0E)  // 0x0E = Reset
     // If you want to store a new patch afterwards, at this point, it makes sense to check the remaining
     // space in the partition (see Section 6.8.2) and to make sure that there is enough space left

// 3. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0x10  // File type to invalidate (Patch = 0x10)
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 0x01  // Partition number (usually 1 for patch partition)
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 4. Write MFR_FIRMWARE_COMMAND with the data byte 0x12 to invalidate the patch
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x12)

// 5.  Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0  // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While  // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)  // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

Code Listing 10

```
// 6. Enable the MMU by restoring the boot descriptor which was saved in the first step into temp
     i2c Write 4 Bytes (0x2005C544, temp)

// 7. Reset after enabling the MMU
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x0E)   // 0x0E = Reset
```

6.6.2 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.

Setting header_code determines the type of configuration to be invalidated:

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

Code Listing 11

```
// 1. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = header_code   // Configuration type to invalidate (see description above)
     writeData(1) = 0             // Xvalent number  (usually 0 if multi-config is not used)
     writeData(2) = 0
     writeData(3) = 0             // Partition number (0 for data partition)
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 2. Write MFR_FIRMWARE_COMMAND with the data byte 0x12 to invalidate the config
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x12)

// 3.  Wait until writing to OTP is completed
     data(0) = 1
     i = 0
     While data(0) <> 0   // As long as data(0) is not 0, writing to OTP is ongoing
          If i >= 10 Then
               Exit While   // Exit if writing was not successful after 1s – do some fault handling
          End If
          i++
          Wait(100ms)
          PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)   // 0x05 = Get Configurator Status
          PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     End While
```

6.7 Configurator status check

Follow the steps below to verify if storing to OTP was successful (after writing the trim, register configuration, PMBus configuration or partials to OTP) or if data transfer to/from OTP is still ongoing and the configurator is busy.

Code Listing 12

```
// 1. Write MFR_FIRMWARE_COMMAND with the data byte 0x05 to request the status of the configurator
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x05)

// 2.  Read MFR_FIRMWARE_COMMAND_DATA to get the status of the configurator
     PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     If data(0) = 0 then
          Message ”configurator is free- Write to OTP was successful”
     Else If data(0) = 1 then
          Message ”configurator is busy -data transfer from/to OTP  is ongoing”
     Endif
```

6.8 Procedure to find remaining free memory in OTP

This 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.

6.8.1 Procedure to find remaining space in data partition to store a new configuration

Code Listing 13

```
// 1. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 0  // Partition number (0 for data partition)
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 2. Write MFR_FIRMWARE_COMMAND with the data byte 0x10 to request the remaining space in
//    the selected partition in OTP
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x10)

// 3. Read MFR_FIRMWARE_COMMAND_DATA. The return value provides the remaining space in Bytes.
     PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     // If the data value is e.g. 0x0000222C, the free space in bytes = 0x222C = 8748 ≈ 8.54 kB
```

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

The MMU (memory management unit) is mapping the active patch to the beginning of the patch partition, so that it can be executed correctly. To be able to read/write the real (unmapped) content of the OTP, the MMU needs to be disabled.

This procedure will not return the correct result if the MMU is active and if there is at least one invalidated and one active patch in the partition. For updating the firmware patch, it is recommended to combine the reading of the remaining space with invalidating the old patch (see comment in **Section 6.6.1**), because the MMU needs to be disabled for invalidating the patch anyway. In case there is not enough space left to store the new patch, the invalidation can still be canceled at that point.

Code Listing 14

```
// 1. Write MFR_FIRMWARE_COMMAND_DATA with the following data
     writeData(0) = 0
     writeData(1) = 0
     writeData(2) = 0
     writeData(3) = 1   // Partition number (usually 1 for patch partition)
     PMBus Block Write (MFR_FIRMWARE_COMMAND_DATA, writeData)

// 2. Write MFR_FIRMWARE_COMMAND with the data byte 0x10 to request the remaining space in
//    the selected partition in OTP
     PMBus Write Byte (MFR_FIRMWARE_COMMAND, 0x10)

// 3. Read MFR_FIRMWARE_COMMAND_DATA. The return value provides the remaining space in Bytes.
     PMBus Block Read (MFR_FIRMWARE_COMMAND_DATA, data)
     // If the data value is e.g. 0x00007C00, the free space in bytes = 0x7C00 = 31744 = 31 kB.
```

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
V 1.3	02-03-2023	Added pseudo code for programming using the I2c section, Added I2C and PMBus communication protocol block digrams. Added direct programming to OTP via scratchpad

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

Edition 2023-03-02

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2023 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.