### Contents

1. Introduction .........................................................................................................................2  
2. MTP Programming Procedure ...............................................................................................3  
3. MTP Programming Flow Chart .............................................................................................6  
4. MTP CRC Verification Flow Chart ..........................................................................................6  
5. Appendix ................................................................................................................................8  
   5.1 MTP Downloader Code Examples ......................................................................................8  
   5.2 MTP Programming Code Example ....................................................................................11  
6. Revision History ....................................................................................................................15

### List of Figures

- Figure 1. P9415 MTP Downloader and MTP Programming Overview ...........................................2  
- Figure 2. MTP Programming Flow Chart ...............................................................................6  
- Figure 3. MTP CRC Verification Flow Chart .........................................................................7
1. Introduction

In multi-time programmable (MTP) programming mode, the P9415 runs a temporary MTP downloader program that is loaded and executed in the RAM. The application processor (AP) communicates with the downloader to send the data that needs to be programmed by MTP. The MTP downloader will perform the programming and verify the data being programmed. This approach ensures maximum speed, safety, and is the easiest implementation by the customer.

The µC is halted before loading the MTP downloader in the RAM. Any program (or process) running up to this point is interrupted and abandoned. The µC is reset (or power is removed) after MTP programming is completed, followed with fresh start. If the programming was successful, the newly programmed firmware (FW) starts execution. If the programming failed, the µC enters exception state. Any partially programmed sector in the MTP retains its data. The MTP programming procedure can be restarted multiple times until completed successfully. The µC does not need to be reset to restart the programming process – just run the MTP programming process from the beginning until completed with “pass” result. If the power is removed during programming, the process can be restarted. The MTP downloader programs only the bits that need to be programmed and it skips memory locations already containing the correct data.

In order to program the IDTP9415 Receiver, DC power should be supplied at Vout on the P9415 Receiver. Different MTP downloaders are used with each supply location. AC power is provided by a Wireless Power Transmitter (WPTX) that needs periodic communication, which the programming process does not support. The WPTX will remove the power after protocol timeout thus leaving the MTP only partially programmed. Following AC resubmission is not possible because the P9415 is left with partially programmed MTP memory and non-functioning FW needed to complete a power transfer contract.

The MTP programming hardware charge pump is connected to Vout on the P9415. When the power is applied to Vout, the circuit is initially powered through the LDO body diode and after the downloader starts execution the LDO is turned ON, connecting Vrect to Vout.

Figure 1. P9415 MTP Downloader and MTP Programming Overview
2. MTP Programming Procedure

1. Power up the P9415 Rx with 5V ±10% at Vout
2. Open the IDTP9415 GUI and establish I2C connection using device address 0x3B
3. Write address 0x3000 = [0x5A] to unlock system registers
4. Set timing and clock
   a) Configure clocks and timing
      i. Write 0x00 to address 0x3004 to set HS clock
      ii. Write 0x09 to address 0x3008 to set AHB clock
      iii. Write 0x05 to address 0x300C to configure 1us pulse
      iv. Write 0x1D to address 0x300D to configure 500ns pulse
   b) Pause the processor and enable MTP access via I2C
      i. Write 0x11 to address 0x3040

Wait 10ms

5. Write address 0x3040 = [0x10] to halt the µC

Wait 10ms

6. Load the MTP downloader program in the RAM at address 0x0800. The program is in a text format of a ‘C’ code constant byte array MTPDownloader9415 (1872 bytes)
7. Write address 0x0400 = [0x00] to initialize the programming structure in the RAM
8. Write address 0x3048 = [0xD0] to remap RAM to program memory space
9. Write address 0x3040 = [0x80] and then wait 100mS to reset the µC and run the MTP downloader program. NOTE: The I2C packet will not be acknowledged.

Wait 100ms

10. Program the FW in the MTP. The FW is provided as a “hex” or “bin” file and will be loaded in 128-byte sections. The starting address for programming is 0x0000
    a) Address 0x0400 in RAM must be loaded for every individual section of code before programming begins. The structure declaration is provided below and the memory is in Little Endian format:

    \begin{verbatim}
    \typedef struct { // the structure is mapped to RAM address 0x0400
    \    u16 Status;
    \    u16 StartAddr;
    \    u16 CodeLength;
    \    u16 DataChksum;
    \    u8   DataBuf[128];
    \} P9415PgmStrType;
    \end{verbatim}
    i. Load Status = 0x0000;
    ii. Load StartAddress = address_of_the_current_section_of_FW;
    iii. Load CodeLength = section_length_in_bytes; // usually 128
    iv. Load DataChksum = sum of StartAddress, CodeLength, and data bytes.
    v. Load DataBuf with data from the FW source file
    b) Write 0x0400 = [0x01] to start programming cycle. The downloader executes the MTP programming FW and checks the integrity of the data. The MTP memory is verified byte by byte against the structure in the RAM.
c) At completion of the programming cycle, the MTP downloader clears BIT0 at address 0x0401 and loads Status Code. The Status Codes are:
   - 0x02 – “OK”. Programming was successful.
   - 0x04 – “MTP Write Error”. Read back value does not match value in the RAM
   - 0x08 – “Check Sum Error”. The programming structure integrity is corrupt
   Codes not mentioned above are reserved

d) The Application Processor reads the code and proceeds to the next logical step

11. After programming is completed, power cycle the IDTP9415
12. Write address 0x3000 = [0x5A] to unlock system registers
13. Set timing and clock
   a) Configure clocks and timing
      i. Write 0x00 to address 0x3004 to set HS clock
      ii. Write 0x09 to address 0x3008 to set AHB clock
      iii. Write 0x05 to address 0x300C to configure 1us pulse
      iv. Write 0x1D to address 0x300D to configure 500ns pulse
      v. Write 0x00 to address 0x3004 to remove MTP write protection
      vi. Write 0x00 to address 0x300D to reset M0

   b) Pause the processor and enable MTP access via I2C
      i. Write 0x11 to address 0x3040

      Wait 10ms

14. Write address 0x3040 = [0x10] to halt the µC

      Wait 10ms

15. Load the MTP downloader program in the RAM at address 0x0800. The program is in a text format of a ‘C’ code constant byte array MTPDownloader9415 (1872 bytes)
16. Write address 0x0400 = [0x00] to initialize the programming structure in the RAM
17. Write address 0x3048 = [0xD0] to remap RAM to program memory space
18. Write address 0x3040 = [0x80] and then wait 100mS to reset the µC and run the MTP downloader program. NOTE: The I2C packet will not be acknowledged.

      Wait 100ms

19. Configure MTP CRC Check Utility:
   a) Load 16-bit MTP start address to 0x402. Note: 0x402 (low byte), 0x403 (high byte) Start address will be set to 0x0000 to run the verifier from the beginning of the MTP.
   b) Load 16-bit MTP data size 0x6000 to 0x404. Note: 0x404 (low byte), 0x405 (high byte) MTP data size. The data size is 24K bytes
   c) Write the 16-bit CRC into address 0x406. Note: 0x406 (low byte), 0x407 (high byte) Checksum value is provided as part of release package for each FW. Calculation is done using following steps: The first byte is the data in MTP start address and the last byte is the data in MTP start_address + data_size -1.
   d) Write 0x0400 = [0x11] to start MTP data CRC-16 check.
   e) Wait 20ms.
f) The MTP verifier clears BIT0 at address 0x400 when the program finishes, and loads the Status Code to 0x401. The Status Codes are:

- 0x01 – CRC Busy
- 0x02 – CRC OK
- 0x08 – CRC ERROR (go to step 21)

20. After programming is completed, either cycle power to the IDTP9415.

21. In case of CRC ERROR, repeat the process from steps 1–20 for 2 retries.
3. MTP Programming Flow Chart

Figure 2. MTP Programming Flow Chart

4. MTP CRC Verification Flow Chart

See Figure 3 on the following page.
Figure 3. MTP CRC Verification Flow Chart

1. MTP CRC Verification Start
2. Initialize SRAM for CRC verification
3. Write MTP start address and MTP data size to be verified
4. Write FW CRC (from release package)
5. Trigger Start Program (0x400 = 0x11)
6. Wait and read status code
7. Check Status Code
   - CRC BUSY? (YES)
   - CRC OK (NO)
   - CRC ERROR
8. Write Key
9. In case of CRC ERROR, repeat steps from 1-20
10. End

Timeout Path:
- Timeout
- Timeout Error
5. Appendix

5.1 MTP Downloader Code Examples

1. MTP downloader code for powering P9415 at Vout

// this byte array needs to be loaded at RAM address 0x0800. This is the MTP downloader and it also verifies CRC.

// use this array with power supply connected to Vout

// Do not modify this code

static byte[] MTPDownloader9415 = {
  0x00, 0x02, 0x00, 0x20, 0x99, 0x00, 0x00, 0x00, 0x9D, 0x00, 0x00, 0x00, 0x9F, 0x00, 0x00, 0x00,
  0xFE, 0xE7, 0x00, 0x00, 0x03, 0x48, 0x04, 0x49, 0x02, 0x4A, 0x04, 0x4B, 0x70, 0x47, 0x00, 0x00,
  0x03, 0xBC, 0x00, 0xF0, 0x4D, 0xFB, 0x00, 0x00, 0x05, 0x48, 0x00, 0x47, 0xFE, 0xE7, 0xFE, 0xE7,
  0xF7, 0xFF, 0x00, 0xF0, 0x45, 0xFA, 0x00, 0xF0, 0x47, 0xFB, 0x03, 0xB4, 0xFF, 0xF7, 0xF2, 0xFF,
  0x78, 0xC1, 0xFB, 0xD8, 0x52, 0x07, 0x00, 0xD3, 0x30, 0xC1, 0x00, 0xD5, 0x0B, 0x60, 0x70, 0x47,
  0x04, 0x07, 0x00, 0x00, 0x00, 0x23, 0x00, 0x24, 0x00, 0x25, 0x00, 0x26, 0x10, 0x3A, 0x01, 0xD3,
  0x00, 0xD0, 0xFB, 0x1A, 0xA2, 0x46, 0xAB, 0x46, 0x33, 0x43, 0x18, 0x47, 0xF4, 0x06, 0x00, 0x00,
  0x00, 0xBF, 0xE6, 0x48, 0x00, 0x88, 0x40, 0x06, 0xC0, 0x0F, 0x00, 0x28, 0xF9, 0xD1, 0x1E, 0xE0,
  0xEA, 0x48, 0x85, 0x80, 0x30, 0x46, 0x85, 0x80, 0x30, 0x20, 0x20, 0x40, 0x10, 0x07, 0xD1,
  0x11, 0x00, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x20, 0x60, 0x01, 0x00, 0x20, 0x60, 0x00, 0x00,
  0xFE, 0x05, 0x00, 0x00, 0x00, 0x60, 0x01, 0x00, 0x20, 0x60, 0x00, 0x00, 0x20, 0x60, 0x01, 0x00,
  0x08, 0x0C, 0x70, 0x83, 0x00, 0x22, 0x44, 0xE0, 0xF6, 0x48, 0x40, 0x88, 0x08, 0x18, 0x83, 0xB2,
  0x01, 0x93, 0xD8, 0x13, 0x00, 0x00, 0x28, 0x03, 0xD1, 0x01, 0x20, 0xC0, 0x03, 0x18, 0x43, 0x01,
  0x0E, 0x48, 0x48, 0x03, 0x08, 0x00, 0x80, 0x4F, 0x01, 0x98, 0x01, 0x98, 0x05, 0x78, 0x01, 0x93,
  0x11, 0x00, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x20, 0x60, 0x01, 0x00, 0x20, 0x60, 0x00, 0x00,
  0x0E, 0x05, 0x00, 0x00, 0x00, 0x60, 0x01, 0x00, 0x20, 0x60, 0x00, 0x00, 0x20, 0x60, 0x01, 0x00,
  0x08, 0x0C, 0x70, 0x83, 0x00, 0x22, 0x44, 0xE0, 0xF6, 0x48, 0x40, 0x88, 0x08, 0x18, 0x83, 0xB2,
  0x01, 0x93, 0xD8, 0x13, 0x00, 0x00, 0x28, 0x03, 0xD1, 0x01, 0x20, 0xC0, 0x03, 0x18, 0x43, 0x01,
  0x0F, 0x83, 0x00, 0x22, 0x44, 0xE0, 0xF6, 0x48, 0x40, 0x88, 0x08, 0x18, 0x83, 0xB2,
  0x01, 0x93, 0xD8, 0x13, 0x00, 0x00, 0x28, 0x03, 0xD1, 0x01, 0x20, 0xC0, 0x03, 0x18, 0x43, 0x01,
  0x0F, 0x83, 0x00, 0x22, 0x44, 0xE0, 0xF6, 0x48, 0x40, 0x88, 0x08, 0x18, 0x83, 0xB2,
  0x01, 0x93, 0xD8, 0x13, 0x00, 0x00, 0x28, 0x03, 0xD1, 0x01, 0x20, 0xC0, 0x03, 0x18, 0x43, 0x01,
  0x0F, 0x83, 0x00, 0x22, 0x44, 0xE0, 0xF6, 0x48, 0x40, 0x88, 0x08, 0x18, 0x83, 0xB2,
5.2 MTP Programming Code Example

```c
byte i2cDeviceAddr = 0x3b; // IDTP9415 i2c device address

bool Xfer9415i2c(byte i2cAddr, ushort MemAddr, byte[] bBuf, int bOffs, int bSize, ReadWrite RW);
// prototype of i2c read/write function used in the example. The implementation is processor/OS dependent
// i2cAddr is device address.
// MemAddr is memory destination/source address
// bBuf is the array holding data to be written, or the destination of the read data
// bOffs is the offset in bBuf of the current data chunk
// bSize is the size of the data in bytes
// RW is the direction of the i2c transaction
// the function returns TRUE if OK, and FALSE if it fails.

private bool PgmMTPwRAM(ushort MtpAddr, byte[] srcData, int size)
// this is the main programming function
// MtpAddr is the starting address in MTP, default value is 0x0000.
// srcData is the array populated with the FW code provided in the hex/bin file
// size is the FW size
{
    ignoreNAK = false; // restore to default in case previous call did not finish
    // configure the system
    if (!WriteI2cByte(0x3000, 0x5a)) return false; // write key
    Thread.Sleep(10);

    if (!WriteI2cByte(0x3004, 0x00)) return false; // Set HS clock
    Thread.Sleep(10);
    if (!WriteI2cByte(0x3008, 0x09)) return false; // set AHB clock
    Thread.Sleep(10);
    if (!WriteI2cByte(0x300C, 0x05)) return false; // configure 1us pulse
    Thread.Sleep(10);
    if (!WriteI2cByte(0x300D, 0x1D)) return false; // configure 500ns pulse
    Thread.Sleep(10);
    if (!WriteI2cByte(0x3040, 0x11)) return false; // Enable MTP access via I2C
    Thread.Sleep(10);
    if (!WriteI2cByte(0x3040, 0x80)) return false; // run M0
    ignoreNAK = false;
    Thread.Sleep(100);

    for (int i = 0; i < size; i += 128) // program pages of 128 bytes
    {
        byte[] sBuf = new byte[144];
        // 8-bytes header, 128-bytes data, 8-bytes padding to round to 16-byte boundary
        ushort StartAddr = (ushort)i;
        ushort CheckSum = StartAddr;
        ushort CodeLength = 128;
        Array.Copy(srcData, i, sBuf, 8, 128);

        int j = size - i;
        if (j < 128)  
        
```
{
    j = ((j + 15) / 16) * 16;
    CodeLength = (ushort)j;
} else {
    j = 128;
} j -= 1;
for (; j >= 0; j--)
    CheckSum += sBuf[j + 8]; // add the non zero values
CheckSum += CodeLength; // finish calculation of the check sum
Array.Copy(BitConverter.GetBytes(StartAddr), 0, sBuf, 2, 2);
Array.Copy(BitConverter.GetBytes(CodeLength), 0, sBuf, 4, 2);
Array.Copy(BitConverter.GetBytes(CheckSum), 0, sBuf, 6, 2);

//typedef struct { // write to structure at address 0x400
//  u16 Status;
//  u16 StartAddr;
//  u16 CodeLength;
//  u16 DataChksum;
//  u8 DataBuf[128];
// } P9220PgmStrType;
// read status is guaranteed to be != 1 at this point
if (!Xfer9415i2c(i2cDeviceAddr, 0x400, sBuf, 0, ((CodeLength + 8 + 15) / 16) * 16, ReadWrite.WRITE)) {
    addText("ERROR: on writing to OTP buffer");
    return false;
}
sBuf[0] = 0x01;
if (!Xfer9415i2c(i2cDeviceAddr, 0x400, sBuf, 0, 1, ReadWrite.WRITE)) {
    addText("ERROR: on OTP buffer validation");
    return false;
}
do {
    Thread.Sleep(20);
    if (!Xfer9415i2c(i2cDeviceAddr, 0x401, sBuf, 0, 1, ReadWrite.READ)) {
        addText("ERROR: on reading OTP buffer status");
        return false;
    }
} while ((sBuf[0] & 1) != 0);

// check status
if (sBuf[0] != 2) // not OK
{
    string er;
    if (sBuf[0] == 4)
        er = " WRITE ERR";
    else if (sBuf[0] == 8)
        er = " CHECK SUM ERR";
    else
        er = " UNKNOWN ERR";
    addText("ERROR: buffer write to OTP returned status " + sBuf[0] + er + " in 128-byte block @ 0x" + i.ToString("X4"));
    return false;
}
// Power cycle 9415 chip
if (!WriteI2cByte(0x3000, 0x5A)) return false;
if (!WriteI2cByte(0x3004, 0x00)) return false;
if (!WriteI2cByte(0x3008, 0x09)) return false;
if (!WriteI2cByte(0x300C, 0x05)) return false;
if (!WriteI2cByte(0x300D, 0x1D)) return false;
if (!WriteI2cByte(0x304C, 0x00)) return false; // remove MTP protection
if (!WriteI2cByte(0x304D, 0x00)) return false; // reset M0
if (!WriteI2cByte(0x3040, 0x11)) return false; // Enable MTP access via I2C
Thread.Sleep(10);

if (!WriteI2cByte(0x3040, 0x10)) return false;  // Halt M0

if (!Xfer9415I2c(i2cDeviceAddr, 0x0800, MTPDownloader9415, 0, MTPDownloader9415.Length, ReadWrite.WRITE)) // load provided by IDT array
    return false;

if (!WriteI2cByte(0x400, 0x00)) return false; // initialize buffer
if (!WriteI2cByte(0x3048, 0xD0)) return false; // map RAM address 0x0800 to OTP 0x0000
ignoreNAK = true; // this global variable is used by the I2C driver to block ACK error message
if (!WriteI2cByte(0x3040, 0x80)) return false; // run M0
ignoreNAK = false;
Thread.Sleep(100);

if (!WriteI2cByte(0x402, 0x00)) return false; // write start address
if (!WriteI2cByte(0x404, 0x00)) return false; // write FW length low byte
if (!WriteI2cByte(0x405, 0x60)) return false; // write FW length high byte
if (!WriteI2cByte(0x406, FW CRC)) return false; // write CRC from FW release package
sBuf[0] = 0x11;
if (!Xfer9415I2c(i2cDeviceAddr, 0x400, sBuf, 0, 1, ReadWrite.WRITE))
{
    addText("ERROR: on OTP buffer validation");
    return false;
}
do
{
    Thread.Sleep(20);
    if (!Xfer9415I2c(i2cDeviceAddr, 0x401, sBuf, 0, 1, ReadWrite.READ))
    {
        addText("ERROR: on reading OTP buffer status");
        return false;
    }
} while ((sBuf[0] & 1) != 0);

// check status
if (sBuf[0] != 2) // not OK
{
    string er;
    if (sBuf[0] == 8)
        er = " CHECK SUM ERR";
    else
        er = " UNKNOWN ERR";
    addText("ERROR: CRC returned status " + sBuf[0] + er + " in 128-byte block @ 0x" + i.ToString("X4"));
    return false;
}
// In case of CRC error start the MTP programming and CRC check again for a retry count of 2
addText("OTP Programming finished");
return true;

// power cycle the chip
}
### 6. Revision History

<table>
<thead>
<tr>
<th>Revision Date</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 19, 2019</td>
<td>Updated to corporate compliant template.</td>
</tr>
<tr>
<td>August 15, 2019</td>
<td>Initial Release to customer.</td>
</tr>
<tr>
<td>August 6, 2019</td>
<td>Incorporated review comments.</td>
</tr>
<tr>
<td>July 29, 2019</td>
<td>Initial release of P9415 MTP Programming Guide.</td>
</tr>
</tbody>
</table>