IQRF
DPA Framework

Technical Guide

Version v4.32
IQRF OS v4.06D+/4.06G

5. 11. 2024

Table of Contents

1 Introduction ... 10
2 Basics ... 10
2.1 Device types ... 10
2.2 RF Devices and Networks ... 10
2.2.1 Migration Notes from DPA 3.0x to DPA 4.xx ... 11
2.3 Interfaces ... 11
2.3.1 SPI ... 11
2.3.2 UART ... 11
2.3.3 Peripherals vs. Interfaces ... 12
2.3.3.1 Peripherals ... 12
2.3.3.2 Interface ... 12
2.4 DPA Plug-in filename ... 13
2.5 Message parameters ... 15
2.6 DPA Messages ... 15
2.6.1 DPA Request ... 16
2.6.2 DPA Confirmation ... 16
2.6.3 DPA Notification ... 18
2.6.4 DPA Response ... 19
2.6.5 Examples ... 19
2.7 Device exploration ... 20
2.7.1 Peripheral enumeration ... 20
2.7.1.1 Source code support ... 21
2.7.2 Get peripheral information ... 21
2.7.2.1 Source code support ... 22
2.7.3 Get information for more peripherals ... 22
2.7.3.1 Source code support ... 22
2.8 Memory peripherals ... 23
3 Peripherals ... 23
3.1 Standard operations in general ... 23
3.1.1 Writing to peripheral ... 24
3.1.1.1 Source code support ... 24
3.1.2 Reading from peripheral ... 24
3.1.2.1 Source code support ... 24
3.2 Coordinator ... 24
3.2.1 Peripheral information ... 24
3.2.2 Get addressing information ... 24
3.2.2.1 Source code support ... 25
3.2.3 Get discovered Nodes ... 25
3.2.4 Get bonded Nodes ... 25
3.2.4.1 Source code support ... 25
3.2.5 Clear all bonds ... 25
3.2.6 Bond Node ... 26
3.2.6.1 Source code support ... 26
3.2.7 Remove bonded Node ... 27
3.2.7.1 Source code support ... 27
3.2.8 Discovery ... 27
3.2.8.1 Source code support ... 28
3.2.9 Set DPA Param ... 28
3.2.9.1 Source code support ... 29
3.2.10 Set Hops ... 29
3.2.10.1 Source code support ... 29
3.2.11 Backup ... 29
3.2.11.1 Source code support ... 30
3.2.12 Restore ... 30
3.2.12.1 Source code support ... 31
3.2.13 Authorize bond ... 31
3.2.13.1 Source code support ... 31
3.2.14 Smart Connect ... 32
3.2.14.1 Source code support ... 32

3.2.15 Set MID 33
3.2.15.1 Source code support 33
3.3 Node 33
3.3.1 Peripheral information 34
3.3.2 Read 34
3.3.2.1 Source code support 34
3.3.3 Remove bond 34
3.3.4 Backup 35
3.3.5 Restore 35
3.3.6 Validate bonds 35
3.3.6.1 Source code support 35
3.4 OS 35
3.4.1 Peripheral information 35
3.4.2 Read 36
3.4.2.1 Source code support 36
3.4.3 Reset 37
3.4.4 Restart 37
3.4.5 Read TR Configuration 37
3.4.5.1 Source code support 38
3.4.6 Write TR Configuration 38
3.4.6.1 Source code support 39
3.4.7 Write TR Configuration byte 39
3.4.7.1 Source code support 39
3.4.8 Run RFPGM 40
3.4.9 Sleep 40
3.4.9.1 Source code support 41
3.4.10 Set Security 41
3.4.10.1 Source code support 42
3.4.11 Batch 42
3.4.12 Selective Batch 42
3.4.12.1 Source code support 43
3.4.13 LoadCode 43
3.4.13.1 Source code support 45
3.4.14 Test RF Signal 45
3.4.14.1 Source code support 45
3.4.15 Factory Settings 46
3.4.16 Indicate 46
3.4.16.1 Source code support 46
3.5 EEPROM 47
3.5.1 Peripheral information 47
3.5.2 Read 47
3.5.2.1 Source code support 47
3.5.3 Write 48
3.5.3.1 Source code support 48
3.6 EEEPROM 48
3.6.1 Peripheral information 48
3.6.2 Extended Read 48
3.6.2.1 Source code support 49
3.6.3 Extended Write 49
3.6.3.1 Source code support 49
3.7 RAM 50
3.7.1 Peripheral information 50
3.7.2 Read & Write 50
3.7.2.1 Source code support 50
3.7.3 Read Any 50
3.8 SPI (Slave) 50
3.9 LED 50
3.9.1 Peripheral information 51
3.9.2 Set 51
3.9.3 Pulse 51

3.9.4 Flashing 51
3.10 IO 51
3.10.1 Peripheral information 51
3.10.2 Direction 52
3.10.2.1 Source code support 52
3.10.3 Set 52
3.10.3.1 Source code support 53
3.10.4 Get 54
3.11 Thermometer 54
3.11.1 Peripheral information 54
3.11.2 Read 54
3.11.2.1 Source code support 54
3.12 PWM 55
3.13 UART 55
3.13.1 Peripheral information 55
3.13.2 Open 56
3.13.2.1 Source code support 56
3.13.3 Close 56
3.13.4 Write & Read 57
3.13.4.1 Source code support 57
3.13.5 Clear & Write & Read 58
3.14 FRC 58
3.14.1 Peripheral information 58
3.14.2 Send 58
3.14.2.1 Source code support 59
3.14.3 Extra result 59
3.14.4 Send Selective 59
3.14.4.1 Source code support 60
3.14.5 Set FRC Params 60
3.14.5.1 Source code support 61
3.14.6 Embedded FRC Commands 61
3.14.6.1 Ping 61
3.14.6.2 Acknowledged broadcast - bits 61
3.14.6.3 Prebonded alive 62
3.14.6.4 Supply voltage 62
3.14.6.5 Prebonded memory compare 62
3.14.6.6 Temperature 63
3.14.6.7 Acknowledged broadcast - bytes 63
3.14.6.8 Memory read 63
3.14.6.9 Memory read plus 1 64
3.14.6.10 FRC response time 65
3.14.6.11 Test RF Signal 65
3.14.6.12 Prebonded memory read plus 1 66
3.14.6.13 Memory read 4 bytes 66
4 TR Configuration 67
5 Device Startup 69
5.1 Button Handling and LED Indications 70
5.1.1 RFPGM 70
5.1.2 Node 71
5.1.2.1 Bonded Node 71
5.1.2.2 Unbonded Node 71
5.1.3 Coordinator 72
5.1.4 Custom DPA Handler State 72
6 DPA Menu 73
6.1 Menus 73
6.1.1 DPA Menu “ReadyToBond” 74
6.1.2 DPA Menu “Online” 74
6.1.3 DPA Menu “Beaming” 74
6.1.4 DPA Menu “StandBy” 74
6.2 DPA Menu Content 74

6.2.1 Bond Request ... 75
6.2.2 Beaming ... 75
6.2.3 Connectivity Check ... 75
6.2.4 Exit Standby ... 75
6.2.5 State Indication ... 75
6.2.6 User1 and User2 ... 75
6.2.7 Standby ... 75
6.2.8 Reset ... 75
6.2.9 Restart ... 75
6.2.10 Unbond + Restart ... 75
6.2.11 Factory Settings + Restart ... 75
6.2.12 Unbond + Factory Settings + Restart ... 75
7 Autoexec ... 76
8 IO Setup ... 76
9 Custom DPA Handler ... 77
9.1 Handler Example ... 78
9.2 Events Flow ... 79
9.2.1 Coordinator ... 79
9.2.2 Node ... 80
9.2.3 General events ... 81
9.2.3.1 Interrupt ... 81
9.2.3.2 Disable Interrupts ... 81
9.2.3.3 Sleep Events ... 81
9.2.3.4 Menu Events ... 81
9.2.3.5 InStandby event ... 81
9.3 Events ... 81
9.3.1 Interrupt ... 81
9.3.2 Idle ... 82
9.3.3 Init ... 83
9.3.4 Notification ... 83
9.3.5 AfterRouting ... 84
9.3.6 BeforeSleep ... 84
9.3.7 AfterSleep ... 84
9.3.8 Reset ... 85
9.3.9 Disable Interrupts ... 85
9.3.10 FrcValue ... 86
9.3.11 FrcResponseTime ... 87
9.3.12 ReceiveDpaResponse ... 87
9.3.13 IFaceReceive ... 87
9.3.14 ReceiveDpaRequest ... 88
9.3.15 BeforeSendingDpaResponse ... 89
9.3.16 PeerToPeer ... 89
9.3.17 UserDpaValue ... 90
9.3.18 BondingButton ... 90
9.3.19 Indicate ... 90
9.3.20 VerifyLocalFrc ... 91
9.3.21 MenuActivated ... 91
9.3.22 MenuItemSelected ... 91
9.3.23 MenuItemFinalize ... 92
9.3.24 InStandby ... 92
9.3.25 DPA Request ... 93
9.3.25.1 Enumerate Peripherals ... 93
9.3.25.2 Get Peripheral Info ... 94
9.3.25.3 Handle Peripheral Request ... 94
9.3.25.4 Alternative Event Processing ... 95
9.4 DPA API ... 95
9.4.1 DpaApiRfTxDpaPacket ... 95
9.4.2 DpaApiReadConfigByte ... 96
9.4.3 DpaApiSendToIFaceMaster ... 97
9.4.4 DpaApiRfTxDpaPacketCoordinator ... 97

9.4.5 DpaApiLocalRequest 98
9.4.6 DpaApiReturnPeripheralError 98
9.4.7 DpaApiSetRfDefaults 99
9.4.8 DpaApiLocalFrc 99
9.4.9 DpaApiCrc8 99
9.4.1 DpaApiAggregateFrc 99
9.4.2 DpaApiSetOTK 100
9.4.3 DpaApiSleep 100
9.4.4 DpaApiDeepSleep 100
9.4.5 DpaApiAfterSleep 101
9.4.6 DpaApiI2Cinit 101
9.4.7 DpaApiI2Cstart 101
9.4.8 DpaApiI2Cwrite 101
9.4.9 DpaApiI2Cread 101
9.4.10 DpaApiI2Cstop 101
9.4.11 DpaApiI2CwaitForACK 101
9.4.12 DpaApiI2Cshutdown 102
9.4.13 DpaApiI2CwaitForIdle 102
9.4.14 DpaApiRandom 102
9.4.15 DpaApiMenu 102
9.4.16 DpaApiMenuIndicateResult 103
9.4.17 DpaApiMenuExecute 103
9.5 DPA API Variables 104
9.5.1 bit IFaceMasterNotConnected 104
9.5.2 bit NodeWasBonded 104
9.5.3 bit EnableIFaceNotificationOnRead 104
9.5.4 uns16 DpaTicks 104
9.5.5 uns8 LPtoutRF 104
9.5.6 uns8 ResetType 104
9.5.7 bit DSMactivated 105
9.5.8 uns8 UserDpaValue 105
9.5.9 uns8 NetDepth 105
9.5.10 bit LpRxPinTerminate 105
9.5.11 uns8 RxFilter 105
9.5.12 uns16 BondingSleepCountdown 105
9.5.13 uns16 Random 106
9.5.14 bit AsyncReqAtCoordinator 106
9.5.15 bit NonroutedRfTxDpaPacket 106
9.5.16 uns8 DpaValue 106
9.5.17 uns8 I2Ctimeout 106
9.5.18 bit I2CwasTimeout 106
9.5.19 bit FirstDpaApiSleep 106
9.6 Examples 106
9.6.1 Bonding 107
9.6.2 Coordinator-FRCandSleep 108
9.6.3 FRC-Minimalistic 108
9.6.4 LED-MemoryMapping 108
9.6.5 PeripheralMemoryMapping 109
9.6.6 UserPeripheral-18B20 109
9.6.7 UserPeripheral-18B20-Idle 109
9.6.8 UserPeripheral-ADC 109
9.6.9 UserPeripheral-HW-UART 109
9.6.10 UserPeripheral-i2c 109
9.6.11 UserPeripheral-PWM 110
9.6.12 UserPeripheral-SPImaster 110
9.7 Migration Notes to DPA 3.03 110
10 DPA Peer-to-Peer 111
10.1 DP2P Request 111
10.2 DP2P Response Handshake 112
10.2.1 DP2P Invite 113

10.2.2 DP2P Confirm ... 113
10.2.3 DP2P Response ... 113
11 DPA in Practice ... 114
11.1 Network Deployment ... 114
11.2 Over The Air (OTA) upgrade of IQRF OS and DPA ... 114
11.3 Code Upload ... 116
11.3.1 Storing Code at External EEPROM ... 116
11.3.2 Executing Code Upload ... 116
11.3.3 Executing IQRF OS Change ... 116
12 Constants ... 118
12.1 Peripheral Numbers ... 118
12.2 Response Codes ... 118
12.3 DPA Commands ... 118
12.4 Peripheral Types ... 119
12.5 Custom DPA Handler Events ... 120
12.6 Extended Peripheral Characteristic ... 120
12.7 HW Profile IDs ... 120
12.8 Baud rates ... 120
12.9 User FRC Codes ... 121
13 Appendix ... 122
13.1 CRC Calculation ... 122
13.1.1 CC5X Compiler ... 122
13.1.2 C# ... 122
13.1.3 Java ... 122
13.1.4 Pascal/Delphi ... 123
13.2 One's Complement Fletcher-16 Checksum Calculation ... 123
13.2.1 CC5X Compiler ... 123
13.2.2 C# ... 123
13.2.3 Python ... 124
13.3 Custom DPA Handler Code at .hex File ... 124
13.4 IQRF OS Change ... 125
13.4.1 IQRF OS Change File ... 126
13.5 Code Optimization ... 128
13.5.1 W as a temporary variable ... 128
13.5.2 Variable access reorder ... 128
13.5.3 Variable access decomposition ... 128
13.5.4 Explicit MOVLB omitting ... 128
13.5.5 Direct function parameter usage ... 128
13.5.6 Avoiding else ... 129
13.5.7 Switch instead of if ... 129
13.5.8 Function call before return ... 129
13.5.9 Using goto to avoid redundant code ... 130
13.5.10 Avoiding readFromRAM and getINDFx ... 130
13.5.11 Advanced C-compiler optimized instructions ... 130
13.5.12 do {} while () is preferred ... 130
13.5.13 Use DECFSZ/INCFSZ ... 131
13.5.14 Widening function parameter ... 131
13.5.15 Carry as a variable ... 131
13.5.16 Limiting variable scope ... 132
13.5.17 Using IQRF variables ... 132
13.5.18 Parameter mapped to W ... 132
13.5.19 Pointer parameters mapped to FSRx ... 133
13.5.20 FSRx as a 16-bit variable ... 133
13.5.21 Using FSRx to copy between buffers and variables ... 133
13.5.22 Accessing 16-bit MCU registers ... 134
13.5.23 Using IQRF OS offset and limit variables ... 134
13.5.24 Effective is not always efficient ... 134
13.5.25 The assignment also has a value ... 134
13.5.26 Interval detection optimization ... 134
13.5.27 Optimized constants ... 134

13.5.28 Equality result ... 135
13.5.29 One instruction at the if branch ... 135
13.5.30 Utilization of the preloaded W ... 135
13.5.31 == 1 is more efficient than != 1 ... 136
13.5.32 == 0xFF is more efficient than != 0xFF ... 136
13.5.33 Expression modification ... 136
13.5.34 Computed goto with a table limit ... 136
13.5.35 Default is first at switch ... 136
13.5.36 Better to return from than after the loop ... 137
13.5.37 Modification instead of storing the value ... 137
13.5.38 Assignment compares to 0 ... 138
13.5.39 End condition of the 16-bit loop variable ... 138
13.5.40 Shift for a smart comparison ... 138
13.5.41 Optimized return TRUE/FALSE ... 138
13.5.42 Avoiding MOVLP #1 ... 139
13.5.43 Avoiding MOVLP #2 ... 139
13.5.44 Setting zeroed variables ... 139
13.5.45 Compare to zero is more efficient ... 139
13.5.46 setFSR01 ... 140
13.5.47 Pointer arithmetic ... 140
13.5.48 Circular buffer index increment ... 140
14 DPA Release Notes ... 141
14.1 DPA 4.32 ... 141
14.2 DPA 4.31 ... 141
14.3 DPA 4.30 ... 141
14.4 DPA 4.18 ... 141
14.5 DPA 4.17 ... 142
14.6 DPA 4.16 ... 142
14.7 DPA 4.15 ... 143
14.8 DPA 4.14 ... 143
14.9 DPA 4.13 ... 143
14.10 DPA 4.12 ... 144
14.11 DPA 4.11 ... 144
14.12 DPA 4.10 ... 144
14.13 DPA 4.03 ... 144
14.14 DPA 4.02 ... 145
14.15 DPA 4.01 ... 145
14.16 DPA 4.00 ... 145
14.17 DPA 3.04 ... 146
14.18 DPA 3.03 ... 146
14.19 DPA 3.02 ... 147
14.20 DPA 3.01 ... 147
14.21 DPA 3.00 ... 148
14.22 DPA 2.28 ... 148
14.23 DPA 2.27 ... 149
14.24 DPA 2.26 ... 149
14.25 DPA 2.24 ... 149
14.26 DPA 2.23 ... 150
14.27 DPA 2.22 ... 150
14.28 DPA 2.21 ... 150
14.29 DPA 2.20 ... 150
14.30 DPA 2.13 ... 151
14.31 DPA 2.12 ... 151
14.32 DPA 2.11 ... 151
14.33 DPA 2.10 ... 151
14.34 DPA 2.01 ... 152
14.35 DPA 2.00 ... 152
15 Document Revisions ... 154
16 Acknowledgement ... 154
17 Sales and Service ... 155

1 Introduction

Direct Peripheral Access (DPA) protocol is a simple byte-oriented protocol used to control services and peripherals of IQMESH network devices (Coordinator and Nodes) by SPI or UART interfaces. DPA protocol implementation is distributed in the form of the IQRF plug-in.

2 Basics

DPA protocol uses byte structured messages to communicate at the IQMESH network. Every message always contains four mandatory parameters NADR, PNUM, PCMD, and HWPID (foursome from now). The message can optionally hold data (an array of bytes often referred to as PData throughout the document) to be transmitted or received. They are always described next to the foursome throughout this document. Although foursome parameters are typically described next to each other in this document, they do not have to be stored at consecutive memory addresses in the real scenario. The same rule does not apply to the message data.

Please note that a Response, Confirmation, and Notification (with a small exception) DPA messages always contain the same NADR, PNUM, and PCMD as the original DPA Request message except the response message is flagged by the most significant bit of PCMD.

All values wider than one byte are encoded using little-endian style unless otherwise specified.

Symbols, variables, structures, methods, etc. mentioned in this document are defined in header files `DPA.h` and `DPAcustomHandler.h`. Please consult IQRF OS documentation whenever an *IQRF OS function* is referenced in this document.

2.1 Device types

Two device types are depending on what type of network device it implements. For each device type, there are dedicated DPA plug-ins to upload.

[C] IQMESH Coordinator device
[N] IQMESH Node device ([Ns] stands for Nodes)

2.2 RF Devices and Networks

There are two types of network devices from RF and the power consumption point of view. STD devices support both STD+LP and STD networks (see below). LP devices support only STD+LP networks. LP devices, unlike STD devices, receive packets strictly at LP-RX mode, so they have considerably lower power consumption compared to STD devices and therefore they can be powered from batteries, accumulators, etc.

There are two IQMESH network types: either STD+LP or STD.

If the network is type STD, then the packets are transmitted in STD-TX mode and they can only be received in STD-RX mode. It follows that such a network consists typically only of mains-powered devices as the STD-RX mode would drain batteries or accumulators fast.

If the network is type STD+LP, then the packets are transmitted in LP-TX mode. STD+LP networks are approximately twice as slow compared to the STD networks. Battery-powered devices (LP [N]) receive the LP packets in LP-RX mode (that puts the device regularly into sleep mode for most of the time) to minimize their energy consumption. By contrast, the mains-powered devices (STD [N]) keep receiving the LP packets in STD-RX mode thus they can unlike battery-powered devices take advantage of using interrupts and peripherals.

Mains-powered devices (STD [N]), unlike battery-powered devices (LP [N]), can work both in STD and STD+LP networks.

The IQMESH network type (STD+LP or STD), that [C] controls, is configured by a TR Configuration bit.7 at byte index 0x05 (*NtwType* from now) of the respective device.

The following table depicts network RF modes and the respective [C] and [N] *RF settings* that are automatically set by DPA:

Network	Coordinator		Node	
			STD	LP
STD	*NtwType* = 0	*STD-TX / STD-RX*	*STD-TX / STD-RX*	*n/a*
STD+LP	*NtwType* = 1	*LP-TX / STD-RX*	*LP-TX / STD-RX*	*LP-TX / LP-RX*

2.2.1 Migration Notes from DPA 3.0x to DPA 4.xx

It is important to prepare existing devices before the migration to the DPA 4.xx from DPA 3.0x.

- The [C] that will control the STD+LP network must have *NtwType* correctly bit set before the DPA is migrated if the OTA is not used.
- Some mains-powered devices running with a former DPA version at LP network had to have a special Custom DPA handler just for LP networks in the past. With DPA 4.xx they can have a former STD Custom DPA Handler that will be compatible with both network types. Please consult your device manufacturer with a proper Custom DPA handler to upload.
- Already bonded STD [N] that can run both in STD+LP and STD networks must be upgraded to the DPA 4.xx using OTA only. OTA process ensures that the STD [N] will be properly set up for the future STD+LP or STD network. If the DPA is updated by uploading the DPA plug-in, then it is not possible to find out whether the network the STD [N] is bonded to is STD+LP or STD (the network RF mode can be found out only during bonding or OTA) thus the STD [N] will not work at the new DPA 4.xx network.

2.3 Interfaces

The interface type is chosen by uploading the proper PDA plug-in. The interface transfers the DPA message to/from the connected device. The message consists of the successively stored foursome and optional data. When an interface is supported then UART embedded peripheral is not implemented. The interface is not available for LP [N] as it regularly sleeps.

2.3.1 SPI

The SPI interface is implemented using the IQRF SPI protocol described in the document "SPI Implementation in IQRF TR modules". The document specifies how to set up the SPI master and the communication over the SPI. The device always plays the role of SPI slave and the externally connected device is the SPI master. The DPA protocol corresponds to the DM and DS bytes of the IQRF SPI protocol.

2.3.2 UART

UART is configured with 8 data bits, 1 stop bit, and no parity bit. UART baud rate is specified at TR Configuration. The size of both RX and TX buffers is 64 bytes.

HDLC byte stuffing protocol is used to frame, protect, and encode DPA messages. Every data frame (DPA message) starts and ends with byte 0x7e (Flag Sequence). When the actual data byte (applies to 8-bit CRC value too) equals 0x7e (Flag Sequence) or 0x7d (Control Escape) then it is replaced by two bytes: the 1st byte is 0x7d (Control Escape) and the 2nd byte equals the original byte value XORed by 0x20 (Escape Bit).

An 8-bit CRC is used to protect data. The CRC value is appended after all data bytes and it is coded by the same HDLC byte stuffing algorithm. CRC is compatible with 1-Wire CRC with an initial value of 0xFF, the polynomial is $x^8+x^5+x^4+1$. See CRC Calculation for the implementations of the CRC algorithm. There is also an online calculator available.

Please note that the UART interface is not able to inform that the device is ready to receive the next DPA request. It is recommended to implement timing accurately. If the duration of the request cannot

be estimated (e.g. Discovery), a simple DPA request Get addressing information can be sent periodically to determine if the current request is still being executed.

At TR-7xG transceivers the TX and/or RX signals can be remapped.

Example

The example shows encoded DPA Request “write bytes 0x7E, 0x7D at the RAM address 0 at [N] with address 0x2F”:

`NADR=0x002F`(Node address)`, PNUM=0x05`(RAM peripheral)`, PCMD=0x01`(RAM write)`, HWPID=0xFFFF,`
`PData={0x00}`(address)`, {0x7E,0x7D}`(bytes to write)

CRC from bytes {0x2f, 0x00, 0x05, 0x01, 0xff, 0xff, 0x00, 0x7e, 0x7d} = 0x7e

Data in index		0	1	2	3	4	5	6	7		8		CRC		
Data in		0x2f	0x00	0x05	0x01	0xff	0xff	0x00	0x7e		0x7d		0x7e		
Data out index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Data out	0x7e	0x2f	0x00	0x05	0x01	0xff	0xff	0x00	0x7d	0x5e	0x7d	0x5d	0x7d	0x5e	0x7e
Note	*Flag Sequence*	original byte	original byte	original byte	original byte	original byte	original byte	original byte	*Control Escape*	0x7e XOR 0x20	*Control Escape*	0x7d XOR 0x20	*Control Escape*	0x7e XOR 0x20	*Flag Sequence*

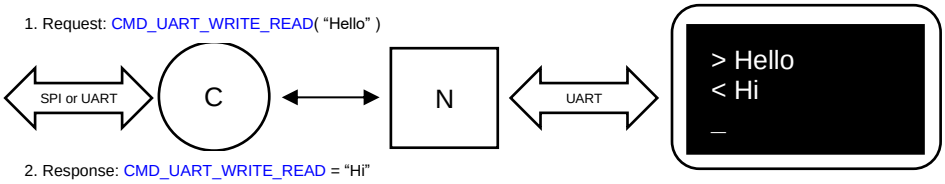
2.3.3 Peripherals vs. Interfaces

UART peripheral differs from the UART interface:

- The peripheral is just a byte-oriented data channel used to exchange data between the network and external devices.
- The interface is used to control network devices from an external device using DPA messages.
  In the case of the SPI interface, the external device must be an SPI master as the DPA network device is always an SPI slave.

2.3.3.1 Peripherals

UART peripheral is typically used to control an external device connected to the [N] device via the HW UART interface. The following picture shows an example where the [C] writes by UART Write & Read DPA Request a text “Hello” to the UART peripheral at [N]. There is a terminal (external device) connected using UART to the [N]. Text “Hello” is then displayed at the terminal and text “Hi” (in this example the terminal automatically answers “Hi” to “Hello”) is read back to the [C] at the corresponding DPA Request.



2.3.3.2 Interface

The interface connects any ([C] or [N]) network device to the external autonomous device and allows the external device to control the network and/or network device. By default the interface is always

enabled at [C] device because it gives an external device means to control the [C] as well as the rest of the network. The interface at [N] device is enabled if the appropriate DPA plug-in is uploaded. See DPA Messages for details of the messages exchanged over the interface. Next table shows some differences in the interface behavior at different network devices:

Topic	*Device*	
	[C]	*[N]*
DPA Messages	DPA Request (in) DPA Confirmation (out) DPA Response (out)	DPA Request (in) DPA Response (out) DPA Notification (out)
NADR at DPA Request	See NADR at General message parameters. Invalid value generates an ERROR_NADR error code. Both values 0x0000 and 0x00FC address the [C] device itself.	Only value 0x00FC is allowed and it addresses the [N] device itself. Other values are silently ignored. There is no way to directly control [C] device coupled to [N] by its interface.

See Examples of interface usage.

2.4 DPA Plug-in filename

DPA protocol implementation is distributed in the form of the IQRF plug-in. The plug-in filename has the following format:

DPA-**[device]**-**[rfmode]**-**[interface]**-**[tr]**-**[version]**-**[date]**.iqrf

Item	Value	Description
[device]	Coordinator	Coordinator device [C]
	Node	Node device [N]
[rfmode]	STD	[N] works in STD-RX mode.
	LP	[N] works in LP-RX mode.
	<empty>	[C] controls both STD+LP and STD networks.
[interface]	SPI	SPI interface
	UART	UART interface
	<empty>	No interface supported by [N]
[tr]	7xD 7xG	For TRs of 7xD series For TRs of 7xG series
[version]	V*abc*	DPA version *a.bc* (e.g. *V416* stands for version 4.16)
[date]	*yymmdd*	Release date (e.g. *210818* stands for August 18th, 2021)

The following table depicts all available DPA plug-ins with supported features:

DPA Plug-in	**Feature**									
	Device		*Network*		*Interface*		*Implemented Peripherals* (1)			
	Coordinator	Node	STD+LP	STD	SPI	UART	Coordinator	Node	UART	FRC
DPA-Coordinator-SPI-7xy-V[version]-[date].iqrf	✓		✓	✓	✓		✓ (2)			✓ (2)
DPA-Coordinator-UART-7xy-V[version]-[date].iqrf	✓		✓	✓		✓	✓ (2)			✓ (2)
DPA-Node-STD-7xy-V[version]-[date].iqrf		✓	✓	✓				✓ (2)	✓	
DPA-Node-LP-7xy-V[version]-[date].iqrf		✓	✓					✓ (2)	✓	
DPA-Node-STD-UART-7xy-V[version]-[date].iqrf		✓	✓	✓		✓		✓ (2)		

(1) All other embedded peripherals are always implemented.
(2) The peripheral is enabled regardless of the configuration settings.

2.5 Message parameters

All numbers are in the hexadecimal format unless otherwise noted.

Parameter	Value [hex]		Description
NADR [2B]	00	IQMESH Coordinator	Network device address. Although it is 2 bytes wide, the 2B addressing is not supported (a higher byte is ignored).
	01-EF	IQMESH Node address	
	F0-FB	Reserved	
	FC	Local (over interface) device	
	FD	Reserved	
	FE	IQMESH temporary address	
	FF	IQMESH broadcast address	
	100-FFFF	Reserved	
PNUM [1B]	00	COORDINATOR	Peripheral number (0x00–0x1F reserved for embedded peripherals) (0x40–0x7F reserved for IQRF standard peripherals)
	01	NODE	
	02	OS	
	03	EEPROM	
	04	EEEPROM	
	05	RAM	
	06	LEDR	
	07	LEDG	
	08	Reserved	
	09	IO	
	0A	Thermometer	
	0B	Reserved	
	0C	UART	
	0D	FRC	
	0E-1F	Reserved	
	20-3E	User peripherals	
	3F	Not available	
	40-7F	Reserved	
	80-FD	Not available	
	FE	PNUM_ERROR_FLAG	
	FF	Not available	
PCMD [1B]	0-3E	Command value	Command specifying an action to be taken. The allowed value range depends on the peripheral type. The most significant bit is reserved for the indication of the DPA response message.
	3F	Not available	
	40-7F	Command value	
	80-FF	Not available	
HWPID [2B]	0000	Default HW Profile	HW profile ID (HWPID from now) uniquely specifies (the functionality of) the device, the user peripherals it implements, its behavior, etc. The only device having the same HWPID as the DPA Request will execute the request. When 0xFFFF is specified then the device with any HW profile ID will execute the request. Note - HWPID numbers used throughout this document are fictitious ones.
	0001-xxxE	Certified HW Profiles	
	C05E	OTA Handler	
	xxxF	User HW Profiles	
	FFFF	Reserved	
PData [0-56B]	An array of bytes. The maximum length is limited to 56 bytes (decimal).		Optional message data.

2.6 DPA Messages

DPA protocol (messages) is transferred over an interface that connects TR module (“slave”) to a superordinate system (”master”).

- Master sends DPA Request.
- If addressee (NADR) is a (remote) IQMESH [N], not a local over the interface connected device (applies only to [C]), then:

    - The device immediately sends DPA Confirmation back to the interface master.
    - [N] processes the DPA message.
- If the DPA message does not have a read-only (can be configured by EnableIFaceNotificationOnRead) side-effect and the interface is enabled for the DPA communication at the [N] side, then the [N] sends DPA Notification to its interface.
    - If the DPA message was not sent using the broadcast address.
        - [N] returns DPA response to [C] via RF.
        - [C] receives the DPA response and re-sends it to the interface master.
- In the case of a local device
    - The device processes the DPA Request. In this case, both the sender and addressee address values of the request equal 0xFC (local address).
    - The device returns the DPA response to the interface master.

2.6.1 DPA Request

DPA Request consists of a foursome with optional data, depending on the actual request. DPA Request is executed only if the specified HW profile ID matches the HW profile ID of the device unless the HW profile ID in the foursome equals 0xFFFF (*HWPID_DoNotCheck*). In some scenarios, the request can be asynchronously sent from [N] to [C]. Then it is marked as asynchronous the same way as asynchronous DPA Response.

2.6.2 DPA Confirmation

DPA Confirmation confirms the reception of DPA Request by interface slave to interface master at the [C]. It consists of the same foursome that was part of the original DPA Request plus the following 5 additional data bytes. The Confirmation is not returned if the Request is incorrect (e.g. if the request NADR is not valid). In this case, a Response with an error code is returned.

The format of the Confirmation data bytes is the following

0	1	2	3	4
STATUS_CONFIRMATION	DPA Value	Hops	Timeslot length in 10 ms units	Hops Response

DPA Value — DPA value of the device.

Hops — Number of hops used to deliver the DPA Request to the addressed [N]. A hop represents any sending of a packet including sending from the sender as well as from any routing [N].

Timeslot length — Timeslot length used to deliver the DPA Request to the addressed [N]. Please note that the timeslot used to deliver the response message from [N] to [C] can have a different length.

Hops Response — Number of hops used to deliver the DPA Request from the addressed [N] back to the [C]. In the case of broadcast, this parameter is 0 as there is no response sent back to the [C].

IQMESH timeslot length depends on the PData length of the DPA messages (the values may change in the future depending on the version of the DPA protocol and IQRF OS version) and the current network type (STD+LP, STD).

PData length [bytes]	**Timeslot length** [ms]	
	STD	**STD+LP**
< 17	40	80
17 - 40	50	90
> 40	60	100

This information can be used to implement a precise timing of the control system (master) connected to the [C] device by the interface to prevent data collision (e.g. when another DPA Request is sent to the network before routing of the previous communication is finished) at the network.

1. Wait till the previous IQMESH routing is finished (see step 7).

2. Make sure the interface is ready (e.g. SPI status is *ReadyCommunication*) and no data remained for reading from the interface.
3. Send DPA Request via the interface.
4. Receive DPA Confirmation via the interface. Remember the time when the DPA Confirmation was received (to be used later in step 7).
5. Now, wait (*Hops* + *1*) × *Timeslot length* × *10 ms* till the DPA Request routing is finished. Note: if it takes some extra time to prepare and send the response back at the [N] side then, this time, it must be considered (added) to the total routing time.
6. Read DPA Response from the interface within the time (*Hops Response* + *1*) × *Estimated response timeslot length* × *10 ms* + *Safety timeout*. Estimated response timeslot length is the value based on the expected length of data returned within the DPA Request or it can be the worst case (e.g. 6 = 60 ms at STD mode).
7. Find out the Actual response timeslot length from the PData length of the actual DPA Response. Now the earliest time to send something to the IQMESH network equals *Time the DPA Confirmation was received* + *(Hops + 1)* × *Timeslot length* × *10 ms* + *(Hops Response + 1)* × *Actual response timeslot length* × *10 ms*. This time is used for waiting at step 1.

Using this technique ensures reliable and optimal speed data delivery at the IQMESH network. Pay attention to the DPA Requests that produce an intentional delay at the addressed device side (e.g. UART Write& Read, IO Set, OS Sleep, OS Reset, LoadCode, Run RFPGM). Such delay (time) must be added to the total response time. Also, the response time for Discovery and Bond Node requests is not predictable at all.

Please note that the OS Read command returns the shortest and the longest timeslot length. Please also see IQMESH Timing Calculator.

Example
The next figure shows processing UART Write & Read request. The DPA Request is marked Request 1. It writes 5 bytes of data to [N_n] UART peripheral, it waits 20 ms, and then it reads a number (unknown in advance) of bytes back from the UART peripheral. The network is operated at STD mode.

After sending **Request 1** to the [C] the [C] replies by **Confirmation 1**. The DPA Confirmation reports *q* hops to deliver a DPA Request from [C] to [N_n] with a timeslot of 40 ms and also *r* hops to deliver the response back from [N_n] to [C]. After the DPA Confirmation is sent the [C] transmits the RF packet to the network (1st hop). The packet is received by [N_1] and [N_1] routes the packet further (2nd hop). The routed packet is received by [N_2] as expected. The routing continues. Last but one [N_{n-1}] receives the routed packet and because of positive RF conditions and network topology the routed packet is also early received by the addressed [N_n]. Then [N_{n-1}] makes the very last routing but [N_n] does not receive the packet again.

Then DPA writes 5 bytes of data to the UART, waits another 20 ms, and reads data from UART. In our example total of 20 bytes is read which results in the real timeslot of 50 ms to be used to deliver the response back from [N3] to [C].

Then [N_n] waits for the still running routing to finish. After that [N_n] transmits the response packet to the network (1st hop). The packet is received by [N_{n-1}] which routes the packet further (2nd hop). The routing continues. The routed packet is received by [N_2]. [N_2] routes the packet to [N_1]. The packet is also received by [C]. [C] immediately delivers **Response 1** to its interface. At the same time [N_1] finally routes the packet to the [C] which receives it but identifies it as the already received response thus [C] does not report it to the interface again.

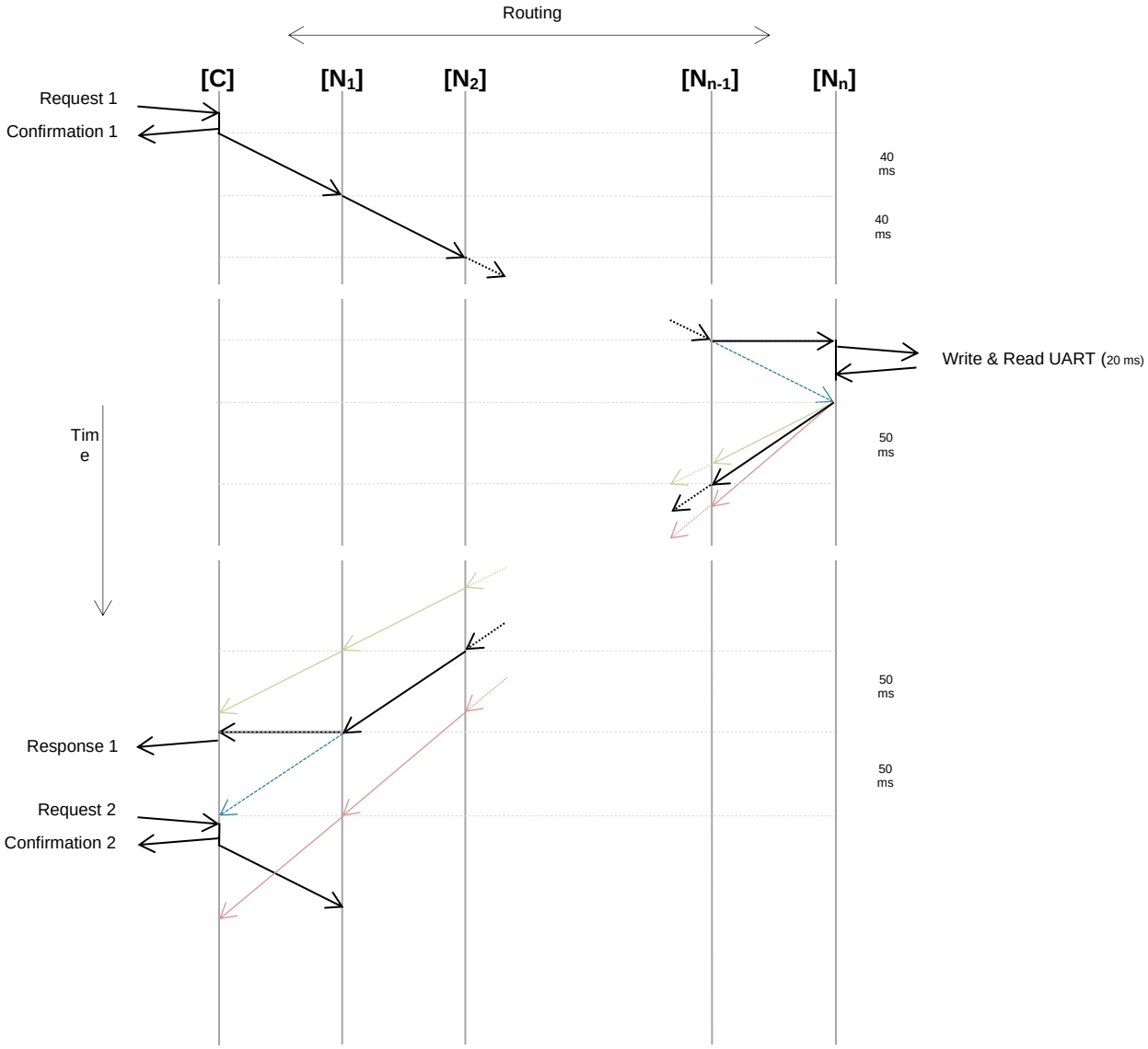
The optimistic response time is:
((*q* + 1) × 40 ms) + 20 ms + ((*r* + 1) × 40 ms)

The pessimistic response time is:
((*q* + 1) × 40 ms) + 20 ms + ((*r* + 1) × 60 ms)

But the real response time was:
((*q* + 1) × 40 ms) + 20 ms + ((*r* + 1) × 50 ms)

An optimistic response routing scenario is represented by dotted green arrows (potential 40 ms timeslot) and a pessimistic scenario is shown by dotted red arrows (potential 60 ms timeslot).

The next **Request 2** cannot be sent to the network immediately after **Response 1** is received. The RF collision would occur. **Request 2** can be issued after the actual routing finishes (end of the dotted blue arrow) the soonest. Another approach is to send the next DPA Request to the [C] after the pessimistic (using the longest 60 ms response timeslot) is finished. For many applications that do not have to be time optimized this is the reasonable and easy to compute way of timing.



Throughout the document in the following examples of the DPA communication, the DPA Confirmation is not usually stated as the emphasis is put on DPA Request-Response pair messages.

2.6.3 DPA Notification

DPA Notification notifies a connected master device at the [N] side that there was a DPA Request without a “read-only” (can be configured by *EnableIFacenotificationOnRead*) side-effect processed by the [N]. It consists of the same foursome that was part of the original DPA Request except for NADR that stores the address of the sender, not the addressee, and the HWPID that contains the actual HW Profile ID of the device. DPA Notification is therefore always 6 bytes long. DPA Request is considered “read-only” when the corresponding DPA Request returns some data, otherwise, it is considered a “write” request.

DPA Notification is issued to the connected master interface when DPA Request is sent from the [C] or when the DPA Request is part of the FRC acknowledged broadcast (see Acknowledged broadcast - bits and Acknowledged broadcast - bytes).

DPA Notification is not issued in the case of DPA Request invoked from a local interface, from DpaApiLocalRequest, or predefined FRCs Memory read and Memory read plus 1.

2.6.4 DPA Response

DPA Request is an actual answer to the DPA Request. DPA Request consists of the same foursome that was part of the original DPA Request except the response message is flagged by the most significant bit of PCMD and HWPID contains the actual HW profile ID of the addressed device. Then come 2 bytes containing the Response code and DPA Value. In the case of error (response code is NOT equal *STATUS_NO_ERROR*), no additional data is present. In the case of the *STATUS_NO_ERROR* response code, the presence of the additional data depends on the DPA Request type. If the response is asynchronous, i.e. it is not a response to the previously sent request, then the response code is marked by the highest bit set (`STATUS_ASYNC_RESPONSE`).

When composing DPA Request in the Custom DPA Handler there is sometimes a need to signalize an error response with a certain Response Code. The way how to return such a response is described in the chapter Handle Peripheral Request.

2.6.5 Examples

Note**:** DPA Value, HWPID, and data read from the memory shown in the following examples may differ in the real scenario.

Example 1

Switching on a red LED at Coordinator:

- **DPA Request** (master → slave)

```
NADR=0x0000, PNUM=0x06, PCMD=0x01, HWPID=0xFFFF
```

- **DPA Response** (slave → master)

```
NADR=0x0000, PNUM=0x06, PCMD=0x81, HWPID=0xABCD, PData={0x00}(No error), {0x07}(DPA Value)
```

Notes:

- NADR 0x0000 Specifies [C] address (0x00FC can be used too)
- PNUM 0x06 Specifies red LED peripheral
- PCMD 0x01 Set LED On command
- DPA Value Coordinator's DPA value

Example 2

Reading 2 bytes from RAM at address 1 of the local [N]:

- **DPA Request** (master → slave)

```
NADR=0x00FC, PNUM=0x05, PCMD=0x00, HWPID=0xFFFF, PData={0x01}(Address), {0x02}(Length)
```

- **DPA Response** (slave → master)

```
NADR=0x00FC, PNUM=0x05, PCMD=0x80, HWPID=0xABCD
PData={0x00}(No error), {0x07}(DPA Value), {0xAB,0xCD}(Read data)
```

Notes:

- NADR 0x00FC Specifies local device address
- PNUM 0x05 Specifies RAM peripheral
- PCMD 0x00 Read command
- DPA Value Local Node's value

Example 3

Switching on a green LED at a remote [N] with address 0x0A:

- **DPA Request** (master → slave)

`NADR=0x000A, PNUM=0x07, PCMD=0x01, HWPID=0xFFFF`

- **DPA Confirmation** (slave → master)

`NADR=0x000A, PNUM=0x07, PCMD=0x01, HWPID=0xFFFF, PData={0xFF}`(Confirmation), `{0x07}`(DPA Value), `{0x06,0x04,0x06}`(Hops, Timeslot length, Hops response)

- **DPA Notification** (slave → master) at remote [N] side

`NADR=0x0000, PNUM=0x07, PCMD=0x01, HWPID=0xABCD`

- **DPA Response** (slave → master)

`NADR=0x000A, PNUM=0x07, PCMD=0x81, HWPID=0xABCD, PData={0x00}`(No error), `{0x06}`(DPA Value)

Notes:

- PNUM 0x07 Specifies the green LED peripheral.
- NADR 0x0000 At the notification specifies that the [C] sent the original DPA Request.
- DPA Value DPA Confirmation: Coordinator's value
  DPA Request: remote Node's value

2.7 Device exploration

Device exploration is used to obtain information about individual devices and their implemented peripherals.

2.7.1 Peripheral enumeration

Request

NADR	PNUM	PCMD	HWPID
NADR	0xFF	0x3F	?

The HWPID value is ignored at this command.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0…1	2	3…6	7…8	9…10	11	(12…23)
NADR	0xFF	0xBF	?	0	?	DpaVer	UserPerNr	EmbeddedPers	HWPID	HWPIDver	Flags	UserPer

DpaVer DPA protocol version
- 1st byte: bits 0-6 = minor version
- 2nd byte: major version

BCD coding is used, e.g. version 12.34 is coded as 0x1234, i.e. 1st byte 0x34, 2nd byte 0x12

UserPerNr Number of all non-embedded peripherals implemented by Custom DPA Handler. Implemented peripherals are flagged at the UserPer variable-size bitmap array. At the Enumerate Peripherals event, the field is prefilled by 0x00.

EmbeddedPers Bits array (starting from LSb of the 1st byte) specifying which of 32 embedded peripherals are enabled in the TR Configuration (it is a copy of the first 4 bytes of the configuration area). If a peripheral is enabled in the configuration although it is not supported by the device, then calling Get peripheral information or Get information for more peripherals will return *PERIPHERAL_TYPE_DUMMY* peripheral type for this peripheral thus indicating that the peripheral is not available.
Bit values for Coordinator (bit 0) and Node (bit 1) peripherals are set according to the device support of these peripherals regardless of actual bit values stored at TR Configuration. The bit for OS (bit 2) is always set. The bit for FRC (bit 5 at byte index 4) is always set at [C] device.

HWPID Hardware profile ID, 0x0000 if default. At the Enumerate Peripherals event, the field is prefilled by 0x0000.

HWPIDver Hardware profile version, 1st byte = minor version, 2nd byte = major version. At the Enumerate Peripherals event the field is prefilled by 0x0000.

Flags Various flags:
- bit 0 Device works in STD-RX mode.

- bit 1 Device works in LP-RX mode.
  Bits 0 and 1 are mutually exclusive.
- bit 2 STD+LP network is running, otherwise STD network.
  The value is undefined in the case of unbonded [N].
- bit 3-7 Reserved

UserPer Bits array (starting from LSb of the 1st byte) specifying which of the non-embedded peripherals are implemented. 1st bit corresponds to the peripheral 0x20 = `PNUM_USER`. The corresponding bits must be set at the Enumerate Peripherals event. The length of this array can be from 0 to 12 bytes depending on the last implemented user peripheral number. the number of bits set in the bitmap must equal the UserPerNr.

Example

- **Request**

`NADR=0x0000, PNUM=0xFF, PCMD=0x3F, HWPID=0xFFFF`

- **Response**

`NADR=0x0000, PNUM=0xFF, PCMD=0xBF, HWPID=0xABCD, PData={0x00}`(No error)`, {0x07}`(DPA Value)`,{02,03}`(DpaVer 3.02)`, {02}`(UserPerNr)`, {E6,06,00,00}`(StdPers)`, {CD,AB}`(HWPID)`, {01,00}`(HWPIDver)`, {41}`(Flags)`, {02,01}`(UserPer)

[C] (NADR=0x0000) having 2 user defined peripheral, Hardware profile ID of type 0xABCD (version 0x0001), DPA version 2.12.
The following embedded peripherals are enabled:

- 0x01 NODE
- 0x02 OS
- 0x05 RAM
- 0x06 LEDR
- 0x07 LEDG
- 0x09 IO
- 0x0A Thermometer
  bit array (E6,06,00,00): `11100110.00000110.00000000.00000000`

The following user peripherals are implemented:

- 0x21
- 0x28
  bit array (02,01): `00000010.00000001`

2.7.1.1 Source code support

```
typedef struct
{
  uns16       DpaVersion;
  uns8        UserPerNr;
  uns8        EmbeddedPers[ PNUM_USER / 8 ];
  uns16       HWPID;
  uns16       HWPIDver;
  uns8        Flags;
  uns8        UserPer[ ( PNUM_MAX - PNUM_USER + 1 + 7 ) / 8 ];
} TEnumPeripheralsAnswer;

TEnumPeripheralsAnswer      _DpaMessage.EnumPeripheralsAnswer;
```

2.7.2 Get peripheral information

Returns detailed information about the peripheral.

Request

NADR	PNUM	PCMD	HWPID
NADR	PNUM	0x3F	?

The HWPID value is ignored at this command.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1	2	3
NADR	PNUM	0xBF	?	0	?	PerTE	PerT	Par1	Par2

PerTE — Extended peripheral characteristic. See Extended Peripheral Characteristic constants.
PerT — Peripheral type. If the peripheral is not supported or enabled, then PerTx = *PERIPHERAL_TYPE_DUMMY*. See Peripheral Types constants.
Par1 — Optional peripheral specific information.
Par2 — Optional peripheral specific information.

2.7.2.1 Source code support

```
typedef struct
{
  uns8 PerTE;
  uns8 PerT;
  uns8 Par1;
  uns8 Par2;
} TPeripheralInfoAnswer;

TPeripheralInfoAnswer       _DpaMessage.PeripheralInfoAnswer;
```

2.7.3 Get information for more peripherals

Returns the same information as Get peripheral information but for up to 14 peripherals of consecutive indexes starting with the specified PCMD.

Request

NADR	PNUM	PCMD	HWPID
NADR	0xFF	Per	?

Per — Number of the first peripheral from the list to get the information about. The parameter value cannot be 0x3F because it would collide with the Peripheral enumeration command.

The HWPID value is ignored at this command.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1	2	3	…	4×(n-1)	4×(n-1)+1	4×(n-1)+2	4×(n-1)+3
NADR	0xFF	RPer	?	0	?	$PerTE_1$	$PerT_1$	$Par1_1$	$Par2_1$	…	$PerTE_n$	$PerT_n$	$Par1_n$	$Par2_n$

RPer — Same as Per at the request but with the most significant bit set to indicate a response message.
n ∈ [0,14] — Number of peripherals the information was returned about. n = 0 when no peripheral information is returned.

If the peripheral at index x is not supported or enabled, then PerTx = *PERIPHERAL_TYPE_DUMMY*. The response data is always right-trimmed to the last supported or enabled peripheral that can fit in the data array i.e. the data never ends with one or more peripheral information with PerTx = *PERIPHERAL_TYPE_DUMMY*.

2.7.3.1 Source code support

```
TPeripheralInfoAnswer _DpaMessage.PeripheralInfoAnswers[MAX_PERIPHERALS_PER_BLOCK_INFO];
```

2.8 Memory peripherals

The following table lists the characteristics of DPA memory peripherals.

	RAM
Size	`PERIPHERAL_RAM_LENGTH` TR-7xD: 48 bytes TR-7xG: 80 bytes
DPA addressing	`0…PERIPHERAL_RAM_LENGTH-1` TR-7xD: 0…47 TR-7xG: 0…79
C code addressing	`uns8 PeripheralRam[PERIPHERAL_RAM_LENGTH];`

	Internal EEPROM
Size	`PERIPHERAL_EEPROM_LENGTH` [N]: 192 bytes [C]: 64 bytes
DPA addressing	`0…PERIPHERAL_EEPROM_LENGTH-1` [N]: 0…191 [C]: 0…63
C code addressing	using eeReadByte, eeReadData, eeWriteByte, eeWriteData address range `PERIPHERAL_EEPROM_START … (PERIPHERAL_EEPROM_START+ PERIPHERAL_EEPROM_LENGTH-1)`
C code initialization	`#pragma cdata[ __EESTART + PERIPHERAL_EEPROM_START + MY_ADDRESS ] = MY_BYTE0, …`

	External EEPROM
Size	`EEEPROM_READ_LENGTH` 32 Kbytes
DPA addressing	read: `0…EEEPROM_READ_LENGTH` (32 Kbytes) write: `0…EEEPROM_WRITE_LENGTH` (16 Kbytes)
C code addressing	same as DPA addressing using eeeReadData and eeeWriteData
C code initialization	`#pragma cdata[ __EEESTART + MY_ADDRESS ] = MY_BYTE0, …`

3 Peripherals

This (the longest) chapter documents all available embedded peripherals and their commands. Nested chapters named *Source code support* show prepared C code types and variables to access the peripheral command from the code. This is done typically at Custom DPA Handler code.

3.1 Standard operations in general

Commands marked *[sync]* are executed after IQMESH routing is finished thus this event is synchronized among all devices that handled the original DPA Request. This applies to the DPA Request being sent using the broadcast address. If the *[sync]* command is executed on [C] it important to send a next command after it is fully executed or the subsequent command will be ignored.

Commands marked *[comdown]* wait for a maximum of 100 ms to flush output buffers of SPI/UART Peripheral/Interface and then shut it down. This is to prevent raising HW interrupts or to release the OS *bufferCOM* variable that is used internally. After the command is finished the object is restarted.

DPA Requests may return the following error codes:

`ERROR_PCMD` The PNUM does not support the specified PCMD.

`ERROR_PNUM` The specified PNUM is not supported or the PNUM does not support the specified PCMD.

`ERROR_DATA_LEN` The number of bytes at the PData message parameter is not appropriate for the specified PNUM/PCMD pair.

`ERROR_HWPID` The specified HWPID does not correspond to an HWPID of the device.

`ERROR_NADR` The NADR specifies the non-bonded device.

3.1.1 Writing to peripheral

Request

NADR	PNUM	PCMD	HWPID	0	…	n - 1
NADR	PNUM	PCMD	?	$PData_0$	…	$PData_{n-1}$

n Data length

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue
NADR	PNUM	PCMD	?	0	?

PCMD Same as PCMD at the request but with the most significant bit set to indicate the response message.

3.1.1.1 Source code support

```
uns8  _DpaMessage.Request.PData[DPA_MAX_DATA_LENGTH];
```

3.1.2 Reading from peripheral

Request

NADR	PNUM	PCMD	HWPID
NADR	PNUM	PCMD	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	…	n - 1
NADR	PNUM	PCMD	?	0	?	$PData_0$	…	$PData_{n-1}$

PCMD Same as PCMD at the request but with the most significant bit set to indicate the response message.
n Data length

3.1.2.1 Source code support

```
uns8  _DpaMessage.Response.PData[DPA_MAX_DATA_LENGTH];
```

3.2 Coordinator

PNUM = 0x00

This peripheral is implemented at [C] device and it is always enabled there regardless of the configuration settings.

General note: The bond state of the [N] is not synchronized between the [N] and [C]. There are separate requests concerning the bonding at a [N] and a [C].

3.2.1 Peripheral information

PerT `PERIPHERAL_TYPE_COORDINATOR`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1 Maximum number of data (PData) bytes that can be sent in the DPA messages
Par2 Undocumented

3.2.2 Get addressing information

Returns basic network information.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x00	0x00	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1
NADR	0x00	0x80	?	0	?	DevNr	DID

DevNr Number of bonded network Nodes
DID Discovery ID of the network

3.2.2.1 Source code support

```
typedef struct
{
  uns8 DevNr;
  uns8 DID;
} TPerCoordinatorAddrInfo_Response;

TPerCoordinatorAddrInfo_Response _DpaMessage.PerCoordinatorAddrInfo_Response;
```

3.2.3 Get discovered Nodes

Returns a bit map of discovered Nodes.

Same as Get bonded Nodes but PCMD = 0x01.

3.2.4 Get bonded Nodes

Returns a bitmap of bonded Nodes.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x00	0x02	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	…	31
NADR	0x00	0x82	?	0	?	$PData_0$	…	$PData_{31}$

$PData_{0-31}$ Bit array indicating bonded Nodes (addresses). Address 0 at bit_0 of $PData_0$, Address 1 at bit_1 of $PData_0$, etc. Bit values corresponding to the addresses out of the IQMESH address space range are undefined.

3.2.4.1 Source code support

```
uns8    _DpaMessage.Response.PData[DPA_MAX_DATA_LENGTH];
```

3.2.5 Clear all bonds

The command removes all Nodes from the list of bonded Nodes at [C] memory. It deletes the network from the [C] point of view.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x00	0x03	?

Response: General response to writing request with STATUS_NO_ERROR Error code

3.2.6 Bond Node

This command bonds a new [N] by the [C]. There is a maximum of approx. 10 s blocking delay when this function is called. The command must not be used inside Batch or Selective Batch.

Please note that the bonded [N] does not have to be configured for a working network RF channel as the channel is automatically inherited from the network member that provided the bonding and then written to the configuration.

Request

NADR	PNUM	PCMD	HWPID	0	1
NADR	0x00	0x04	?	ReqAddr	BondingTestRetries

ReqAddr — A requested address for the bonded [N]. The address must not be used (bonded) yet.
If this parameter equals 0, then the 1st free address is assigned to the [N]. If this parameter equals 0xF0 and the next parameter BondingTestRetries is 0, then activation of the IQuip is performed for the later OTK bonding. See IQuip users guide for more details.

BondingTestRetries — Maximum number of FRCs used to test whether the [N] was successfully bonded. If the [N] does not respond, it is unbonded at the Coordinator's side. If the value is 0, then no test is performed. If the [N] is connected to and bonded from DSM then this testing never succeeds.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1
NADR	0x00	0x84	?	0	?	BondAddr	DevNr

BondAddr — Address of the [N] newly bonded to the network. The value is undefined in the case of IQuip activation.

DevNr — Total number of bonded Nodes.

Error codes

ERROR_FAIL
a. Nonzero ReqAddr is already used.
b. No free address is available when ReqAddr equals 0.
d. ReqAddr is out of range of valid addresses.
e. Internal call to *bondNewNode* failed.
f. Bonded [N] did not respond to the testing FRC.

3.2.6.1 Source code support

```
typedef struct
{
  uns8 ReqAddr;
  uns8 BondingTestRetries;
} TPerCoordinatorBondNode_Request;

TPerCoordinatorBondNode_Request _DpaMessage.PerCoordinatorBondNode_Request;

typedef struct
{
  uns8 BondAddr;
  uns8 DevNr;
} TPerCoordinatorBondNodeSmartConnect_Response;

TPerCoordinatorBondNodeSmartConnect_Response _DpaMessage.PerCoordinatorBondNodeSmartConnect_Response;
```

3.2.7 Remove bonded Node

Removes [N] from the list of bonded (only already bonded [N]) and discovered (even not bonded [N]) [Ns] at [C] memory.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x00	0x05	?	BondAddr

BondAddr Address of the [N] to remove the bond to

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x00	0x85	?	0	?	DevNr

DevNr Number of bonded network Nodes

Error codes

ERROR_FAIL BondAddr does not specify a bonded [N].

3.2.7.1 Source code support

```
typedef struct
{
  uns8 BondAddr;
} TPerCoordinatorRemoveBond_Request;

TPerCoordinatorRemoveBond_Request
      _DpaMessage.PerCoordinatorRemoveBond_Request;

typedef struct
{
  uns8 DevNr;
} TPerCoordinatorRemoveBond_Response;

TPerCoordinatorRemoveBond_Response
      _DpaMessage.PerCoordinatorRemoveBond_Response;
```

3.2.8 Discovery

[comdown] Runs IQMESH discovery process. The time when the response is delivered depends highly on the number of network devices, the network topology created using specified TxPower, and RF mode, thus, it is not predictable. It can take from a few seconds to many minutes.

Request

NADR	PNUM	PCMD	HWPID	0	1
NADR	0x00	0x07	?	TxPower	MaxAddr

TxPower TX Power used for discovery.

MaxAddr Zero-value specifies, that all bonded [Ns] will take part in the discovery process.
The nonzero value specifies the maximum [N] address to be part of the discovery process. This feature allows splitting all [Ns] into two parts: [1] devices having an address from 1 to MaxAddr will be part of the discovery process thus they become routers, [2] devices having an address from MaxAddr+1 to 239 will not be routers. See IQRF OS documentation for more information.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x00	0x87	?	0	?	DiscNr

DiscNr Number of discovered network Nodes

Error codes

ERROR_FAIL When the internal call of *discovery* fails.

3.2.8.1 Source code support

```
typedef struct
{
  uns8 TxPower;
  uns8 MaxAddr;
} TPerCoordinatorDiscovery_Request;

TPerCoordinatorDiscovery_Request _DpaMessage.PerCoordinatorDiscovery_Request;

typedef struct
{
  uns8 DiscNr;
} TPerCoordinatorDiscovery_Response;

TPerCoordinatorDiscovery_Response _DpaMessage.PerCoordinatorDiscovery_Response;
```

3.2.9 Set DPA Param

Sets DPA Param. DPA Param (DPA Parameter) is a one-byte parameter stored in the [C] RAM that configures network behavior. Default value 0x00 is set upon [C] reset or restart.

Bit	Description	
0-1	Specifies which type of DPA Value is returned in every DPA Request or DPA Confirmation message:	
	00	*lastRSSI*: IQRF OS variable value (*). In the case of the [C] device, the value is 0 until some RF packet is received.
	01	*voltage*: Value returned by *getSupplyVoltage* IQRF OS function (*).
	10	*system*:
		bit 0: Equals *bit DSMactivated*.
		bits 1-6: Reserved
		bit 7: (*)
	11	*user*-specified DPA Value. See UserDpaValue.
2-7	Reserved	

(*) The highest bit.7 indicates that the supply voltage is out of the recommended range. See Supply Voltage for more information about supply voltage values.

DPA Param is transparently sent with every DPA message from the [C] and thus, it controls the network behavior “on the fly”. It is not permanently stored at Nodes.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x00	0x08	?	DpaParam

DpaParam DPA Param to set.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x00	0x88	?	0	?	DpaParam

DpaParam Previous value

3.2.9.1 Source code support

```
typedef struct
{
  uns8 DpaParam;
} TPerCoordinatorSetDpaParams_Request_Response;

TPerCoordinatorSetDpaParams_Request_Response
      _DpaMessage.PerCoordinatorSetDpaParams_Request_Response;
```

3.2.10 Set Hops

Allows the specifying fixed number of hops used to send the DPA Request/Response or to specify an optimization algorithm to compute the number of hops. The default value 0xFF is set upon device reset.

Request

NADR	PNUM	PCMD	HWPID	0	1
NADR	0x00	0x09	?	RequestHops	ResponseHops

Hops values:
0x00, 0xFF: See a description of the parameter of function *optimizeHops* in the IQRF OS documentation. 0x00 does not make sense for the Response Hops parameter.
0x01 - 0xEF: Sets number of hops to the value *Request/ResponseHops - 1*.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1
NADR	0x00	0x89	?	0	?	RequestHops	ResponseHops

Request/Response Hops Previous values

3.2.10.1 Source code support

```
typedef struct
{
  uns8 RequestHops;
  uns8 ResponseHops;
} TPerCoordinatorSetHops_Request_Response;

TPerCoordinatorSetHops_Request_Response
      _DpaMessage.PerCoordinatorSetHops_Request_Response;
```

3.2.11 Backup

This command reads [C] network information data that can be then restored to another [C] to make a clone of the original [C]. The backup data structure is not public and it is encrypted (except the very last byte) by an AES-128 algorithm using an access password as a key. Backup data does not contain the device configuration. Use Read TR Configuration and Write TR Configuration instead.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x00	0x0B	?	Index

Index Index of the block of data

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … 48
NADR	0x00	0x8B	?	0	?	NetworkData

NetworkData One block of the [C] network info data

To read all data blocks just start with Index = 0 and execute the Backup request. Then store the received data block from the response. The last byte of the read data specifies how many data blocks remain to be read. So, if this byte is not 0 just increment Index (0, 1, …) and execute another Backup request.

Error codes

`ERROR_DATA` Index is out of range.
`ERROR_FAIL` Error accessing serial EEPROM chip.

3.2.11.1 Source code support

```
typedef struct
{
  uns8 Index;
} TPerCoordinatorNodeBackup_Request;

TPerCoordinatorNodeBackup_Request _DpaMessage.PerCoordinatorNodeBackup_Request;

typedef struct
{
  uns8 NetworkData[49];
} TPerCoordinatorNodeBackup_Response;

TPerCoordinatorNodeBackup_Response _DpaMessage.PerCoordinatorNodeBackup_Response;
```

3.2.12 Restore

The command allows writing previously backed up [C] network data to the same or another [C] device. To execute the full restore all data blocks (in any order) obtained by Backup commands must be written to the device. Because the data to restore is encrypted by an AES-128 algorithm using an access password as a key, the access password at the device must be the same as the access password at the device that was originally backed up.

The following conditions must be met to make the [C] backup fully functional:

- Backed-up and restored devices have the same access password.
- No network traffic comes from/to restored [C] during the restore process.
- [C] device is reset or restarted after the whole restore is finished.
- It is recommended to run the Discovery command before the network is used after the restoration because of possible RF differences between new and previous [C] device HW.

Request

NADR	PNUM	PCMD	HWPID	0 … 48
NADR	0x00	0x0C	?	NetworkData

NetworkData One block of the [C] network info data that was previously obtained by the Backup command.

Response: General response to writing request with `STATUS_NO_ERROR` Error code

Error codes

`ERROR_DATA` Invalid (access password does not match) or inappropriate (e.g. [C] data used to restore [N] or vice versa) NetworkData content.

ERROR_FAIL Error accessing serial EEPROM chip.

3.2.12.1 Source code support

```
typedef struct
{
  uns8 NetworkData[49];
} TPerCoordinatorNodeRestore_Request;

TPerCoordinatorNodeRestore_Request _DpaMessage.PerCoordinatorNodeRestore_Request;
```

3.2.13 Authorize bond

Authorizes previously prebonded [Ns]. This assigns the [Ns] to the final network address. The command must not be used inside Batch or Selective Batch.

Request

NADR	PNUM	PCMD	HWPID	0	1…4	…	n × 5	n × 5 + 1… n × 5 + 4
NADR	0x00	0x0D	?	$ReqAddr_0$	MID_0	…	$ReqAddr_n$	MID_n

ReqAddr See Bond Node request.
If 0xFF is specified then the prebonded [N] is unbonded and then reset.
Values 0x00 and 0xFF are not allowed if multiple [Ns] (more than 1) are validated.
MID Module ID of the [N] to be authorized. Module ID is typically obtained by PrebondedMemoryReadPlus1.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1
NADR	0x00	0x8D	?	0	?	BondAddr	DevNr

BondAddr Single [N] authorization: address of the [N] newly bonded to the network
Multiple [Ns] authorizations: 0
DevNr Total number of bonded Nodes

Error codes

ERROR_FAIL a. Nonzero ReqAddr is already used.
b. No free address is available when ReqAddr equals 0.
c. Internal call to *nodeAuthorization* failed when single [N] was validated.
d. Some ReqAddr is out of an interval [1;239] but multiple [Ns] are validated.

3.2.13.1 Source code support

```
typedef struct
{
  uns8        ReqAddr;
  uns8        MID[4];
} TPerCoordinatorAuthorizeBond_Request;

TPerCoordinatorAuthorizeBond_Request _DpaMessage.PerCoordinatorAuthorizeBond_Request;

typedef struct
{
  uns8 BondAddr;
  uns8 DevNr;
} TPerCoordinatorAuthorizeBond_Response;

TPerCoordinatorAuthorizeBond_Response
       _DpaMessage.PerCoordinatorAuthorizeBond_Response;
```

3.2.14 Smart Connect

This command bonds [N] using the Smart Connect process. For details please see IQRF OS User's Guide. The command must not be used inside Batch or Selective Batch. The Smart Connect parameters can be effectively stored at IQRF Code to automate the bonding process.

Request

NADR	PNUM	PCMD	HWPID	0	1	2 … 17	18 … 21	22
NADR	0x00	0x12	?	ReqAddr	BondingTestRetries	IBK	MID	res0

23	24 … 27	28 … 37
VirtualDeviceAddress	UserData	res1

ReqAddr — A requested address for the bonded [N]. The address must not be used (bonded) yet. If this parameter equals 0, then the 1st free address is assigned to the [N].

If this parameter equals 0xFE (IQMESH temporary address) and MID equals 0x00000000 then all unbonded Nodes within the RF reach of the current network and having the same Access Password as existing members of the network will be prebonded with the address 0xFE. When the 0xFE parameter is used please make sure all other command parameters are zeroed.

If this parameter equals 0xFE and MID is nonzero then only [Ns] whose remainder by dividing their MID by MID[1] (i.e. PData[19]) equals MID[0] (i.e. PData[18]) will be prebonded to the network. This feature is intended for building so-called overlapping networks based on the Law of large numbers. Again, the same Access Password is required.

BondingTestRetries — Maximum number of FRCs used to test whether the [N] was successfully bonded. If the [N] does not respond, it is unbonded at the Coordinator's side. If the value is 0, then no test is performed and the command always succeeds.

IBK — Individual Bonding Key of the [N] to bond. Must be zeroed if ReqAddr equals 0xFE.

MID — MID of the [N] to bond. Must be zeroed if ReqAddr equals 0xFE.

res0 — Reserved. Must be zero.

VirtualDeviceAddress — Virtual device address. Must equal 0xFF if not used.

UserData — Reserved for future optional data passed to the bonded [N].

res1 — Reserved. It must be filled with zeros.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1
NADR	0x00	0x92	?	0	?	BondAddr	DevNr

BondAddr — Address of the [N] newly bonded to the network.

DevNr — The number of bonded Nodes in the network.

Error codes

ERROR_FAIL
a. Nonzero ReqAddr is already used.
b. No free address is available when ReqAddr equals 0.
c. ReqAddr is out of range of valid addresses.
d. Internal call to *smartConnect* failed.
e. None of the testing FRCs used to test the connection to the bonded [N] succeeded.

3.2.14.1 Source code support

```
typedef struct
{
  uns8  ReqAddr;
  uns8  BondingTestRetries;
```

```
  uns8  IBK[16];
  uns8  MID[4];
  uns8  reserved0;
  uns8  VirtualDeviceAddress;
  uns8  UserData[4];
  uns8  reserved1[10];
} TPerCoordinatorSmartConnect_Request;

TPerCoordinatorSmartConnect_Request _DpaMessage.PerCoordinatorSmartConnect_Request;

typedef struct
{
  uns8 BondAddr;
  uns8 DevNr;
} TPerCoordinatorBondNodeSmartConnect_Response;

TPerCoordinatorBondNodeSmartConnect_Response _DpaMessage.PerCoordinatorBondNodeSmartConnect_Response;
```

3.2.15 Set MID

Sets the MID value to a [N] with a specified address in the Coordinator's database. This feature is used after [N] restore which typically changes Node's MID. It can be also used to make sure the Coordinator's database contains real MIDs after the [C] was updated from the former IQRF OS version that did not store MIDs at [C] at all (MIDs are read as 0xFFFFFFFF). Use OS Read to obtain Node's MID. See also Node Restore.

Request

NADR	PNUM	PCMD	HWPID	0…3	4
NADR	0x00	0x13	?	MID	BondAddr

MID The MID is written to the Coordinator's database in the external EEPROM.
BondAddr Address of the [N] to set the MID to.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Error codes
`ERROR_FAIL` Error accessing serial EEPROM chip.

3.2.15.1 Source code support

```
typedef struct
{
  uns8 MID[4];
  uns8 BondAddr;
} TPerCoordinatorSetMID_Request;

TPerCoordinatorSetMID_Request
      _DpaMessage.PerCoordinatorSetMID_Request;
```

3.3 Node

PNUM = 0x01

This peripheral is implemented at [N] devices and it is always enabled there regardless of the configuration settings.

General note: Bond state of the [N] is not synchronized between the [N] and [C]. There are separate requests for [N] and [C] concerning the bonding.

3.3.1 Peripheral information

PerT `PERIPHERAL_TYPE_NODE`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1 Maximum number of data (PData) bytes that can be sent in the DPA messages
Par2 Undocumented

3.3.2 Read

Returns IQMESH specific [N] information.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x01	0x00	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … 10	11
NADR	0x01	0x80	?	0	?	ntwADDR … ntwCFG	Flags

ntwADDR … ntwCFG Block of all *ntw** IQRF OS variables (*ntwADDR*, *ntwVRN*, *ntwZIN*, *ntwDID*, *ntwPVRN*, *ntwUSERADDRESS*, *ntwID*, *ntwVRNFNZ*, *ntwCFG*) in the same order and size as located in the IQRF OS memory. See IQRF OS documentation for more information.
Flags bit 0 Indicates whether the [N] is bonded.
bit 1-7 Reserved

3.3.2.1 Source code support

```
typedef struct
{
  uns8  ntwADDR;
  uns8  ntwVRN;
  uns8  ntwZIN;
  uns8  ntwDID;
  uns8  ntwPVRN;
  uns16 ntwUSERADDRESS;
  uns16 ntwID;
  uns8  ntwVRNFNZ;
  uns8  ntwCFG;
  uns8  Flags;
} TPerNodeRead_Response;

TPerNodeRead_Response _DpaMessage.PerNodeRead_Response;
```

3.3.3 Remove bond

[sync] [comdown] The [N] is marked as unbonded (removed from network) using the *removeBond* IQRF OS function and then restarted (except in DSM). The bonding state of the [N] on the [C] side is not affected at all. Please use for instance Remove bonded Node to unbond [N] at [C].

Request

NADR	PNUM	PCMD	HWPID
NADR	0x01	0x01	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.3.4 Backup

Same as [C] Backup except PNUM = 0x01 and PCMD = 0x06.

3.3.5 Restore

Same as [C] Restore except PNUM = 0x01 and PCMD = 0x07.

3.3.6 Validate bonds

[sync] [comdown] (only when a [N] is unbonded and restarted) This command can be used to resolve the situation when there are more [Ns] with the same address in the network. This incorrect situation might happen usually in the case of Smart Connect, or due to unintended user interference when the [N] was bonded but for some reason, it does not communicate afterward and therefore it is unbonded at the [C] side only and its address is recycled for another [N] later.

The command can hold up to 11 pairs of the network [N] address and [N] MID in the data part. When the command is broadcast and the [N] finds its address in the data but the MID does not equal its MID, the [N] unbonds itself, and then it restarts (it might skip optional RFPGM after module reset).

The typical algorithm is to loop all bonded [Ns] and for each [N] to read its MID from the [C] external EEPROM. Then pack up to 11 address/MID pairs into the command and send a series of broadcast commands into the network.

Request

NADR	PNUM	PCMD	HWPID	0	1…4	…	n × 5	n × 5 + 1… n × 5 + 4
NADR	0x01	0x08	?	$Address_0$	MID_0	…	$Address_n$	MID_n

n ∈ [0,10] Number of validated addresses minus 1.
Address Node's address.
MID Node's MID. Value 0xFFFFFFFF is ignored.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.3.6.1 Source code support

```
typedef struct
{
  uns8 Address;
  uns8 MID[4];
} TPerNodeValidateBondsItem;

typedef struct
{
  TPerNodeValidateBondsItem Bonds[DPA_MAX_DATA_LENGTH / sizeof(TPerNodeValidateBondsItem)];
} TPerNodeValidateBonds_Request;

TPerNodeValidateBonds_Request
_DpaMessage.PerNodeValidateBonds_Request;
```

3.4 OS

PNUM = 0x02

This peripheral is always enabled regardless of the configuration settings.

3.4.1 Peripheral information

PerT `PERIPHERAL_TYPE_OS`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`

Par1 Date of the DPA build coded using BCD.
Par2 Lower nibble contains the month of the DPA build date. Higher nibble contains the year above 2010 modulo 16.

Example: Par1=0x31, Par2=0x4A => build date is 31. 10. 2014.

3.4.2 Read

Returns some useful system information about the device.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x00	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … 3	4	5	6 … 7
NADR	0x02	0x80	?	0	?	MID	OsVersion	TrType	OsBuild

8	9	10	11	12 … 27	28 … (40…51)
Rssi	SupplyVoltage	Flags	SlotLimits	IBK	PerEnum

MID,
OsVersion,
TrType,
OsBuild See *moduleInfo* at IQRF OS Reference Guide.
Rssi See *lastRSSI* at IQRF OS Reference Guide. In the case of the [C] device, the value is 0 until some RF packet is received.
SupplyVoltage See *getSupplyVoltage* at IQRF OS Reference Guide.
Flags bit.0 is 1 if there is an insufficient OsVersion for the used DPA version.
bit.1 is 0 if the SPI interface is supported (*); 1 if the UART interface is supported (*). This bit is valid only if bit.4 is 0.
bit.2 is 1 if Custom DPA Handler was detected.
bit.3 is 1 if Custom DPA Handler is not detected but enabled at TR Configuration. See details of the handling of this erroneous state.
bit.4 is 1 if no interface is supported (*).
bit.5 is 1 if IQRF OS is changed from the originally manufactured version.
bit.6 is 1 if the FRC Aggregation feature is enabled by the TR manufacturer.
bit.7 is reserved.
SlotLimits Lower nibble stores the shortest timeslot length in 10 ms units, upper nibble stores the longest timeslot respectively. The stored length value is lowered by 3. So a value 0x31 specifies the shortest timeslot of 40 ms and the longest of 60 ms. The value is undefined in the case of unbonded [N].
IBK Individual Bonding Key.
PerEnum See the response of Peripheral enumeration.

(*) “Supported” means the interface is supported by the uploaded DPA plug-in.

3.4.2.1 Source code support

```
typedef struct
{
  uns8        MID[4];
  uns8        OsVersion;
  uns8        TrType;
  uns16       OsBuild;
  uns8        Rssi;
  uns8        SupplyVoltage;
  uns8        Flags;
```

```
  uns8        SlotLimits;
  uns8        IBK[16];
  // Enumerate peripherals part, variable length because of UserPer field
  uns16       DpaVersion;
  uns8        UserPerNr;
  uns8        EmbeddedPers[PNUM_USER / 8];
  uns16       HWPID;
  uns16       HWPIDver;
  uns8        FlagsEnum;
  uns8        UserPer[( PNUM_MAX - PNUM_USER + 1 + 7 ) / 8];
} TPerOSRead_Response;

TPerOSRead_Response _DpaMessage.PerOSRead_Response;
```

3.4.3 Reset

[sync] [comdown] Forces TR transceiver module to carry out reset.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x01	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.4.4 Restart

[sync] [comdown] Forces TR transceiver module to restart. It is similar to reset (the device starts, RAM, and global variables are cleared) except MCU is not reset from the HW point of view (MCU peripherals are not initialized) and RFPGM on reset (when it is enabled) is always skipped.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x08	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.4.5 Read TR Configuration

Reads a raw TR Configuration memory. Bit values for [C] (bit 0), [N] (bit 1), and OS (bit 2) peripherals stored at TR Configuration byte index 1 are set the same way as in Peripheral enumeration.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x02	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1 … 31	32	33
NADR	0x02	0x82	?	0	?	Checksum	Configuration	RFPGM	InitPHY

Checksum	The Checksum byte XORed with all Configuration bytes gives 0x5F.
Configuration	Content the configuration memory block from address 0x01 to 0x1F.
RFPGM	See the parameter of setupRFPGM IQRF OS function.
InitPHY	This value is read-only. bits.0-1 RF band

00	868 MHz
01	916 MHz
10	433 MHz
11	Reserved
bits.2-3	Reserved
bit.4	1 if on-board thermometer sensor chip is present.
bit.5	1 if serial EEPROM chip is present.
bit.6	1 if transceiver is IL type i.e., for Israel region. See Device Startup.
bit.7	Reserved

3.4.5.1 Source code support

```
typedef struct
{
  uns8 Checksum;
  uns8 Configuration[31];
  uns8 RFPGM;
  uns8 Undocumented[1];
} TPerOSReadCfg_Response;

TPerOSReadCfg_Response _DpaMessage.PerOSReadCfg_Response;
```

3.4.6 Write TR Configuration

Writes TR Configuration memory. It is a programmer's responsibility to prepare the correct configuration block including the checksum byte. This command is for advanced users only. Please note that the device should be restarted for all configuration changes to take effect. See TR Configuration for details.

Request

NADR	PNUM	PCMD	HWPID	0	1 … 31	32
NADR	0x02	0x0F	?	Undefined	Configuration	RFPGM

Undefined	Value does not matter. The checksum value that is read at this same position will be computed automatically.
Configuration	Content the configuration memory block from address 0x01 to 0x1F.
RFPGM	See the parameter of *setupRFPGM* IQRF OS function.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Example

The following example shows writing RF output power value to the configuration in the Custom DPA Handler code. Note - for changing just a few configuration values (bytes or bits) it is more efficient to use Write TR Configuration byte.

```
// Read configuration
_PNUM = PNUM_OS;
_PCMD = CMD_OS_READ_CFG;
_DpaDataLength = 0;
DpaApiLocalRequest();

// Update TX power
_DpaMessage.PerOSWriteCfg_Request.Configuration[ CFGIND_TXPOWER -
offsetof(TPerOSWriteCfg_Request, Configuration) ] = txPowerToSet;

// Write configuration
_PCMD = CMD_OS_WRITE_CFG;
_DpaDataLength = sizeof( TPerOSWriteCfg_Request );
```

```
DpaApiLocalRequest();
```

3.4.6.1 Source code support

```
typedef struct
{
  uns8 Undefined;
  uns8 Configuration[31];
  uns8 RFPGM;
} TPerOSWriteCfg_Request;

TPerOSWriteCfg_Request _DpaMessage.PerOSWriteCfg_Request;
```

3.4.7 Write TR Configuration byte

Writes multiple bytes (or just bits) to the TR Configuration memory. This command is for advanced users only. The Acknowledged broadcast is recommended for writing configuration values to all or selected Nodes as it also confirms which Nodes performed the configuration write. Please note that the device should be restarted for some configuration changes to take effect. See TR Configuration for details.

Request

NADR	PNUM	PCMD	HWPID	0	1	2	…	n × 3	n × 3 + 1	n × 3 + 2
NADR	0x02	0x09	?	$Address_0$	$Value_0$	$Mask_0$	…	$Address_n$	$Value_n$	$Mask_n$

n ∈ [0,17]	Number of configuration items to write minus 1.
Address	Address of the item at configuration memory block. The valid address range is 0x00-0x1F for configuration values. Also, address 0x20 is a valid value for RFPGM settings. See the parameter of *setupRFPGM* IQRF OS function.
Value	Value of the configuration item to write.
Mask	Specifies bits of the configuration item (i.e. byte) to be modified by the corresponding bits of the Value parameter. Only bits that are set at the Mask will be written to the configuration byte i.e. when Mask equals 0xFF then the whole Value will be written to the configuration byte. For example, when Mask equals 0x12 then only bit.1 and bit.4 from Value will be written to the configuration byte.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Error codes

`ERROR_DATA`	Address is out of range.

3.4.7.1 Source code support

```
typedef struct
{
  uns8 Address;
  uns8 Value;
  uns8 Mask;
} TPerOSWriteCfgByteTriplet;

typedef struct
{
  TPerOSWriteCfgByteTriplet
       Triplets[DPA_MAX_DATA_LENGTH / sizeof( TPerOSWriteCfgByteTriplet )];
} TPerOSWriteCfgByte_Request;

TPerOSWriteCfgByte_Request _DpaMessage.PerOSWriteCfgByte_Request;
```

3.4.8 Run RFPGM

[sync] [comdown] Puts the device into RFPGM mode configured at TR Configuration. The device is always reset when the RFPGM process is finished for any reason. RFPGM runs at the same channels (configured at TR Configuration) the network is using.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x03	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.4.9 Sleep

Puts the device into sleep (power saving) mode.

[sync] [comdown] This command is implemented at the [N] device only. Brown-Out Reset (BOR) is disabled during the execution and enabled on exit at TR-7xD transceivers.

The (in)accuracy of the real sleep time depends on the PIC LFINTOSC oscillator that runs the watchdog timer. The oscillator frequency is mainly influenced by the device supply voltage and temperature volatility. See the PIC MCU datasheet for more details.

If the interface is used then it is disabled before going to sleep and enabled after the device wakes up.

Before going to sleep mode the UART DPA peripheral or DPA interfaces are automatically shut down and later restarted when the device wakes up. Please consider implementing BeforeSleep and AfterSleep events to handle MCU peripherals and pins to obtain the lowest possible device consumption.

The command provides two sleep modes. Standard sleep (with RF transceiver chip in a ready state) and Deep sleep (with RF transceiver chip in sleep state). It might seem that the deep sleep one is always the best choice because of lower power consumption but one must consider the time (i.e. power consumption) needed to switch the RF transceiver from the sleep mode into the fully operational mode. Please note Online DPA Menu at TR-7xG is not active during sleep unless wake up is enabled using Control.bit.0.

Request

NADR	PNUM	PCMD	HWPID	0	1	2
NADR	0x02	0x04	?	Time		Control

Time — Sleep time in 2.097 s or 32.768 ms units. See Control.bit.4. Maximum sleep time is 38 hours 10 minutes 38.95 seconds or 35 minutes 47.48 seconds respectively. 0 specifies endless sleep (except Control.bit1 is set to run the calibration process without performing sleep). In case of endless sleep make sure to set Control.bit.0 or Control.bit.3 to enable wake up.

Control
- bit 0 — Wake up on PORTB.4 pin negative edge change. See the *iqrfSleep* IQRF OS function for more information.
- bit 1 — Runs the calibration process before going to sleep. Calibration takes approximately 16 ms and this time is subtracted from the requested sleep time. Calibration time deviation may produce an absolute sleep time error at short sleep times. But it is worth running the calibration always before a long sleep because the calibration time deviation then accounts for a very small total relative error. The calibration is always run before a first sleep with nonzero

Time after the module reset if the calibration was not already initiated by Time=0 and Control.bit.1=1.

- bit 2 If set, then when the device wakes up after the sleep period, a green LED once shortly flashes. It is useful for diagnostic purposes.
- bit 3 Wake up on PORTB.4 pin positive edge change. See the *iqrfSleep* IQRF OS function for more information.
- bit 4 If set then the unit is 32.768 ms instead of the default 2.097 s (i.e. 2048 × 1.024 ms).
- bit 5 *iqrfDeepSleep* instead of *iqrfSleep* is used. See IQRF OS documentation for more information.
- bit 6-7 Reserved.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Example 1

[N] #1 sleep for 1 minute with green LED flash after waking up:

Unit is 32.768 ms => sleep time is 1831 = 0x0727 units:

`NADR=0x0001, PNUM=0x02, PCMD=0x04, HWPID=0xFFFF, PData={0x27}`(time lower byte) `{0x07}`(time higher byte) `{0x14}`(LED flash + finer unit)

Example 2

[N] #10 deep sleep for 1 hour with forced calibration and wake up on negative edge change:

Unit is 2.097 s => sleep time is 1717 = 0x06B5 units:

`NADR=0x000A, PNUM=0x02, PCMD=0x04, HWPID=0xFFFF, PData={0xB5}`(time lower byte) `{0x06}`(time higher byte) `{0x43}`(calibration + negative edge + deep sleep)

3.4.9.1 Source code support

```
typedef struct
{
  uns16       Time;
  uns8        Control;
} TPerOSSleep_Request;

TPerOSSleep_Request  _DpaMessage.PerOSSleep_Request;
```

3.4.10 Set Security

This command allows setting various security parameters.

Request

NADR	PNUM	PCMD	HWPID	0	1 … 16
NADR	0x02	0x06	?	Type	Data

Type 0 Sets an access password using the *setAccessPassword* IQRF OS function.
1 Sets a user key using the *setUserKey* IQRF OS function.
other Reserved

Data Data written to the specified Type of security parameter.

Response

The general response to writing request with STATUS_NO_ERROR Error code.

Error codes

ERROR_DATA Invalid Type value.

3.4.10.1 Source code support

```
typedef struct
{
  uns8 Type;
  uns8 Data[16];
} TPerOSSetSecurity_Request;

TPerOSSetSecurity_Request  _DpaMessage.PerOSSetSecurity_Request;
```

3.4.11 Batch

[sync] Batch command allows executing more individual DPA Requests within one original DPA Request. Both the sender's and addressee's addresses of each embedded DPA Request equal the corresponding addresses of the original Batch DPA Request. It is not allowed to embed the Batch command itself within a series of individual DPA Requests (recursion is not possible). Using Run discovery is not allowed inside the batch command list. Batch command is useful not only to group commands but also to execute the asynchronous command(s) synchronously (after the Batch response is sent).

Request

NADR	PNUM	PCMD	HWPID	0 …	n
NADR	0x02	0x05	?	Requests	0

Requests — It contains more DPA Requests to be executed. The format in which the DPA Requests are stored is the same as the format of IO Setup DPA Requests. See IO Setup for more information.

Example

The following example runs a simple broadcast set of 5 DPA Requests. It switches on the red LED at devices with HW profile ID 0x1234 or green LED at devices with HW profile ID 0x5678 respectively, then waits for 200 ms (using I/O peripheral) and finally switches the same LEDs off.

NADR=0x00FF, PNUM=0x02, PCMD=0x05, HWPID=0xFFFF, PData=
[1st command] {0x05(length), 0x06(PNUM=LEDR), 0x01(PCMD=LED on), 0x1234(HWPID)},
[2nd command] {0x05(length), 0x07(PNUM=LEDG), 0x01(PCMD=LED on), 0x5678(HWPID)},
[3rd command] {0x08(length), 0x09(PNUM=I/O), 0x01(PCMD=Set),0xFFFF(HWPID),0xFF(Delay command),0x00C8(200 ms)}
[4th command] {0x05(length), 0x06(PNUM=LEDR), 0x00(PCMD=LED off),0x1234HWPID)},
[5th command] {0x05(length), 0x07(PNUM=LEDG), 0x00(PCMD=LED off),0x5678HWPID)},
{0x00(end of the batch)}

Response

The general response to writing request with STATUS_NO_ERROR Error code.

3.4.12 Selective Batch

[sync] This command is similar to the Batch but besides, it allows specifying Nodes that execute the batch. This implies that the command is typically used at broadcast. This command is not implemented at the [C] device.

Request

NADR	PNUM	PCMD	HWPID	0 … 29	30 …	n

NADR	0x02	0x0B	?	SelectedNodes	Requests	0

SelectedNodes See identically named field at Send Selective command.
Requests See identically named field at Batch command.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.4.12.1 Source code support

```
typedef struct
{
  uns8 SelectedNodes[30];
  uns8 Requests[DPA_MAX_DATA_LENGTH - 30];
} TPerOSSelectiveBatch_Request;

TPerOSSelectiveBatch_Request      _DpaMessage.PerOSSelectiveBatch_Request;
```

3.4.13 LoadCode

[sync] [comdown] Implemented at [C] and [N] devices. This advanced command allows OTA (over the air] update of the firmware as it loads a code previously-stored at external EEPROM to the MCU Flash memory. Then the device is reset. External EEPROM can store more code images at one time. When storing the code for upload at the external EEPROM, make sure you do not overwrite another stored code or IO Setup.

Please note, that there might be a considerable delay before a response is ready because the command needs to read a larger amount of external EEPROM memory and compute the checksum.

The command can load three types of code:

1. Custom DPA Handler code from the .hex file.

   Custom DPA Handler code (but not the optional content of EEPROM and/or external EEPROM required by the handler) can be uploaded, updated, or just “switched” “over the air” without the need to reprogram the device using a hardware programmer.

   It is necessary to read the output .hex file containing compiled Custom DPA Handler code to obtain the code before it can be stored as an image at external EEPROM. The continuous code block starts from the PIC address `CUSTOM_HANDLER_ADDRESS = 0x3A20` and is located up to address `CUSTOM_HANDLER_ADDRESS_END - 1 = 0x3D7F`. Because each MCU instruction takes 2 bytes the address inside the .hex file is doubled so the code starts from address 0x7440 at the .hex file. Please read the Custom DPA Handler Code from .hex File for more details.

   The length of the image stored in the external EEPROM must be a multiple of 64 (the used Flash memory page of MCU is 32 words long) otherwise the result is undefined. The checksum value is calculated from all the code bytes including unused trailing bytes that fill in the last 64-byte block. We recommend filling in unused trailing bytes by value 0x34FF to get the same checksum value as IQRF IDE. The initial value of the Fletcher-16 checksum is 0x0001.

   If loaded Custom DPA Handler code needs to use the certain content of EEPROM and/or external EEEPROM memory, then EEPROM and/or EEEPROM peripherals can be used to prepare the content before the handler is loaded. Disabling former Custom DPA Handler using Write TR Configuration byte (configuration byte at index 0x5, bit 0) and Restart is highly recommended (both commands might be the content of one Batch or Acknowledged broadcast - bits) if old or a new handler use EEPROM and/or EEEPROM peripherals. After a new handler is loaded it must be then enabled back.

2. IQRF plug-in containing DPA protocol implementation (to perform DPA version change on the fly), Custom DPA Handler, or IQRF OS patch. The feature is supported starting from IQRF OS version 3.08D and the corresponding DPA version.

IQRF plug-in file is a text file containing an encrypted code. Only lines of the file that do not start with character # contain the code. Such lines contain 20 bytes stored by 2 hexadecimal characters (thus every line contains 40 characters in total). To create a code image for the external EEPROM from IQRF plug-in file just read all the consequential hexadecimal bytes from all code lines from the beginning to the end of the file, convert them to the real bytes and store them in the external EEPROM.

The length of the image stored in the external EEPROM must be multiple of 20. The initial value of the Fletcher-16 checksum is 0x0003.

Please note that only DPA IQRF plug-in version 2.26 or higher can be loaded.

3. IQRF OS Change File containing IQRF OS and/or DPA update. See chapter IQRF OS Change and below for more information. Using LoadCode command at TR-7xG transceivers, the special handler *CustomDpaHandler-ChangeIQRFOS.iqrf* is not needed anymore.

Request

NADR	PNUM	PCMD	HWPID	0	1 … 2	3 … 4	5 … 6
NADR	0x02	0x0A	?	Flags	Address	Length	CheckSum

Flags bit 0 Action:
0 Computes and matches the checksum only without loading code.
1 Same as above plus loads the code into Flash if the checksum matches.
bits 1-2 Code type:
00 Loads Custom DPA Handler from .hex file.
01 Loads IQRF plug-in from .iqrf file.
1x Loads IQRF OS Change File .bin file. This option is available at TR-7xG.
bits 3-7 Reserved, must equal 0.

Address A physical address at external EEPROM memory to load the code image from.

Length Length of the code image in bytes at the external EEPROM. See the text above. When IQRF OS Change File .bin file is loaded, then the value must be 0x??55 otherwise the actual upload will not be performed but device only resets.

CheckSum One's complement Fletcher-16 checksum of the code image. If the checksum does not match a checksum of the code stored in external EEPROM then writing the code to the Flash memory is not performed. See source code examples of the checksum calculation. For an initial checksum value see the text above. Different initial checksum values for both types of upload code ensure that code types cannot be confused.
When IQRF OS Change File .bin file is loaded then this parameter is ignored.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x02	0x8A	?	0	?	Result

Result bit 0

0	An error occurred. The following bits of the result contain the detail.
1	All conditions are met to load the code if requested. The code will be loaded if Flags.0 was set at the request.

bits 1-7

0000.000	The checksum at the external EEPROM does not match the provided checksum in case of .hex/.iqrf upload.
0000.011	Old IQRF OS is not present (old checksum does not match) in case of .bin upload.
0000.100	The checksum of the IQRF OS Change File in the external EEPROM does not match in case of .bin upload.
0000.111	IQRF OS change file stored in the external EEPROM has an unsupported version in case of .bin upload.
other values	Reserved.

Error codes

ERROR_FAIL	Invalid Flags value.

3.4.13.1 Source code support

```
typedef struct
{
  uns8        Flags;
  uns16       Address;
  uns16       Length;
  uns16       CheckSum;
} TPerOSLoadCode_Request;

TPerOSLoadCode_Request _DpaMessage.PerOSLoadCode_Request;
```

3.4.14 Test RF Signal

This command tests the RF signal in the same way as FRC command Test RF Signal. The command is implemented at the [C] device only.

Request

NADR	PNUM	PCMD	HWPID	0	1	2 … 3
NADR	0x02	0x0C	?	Channel	RXfilter	Time

Channel	See the same parameter at Test RF Signal.
RXfilter	See the same parameter at Test RF Signal.
Time	Time interval to test the signal. The unit is 10 ms.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x02	0x8C	?	0	?	Counter

Counter See output value of Test RF Signal.

3.4.14.1 Source code support

```
typedef struct
{
  uns8  Channel;
  uns8  RXfilter;
  uns16 Time;
} TPerOSTestRfSignal_Request;

TPerOSTestRfSignal_Request _DpaMessage.PerOSTestRfSignal_Request;

typedef struct
{
```

```
  uns8  Counter;
} TPerOSTestRfSignal_Response;

TPerOSTestRfSignal_Response _DpaMessage.PerOSTestRfSignal_Response;
```

3.4.15 Factory Settings

[sync] [comdown] This command is implemented at the [N] device only. It executes the following settings and actions:

1. Sets RF output power in the configuration to the default and maximum value 7.
2. Sets RF signal filter in the configuration to default value 5.
3. Clears Access Password to the default value. Important - it is not recommended to keep default (full of zeros) Access Password in the production network as it decreases the overall network security.
4. Removes bond ([N] restart is included by default except in DSM).

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x0D	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.4.16 Indicate

[sync] This command is implemented at the [N] devices only. The command controls the device indication (by default visual). The indication is usually used during inventory or device localization. By default, the command uses both (red and green) default IQRF LEDs at (or connected to) the [N].

The default indication behavior can be changed using an Indicate event. The custom indication can be visual (a light), acoustic (a buzzer), action (a motor), etc.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x02	0x07	?	Control

Control bits.0-1
- 0b00 Indication is switched off.
- 0b01 Indication is switched on [*].
- 0b10 Indication is on for 1 s [**].
- 0b11 Indication is on for 10 s [**].

[*] The default LED indication will be immediately off at [N] at LP mode. Use timed indication at LP [N] devices instead.
[**] During the indication, the [N] is blocked and any network traffic will not be processed.

bits.2-7 Reserved.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.4.16.1 Source code support

```
typedef struct
{
  uns8  Control;
} TPerOSIndicate_Request;
```

```
TPerOSIndicate_Request _DpaMessage.PerOSIndicate_Request;
```

3.5 EEPROM

PNUM = 0x03

This peripheral controls internal MCU EEPROM memory. See also Memory peripherals.

3.5.1 Peripheral information

PerT	PERIPHERAL_TYPE_EEPROM
PerTE	PERIPHERAL_TYPE_EXTENDED_READ_WRITE
Par1	Size in bytes. In the current version of DPA, it equals 192 at [N] device or 64 at [C] respectively.
Par2	Maximum data block length. In the current version of DPA, it equals 55 bytes.

Actual EEPROM address space starts at address 0x00 at [N] device or 0x80 at [C] devices. There is a predefined symbol *PERIPHERAL_EEPROM_START* that equals the actual starting address.

3.5.2 Read

Reads data from the memory.

Request

NADR	PNUM	PCMD	HWPID	0	1
NADR	0x03	0x00	?	Address	Length

Address	An address to read data from.
Length	Length of the data in bytes.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	…	Len-1
NADR	0x03	0x80	?	0	?	$PData_0$	…	$PData_{Len-1}$

Len	Read data length.

Error codes

ERROR_ADDR	Data address is out of range.

3.5.2.1 Source code support

```
typedef struct
{
  uns8 Address;

  union
  {
       struct
       {
         uns8 Length;
       } Read;
  } ReadWrite;
} TPerMemoryRequest;

TPerMemoryRequest _DpaMessage.MemoryRequest;
```

3.5.3 Write

Writes data to the memory.

Request

NADR	PNUM	PCMD	HWPID	0	1	…	n+1
NADR	0x03	0x01	?	Address	$PData_0$	…	$PData_{n-1}$

Address An address to write data to.
PData Actual data to be written to the memory.
n Written data length.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Error codes

`ERROR_ADDR` Data address is out of range.

3.5.3.1 Source code support

```
typedef struct
{
  uns8 Address;

  union
  {
       #define       MEMORY_WRITE_REQUEST_OVERHEAD      ( sizeof( uns8 ) )
       struct
       {
         uns8 PData[DPA_MAX_DATA_LENGTH - MEMORY_WRITE_REQUEST_OVERHEAD];
       } Write;

  } ReadWrite;
} TPerMemoryRequest;

TPerMemoryRequest _DpaMessage.MemoryRequest;
```

3.6 EEEPROM

PNUM = 0x04

This peripheral controls external serial EEPROM memory. If the external serial EEPROM memory is not accessible or missing, the `ERROR_FAIL` code is returned. Please note that the part of the external EEPROM memory space can be used for IO Setup. See also Memory peripherals.

3.6.1 Peripheral information

PerT `PERIPHERAL_TYPE_BLOCK_EEPROM`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1 Memory size in 256 bytes blocks. In the current version of DPA, it equals 0x80 (memory size is 32 kB). Value 0x00 represents 0x100 thus memory size would be 64 kB.
Par2 Page size in bytes. Non-zero page boundaries must not be exceeded during writing. When page size is 0 then there is no writing limitation.

3.6.2 Extended Read

This command allows reading data from the whole physical address space of the external EEPROM.

Request

NADR	PNUM	PCMD	HWPID	0 … 1	2
NADR	0x04	0x02	?	Address	Length

Address A physical address to read data from.
Length Length of the data to read in bytes. The allowed range is 0-54 bytes. Reading behind the maximum address range is undefined.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	…	Len-1
NADR	0x04	0x82	?	0	?	$PData_0$	…	$PData_{Len-1}$

Len Read data length.

Error codes
`ERROR_ADDR` Starting address is out of range.
`ERROR_FAIL` Error accessing serial EEPROM chip.

3.6.2.1 Source code support

```
typedef struct
{
  uns16       Address;

  union
  {
       struct
       {
         uns8 Length;
       } Read;
  } ReadWrite;
} TPerXMemoryRequest;

TPerXMemoryRequest _DpaMessage.XMemoryRequest;
```

3.6.3 Extended Write

This command allows writing data to the address space of the external EEPROM.

Request

NADR	PNUM	PCMD	HWPID	0 … 1	2	…	n+2
NADR	0x04	0x03	?	Address	$Data_0$	…	$Data_{n-1}$

Address The allowed address range is 0x0000-0x3FFF.
Data Actual data to be written to memory.
n Length of the data to write in bytes. The allowed range is 1-54 bytes.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Error codes
`ERROR_ADDR` Starting address is out of range.
`ERROR_FAIL` Error accessing serial EEPROM chip.

3.6.3.1 Source code support

```
typedef struct
{
```

```
  uns16       Address;

  union
  {
#define       XMEMORY_WRITE_REQUEST_OVERHEAD     ( sizeof( uns16 ) )

      struct
      {
        uns8 PData[DPA_MAX_DATA_LENGTH - XMEMORY_WRITE_REQUEST_OVERHEAD];
      } Write;

  } ReadWrite;
} TPerXMemoryRequest;

TPerXMemoryRequest _DpaMessage.XMemoryRequest;
```

3.7 RAM

PNUM = 0x05

This peripheral controls a block of internal MCU RAM. The address space of the peripheral occupies the whole bank 12 of the MCU RAM and can be accessed by an array variable *PeripheralRam* from Custom DPA Handler code. See also Memory peripherals.

3.7.1 Peripheral information

PerT	`PERIPHERAL_TYPE_RAM`
PerTE	`PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1	Size in bytes. In the current version of DPA, it equals 48 at TR-7xD transceivers and 80 at TR-7xG transceivers.
Par2	Maximum data block length. In the current version of DPA, it equals 48 at TR-7xD transceivers or 55 at TR-7xG transceivers respectivelly.

3.7.2 Read & Write

See EEPROM.

3.7.2.1 Source code support

```
bank12 uns8  PeripheralRam[PERIPHERAL_RAM_LENGTH] @ 0x620;
```

3.7.3 Read Any

This command can read any addressable memory. The command is same as EEEPROM extended Read except PNUM = 0x05 and PCMD = 0x0F.

3.8 SPI (Slave)

This peripheral (formerly available only at [N] without interface) is depreciated. See CustomDpaHandler-UserPeripheral-SPIslave for the implementation that mimics the former embedded SPI peripheral behavior.

3.9 LED

PNUM = 0x06 or 0x07 for standard red respectively green LED at IQRF TR module.

Please note that the LP [N] regularly enters a sleep mode in LP-RX mode when waiting for a packet so the LEDs are switched off. To keep the LED on for some time use LED request together with IO Set request with a delay. Both requests can be stored in one Batch request so the packet will not be received after the LED command.

3.9.1 Peripheral information

PerT `PERIPHERAL_TYPE_LED`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1 `LED_COLOR_*` where * specifies one of the predefined color constants.
Par2 Not used

3.9.2 Set

Controls the state of the LED peripheral.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x06 or 0x07	OnOff	?

OnOff 0x01 to switch LED on, 0x00 to switch LED off

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.9.3 Pulse

Generates one LED pulse using IQRF OS function *pulseLEDx*.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x06 or 0x07	3	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.9.4 Flashing

Enables continuous LED flashing using IQRF OS function *pulsingLEDx*.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x06 or 0x07	4	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.10 IO

PNUM = 0x09

The peripheral is not available at the [C]. It controls the IO pins of the MCU. Please note that the pins used by an internal IQRF TR module circuitry cannot be used and their control by this peripheral is blocked. See a corresponding IQRF TR module datasheet for the IO pins that are available.

3.10.1 Peripheral information

PerT `PERIPHERAL_TYPE_IO`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1 Bitmask specifying user available MCU ports (b0=PORTA, b1=PORTB, …, b7=PORTH)
Par2 Not used

3.10.2 Direction

This command sets the direction of the individual IO pins of the individual ports. Additionally, the same command can be used to set up weak pull-ups at the pins where available. See the datasheet of the PIC MCU for a description of IO ports.

The correctness of the embedded subcommands (port + mask + value) is not checked before but during their execution. If an error is detected during the execution of the subcommands the execution is aborted. If the invalid port number is specified the `ERROR_DATA` is returned. If the length of the very last subcommand is not 3 bytes, the error is not returned.

Request

NADR	PNUM	PCMD	HWPID	0	1	2	…	n × 3	n × 3 + 1	n × 3 + 2
NADR	0x09	0x00	?	$Port_0$	$Mask_0$	$Value_0$	…	$Port_n$	$Mask_n$	$Value_n$

n ∈ [0,17]	Number of subcommands minus 1.
Port	a. Specifies port to setup a direction to. 0x00=TRISA, 0x01=TRISB, …(predefined symbols *PNUM_IO_TRISx*) or b. Specifies port to setup a pull-up. 0x10=WPUA, 0x11=WPUB, … (predefined symbols *PNUM_IO_WPUx*)
Mask	Masks pins of the port.
Value	a. Actual direction bits for the masked pins. 0=output, 1=input., … or b. Pull-up state. 0=disabled, 1=enabled.

Error codes

`ERROR_DATA`	Invalid Port value.

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.10.2.1 Source code support

```
typedef struct
{
  uns8  Port;
  uns8  Mask;
  uns8  Value;
} TPerIOTriplet;

typedef union
{
  TPerIOTriplet Triplets[DPA_MAX_DATA_LENGTH / sizeof( TPerIOTriplet )];
} TPerIoDirectionAndSet_Request;

TPerIoDirectionAndSet_Request _DpaMessage.PerIoDirectionAndSet_Request;
```

3.10.3 Set

[sync] This command sets the output state of the IO pins. It also allows inserting an active waiting delay between IO pins settings. This feature can be used to generate arbitrary time-defined signals on the IO pins of the MCU. During the active waiting, the device is blocked and any network traffic will not be processed.

This command is executed after the DPA Request is sent back to the device that sent the original DPA IO Set request. The correctness of the embedded triplet is not checked before but during their execution. Therefore, if an invalid port is specified an error code is not returned inside DPA Request but the rest of the request execution is skipped. Also when the length of the very last $triplet_n$ is not 3 bytes, the preceding commands are executed and the error is not returned.

Request

NADR	PNUM	PCMD	HWPID	0	1	2	…	n × 3	n × 3 + 1	n × 3 + 2
NADR	0x09	0x01	?	triplet$_0$			…	triplet$_n$		

n ∈ [0,17] Number of triplets minus 1.
triplet There are 2 types of 3-byte triplets (subcommands) allowed:
a. Setting an output value
port Specifies the port to set up an output state. 0=PORTA, 1=PORTB, … (predefined symbols *PNUM_IO_PORTx*)
mask Masks pins of the port to setup.
value Actual output bit value for the masked pins.
b. Delay
0xFF Specifies a delay command (predefined symbol *PNUM_IO_DELAY*).
delayL Lower byte of the 2-byte delay value, unit is 1 ms.
delayH Higher byte of the 2-byte delay value, unit is 1 ms.

Response

The general response to writing request with STATUS_NO_ERROR Error code.

Example 1

The setting of PORTA.0 and PORTC.2 as output, PORTC.3 as input.

- **Request**

`NADR=0x0001, PNUM=0x09, PCMD=0x00, HWPID=0xFFFF, PData={0x00`(PORTA)`, 0x01`(bit0=1)`, 0x00`(bit0=output)`} {0x02`(PORTC)`, 0x0C`(bit2=1, bit3=1)`, 0x08`(bit2=output, bit3=input)`}`

- **Response**

`NADR=0x0001, PNUM=0x09, PCMD=0x80, HWPID=0xABCD, PData={00}`(No error)`, {0x07}`(DPA Value)

Example 2

Set PORTA.0=1, PORTC.2=1, then wait for 300 ms, set PORTA.0=0.

- **Request**

`NADR=0x0001, PNUM=0x09, PCMD=0x01, HWPID=0xFFFF, PData={0x00`(PORTA)`, 0x01`(bit0=1)`, 0x01`(bit0=1)`} {0x02`(PORTC)`, 0x04`(bit2=1)`, 0x04`(bit2=1)`} {0xFF`(delay)`, 0x2C `(low byte of 300)`, 0x01`(high byte of 300)`} {0x00`(PORTA)`, 0x01`(bit0=1)`, 0x00`(bit0=0)`}`

- **Response**

`NADR=0x0001, PNUM=0x09, PCMD=0x81, HWPID=0xABCD, PData={00}`(No error)`, {0x07}`(DPA Value)

3.10.3.1 Source code support

```
typedef struct
{
  uns8  Port;
  uns8  Mask;
  uns8  Value;
} TPerIOTriplet;

typedef struct
{
  uns8  Header;      // == PNUM_IO_DELAY
  uns16 Delay;
} TPerIODelay;

typedef union
{
  TPerIOTriplet Triplets[DPA_MAX_DATA_LENGTH / sizeof( TPerIOTriplet )];
```

```
  TPerIODelay    Delays[DPA_MAX_DATA_LENGTH / sizeof( TPerIODelay )];
} TPerIoDirectionAndSet_Request;

TPerIoDirectionAndSet_Request _DpaMessage.PerIoDirectionAndSet_Request;
```

3.10.4 Get

This command is used to read the input state of all MCU ports (PORTx).

Request

NADR	PNUM	PCMD	HWPID
NADR	0x09	0x02	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … n
NADR	0x09	0x82	?	0	?	Port data

Port data — Array of bytes representing the state of port PORTA, PORTB, …, ending with the last supported MCU port.

3.11 Thermometer

PNUM = 0x0A for standard on-board thermometer peripheral

3.11.1 Peripheral information

PerT — `PERIPHERAL_TYPE_THERMOMETER`
PerTE — `PERIPHERAL_TYPE_READ`
Par1 — Not used
Par2 — Not used

3.11.2 Read

Reads on-board thermometer sensor value.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x0A	0x00	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1	2
NADR	0x0A	0x80	?	0	?	IntegerValue	SixteenthValue	

IntegerValue — Temperature in °C, integer part, not rounded.
See the return value of the *getTemperature* *IQRF* OS function. If the temperature sensor is not installed (see TR Configuration) then the returned value is 0x80 = -128 °C.

SixteenthValue — Complete 12 bit value of the temperature in 1/16 = 0.0625 °C units with 0.5 °C resolution. See the *param3* output value of the *getTemperature* IQRF OS function. If the temperature sensor is not installed the value is undefined.

3.11.2.1 Source code support

```
typedef struct
{
  int8  IntegerValue;
```

```
  int16 SixteenthValue;
} TPerThermometerRead_Response;

TPerThermometerRead_Response _DpaMessage.PerThermometerRead_Response;
```

3.12 PWM

This peripheral (formerly available at demo version only) is depreciated. See UserPeripheral-PWM for using PWM.

3.13 UART

PNUM = 0x0C for embedded UART peripheral

The peripheral is not available at the [C] and at [N] supporting an interface. The size of both TX and RX buffers is 64 bytes.

When UART Peripheral is enabled at the [N] configuration then the UART is automatically opened by the DPA shortly before the Init event is raised. The baud rate is set in the configuration. If this behavior is not intended then the UART can be closed or opened with a different baud rate immediately in the Init event or later as needed.

The usage of the peripheral is limited in LP [Ns] because they regularly sleep in its main receiving loop. The peripheral works only when the device does not sleep or during a time defined by a *ReadTimeout* parameter of a Write & Read command. Please see the details below.

PIC HW UART peripheral interrupts can be handled at the Custom DPA Handler Interrupt event unless the DPA UART peripheral is not open or DPA UART Interface is not used.

For TR-7xG transmitters, TX and/or RX signals can be remapped. By default, the TX and RX signals are mapped to pins RC6 and RC7. The default mapping is applied whenever the UART peripheral (or interface) is opened. This occurs at the following moments:

1. When the DPA is started just before the Init event is called.
2. When the embedded UART peripheral is explicitly opened.
3. When the UART is reopened after it was previously automatically closed before:
   a. Discovery
   b. Sleep
   c. Standby

The easiest way to force a custom mapping of the TX and RX signals is to apply the mapping (if necessary) in the Idle event, as shown in the example:

```
case DpaEvent_Idle:
  if ( RC3PPS != 0x10 ) // TX != RC3 ?
  {
    unlockPPS();
    RC3PPS = 0x10;    // RC3 = TX
    RXPPS = 0b10.100; // RC4 = RX
    lockPPS();
  }
  break;
```

3.13.1 Peripheral information

PerT	PERIPHERAL_TYPE_UART
PerTE	PERIPHERAL_TYPE_READ_WRITE
Par1	Maximum data block length for reading and writing. Currently, it equals 55 bytes.
Par2	Not used

3.13.2 Open

This command opens the UART peripheral at a specified baud rate (predefined symbols *DpaBaud_xxx* can be used in the code) and discards internal read and write buffers. The size of the read and write buffers is 64 bytes.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x0C	0x00	?	BaudRate

BaudRate specifies baud rate:

- 0x00 1 200 Baud
- 0x01 2 400 Baud
- 0x02 4 800 Baud
- 0x03 9 600 Baud
- 0x04 19 200 Baud
- 0x05 38 400 Baud
- 0x06 57 600 Baud
- 0x07 115 200 Baud
- 0x08 230 400 Baud

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

Error codes

`ERROR_DATA` Invalid BaudRate value.

Example 1

Open UART for communication with 9 600 baud rate:

- **DPA Request** (master → slave)
  `NADR=0x0001, PNUM=0x0C, PCMD=0x00, HWPID=0xFFFF, PData={0x03}`(9 600 Baud)
- **DPA Response** (slave → master)
  `NADR=0x0001, PNUM=0x0C, PCMD=0x80, HWPID=0xABCD, PData={0x00}`(No error)`, {0x07}`(DPA Value)

3.13.2.1 Source code support

```
typedef struct
{
  uns8  BaudRate;
} TPerUartOpen_Request;

TPerUartOpen_Request _DpaMessage.PerUartOpen_Request;
```

3.13.3 Close

Closes UART peripheral.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x0C	0x01	?

Response

The general response to writing request with `STATUS_NO_ERROR` Error code.

3.13.4 Write & Read

Writes and/or reads data to/from UART peripheral. If UART is not open, the request fails with `ERROR_FAIL`.

Request

NADR	PNUM	PCMD	HWPID	0	1 … n
NADR	0x0C	0x02	?	ReadTimeout	WrittenData

ReadTimeout Specifies timeout in 10 ms unit to wait for data to be read after data is (optionally) written. 0xFF specifies that no data should be read.
WrittenData Optional data to be written to the UART TX buffer.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … n-1
NADR	0x0C	0x82	?	0	?	ReadData

ReadData Optional data read from UART RX buffer if the reading was requested and data is available. Please note that the internal buffer limits a maximum number of bytes to *PERIPHERAL_UART_MAX_DATA_LENGTH*.

Error codes

`ERROR_FAIL` UART peripheral is not open.

Example 1

Write three bytes (0x00, 0x01 and 0x02) to UART, no reading:

- **DPA Request** (master → slave)
  `NADR=0x0001, PNUM=0x0C, PCMD=0x02, HWPID=0xFFFF, PData={0xff}`(No reading)
  `{0x00,0x01,0x02}`(written data)

- **DPA Response** (slave → master)
  `NADR=0x0001, PNUM=0x0C, PCMD=0x82, HWPID=0xABCD, PData={0x00}`(No error), `{0x07}`(DPA Value)

Example 2

Write three bytes (0x00, 0x01, and 0x02) to UART, read 4 bytes after 10 ms:

- **DPA Request** (master → slave)
  `NADR=0x0001, PNUM=0x0C, PCMD=0x02, HWPID=0xFFFF, PData={0x01}`(10 ms timeout) ,
  `{0x00,0x01,0x02}`(written data)

- **DPA Response** (slave → master)
  `NADR=0x0001, PNUM=0x0C, PCMD=0x82, HWPID=0xABCD,`
  `PData={0x00}`(No error),`{0x07}`(DPA Value),`{0xaa,0xbb,x0cc,0xdd}`(read data)

3.13.4.1 Source code support

```
typedef struct
{
  uns8 ReadTimeout;
  uns8 WrittenData[DPA_MAX_DATA_LENGTH - sizeof( uns8 )];
} TPerUartWriteRead_Request;

TPerUartWriteRead_Request _DpaMessage.PerUartWriteRead_Request;
```

3.13.5 Clear & Write & Read

Same as Write & Read from above except it clears the UART RX buffer at the start and then it executes write and read. Also PCMD = 0x03.

3.14 FRC

PNUM = 0x0D for embedded FRC peripheral.

The peripheral is implemented at the [C] devices only and it is always enabled there regardless of the configuration settings. The peripheral is always disabled at the [N] device.

3.14.1 Peripheral information

PerT `PERIPHERAL_TYPE_FRC`
PerTE `PERIPHERAL_TYPE_READ_WRITE`
Par1 Length of FRC data returned by Send command.
Par2 Not used

3.14.2 Send

This command starts the Fast Response Command (FRC) process supported by IQRF OS. It allows quick and using only one request to collect the same type of information (data length) from multiple Nodes in the network. The type of the collected information is specified by a byte called the FRC command. Currently, IQRF OS allows collecting either 2 bits from all (up to 239) Nodes, 1 byte from up to 63 Nodes (having logical addresses 1-63), 2 bytes from up to 31 Nodes (having logical addresses 1-31), or 4 bytes from up to 15 Nodes (having logical addresses 1-15). The type of collected data is specified by the FRC command value:

Type of collected data	FRC Command interval	Reserved interval	User interval
2 bits	`0x00 - 0x7F`	`0x00 - 0x3F`	`0x40 - 0x7F`
1 byte	`0x80 - 0xDF`	`0x80 - 0xBF`	`0xC0 - 0xDF`
2 bytes	`0xE0 - 0xF7`	`0xE0 - 0xEF`	`0xF0 - 0xF7`
4 bytes	`0xF8 - 0xFF`	`0xF8 - 0xFB`	`0xFC - 0xFF`

When 2 bits are collected, then the 1st bits from the Nodes are stored in the bytes of index 0-29 of the output buffer, 2nd bits from the Nodes are stored in the bytes of index 32-61.

When 1 byte is collected then bytes from each [N] (1-63) are stored in bytes 1-63 of the output buffer.

When 2 bytes are collected then byte pairs for each [N] (1-31) are stored in bytes 2-63 of the output buffer.

When 4 bytes are collected then byte foursomes for each [N] (1-15) are stored in bytes 4-63 of the output buffer.

For more information see IQRF OS manuals. If the [N] does not return an FRC value for some reason, then either returned bits or bytes are equal to 0. This is why it is necessary to code the zero return value into a non-zero one.

The time when the response is delivered depends on the type of the FRC command and used RF mode. Consult IQRF OS guides for the response time calculation and the IQMESH Timing Calculator.

Request

NADR	PNUM	PCMD	HWPID	0	1 … n
NADR	0x0D	0x00	?	FrcCommand	UserData

FrcCommand Specifies data to be collected.

UserData — User data that are available at IQRF OS array variable *DataOutBeforeResponseFRC* at FRC Value event. The length **n** is from 2 to 30 bytes. Unused user data is null when received by the node.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1 … n
NADR	0x0D	0x80	?	0	?	Status	FrcData

Status — Return code of the *sendFRC* IQRF OS function (typically number of responding [Ns]). See IQRF OS documentation for more information.

FrcData — Data collected from the Nodes. Because the current version of DPA cannot transfer the whole FRC output buffer at once (currently only up to 55 bytes), the remaining bytes of the buffer can be read by the next described Extra result command.

3.14.2.1 Source code support

```
typedef struct
{
  uns8 FrcCommand;
  uns8 UserData[30];
} TPerFrcSend_Request;

TPerFrcSend_Request _DpaMessage.PerFrcSend_Request;

typedef struct
{
  uns8 Status;
  uns8 FrcData[DPA_MAX_DATA_LENGTH - sizeof( uns8 )];
} TPerFrcSend_Response;

TPerFrcSend_Response _DpaMessage.PerFrcSend_Response;
```

3.14.3 Extra result

Reads remaining bytes of the FRC result, so the total number of bytes obtained by both commands will be a total of 64. It is needed to call this command immediately after the FRC Send command to preserve previously collected FRC data.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x0D	0x01	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … n
NADR	0x0D	0x81	?	0	?	FrcData

FrcData — Remaining FRC data that could not be read by FRC Send command because of DPA data buffer size limitations.

3.14.4 Send Selective

Similar to Send but allows to specify a set of Nodes that will receive the FRC command and return FRC data. Together with Acknowledged broadcast - bits it can be then used to execute a DPA Request at selected Nodes only and get the DPA Confirmation plus one data bit from selected Nodes. Both DPA Request and DPA Response have the same structure as Send except the SelectedNodes field. Also, the length of the UserData field is limited to 25 bytes. When 1 byte or 2 bytes are collected then results from all selected Nodes are adjacent, so there are no gaps filled with 0s for unselected Nodes (unlike

Send command). IQRF OS function *amIRecipientOfFRC* can be used at [N] side to find out if the result value is to be returned.

Request

NADR	PNUM	PCMD	HWPID	0	1 … 30	31 … n
NADR	0x0D	0x02	?	FrcCommand	SelectedNodes	UserData

FrcCommand — Specifies data to be collected.

SelectedNodes — Specifies a bitmap with selected Nodes. Bit_1 of the 1st byte of the bitmap represents [N] with address 1, bit_2 of the 1st byte of the bitmap represents [N] with address 2, …, bit_7 of the 30th byte of the bitmaps represents Nodes with address 239.

UserData — User data that are available at IQRF OS array variable *DataOutBeforeResponseFRC* at FRC Value event. The length of the data is from 2 to 25 bytes. Unused user data is null when received by the node.

Response

See Send DPA Request.

3.14.4.1 Source code support

```
typedef struct
{
  uns8 FrcCommand;
  uns8 SelectedNodes[30];
  uns8 UserData[25];
} TPerFrcSendSelective_Request;

TPerFrcSendSelective_Request _DpaMessage.PerFrcSendSelective_Request;
```

3.14.5 Set FRC Params

Sets global FRC parameters.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x0D	0x03	?	FrcParams

FrcParams — Value corresponding to the parameter of the *setFRCparams* macro defined at *IQRF-macros.h*. See IQRF OS documentation for more details.

- bit.0-2 Reserved
- bit.3 If set, then so-called offline FRC is performed. Offline FRC is missing an individual FRC phase and can be used only with Beaming sensors and repeaters that aggregate data from Beaming sensors. The bit is automatically reset after the FRC is performed or after the startup.
- bit.4-6 Specifies FRC response time i.e. a maximum time reserved for preparing return FRC value. See `_FRC_RESPONSE_TIME_??_MS` constants. The setting is persistent till the next startup.
- bit.7 Reserved

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x0D	0x83	?	0	?	FrcParams

FrcParams — Previous FrcParams value.

3.14.5.1 Source code support

```
typedef struct
{
  uns8 FrcParams;
} TPerFrcSetParams_RequestResponse;

TPerFrcSetParams_RequestResponse _DpaMessage.PerFrcSetParams_RequestResponse;
```

3.14.6 Embedded FRC Commands

There are a few embedded FRC commands. The user can implement a custom FRC command too. See User FRC Codes intervals for allowed custom FRC command values and FrcValue event.

All embedded FRC commands prepare returned FRC values within the shortest predefined FRC response time of 40 ms (corresponds to `_FRC_RESPONSE_TIME_40_MS` constant). Only in the case of Memory read and Memory read plus 1 command the FRC response time depends on the DPA Request that is specified by the user and executed before the FRC value is returned. Event FrcResponseTime is not implemented for embedded FRC commands, therefore, FRC response time returns 0xFF for them.

3.14.6.1 Ping

FRC_Ping = 0x00

Collects bits. This command is used for “pinging” [Ns] by observing bit.0 of the returned value.

3.14.6.2 Acknowledged broadcast - bits

FRC_AcknowledgedBroadcastBits = 0x02

This command except for collecting bits allows executing DPA Request stored at FRC user data after the FRC result is sent back *[sync]*. When the Send Selective request is used, then the DPA Request is executed at selected Nodes only.

Input FCR user data has the following content. Please note that DPA does not check the correct content or length of FRC user data (except maximum FRC user data length of 30 bytes).

0	1	2	3 … 4	5 … length - 1
Length	PNUM	PCMD	HWPID	PData

Length	Total length of FRC user data containing the DPA Request.
PNUM	Peripheral number of executing DPA Request at.
PCMD	Peripheral command.
HWPID	HWPD of the DPA Request.
PData	Optional DPA Request Data.

DPA Request is executed only when HWPID matches the HWPID of the device or *HWPID_DoNotCheck* is specified. In this case, also, the FrcValue event is raised to allow setting resulting FRC bit.1 by the user. The sender’s address of the embedded DPA Request equals 0x00 ([C] address) and the addressee’s address is 0xFF (broadcast address).

Returned FRC value bits:

bit.0	bit.1	Description
0	0	The [N] device did not respond to the FRC command at all.
0	1	HWPID did not match the HWPID of the device.
1	x	HWPID matches the HWPID of the device. Bit.1 can be set by the FrcValue event. In the end, the DPA Request is executed.

Example of FRC user data:

This example will pulse both LEDs after the FRC is collected. To pulse both LEDs by one request a Batch request is used to package individual 2 LED pulse requests into one request.

16{Length}, 2{PNUM=OS}, 5{PCMD=Batch}, 0xffff{HWPID}, [5{LED Request length},7{PNUM=LEDG},3{PCMD=PulseLED}, 0xffff{HWPID}, 5{ LED Request length },6{ PNUM=LEDR},3{ PCMD=PulseLED }, 0xffff{HWPID}, 0{End of Batch}] {PData=Batch PData}

3.14.6.3 Prebonded alive

FRC_PrebondedAlive = 0x03

Collects bits. This command addresses prebonded [Ns] although they all have the same IQMESH temporary address 0xFE. The command assigns (ideally) a unique and imaginary address to each prebonded [N] within the RF reach of the existing network. The address is deterministically computed from the Node's unique MID and a non-zero parameter NodeSeed. The result of this FRC command is a bitmap (1st half of the result containing the bits #0) of the living prebonded [Ns]. The address can be later used with the same NodeSeed value at an FRC command Prebonded memory read plus 1 to read Nodes' MIDs for a subsequent [N] authorization that gives the [Ns] the final unique network addresses. It is necessary to use a different NodeSeed between every use of this FRC command to avoid possible duplication of the generated imaginary [N] addresses.

FRC user data has the following format:

0	1
NodeSeed	0

NodeSeed Non-zero value used to generate (ideally) unique and imaginary addresses.

3.14.6.4 Supply voltage

FRC_SupplyVoltage = 0x04

Collects bits. Returned bits classify an actual supply voltage value into one of three important supply voltage ranges. In general, FRC bit.0 indicates that the supply voltage is out of the optimal recommended range.

Returned FRC value bits:

bit.0	bit.1	Supply voltage [V]	Note
1	0	< 2.97	undervoltage
0	1	>= 2.97 and < 3.39	optimal voltage
1	1	>= 3.39	overvoltage

3.14.6.5 Prebonded memory compare

FRC_PrebondedMemoryCompare2B = 0x05

Collects bits. It returns information whether 2 bytes read from the specified memory address of the prebonded [N] after a provided DPA Request is executed to satisfy the specified condition. Similar to Prebonded memory read plus 1 this command must be also run after Prebonded alive was executed.

Returned FRC value bits:

bit.0	bit.1	Description
0	0	The [N] device did not respond to the FRC command at all.
1	0	The condition was not met.
0	1	The condition was met.
1	1	The FRC command is not supported because of the older DPA version. Make sure to always execute Prebonded alive FRC just before this FRC command in case the network might contain [Ns] with the older DPA

		version to ensure the returned FRC bits is 0b11. Otherwise, the return value is not defined.

FRC user data has the following format:

0	1	3	4 … 5	6 … n
NodeSeed	0	Flags	Value	MemoryRead

NodeSeed Please see the same field at Prebonded memory read plus 1.
Flags bit.0 Specifies the condition between 2 bytes provided by DPA Request at the specified address and Value field.
0: 2 bytes must equal the 2 bytes of the Value field.
1: 2 bytes (unsigned integer) must be greater than or equal to the 2 bytes of the Value field.
bits.1-7 Reserved.
Value Specify 2 bytes value that is compared with the 2 bytes provided by the DPA Request.
MemoryRead Please see the same field at Prebonded memory read plus 1.

3.14.6.6 Temperature

FRC_Temperature = 0x80

Collects bytes. The resulting byte equals the temperature value read by the *getTemperature* IQRF OS function. If the resulting temperature is 0°C, which would normally equal 0, then a fixed value 0x7F is returned instead. This value substitution makes it possible to distinguish between devices reporting 0°C and devices not reporting at all. The device would normally never return a temperature corresponding to the value 0x7F because +127°C is out of the working temperature range.

3.14.6.7 Acknowledged broadcast - bytes

FRC_AcknowledgedBroadcastBytes = 0x81

Collects bytes. The resulting byte equals normally the same temperature value as the Read temperature command, but if this FRC command is caught by the FrcValue event and a nonzero value is stored at *responseFRCvalue* then this value is returned instead of temperature. FRC user data also stores DPA Request to execute after data bytes are collected in the same way as the Acknowledged broadcast - bits FRC command does.

3.14.6.8 Memory read

FRC_MemoryRead = 0x82

Collects bytes. A resulting byte is read from the specified memory address after a provided DPA Request is executed. This allows getting one byte from any memory location (RAM, EEPROM and EEEPROM peripherals, Flash, MCU register, etc.). As the returned byte must not equal 0 there is also a Memory read plus 1 FRC command available.

Input FCR user data has the following content. Please note that DPA does not check the correct content or length of FRC user data. A batch request is not allowed to be a DPA Request being executed. Specified DPA Request is executed with an HWPID the [N] has.

0 … 1	2	3	4	5 … 6 - Length
Memory address	PNUM	PCMD	Length	PData

Memory address Memory address to read the byte from.
PNUM Peripheral number of executing DPA Request at.
PCMD Peripheral command.
Length Length of the optional DPA Request data.
PData Optional DPA Request Data.

Example 1

This example reads the OS version. OS Read DPA Request will be executed and then a byte from *_DpaMessage.PerOSRead_Response.OsVersion* variable (the request stores the result/response there) will be returned. The actual address of this byte is 0x4A4. See *.h* or *.var* files for details.

FRC command = FRC_MemoryRead = 0x82
Memory address = 0x4A4
PNUM = PNUM_OS = 0x02
CMD = CMD_OS_READ = 0x00
Length = 0 = No data bytes
PData *none*

Example 2

This example reads the value of the IQRF OS *lastRSSI* variable. Dummy OS Read_Get DPA Request will be executed and then a byte from the *lastRSSI variable* will be returned. The actual address of this variable is 0x5B6. Open a generated *.var* file of any IQRF compiled project to find out an address of a system variable.

FRC command = FRC_MemoryRead = 0x82
Memory address = 0x5B6
PNUM = PNUM_OS = 0x02
CMD = CMD_OS_READ = 0x00
Length = 0 = No data bytes
PData *none*

Example 3

This example reads a lower byte of the HWPID version from more Nodes at once. Peripheral enumeration DPA Request is executed and the result byte is read. Address 0x4A9 points to the lower byte of HWPID. Use an address from range 0x4A7 to 0x4AA to read any byte of HWPID or HWPID version respectively.

FRC command = FRC_MemoryRead = 0x82
Memory address = 0x4A9
PNUM = PNUM_ENUMERATION = 0xFF
CMD = CMD_GET_PER_INFO = 0x3F
Length = 0 = No data bytes
PData *none*

Example 4

This example returns a supply voltage level using an embedded OS Read command. See *getSupplyVoltage* at IQRF OS Reference Guide for the format of the return value.

FRC command = FRC_MemoryRead = 0x82
Memory address = 0x4A9
PNUM = PNUM_OS = 0x02
CMD = CMD_OS_READ = 0x00
Length = 0 = No data bytes
PData *none*

3.14.6.9 Memory read plus 1

FRC_MemoryReadPlus1 = 0x83

Same as Memory read but 1 is added to the returned byte to prevent returning 0. This means that this FRC command cannot return the 0xFF value.

Example 1

This example returns byte+1 being read from EEPROM peripheral at address 3. EEPROM Read DPA Request will be executed and then a byte from *_DpaMessage.Response.PData[0]* (the request stores the result/response there) will be returned. The actual address of this byte is 0x4A0. See *.h* or *.var* files for details.

FRC command = FRC_MemoryReadPlus1 = 0x83
Memory address = 0x4A0
PNUM = PNUM_EEPROM = 0x03
CMD = CMD_EEPROM_READ = 0x00
Length = 2 = Two data bytes
PData[0] = 3 = Read from EEPROM address 3
PData[1] = 1 = Read one byte from EEPROM

3.14.6.10 FRC response time

FRC_FrcResponseTime = 0x84

Collects bytes. This embedded FRC command is used to find out the FRC response time of the specified user FRC command. This is useful when a network consists of devices with different devices implementing the same user FRC command but in a different way that might result in different FRC response times. In this case, it is necessary to specify the maximum FRC response time that has any [N] from the set of Nodes that will receive the specified FRC command. This FRC command raises the FrcResponseTime event where a user code returns the time. The returned time value equals the value of the corresponding `_FRC_RESPONSE_TIME_??_MS` constant (see IQRF-macros.h) with the lowest bit set (internally by DPA) to prevent returning zero value. If the specified FRC command is not supported (i.e. FrcResponseTime event is not handled) returned value is 0xFF.

Input FRC user data has the following format:

0	1
FRCcommand	0

FRCcommand Value of the user FRC command to read FRC response time of.

3.14.6.11 Test RF Signal

FRC_TestRFsignal = 0x85

Collects bytes. This embedded FRC command tests the RF signal at the given channel using the given RX filter. The command counts and returns the value of *checkRF* IQRF OS function calls returning TRUE during the currently used FRC response time interval. The counter starts initiated with value 1. If the final counter value is less than 128 (0x80 hexadecimal), the unchanged counter value is returned. If the final counter value is greater or equal to 128, then the counter value is divided by 128 and the division byte result with MSB (7th bit) set is returned. So the MSB of the return FRC value is used to find out whether the resolution is fine (1 count) or coarse (128 counts) respectively.

See the pseudo-code below:

```
setRFchannel( DataOutBeforeResponseFRC[0] /* Channel */ );
uns16 counter = 1;
while ( isFrcResponseTime() )
  if ( checkRF( DataOutBeforeResponseFRC[1] /* RX Filter */ ) )
    counter++;

if ( counter < 0x80 )
  return counter;
else
  return ( counter / 0x80 ) | 0x80;
```

Input FCR user data has the following content:

0	1
Channel	RXfilter

Channel The channel to test.
RXfilter RX filter value passed as a parameter to *checkRF* IQRF OS function. See IQRF OS documentation for more details.
Use value 0xFF to get the data from the previous measurement. This can be used to collect the values measured by all Nodes at the same time by the first use of the command. Next selective FRC with a properly set bitmap will then return the value from next up to 63 Nodes. In this case, the shortest FRC response time of 40 ms can be used, so this procedure ensures the fasted acquisition of the test RF signal data measured at the same time.

3.14.6.12 Prebonded memory read plus 1

FRC_PrebondedMemoryRead4BPlus1 = 0xF8

Collects 4 bytes. The command behaves similarly to Memory read plus 1 but it reads 4 bytes from the specified address of the prebonded [Ns] formerly addressed by the FRC command Prebonded alive. The 4 bytes are treated as *unsigned int32* type value increased by 1 to allow returning a value 0x00000000. Therefore a value 0xFFFFFFFF cannot be read. It is necessary to keep the same NodeSeed value formerly used with Prebonded alive to use the same imaginary Nodes' addresses. This command must be used only as a selective FRC command. The SelectedNodes bitmap equals the result of the Prebonded alive command. This ensures the same and accessible prebonded [Ns] are requested to return their bytes. One call of this FRC command can return up to 15 blocks of 4 bytes. If there are more than 4 bytes to read then just use the Offset parameter increased by 15 from the previous call (start with value 0). This will return next up to 15 MIDs starting from 16th, 31st, … [N] from the bitmap.

FRC user data has the following format:

0	1	2 … n
NodeSeed	Offset	MemoryRead

NodeSeed Non-zero value used to generate (ideally) unique and imaginary addresses. Use the same value as for the previous call of Prebonded alive.
Offset Allows reading next up to 15 MIDs from the addressed [Ns]. Values 0, 15, 30, … are typically used.
MemoryRead This variable-length field specified memory address to read after the specified DPA Requests is executed. It has the same format as FRC user data at the Memory read FRC command. The maximum length is 20 bytes.

3.14.6.13 Memory read 4 bytes

FRC_MemoryRead4B = 0xFA

Collects 4 bytes. The command is similar to Memory read and Memory read plus 1. The resulting four bytes are read from the specified memory address after a provided DPA Request is executed.

FRC user data has the following format:

0	1	2 … n
Inc	0	MemoryRead

Inc 0 The original four-byte value is returned.
1 The original four-byte value is increased by 1 and returned.
other Reserved
MemoryRead This variable-length field specifies the memory address to read after the specified DPA Request is executed. It has the same format as FRC user data at the Memory read FRC command. The maximum length is 20 bytes.

4 TR Configuration

TR configuration is stored in the MCU Flash memory. It is necessary to correctly configure the device before DPA is used for the first time. The configuration can be modified by IQRF IDE using SPI or RFPGM programming, by DPA Service Mode, or by Read TR Configuration, Write TR Configuration, and Write TR Configuration byte commands. There are predefined symbols *CFGIND_???* having the address of each configuration byte.

The following table depicts documented configuration items. Other items are reserved or undocumented. The total size of the configuration block is 32 bytes.

Address	Description
0x00	**Checksum:** The checksum of the TR Configuration block. See Read TR Configuration for details.
0x01 [**]	**Embedded peripherals:** An array of 32 bits. Each bit enables/disables one of the embedded 32 predefined peripherals. Peripheral #0 (Coordinator) is represented by bit 0.0, peripheral #31 (currently not used, but reserved) is controlled by bit 3.7. It does not make sense to enable the peripheral that is not implemented in the currently used device (see Peripheral enumeration).
0x02 [**]	
0x03 [**]	
0x04 [**]	
0x05 [*]	DPA configuration bits #0:
bit 0	**Custom DPA Handler:** If set, then a Custom DPA handler is called in case of an event. The handler can define user peripherals, handle messages to embedded peripherals, and add special user-defined device behavior. If set and the Custom DPA handler is not detected the device indicates an error state. Find more information at the Custom DPA Handler chapter.
bit 1	**DPA Peer-to-peer:** If set, then DP2P is enabled at [N].
bit 2	Reserved.
bit 3	**Routing off:** If set, then the [N] does not route packets in the background.
bit 4	**IO Setup:** If set, then DPA IO Setup is run at an early stage of the module boot time.
bit 5	**User Peer-to-peer:** If set, the device receives also peer-to-peer (non-networking) packets and raises the PeerToPeer event.
bit 6	**Stay awake when not bonded:** If set, then unbonded [N] never enters low power sleep if the button is not pressed.
bit 7	**Network type:** If the bit is set, then the [C] controls the STD+LP network; otherwise, it controls the STD network. The bit can only be changed if the network is empty (no [Ns] are bonded) otherwise the network will stop working.
0x08	**TX power:** RF output power. Valid numbers are 0-7. Setting this item does not have an immediate effect except these moments: 1. at Startup, 2. after discovery (both at [C] and [N]), 3. at DpaApiSetRfDefaults API and 4. after DP2P communication. Use the *setRFpower* IQRF OS function to set the power at runtime.
0x09	**RX filter:** RF signal filter. Valid numbers 0-64. Setting this item does not have an immediate effect except these moments: 1. at Startup, 2. at DpaApiSetRfDefaults API and 3. after DP2P communication. Also, see API variable RxFilter.
0x0A [*]	**LP RX timeout:** Timeout for receiving RF packets at LP-RX mode at LP [N]. The unit is one cycle (one cycle is 46 ms at LP-RX mode). Greater values save energy but might decrease responsiveness to the master interface DPA Requests and also decrease Idle event calling frequency. The valid numbers are 1-255. See also API variable LPtoutRF.
0x0B [*]	**UART baud rate:** Baud rate of the UART interface or the UART peripheral. Uses the same baud rate coding as UART Open (i.e. 0x06 = 57 600 Baud)
0x0C	**Alternative DSM channel:** A nonzero value specifies an alternative DPA service mode channel.
0x0D	DPA configuration bits #1:

bit 0	**Local FRC:** If set, then Local FRC reception is enabled at [N].
bits 1-7	Reserved.
0x11	**RF channel A:** Main RF channel A of the main network. Valid numbers depend on the used RF band. Setting this item does not have an immediate effect on [C] or [N] devices except Startup. Use the *setRFchannel* IQRF OS function to change the RF channel at runtime. When the [N] is bonded using the traditional bonding or the Smart Connect the channel is automatically inherited from the network member that provided the bonding and then written to the configuration.
0x12	**RF channel B:** Same as above but the second B channel.

[*] The device must be restarted for configuration item change to take effect.
[**] Same as [*] but only in the case of _SPI_(Slave)UART embedded peripheral bit.

5 Device Startup

When the device **(1)** is reset it first optionally goes into **(2)** RFPGM mode supposed this mode is (enabled) configured on the OS tab of the TR Configuration dialog box at IQRF IDE. RFPGM mode is terminated depending on its configuration. RFPGM mode is fully controlled by IQRF OS.

Brown-out Reset (BOR) is enabled at TR-7xD transceivers (BOR is always enabled by IQRF OS and automatically disabled at sleep at TR-7xG transceivers) and **(3)** IO Setup is executed if one is enabled.

At the very beginning, it is possible to remotely connect to the device at the so-called **(4)** DPA Service Mode (DSM). A special tool e.g. CATS - DPA Service Tool from IQRF IDE is needed to do it. In the DPA Service Mode, the device can be fully controlled by individual DPA commands regardless of the device configuration so it gives the possibility to update or fix a corrupted device configuration, find out its network address, (un)bond it, find out OS information, reprogram the device, etc. The *DSMactivated* API variable indicates whether DSM was started during device startup. Upon the DSM exit, the device is always reset. The device first tries to establish a DSM session at the fixed channel number 0[*] and then it tries an alternative channel optionally specified at TR Configuration. CATS - DPA Service Tool must be set to use the same required channel for the DSM session.

> [*] Due to local government regulation, devices operating in Israel are distributed with a limitation for a 916 MHz band and channels from 133 to 140 only. Therefore fixed DSM channel is set to 133. Furthermore, TR-77xy devices are technically limited by the SAW filter for the 868 MHz band and channels from 45 to 67, therefore the fixed DSM channel is set to 45.

The user interrupt is enabled, so an Interrupt event can be raised if any interrupt source is enabled from now on.

The bonding or unbonding phase being valid only for [N] devices comes next.

At TR-7xD the bonding or the bond removal (unbonding) at [N] side is initiated and controlled by a “default“ IQRF button connected between the ground and `RB4` MCU pin which is normally available at IQRF development tools. The default behavior can be modified by the implementation of the Reset event that is raised during the bonding and/or unbonding phases. To keep the default behavior but with a custom bonding button, an event BondingButton can be used. See details of the button behavior in the next chapter. Bonding and unbonding at TR-7xG are controlled via DPA Menu.

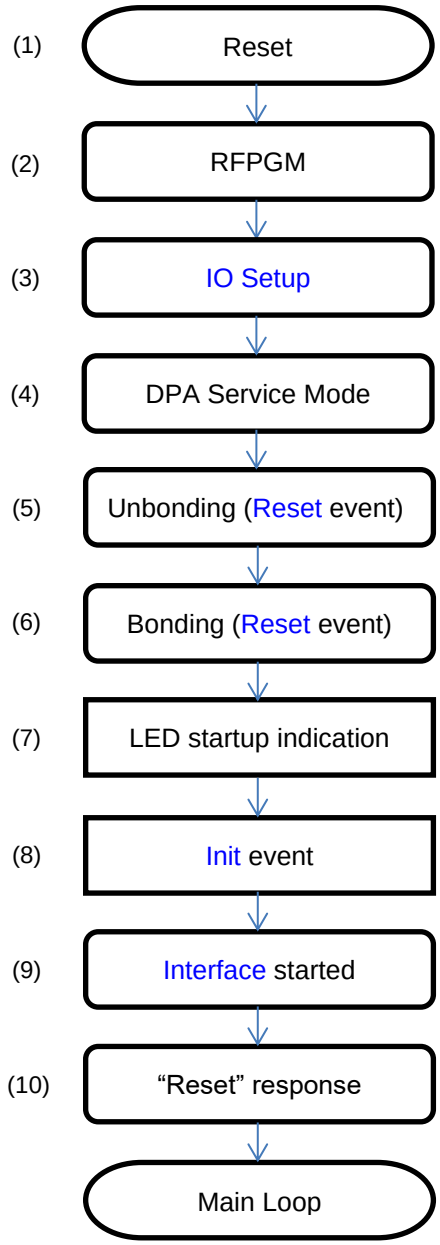
Already bonded [N] can be **(5)** unbonded by a procedure described in the following chapter. If the [N] is not bonded yet then it can be bonded **(6)** by a procedure also described in the next chapter. Please note that the [N] does not have to be configured for a working network RF channel as the channel is automatically inherited from the network member that provided the bonding and then written to the configuration. At this point, the [N] device is bonded and ready to work on the network. This is indicated by a red LED **(7)**.

After that, Init event **(8)** is raised and Interface is started **(9)** (in the case of [N] devices only when the interface is supported).

At **(10)** if the interface is enabled (always at the [C] device) the device (being always slave interface) sends the following asynchronous “Reset” DPA Request equal (except PCMD) Peripheral enumeration response to the interface master. This time the response code is marked by the asynchronous bit `STATUS_ASYNC_RESPONSE`.

NADR	PNUM	PCMD	HWPID	PData
NADR	0xFF	0x3F	?	See DPA Request of Peripheral enumeration

Then the [C] device checks the presence of the connected interface master device during startup. If the data of the “Reset” response are not collected from the interface by the interface master within 100 ms then the device assumes that the interface master is not present. When the interface master is not connected an API variable *IFaceMasterNotConnected* is set to 1.



5.1 Button Handling and LED Indications

Please find below detailed descriptions of the button handling and the LED indication after the device is reset. They are listed in order of appearance.

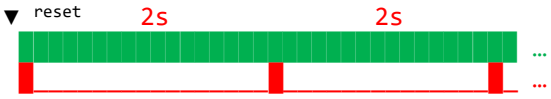
The button is ignored (except RFPGM) at TR-7xG because it is controlled via DPA Menu.

5.1.1 RFPGM

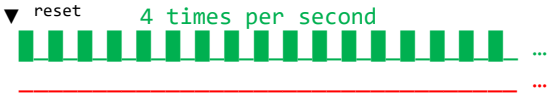
This very first indication is common to both [C] and [N]. The RFPGM indication depends on the RFPGM settings in the device configuration. If RFPGM is enabled, it starts after the device is reset i.e. before [C] or [N] default DPA startup.

A. *RFPGM is not enabled.*
No LED indication.

B. *RFPGM at STD mode.*
The green LED is on and the red LED flashes slowly.



C. *RFPGM at LP mode.*
The green LED flashes quickly.



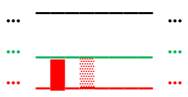
RFPGM is indicated until it is terminated and then the default DPA startup indication starts. It varies for [C] and [N].

5.1.2 Node

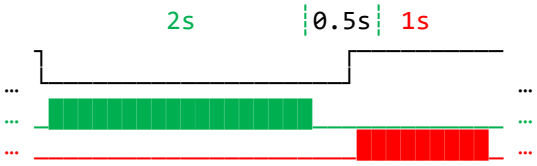
Both the [N] LED indication and button behavior depend on the [N]'s bond state.

5.1.2.1 Bonded Node

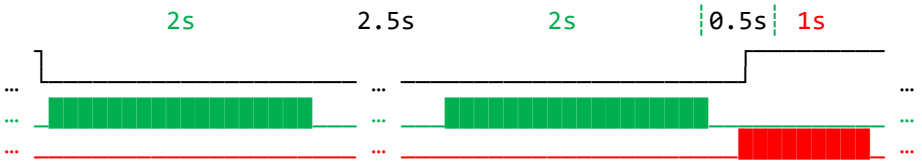
A. *Startup*
If the button is not pressed (at TR-7xD) when the [N] starts, [N] pulses the red LED. The [N] pulses the red LED one more time if [N] has a temporary network address (0xFE).



B. *Unbonding (valid for TR-7xD)*
If the button is already pressed when the [N] starts, then the green LED goes on for 2 seconds while the button is pressed. If the button is released immediately after the green LED goes out within the 0.5-second window, the [N] is unbonded and restarted. The unbonding is confirmed by the red LED for 1 second.

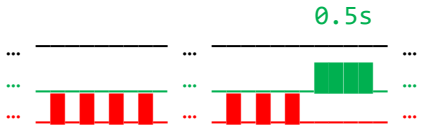


C. *Factory settings (valid for TR-7xD)*
If the button is not released in the previous case after the green LED goes out, the same pattern occurs for the second time after 2.5 s. If the button is then released within the 0.5-second window after the green LED goes out, the factory settings are applied and the [N] is unbonded and restarted. The process is confirmed by the red LED for 1 second.



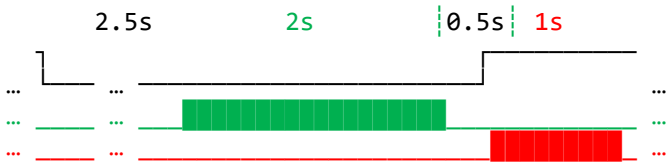
5.1.2.2 Unbonded Node

A. *Smart Connect*
If the button is not pressed, the red LED flashes quickly and the [N] is ready to be bonded using the Smart Connect process (including the Autonetwork process). If Smart Connect bonding is executed, it is indicated by the green (TR-7xD) or red (TR-7xF) LED for 0.5 seconds. If the button is not pressed (while the device operates at LP-RX mode) within approximately 5 hours then the [N] goes into power-saving deep sleep mode and the red LED stops flashing. From the deep sleep mode, the [N] can be woken up by the button press.



B. *Button Bonding (valid for TR-7xD)*
If the button is pressed during Smart Connect bonding (see above), the red LED starts flashing slowly and the [N] continuously requests bonding using *bondRequestAdvanced* IQRF OS function. If bonding is executed, it is indicated by the green LED for 0.5 seconds.

C. *Factory settings (valid for TR-7xD)*
If the button is already pressed when the [N] starts, then after 2.5 s while the button is still pressed the green LED goes on for 2 seconds while the button is pressed. If the button is released immediately after the green LED goes out within the 0.5-second window, the factory settings are applied and the [N] is restarted. The process is confirmed by the red LED for 1 second.



5.1.3 Coordinator

After the [C] starts, it always indicates the Interface state.

A. *The interface is connected.*

B. *The interface is disconnected.*

5.1.4 Custom DPA Handler State

The following indication is common to both [C] and [N]. After they start, there is a Custom DPA Handler availability indication in case the handler is enabled in the configuration.

A. *Custom DPA Handler is present and enabled.*
No LED indication.

B. *Custom DPA Handler is missing though it is enabled.*
Red LED is on until the problem is fixed. Please note, that at LP [N] the red LED goes out periodically during the [N] sleep period.

6 DPA Menu

DPA Menu is implemented at [N] at TR-7xG transceivers. DPA Menu provides uniform and simple device control.

It allows you to perform many useful actions (e.g. bonding, unbonding, factory setting, reset, standby, …). In addition, the menu can be customized and can include user-defined actions. DPA Menu can be activated in any device state.

The DPA Menu is controlled by the default IQRF button connected to the RB4 and active at LOW. The indication is made using a standard red LED at RA2 active at HIGH.

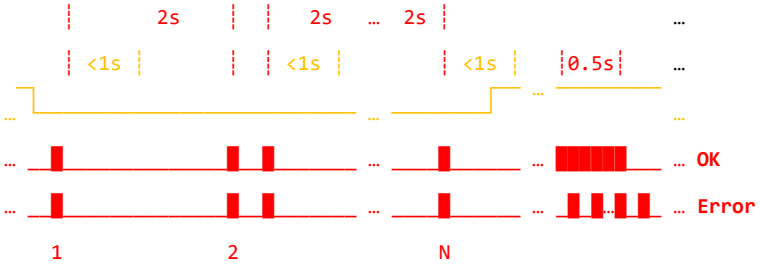
The DPA menu is activated by pressing the default IQRF button. Please note that there may be a delay before the menu is activated as the device may be in the middle of a time-consuming activity (routing, FRC, discovery, ...). Each menu always contains 9 items. As long as the button is pressed, the menu items scroll. Each implemented menu item is indicated by a single blink of the red LED. Unimplemented menu items are indicated by two flashes of the red LED. The time between menu items is 2 seconds. If the end of the menu is reached and the button remains pressed, three red LED flashes are indicated every 2 seconds.

The menu item is selected by releasing the IQRF button within 1 second after the menu item is indicated (by a single blink of the red LED). Some menu items require confirmation. In this case, the red LED will illuminate for 1 second after the menu item is selected. During this time, the IQRF button must be pressed again. The red LED will then remain lit for 1 second. To confirm the selection of an item, you must release the IQRF button within 1 second after the red LED goes off.

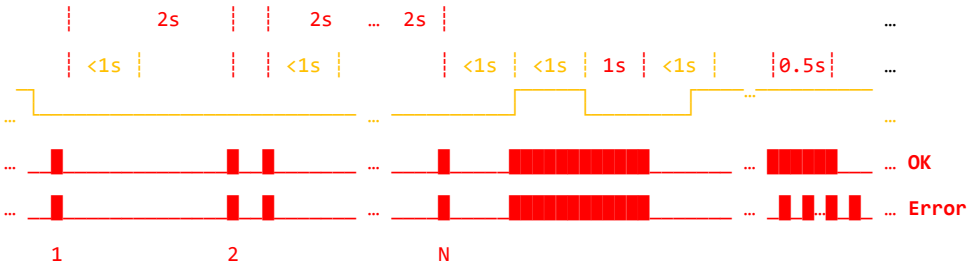
When the menu item is successfully selected, the result will be displayed. The OK result is indicated by the red LED lighting up for 0.5 seconds. In the event of an error, rapid flashing of the red LED is indicated. In addition, some menu items indicate more detailed information by the further flashing of the red LED.

The following diagrams explain menu navigation and an indication of the result. In the example, the 1st and Nth menu items are implemented and the 2nd menu item is not implemented.

- Menu item Nth selection:



- Menu item Nth selection with confirmation:



6.1 Menus

There are four types of menus corresponding to the four device states.

6.1.1 DPA Menu "ReadyToBond"

► When the device is not bonded to the network.

This DPA Menu is always implemented. This device state is indicated by the flashing of the red LED. During this flashing, the device is ready to be bonded via Smart Connect, Autonetwork (also using Smart Connect), or NFC/OTK (e.g. by IQUIP). Some devices might go into sleeping mode if they are not bonded for an extended period. The 1st menu item for this state allows traditional "button" bonding.

From the Custom DPA Handler code point of view, this menu is handled using predefined Menu events. See MenuActivated.

6.1.2 DPA Menu "Online"

► When the device is bonded or prebonded and is receiving IQMESH network traffic.

This DPA Menu is always implemented. From the Custom DPA Handler code point of view, this menu is handled via predefined Menu events. See MenuActivated.

6.1.3 DPA Menu "Beaming"

► When a typically battery-powered sensor device is most of the time in the low power mode and periodically transmits (beams) sensor values using IQRF Standard Sensor.

This DPA Menu is optional. From the Custom DPA Handler code point of view, this menu is handled via a predefined DpaMenu API. See DpaApiMenu.

6.1.4 DPA Menu "StandBy"

► When the device is in very low power mode during storage or transport.

This DPA Menu is always implemented and available after the Standby menu item is selected. The menu cannot be customized.

6.2 DPA Menu Content

General rules:

- The 1st menu item contains the most useful action for the current device state.
- The 2nd menu item is always implemented and allows you to check the status of the device.
- The 7th never-implemented menu item separates the most critical actions at positions 8 and 9.
- Critical actions require confirmation.

Menu Item	Menu			
	ReadyToBond	**Online**	**Beaming**	**Standby**
1	Bond Request	Beaming [1]	Connectivity Check [3]	Exit Standby
2	State Indication (5×)[4]	State Indication (1 or 2×)[4]	State Indication (3×)[4]	State Indication (4×)[4]
3	User1a [1]	User1b [1]	User1c [1]	
4	User2a [1]	User2b [1]	User2c [1]	
5	Standby [2]	Standby [2]	Standby [2]	
6	Reset	Reset	Reset	
7				
8	Restart [2]	Unbond+Restart [2]	Unbond+Restart [2]	
9	Factory Settings+Restart [2]	Unbond+Factory Settings+Restart [2]	Unbond+Factory Settings+Restart [2]	

Legend:

✓	Normal menu item selection
✓✓	Normal menu item selection with confirmation
×	Always unimplemented (empty)

1	Optional (opt-in) and executed in Custom DPA Handler
2	Optionally unimplemented (opt-out)
3	Executed in Custom DPA Handler
4	Number of red LED flashes.

The next chapters describe menu items in more detail.

6.2.1 Bond Request

Executes 10 bonding attempts using *bondRequestAdvanced* IQRF OS function so the total execution might take up to 10 seconds. Between each attempt the red LED flashes. When the button is kept pressed the bonding is aborted and the device continues in ReadyToBond state after the button is released.

6.2.2 Beaming

Starts beaming that must be implemented at Custom DPA Handler. See DPA Menu example.

6.2.3 Connectivity Check

Executes connectivity check at the beaming state. See DpaApiMenu for the reference implementation.

6.2.4 Exit Standby

Exits standby state and indicates the new state i.e. the state before the Standby was selected.

6.2.5 State Indication

A number of the red LED flashes (after the OK indication) indicate the device state:

1×	Online and bonded
2×	Online and prebonded
3×	Beaming
4×	Standby
5×	ReadyToBond

6.2.6 User1 and User2

The execution of these user actions must be implemented in the Custom DPA Handler. See DPA Menu example.

6.2.7 Standby

Switches the device to low power mode. Custom DPA Handler should implement necessary actions before and/or after the standby to achieve the lowest possible power consumption. If the menu is processed using events, the BeforeSleep event is raised before going into standby and the AfterSleep event is raised after standby respectively.

6.2.8 Reset

Resets the device. If the menu is processed using events, the Disable Interrupts event is executed first. After resetting from the Beaming and Online states, the device enters the Online state. Resetting in the ReadyToBond state preserves the ReadyToBond state.

6.2.9 Restart

Just like the Unbond + Restart menu item, except that the device is not unbonded because it is not bonded.

6.2.10 Unbond + Restart

Unbonds and restarts the device. Since the internal microcontroller is not reset, it is necessary to disable all interrupt sources before executing the item. If the menu is processed using events, the Disable Interrupts event is executed before restart. After device restarts it goes into ReadyToBond state.

6.2.11 Factory Settings + Restart

Just like the Unbond + Factory Settings + Restart menu item, except that the device is not unbonded because it is not bonded.

6.2.12 Unbond + Factory Settings + Restart

Same as Unbond + Restart including factory settings.

7 Autoexec

The Autoexec is depreciated from DPA 4.15 because the embedded UART peripheral is automatically opened (since DPA 4.11) at the startup supposed it is enabled in the configuration.

8 IO Setup

IO Setup feature is available at the [N]. It can be used to set up direction, pull-ups, and values of individual IO pins of the MCU at the very beginning of the device startup. Only DPA peripheral IO requests can be executed to make sure the device will always enter DPA Service Mode which can be used to fix an incorrect behavior. Also, every request must use HWPID equal 0xFFFF (*HWPID_DoNotCheck*). IO Setup DPA Requests are stored at external EEPROM memory starting from its physical address *IOSETUP_EEEPROM_ADDR* = 0x0040; the size of the block is 64 bytes. DPA Requests are stored next to each other and are structured according to DPA protocol. There is one exception - the total size of the DPA Request in bytes is stored in the place of a corresponding NADR (in this case, it is only 1 byte wide, not 2 bytes as normal NADR). 0x00 is stored after the very last DPA Request to indicate the end of the IO Setup batch. When executing DPA Request a local interface DPA Notification is not performed although DPA via the interface is enabled. Other events at the user DPA routine are called as usual.

Important: Updating Custom DPA Handler code using the OTA LoadCode command does not allow writing external EEPROM content. Therefore, the update of the IO Setup is not possible. It is recommended to avoid IO Setup when OTA is used.

IO Setup example:

The following example shows the bytes stored in the IO Setup external EEPROM memory space that will run these 2 commands upon the module reset:

1. Sets PORTB.7 (controls green LED) as output
2. Sets green LED on for 1s and then off for 1s

Actual bytes stored at serial EEPROM from address 0x0040:

```
   Len    PNUM   PCMD          HWPID          PData
1. 0x08, 0x09, 0x00(IO Direction), 0xFFFF,       {1,0x80,0x00}(B.7 = output),
2. 0x11, 0x09, 0x01(IO Set),    0xFFFF,       {1,0x80,0x80}(B.7 = 1), {0xff,0xe8,0x03}(1s delay),
{1,0x80,0x00}(B.7 = 0), {0xff,0xe8,0x03}(1s delay),
3. 0x00(end of IO Setup)
```

C code to upload IO Setup example to the external EEPROM:

```
#define NO_CUSTOM_DPA_HANDLER

#include "IQRF.h"
#include "DPA.h"
#include "DPAcustomHandler.h"

#pragma cdata[ __EEESTART + IOSETUP_EEEPROM_ADDR ] = \
8,  PNUM_IO, CMD_IO_DIRECTION, 0xff, 0xff, \
       PNUM_IO_TRISB, 0x80, 0x00, \
17, PNUM_IO, CMD_IO_SET, 0xff, 0xff, \
       PNUM_IO_PORTB, 0x80, 0x80, \
       PNUM_IO_DELAY, 0xe8, 0x03, \
       PNUM_IO_PORTB, 0x80, 0x00, \
       PNUM_IO_DELAY, 0xe8, 0x03, \
0
```

☼ See example code *DpaIoSetup.c* for more details.

9 Custom DPA Handler

Custom DPA handler is an optional user-defined C language routine that can handle various events and thus implements user peripherals, handles embedded peripherals, provides peripheral virtualization, adds internal device logic, customizes DPA Menu, and much more. If the custom DPA handler is implemented it must be enabled in the TR Configuration to receive events.

If the Custom DPA handler is enabled in the TR Configuration but it was not detected (see point 2. below) then the device indicates an error by constant switching on the red LED and by returning the *ERROR_MISSING_CUSTOM_DPA_HANDLER* error code to every DPA Request (except to request to OS peripheral, to request Get information for more peripherals and to all DPA Requests at DPA service mode). In this case, the OS peripheral can be used to fix the problem (disable handler and restart the device or load missing handler already stored in the external EEPROM).

Please respect the following rules when implementing Custom DPA handler:

1. Custom DPA handler must be the first C routine declared as `bit CustomDpaHandler()` in your code. It must be located at the fixed address `CUSTOM_HANDLER_ADDRESS` = `0x3A20` of the MCU Flash memory.
2. The very first instruction of the handler must be CLRWDT to indicate its presence. To do it just insert *clrwdt();* statement right after the handler header. This statement/instruction is thus executed at the beginning of every event (except the Interrupt event).
3. There is an 5344 instructions (TR-7xG) or 864 (TR-7xD) long block in the MCU flash memory reserved for a custom DPA handler in the current version of DPA. See `CUSTOM_HANDLER_ADDRESS_END`.
4. “cases:” for unhandled events do not have to be programmed to save memory space and make the code more readable. Please see Interrupt for an exception from this rule.
5. Variables, as well as function parameters, must be allocated in the standard RAM bank 11 only (48 bytes at range 0x5C0-0x5EF).
6. Variables can be also mapped to the RAM bank 12 that equals the peripheral RAM space (48 bytes at range 0x620 - 0x64F).
7. Do not use *bufferRF*, *bufferCOM,* and *bufferAUX* at all (except inside events Reset, Init, Idle, and DisableInterrupts). *bufferAUX* can be used at the FrcValue event.
8. *bufferINFO* can be used inside events but not to carry data between events as its content can change. *bufferINFO* cannot be used at all when an event is raised during processing IO Set, FRC Send, Get Peripheral Info, or FRC Extra result as these DPA Requests use *bufferINFO* internally.
9. Also, do not use *userReg0* and *userReg1* variables unless you do not call any DPA API function.
10. DPA uses bits 0-1 of the *userStatus* IQRF OS variable internally. Usage of other *userStatus* bits is reserved, therefore their future availability is not guaranteed.
11. Maintain the written code as much speed optimized as possible as the long time spent in the user code might negatively influence device behavior. Especially Interrupt and Idle events must be programmed extremely efficiently.
12. Special attention must be paid to the implementation of an Interrupt event. See details in the dedicated chapter.
13. Do not use timer TMR6 at [C]. Use DpaTicks being internally driven by TMR6 instead.
14. Do not use IQRF OS functions *start[Long]Delay* and *waitDelay* (except locally inside of Reset, Init, Idle, and DisableInterrupts event handlers). Use *waitMS* or TMR6 (but not at the [C] device) instead. Also, IQRF OS functions *startCapture* and *captureTicks* can be used for timing purposes. See IQRF OS documentation for existing side effects.
15. Sending and receiving packets by predefined DPA API functions are allowed only at events Reset, Init, Idle, DisableInterrupts, PeerToPeer, and AfterRouting. It is required to keep the same RF settings (see *setRFpower, setRFchannel, set*mode*, etc. IQRF OS functions) that were set at the beginning of the event upon the event exit.
16. Do not modify the content of IQRF OS variables within the event code. It is required to save their values and restore them at the event exit.
17. Starting from the Init event an MCU watchdog timer with a 4 s period is enabled. Do not change WDT settings. Also, make sure to call *clrwdt()* if needed to prevent WDT reset.
18. If possible, try to avoid executing MCU stack demanding complex requests (e.g. Discovery) from subroutines to prevent MCU stack overflow. Such overflow results in the (often irregular) HW device reset.

19. Both FSR0 and FSR1 point to the message PData at the Custom DPA Handler entry (except Interrupt event). This can be used for code optimization.

A Custom DPA handler can be optionally loaded “over the air” into the device. Please see LoadCode.

9.1 Handler Example

The typical skeleton of the Custom DPA Handler looks like this (see *CustomDpaHandler-Template-Node.c* source code example for a complete template for Node):

```
// Default IQRF include
#include "IQRF.h"

// Uncomment to implement Custom DPA Handler for Coordinator
//#define COORDINATOR_CUSTOM_HANDLER

// Default DPA header
#include "DPA.h"
// Default Custom DPA Handler header
#include "DPAcustomHandler.h"

// Real Custom DPA Handler function
bit CustomDpaHandler ()
{
 // Handler presence mark
 clrwdt();

 // Detect DPA event to handle
 switch ( GetDpaEvent() )
 {
  case DpaEvent_Interrupt:
   // …
   return Carry;

  // Other events …
  case DpaEvent_Idle:
   // …
   return Carry;

  case DpaEvent_DpaRequest:
   if ( IsDpaEnumPeripheralsRequest() )
     // Enumerate Peripherals
     {
       // …
       return TRUE;
     }
   else if ( IsDpaPeripheralInfoRequest() )
     // Get Peripheral Info
     {
       // …
       return TRUE;
     }
   else
     // Peripheral Request
     {
       // …
       return TRUE;
     }
 }

 return FALSE;
}

// Default Custom DPA Handler header
// (2nd include implementing a Code bumper to detect too long code of the handler)
```

```
#include "DPAcustomHandler.h"
```

9.2 Events Flow

The following pseudo-codes illustrate the behavior and raising of events at different device types. A notation *[Event]* specifies that the *Event* is raised.

9.2.1 Coordinator

The pseudo-code applies to the [C] device. For details of the device startup please see a dedicated chapter.

```
if IO Setup enabled
    Run IO Setup

DPA Service Mode
[Reset]
[Init]

Send Reset response to Interface
loop
    if DPA Request packet received from Interface
        if [IFaceReceive]
            Return ERROR_IFACE_CUSTOM_HANDLER to Interface
        else
            if [C] is addressed
                if not [ReceiveDpaRequest]
                    if embedded peripheral
                        Execute embedded DPA peripheral Request
                    else
                        [Handle Peripheral Request]
                    [BeforeSendingDpaResponse]
                Send DPA Response to Interface
                [Notification]
                Execute optional [sync] part of the request
                [AfterRouting]
            else
                Wait for the previous routing timeout to finish
                Send DPA Confirmation to Interface
                Transmit DPA Request packet to the network
                Set routing timeout to the real [C]>[N] plus optimistic [N]>[C] routing

    if packet (typically DPA Response) received from the network
        if not system packet
            if not peer to peer packet
                if not same DPA packet was already received last time
                    if not [ReceiveDpaResponse]
                        Set routing timeout to remaining [N]>[C] routing
                        if [C] addressed
                            if not [ReceiveDpaRequest]
                                if embedded peripheral
                                    Execute embedded DPA peripheral Request
                                else
                                    [Handle Peripheral Request]
                                [BeforeSendingDpaResponse]
                            [Notification]
                            Execute optional [sync] part of the DPA Request
                            [AfterRouting]
                        else
                            Send received packet to Interface
            else
                if peer to peer packet enabled
                    [PeerToPeer]
    else
```

```
        [Idle]
endloop
```

9.2.2 Node

Pseudocode applies to the [N] device. For details about the details of the device startup, see Device Startup.

```
if IO Setup enabled
    Run IO Setup

DPA Service Mode
```

TR-7xD

```
if the [N] is bonded and not [Reset]
    Default unbonding procedure
```

```
while the [N] is not bonded
    if not [Reset]
        Default bonding procedure
```

TR-7xG

```
        DpaMenu(ReadyToBond)
```

```
 [Init]

 Send Reset response to Interface
loop
    if DPA Request packet received from the network
        if not system packet
            if not peer to peer packet
                if not FRC request
                    if not [ReceiveDpaRequest]
                        if embedded peripheral
                            Execute embedded DPA peripheral Request
                        else
                            [Handle Peripheral Request]
                        [BeforeSendingDpaResponse]
                    if packet was not broadcasted
                        Wait for [C]>[N] routing to finish
                        Transmit DPA Response back to the network
                    [Notification]
                    if Interface enabled
                        Send DPA Notification to Interface
                    Wait for [C]>[N] routing to finish
                    Execute optional [sync] part of the DPA Request
                    [AfterRouting]
                else
                    Wait for [C]>[N] routing to finish
                    if LocalFRC and ( LocalFRC not enabled or not [VerifyLocalFrc] )
                        Stop processing FRC
                    if not predefined FRC command
                        [FrcValue]
                    Response FRC value
            else
                if peer to peer packet enabled
                    [PeerToPeer]
    else
        [Idle]
```

TR-7xG

```
        DpaMenu(Online)
```

```
    if local DPA Request packet received from enabled Interface
          if not [ReceiveDpaRequest]
            if embedded peripheral
```

```
                Execute embedded DPA peripheral Request
            else
                [Handle Peripheral Request]
            [BeforeSendingDpaResponse]
        Send DPA Response back to Interface
        [Notification]
        Execute optional [sync] part of the DPA Request
        [AfterRouting]
endloop
```

9.2.3 General events

The next chapters show pseudo-codes illustrating the logic of raising general events at any device where the described evens make sense.

9.2.3.1 Interrupt

An interrupt event is raised whenever an MCU interrupt occurs.

```
if MCU interrupt
    [Interrupt]
```

9.2.3.2 Disable Interrupts

Disable interrupts event is raised at Reset, Restart, LoadCode, Run RFPGM, Remove bond, Factory Settings, and Validate bonds (when [N] is restarted) commands as all of them cause the device to reset or restart. It is also raised at Reset, Unbond + Restart, and Unbond + Factory Settings + Restart.

```
if Reset/Restart/LoadCode/Run RFPGM/Remove bond/Factory Settings/(Validate bonds)
    [Disable Interrupts]
    The device will reset or restart
```

9.2.3.3 Sleep Events

Sleep events (BeforeSleep and AfterSleep) are raised around precise Sleep command.

```
if Sleep
    [BeforeSleep]
    Execute sleep
    [AfterSleep]
```

9.2.3.4 Menu Events

Menu events MenuActivated, MenuItemSelected, and MenuItemFinalize are raised when menus ReadyToBond and Online are handled. See referenced chapters and example *CustomDpaHandler-DpaMenu* for more details.

9.2.3.5 InStandby event

The event is optionally periodically raised during Standby event either activated from Standby menu item or when the [N] is not bonded for certain time at DPA Menu ReadyToBond state. This event allows to optionally specify the watchdog time for its the periodical activation. See also Unbonded Node and BondingSleepCountdown.

9.3 Events

The following paragraphs describe available events in more detail. Unless otherwise specified then the return value from the event does not matter. The code fragments are for illustration purposes only. Please use the C code template and examples distributed with the DPA package instead.

9.3.1 Interrupt

This event is not raised at [C] devices. The event is called whenever an MCU interrupt occurs. Interrupt events might be blocked by IQRF OS during packet reception so the event might not be suitable for high frequency and low jitter interrupts.

Please make sure the following rules are met when implementing an Interrupt event:

1. The time spent handling this event is critical. If there is no interrupt to handle then return immediately otherwise keep the code as fast as possible.
   Make sure the event is the 1st *case* in the main *switch* statement at the handler routine. This ensures that the event is handled as the 1st one.
   This event should be handled with an immediate *return Carry;* even if it is not used by the custom handler because the Interrupt event is raised on every MCU interrupt and the "empty" *return Carry;* handler ensures the shortest possible interrupt routine response time.
2. Only global variables or local ones marked by a keyword *static* can be used to allow reentrancy.
3. Make sure race condition does not occur when accessing those variables at other places.
4. Make sure (inspect .lst file generated by C compiler) compiler does not create any hidden temporary local variable (occurs when using division, multiplication, or bit shifts) at the event handler code. The name of such a variable is usually C*number*cnt. Such hidden variables would cause memory overwrites and code malfunction.
5. Do not call any OS functions except *setINDFx*. Use direct reading by FSRx or INDFx registers instead of calling obsolete and ineffective *getINDFx/readFromRAM* IQRF OS functions.
6. Do not use any OS variables especially for writing access.
7. All the above rules apply also to any other function being called from the Interrupt event handler code, although calling any function from the Interrupt event is not recommended because of additional MCU stack usage that might result in stack overflow and HW device reset.

Example

```
case DpaEvent_Interrupt:

        if ( !TMR6IF )
          return Carry;

        TMR6IF = FALSE;

       // timerOccured is (must be) a global or static variable
        timerOccured = TRUE;

        return Carry;
```

☼ *See example code CustomDpaHandler-Timer.c or CustomDpaHandler-TimerCalibrated.c for more details.*

9.3.2 Idle

This event is periodically raised when the main loop is waiting for incoming RF (or interface) messages to handle. The time spent handling this event is critical. When there is no RF signal then the event is raised in STD mode approximately every 1.0 ms. When there is an RF signal, the time might be up to 2.8 ms.

Note that the frequency at which the event is called depends mainly on the time spent inside the *RFRXpacket* IQRF OS function (used to receive network packets) located in the main DPA loop. In the case when there is a full IQMESH network consisting of 239 devices and the timeslot equals 100 ms, the Idle event might not be called even for 239 × 100 ms = 23.9 s. Even a long time the Idle event is not called can happen during FRC and especially discovery.

Example

```
case DpaEvent_Idle:
        // Go sleep?
        if ( sleepTime != 0 )
        {
              // Prepare OS Sleep DPA Request
              // Time in 2.097 s units
              _DpaMessage.PerOSSleep_Request.Time = sleepTime;
              sleepTime = 0;
              _PNUM = PNUM_OS;
              _PCMD = CMD_OS_SLEEP;
              // LEDG flash after wake up
```

```
            _DpaMessage.PerOSSleep_Request.Control = 0b0100;
            _DpaDataLength = sizeof ( TPerOSSleep_Request );
            // Perform local DPA Request
            // BeforeSleep and AfterSleep events will not be called in this case!
            DpaApiLocalRequest();
        }

        // Return user DPA value
        UserDpaValue = myUserDpaValue;
        return Carry;
```

☼ *See example code CustomDpaHandler-Timer.c, CustomDpaHandler-Coordinator-ReflexGame.c for more details.*

9.3.3 Init

This event is called just before the main loop starts after the Reset event i.e. where the [N] might be (un)bonded. Also, Enumerate Peripherals is called before this event is raised to find out the hardware profile ID (HWPID). This event is typically used to initialize peripherals and global variables. If the initialization is needed as soon as possible and even if the device is not bonded yet then it can be implemented inside the 1st call of a Reset event. Make sure the Init event is processed quickly especially when the [N] was bonded in the Reset event (NodeWasBonded indicates the [N] was just bonded) to process Ping FRC issued by [C] to verify the [N] is bonded.

Example

```
case DpaEvent_Init:
       myVariable = 123;
       T6CON = 0b0.0110.1.00;
       TMR6IE = 1;
       return Carry;
```

☼ *See example code CustomDpaHandler-Timer.c for more details.*

9.3.4 Notification

This event is called when a DPA Request was successfully processed and the DPA Response was sent. DPA Response (but not the original DPA Request) is available at this event. The user can sense what peripheral was accessed and react accordingly. _NADR contains the address of the sender of the original DPA Requests i.e. address to send DPA Request to.

Example

```
case DpaEvent_Notification:
// Anything was written to the RAM?
if ( _PNUM == PNUM_RAM && _PCMD == CMD_RAM_WRITE )
{
       if ( PeripheralRam[0] == 0xAB )
              setLEDR();
       else
              setLEDG();

       ramWritten = TRUE;
}

if ( _PNUM == PNUM_EEPROM && _PCMD == CMD_EEPROM_WRITE )
{
       uns16 someData @ bufferINFO;

       eeReadData( PERIPHERAL_EEPROM_START, sizeof( someData ) );
       if ( someData == 0 )
       {
        // …
       }
}
```

```
return Carry;
```

☼ See example *code CustomDpaHandler-LED-MemoryMapping.c, CustomDpaHandler-PeripheralMemoryMapping.c for more details.*

9.3.5 AfterRouting

[sync] This event is called after the DPA Request was sent and (optional) Notification event and (optional) Interface Notification is sent. In any case, the packet routing of the original DPA Request is finished.

Please note that the RF channel is not defined but if it is changed by the user code (e.g. before calling DpaApiRfTxDpaPacket) its value must be restored. Also, note that the original DPA Request nor Response foursome, as well as DPA data, are not available anymore.

Example

```
case DpaEvent_AfterRouting:
      if ( ramWritten )
      {
            ramWritten = FALSE;
            stopLEDR();
            stopLEDG();
      }
      return Carry;
```

☼ *See example code CustomDpaHandler-PeripheralMemoryMapping.c for more details.*

9.3.6 BeforeSleep

This event is called before the device goes to Sleep mode and at Standby. The code must shut down all HW and MCU peripherals and circuitry not handled by DPA by default. Especially custom handling of SPI and I²C MCU peripherals in a non-DPA way must be handled. Also, to minimize the power consumption, no MCU pin must be left as digital input without a defined input level value. So, unused pins in the given hardware should be set as outputs.

☼ See example code *CustomDpaHandler-Timer.c*.

This event is not implemented at [C].

Example

```
case DpaEvent_BeforeSleep:
      StopMyPeripherals();
      return Carry;
```

☼ *See example code CustomDpaHandler-Timer.c, CustomDpaHandler-UserPeripheral-i2c.c for more details.*

9.3.7 AfterSleep

This event is called after the device wakes up from Sleep mode and at Standby. The event handler is the opposite of the BeforeSleep event handler.

This event is not implemented at [C].

Example

```
case DpaEvent_AfterSleep:
      StartMyPeripherals();
      return Carry;
```

☼ *See example code CustomDpaHandler-Timer.c, CustomDpaHandler-UserPeripheral-i2c.c for more details.*

9.3.8 Reset

The event is called just after the module was reset. It can be used to implement the custom bonding/unbonding of the [N] devices. In this case, the event handler must return TRUE and so the default internal DPA bonding/unbonding code is skipped. If [N] is bonded, the event is raised only once to allow unbonding. If the [N] is not bonded the event is called until the [N] is bonded. The code does not have to handle the setting of the NodeWasBonded variable. For details see Node events flow. See also Init event concerning the initialization options. The interrupt is enabled so the Interrupt event can be already called. [N] devices are set to the Node mode by calling the *setNodeMode* IQRF OS function before this event is raised. After the bonding is done the content of the *bufferRF* must stay intact.

The Reset event is also once raised at the [C] device for the sake of the same behavior of all device types. In this case, it is not used to do bonding or unbonding of course. The [C] devices are at non-network mode because of the previous call of *setNonetMode* IQRF OS function.

Example

```
// Illustrative code
case DpaEvent_Reset:
	if (!doCustomBonding)
		return FALSE;

	if ( amIBonded() )
	{
		if ( unBondCondition )
		{
			removeBond();
			setLEDR();
			waitDelay( 100 );
			stopLEDR();
		}
	}
	else
	{
		while ( !amIBonded() )
		{
			if ( bondRequestCondition )
			{
				bondRequestAdvanced();
				setWDToff();
			}
		}
	}

	return TRUE;
```

☼ *See example code CustomDpaHandler-Bonding.c for more details.*

9.3.9 Disable Interrupts

The event is called when the device needs all hardware interrupts to be disabled. Such a moment occurs at Reset, Restart, LoadCode, Run RFPGM, Remove bond, Factory Settings, and Validate bonds (when [N] is restarted) commands as all of them cause the device to reset or restart. It is also raised at Reset, Unbond + Restart, and Unbond + Factory Settings + Restart.

Example

```
case DpaEvent_DisableInterrupts:
	// ADC Interrupt Enable - off
	ADIE = 0;
	return Carry;
```

☼ *See example code CustomDpaHandler-Timer.c for more details.*

9.3.10 FrcValue

[sync] This event is called whenever the [N] is asked to provide data to be collected by FRC (see Send) and specified FRC Command is not handled by DPA itself (see Predefined FRC Commands). FRC Command value is accessible at the *_PCMD* variable. FRC data to collect must be stored at the *responseFRCvalue* IQRF OS variable. If 2 bytes are collected then the data must be stored at *responseFRCvalue2B* variable instead and at *responseFRCvalue4B* variable when 4 bytes are collected respectively. If bits are collected then only the lowest 2 bits of *responseFRCvalue* are used. Before calling the variables *responseFRC** are prefilled with 1 (except Acknowledged broadcast - bytes).

The code must take less than 40 ms at all Nodes to keep them synchronized (the event is fired at the same time at all Nodes) and to avoid RF collisions. If 40 ms is not enough to prepare data then use Set FRC Params to set a longer time to prepare data for FRC to return.

Important: If the event handler exceeds the selected time then the device does not respond via FRC at all thus “returning” 0 value.

Important: The event is raised even at the Nodes that are not addressed by the current FRC command. IQRF OS function *amIRecipientOfFRC* can be used to find out if the result value is to be returned.

User data passed by Send are accessible at the *DataOutBeforeResponseFRC* IQRF OS variable. This event is implemented at [N] devices only.

Example

```
case DpaEvent_FrcValue:
{
  switch ( _PCMD )
  {
    // This example is sensitive to the bit FRCommand 0x40
    case FRC_USER_BIT_FROM:
      // Return info about power supply voltage
      if ( getSupplyVoltage() < 40 ) // < 3.00 V?
        // Both bits bit0 and bit1 are set now
        responseFRCvalue.1 = 1;
      break;

      // This example is sensitive to the byte FRCommand 0xC0
    case FRC_USER_BYTE_FROM:
      // Just return your logical address as an example
      responseFRCvalue = ntwADDR;
      break;

      // This example is sensitive to the byte FRCommand 0xF0
    case FRC_USER_2BYTE_FROM:
      // Return 2 byte value
      responseFRCvalue2B = Measure2Bytes();
      break;

      // This example is sensitive to the byte FRCommand 0xF8
    case FRC_USER_4BYTE_FROM:
      // Return 4 byte value
      // Use .low16, .high16, … to access this variable at the free CC5X edition
      responseFRCvalue4B = Measure4Bytes();
      break;
  }

  return Carry;
}
```

☼ *See example code CustomDpaHandler-FRC.c for more details.*

9.3.11 FrcResponseTime

This event is raised by predefined FRC response time command. 1st FRC user data byte (i.e. variable `DataOutBeforeResponseFRC[0]`) specifies the value of the user FRC command the FRC response time is requested. The byte return value corresponds to one of the corresponding `_FRC_RESPONSE_TIME_??_MS` constant (see IQRF-macros.h). It is highly recommended to implement this event for every user-defined FRC command. This allows the control system connected to the [C] to find out the longest FRC response time in the network consisting of "unknown" heterogeneous [N] devices. DPA internally sets the lowest bit of the return value to prevent returning zero (equals `_FRC_RESPONSE_TIME_40_MS`) value. If the handler does not handle this event a value 0xFF is returned. The event is raised even at the Nodes that are not addressed by the current FRC response time command. IQRF OS function *amIRecipientOfFRC* can be used to find out if the result value is returned.

Example

```
case DpaEvent_FrcResponseTime:
  switch ( DataOutBeforeResponseFRC[0] )
  {
       case FRC_USER_BIT_FROM + 0:
       case FRC_USER_BIT_FROM + 1:
         responseFRCvalue = _FRC_RESPONSE_TIME_40_MS;
         break;

       case FRC_USER_BYTE_FROM + 0:
         responseFRCvalue = _FRC_RESPONSE_TIME_640_MS;
         break;
  }
  return Carry;
```

☼ *See example code CustomDpaHandler-FRC.c for more details.*

9.3.12 ReceiveDpaResponse

This event is implemented at [C] devices. It is called when a DPA Request packet was received from the network. If the event handler returns TRUE, then further standard DPA Request processing (passing DPA Request to the interface master internally by DpaApiSendToIFaceMaster) is skipped. The event is raised even when HWPID does not match. At this time, system variables RTTSLOT and RTHOPS have valid numbers corresponding to the received DPA Response.

Example

```
case DpaEvent_ReceiveDpaResponse:
 {
       // This example just for demonstration purposes consumes any
       // DPA Request CMD_LED_PULSE at peripheral PNUM_LEDG and pulses LEDR locally
       if ( _PNUM == PNUM_LEDG && _PCMD == ( CMD_LED_PULSE | RESPONSE_FLAG ) )
       {
              pulseLEDR();
              return TRUE;
       }

       return FALSE;
 }
```

☼ See example code *CustomDpaHandler-Coordinator-PollNodes.c for more details.*

9.3.13 IFaceReceive

This event is implemented at the [C] device. It is called when a DPA Request packet was received from the interface master. If the event handler returns TRUE, then further standard DPA Request processing (sending DPA Confirmation back to the interface master, passing DPA Request to the network internally by DpaApiRfTxDpaPacketCoordinator) is skipped. In this case, the interface master receives an error

DPA Request with *ERROR_INTERFACE_CUSTOM_HANDLER* Response Code. The event is raised even when HWPID does not match.

Example

```
case DpaEvent_IFaceReceive:
 {
       // This example just for demonstration purposes consumes any DPA Request
       // CMD_LED_PULSE at peripheral PNUM_LEDR and pulses LEDG locally
        if ( _PNUM == PNUM_LEDR && _PCMD == CMD_LED_PULSE )
        {
              pulseLEDG();
              return TRUE;
        }

        return FALSE;
 }
```

9.3.14 ReceiveDpaRequest

The event is called when a DPA Request (except Get information for more peripherals and Remove bond) is received from the network or from the interface master (if applicable). If the event handler returns TRUE, then the request is not passed to the default handling by the DPA Request event. In this case, the programmer is fully responsible for preparing a valid DPA Response that will be returned to the device that sent the original DPA Request. Also, the BeforeSendingDpaResponse event is skipped. The event is raised even when HWPID does not match.

Example #1

```
case DpaEvent_ReceiveDpaRequest:
// Returns error when there is an attempt to write to the address 0 of RAM peripheral
if ( _PNUM==PNUM_RAM && _PCMD==CMD_RAM_WRITE && _DpaMessage.MemoryRequest.Address==0)
{
       _PCMD |= RESPONSE_FLAG;
       DpaApiSetPeripheralError( ERROR_FAIL );
       return TRUE;
}

return FALSE;
```

Example #2

```
case DpaEvent_ReceiveDpaRequest:
// Do not allow DPA Request from Interface
if ( TX == LOCAL_ADDRESS )
{
       _PCMD |= RESPONSE_FLAG;
       DpaApiSetPeripheralError( ERROR_NADR );
       return TRUE;
}

return FALSE;
```

Example #3

```
case DpaEvent_ReceiveDpaRequest:
// Beaming packet received and beaming command we understand?
if ( !_ROUTEF &&
     _PNUM == PNUM_STD_SENSORS &&
     _PCMD == ( PCMD_STD_SENSORS_READ_TYPES_AND_FRC_VALUES | RESPONSE_FLAG ) )
  {
       …
  }
```

```
return FALSE;
```

☼ See example codes *CustomDpaHandler-PeripheralMemoryMapping.c*, *CustomDpaHandler-HookDpa.c* and *CustomDpaHandler-BeamingAggregation.c* for more details.

9.3.15 BeforeSendingDpaResponse

The event is called when a DPA Response (except a response to Get information for more peripherals) is ready to be returned to the device that sent a DPA Request via a network or from the interface master (if applicable). The event handler can inspect or modify the DPA Response even in the way that the error code is returned.

Example

```
case DpaEvent_BeforeSendingDpaResponse:
 // Always adds one more read byte from EEEPROM peripheral and sets it to 0x55
 if ( _PNUM == PNUM_EEEPROM && _PCMD == CMD_RAM_READ )
 {
       _DpaDataLength++;
       FSR0 = _DpaMessage.Response.PData + _DpaDataLength - 1;
       setINDF0( 0x55 );
 }

 return Carry;
```

Example

```
case DpaEvent_BeforeSendingDpaResponse:
// This example hides even enabled and implemented PNUM_IO peripheral
if ( IsDpaEnumPeripheralsRequest() )
 _DpaMessage.EnumPeripheralsAnswer.EmbeddedPers[ PNUM_IO/8 ] &= ~( 1 << ( PNUM_IO % 8 ) );
else
 if ( _PNUM == PNUM_IO && _PCMD == CMD_GET_PER_INFO )
  _DpaMessage.PeripheralInfoAnswer.PerT = PERIPHERAL_TYPE_DUMMY;
return Carry;
```

9.3.16 PeerToPeer

When peer-to-peer (non-networking) packets are enabled at TR Configuration then the device raises this event when such a packet is received. Peer-to-peer packets are received by all devices receiving at the same RF channel. The peer-to-peer packets can be used to implement e.g. simple battery-operated remote control device that is not part of the DPA network. It is highly recommended to use additional security techniques (e.g. encryption, rolling code, checksum, CRC) against packet sniffing, spoofing, and eavesdropping. As the peer-to-peer packets are not networked ones, optional addressing (*_DpaParams* DPA variable can be misused for this purpose) must be implemented in a custom way. It is also recommended to use the lowest possible RF output power and listen-before-talk technique to minimize the risk of RF collision that might cause the main network RF traffic to fail. The following minimalistic examples show only the basic usage.

Example - Transmitter

```
// Set RF mode to STD-TX
setRFmode( _TX_STD );
// Prepare default PIN
PIN = 0;
// Prepare "DPA" peer-to-peer packet

// DPA packet fields will be used
_DPAF = 1;
// Fill in PNUM and PCMD
_PNUM = PNUM_LEDG;
_PCMD = CMD_LED_PULSE;
// No DPA Data
_DpaDataLength = 0;
// Transmit the prepared packet
```

```
RFTXpacket();
```

Example - Handler

```
case DpaEvent_PeerToPeer:
  // Peer-to-peer "DPA" packet?
  if ( _DPAF )
    // Just execute the DPA Request locally
    DpaApiLocalRequest();

  return Carry;
```

☼ See example *code Peer-to-Peer-Transmitter.c, CustomDpaHandler-Peer-to-Peer.c, CustomDpaHandler-PIRlighting.c for more details.*

9.3.17 UserDpaValue

This event is raised whenever DPA is internally required to return user-defined DPA value in the DPA Response. This event is raised the very last time when it is necessary to fill in the UserDpaValue variable but the user can also fill in this variable at any other event before and ignore this event.

Example

```
case DpaEvent_UserDpaValue:
  UserDpaValue = myDpaValue;
  return Carry;
```

9.3.18 BondingButton

This event is called only at TR-7xD [N] during the standard DPA (un)bonding process at [N] and it allows to redefine (un)bonding button. If the event handler returns FALSE the default button is used. If the event handler returns TRUE then the bit at *userReg1.0* specifies whether the used bonding button is pressed or not. When a custom button is used then the [N] does not go into a power-saving sleep mode during bonding. IQRF OS function *amIBonded* can distinguish between bonding and unbonding.

Since [N] spends most of its time in the LP reception during bonding, interrupts cannot be raised even though they are enabled. This can be solved by adding a 10 ms delay by calling *waitMS* to this event. Such a delay does not block the SmartConnect bonding and allows enough interrupts (e.g., from the custom UART interrupt handler) to be received during this delay.

This event is also used to modify a default bonding button timeout using the BondingSleepCountdown variable.

Example

```
case DpaEvent_BondingButton:
  userReg1.0 = 0;
  if ( !PORTA.0 )
    userReg1.0 = 1;
  return TRUE;
```

☼ *See example CustomDpaHandler-BondingButton.c for more details.*

9.3.19 Indicate

[sync] This event is raised at [N] before the embedded *[sync]* Indicate command is executed i.e. after the IQMESH routing is finished. When the event handler returns FALSE, the default indication is processed.
The event handler may return TRUE to implement a custom indication according to the Control byte of the Indicate command passed at the *userReg1* variable (only valid bits of Control byte are passed) and to skip the default device indication. Please see this example for the recommended implementation. If the device indication is to be off when the device sleeps, the BeforeSleep event must be handled too.

☼ *See example CustomDpaHandler-CustomIndicate.c for more details.*

9.3.20 VerifyLocalFrc

This event is used to verify the Local FRC command received at the [N] (sometimes called actuator). The event is raised only when the Local FRC is enabled at the configuration. The execution of the FRC command continues only if the function returns TRUE. The event typically checks the following variables to verify the received FRC command:

- *TX* = address of the [N] (sometimes called a controller) that sent the local FRC command.
- *_PCMD* = FRC command value.
- *DataOutBeforeResponseFRC* = FRC user data that were stored in the *DataInSendFRC* at controller's side.

☼ *See example CustomDpaHandler-LocalFRC.c for more details.*

9.3.21 MenuActivated

This event is raised only at TR-7xG [N] when Online, ReadyToBond or StandBy DPA Menus are about to be activated. The *userReg1* variable at the entry is the menu to be activated (`DMENU_Online`, `DMENU_ReadyToBond` or `DMENU_StandBy`). On exit *userReg1* variable must contain flags for enabling or disabling implementation of the optional menu items (or 0, if menu is not customized). See DpaApiMenu parameter flags for details. The event must return TRUE to be processed, otherwise the menu is not customized. Please note that StandBy DPA Menu cannot be customized.

The event may return FALSE and *userReg1* = `DMENU_MenuActivated_DoNotOpen` to disable opening the menu at all.

Example

```
case DpaEvent_MenuActivated:
  switch ( userReg1 )
  {
    case DMENU_Online:
       if ( PeripheralRam[0] )
       {
         userReg1 = DMENU_MenuActivated_DoNotOpen;
         return FALSE;
       }

      userReg1 = DMENU_Item_Implemented_GoBeaming | DMENU_Item_Implemented_User1;
      return TRUE;

    case DMENU_ReadyToBond:
      userReg1 = DMENU_Item_Unimplemented_UnbondFactorySettingsAndRestart;
      return TRUE;

    case DMENU_StandBy:
       if ( PeripheralRam[1] )
         userReg1 = DMENU_MenuActivated_DoNotOpen;
       return FALSE;
  }

return FALSE;
```

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.3.22 MenuItemSelected

This event is raised only at TR-7xG [N] when a menu item from Online or ReadyToBond DPA Menus was selected. The *userReg1* variable at the entry contains menu and menu item, that was selected. If the user item is optional and implemented in the Custom DPA Handler the event must return TRUE to indicate OK result of the menu item, otherwise it must return FALSE to indicate error. Predefined macro `MakeDMenuAndItem` constructs menu&menuItem value. Predefined macros `GetDMenu` and `GetDMenuItem` get menu or menu item part from the menu&menuItem value respectively. Menu item

action might be executed at this event to find out the its result or it can be executed at the MenuItemFinalize event later, if the result is known.

Example

```
static bit startBeamingAtIdle;

…

case DpaEvent_MenuItemSelected:
  switch ( userReg1 )
  {
    case MakeDMenuAndItem( DMENU_Online, DMENU_Item_GoBeaming ):
      if ( amIBonded() )
      {
         startBeamingAtIdle = TRUE;
         return TRUE;
      }
      break;

    case MakeDMenuAndItem( DMENU_Online, DMENU_Item_User1 ):
      return TRUE;
  }

return FALSE;
```

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.3.23 MenuItemFinalize

This event is raised only at TR-7xG [N] when a previously selected menu item from Online or ReadyToBond DPA Menu is to be finalized. The *userReg1* variable at the entry contains menu and menu item, that was previously selected. If the user item is implemented in the Custom DPA Handler the event must execute the menu action if it was not already executed at MenuItemSelected event.

Example

```
case DpaEvent_MenuItemFinalize:
  switch ( userReg1 )
  {
    case MakeDMenuAndItem( DMENU_Online, DMENU_Item_User1 ):
      ExecuteMyUser1MenuItem();
      break;
  }

return Carry;
```

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.3.24 InStandby

This event is raised only at TR-7xG [N]. Allows you to optionally set the value of the watchdog timer (WDT) so that the event is periodically triggered in the Standby state. The event is always triggered at the beginning of the standby mode. The event stores the WDT value in the *userReg1* variable and returns TRUE. If the event returns FALSE (i.e. it is not implemented), the WDT is disabled and the standby mode can only be interrupted by the IQRF button. If a WDT time has been set and an interrupt from WDT has occurred, the event is periodically called again. Reentrancy of the Custom DPA handler may occur because this event can be triggered in a standby state activated from the DPA Menu Beaming, which is normally handled in an Idle event. Therefore, local variables must not be used in event processing, but global or local static variables. No interrupt-dependent routines (e.g., from a timer) must be used because interrupts are disabled. Because of the standby mode, power consumption must of course be minimal.

Example

```
static uns8 inStanbyCnt;
…
  case DpaEvent_InStandby:
    // Pulse LEDR or LEDG for 8 ms every 4 s in Standby
    if ( inStanbyCnt.1 )
      _LEDR = 1;
    else
      _LEDG = 1;

    if ( inStanbyCnt.0 )
    {
      _LEDR = 0;
      _LEDG = 0;
      userReg1 = WDTCON_4s;
    }
    else
      userReg1 = WDTCON_8ms;

    inStanbyCnt++;
    return TRUE;

  case DpaEvent_MenuActivated:
    inStanbyCnt = 0;
    break;
```

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.3.25 DPA Request

DPA Requests to peripherals are handled in the same way as the built-in DPA interpreter does it. If DPA Request is passed an event DpaEvent_DpaRequest is signaled.

☼ See example codes *CustomDpaHandler-UserPeripheral???.c* for more details.

9.3.25.1 Enumerate Peripherals

This DPA Request is executed as a part of the peripheral enumeration.

The purposes of the request are:

1. Specify how many user peripherals are implemented.
2. Set bits corresponding to the user peripherals at the `UserPer` array. Predefined macro `FlagUserPer` can be used.
3. If any embedded peripheral is handled by a custom DPA handler instead of the default handler (overriding embedded peripherals).
4. Specify HW profile ID and its version if one is implemented.

Example

```
case DpaEvent_DpaRequest:
if ( IsDpaEnumPeripheralsRequest() )
{
  // One user peripheral defined
  _DpaMessage.EnumPeripheralsAnswer.UserPerNr = 1;
  FlagUserPer( _DpaMessage.EnumPeripheralsAnswer.UserPer, PNUM_USER );
  // We override embedded EEEPROM peripheral
  _DpaMessage.EnumPeripheralsAnswer.DefaultPer[PNUM_EEEPROM/8] |= 1 << (PNUM_EEEPROM % 8);
  // HW profile ID and version
  _DpaMessage.EnumPeripheralsAnswer.HWPID = 0x123F;
  _DpaMessage.EnumPeripheralsAnswer.HWPIDver = 0xABCD;

  return TRUE;
}
```

9.3.25.2 Get Peripheral Info

If the user code handles user peripherals or overrides embedded peripherals then this request is used to return information about the peripheral in the peripheral information format. If the handler does not handle the DPA "Get peripheral info request" then it must return FALSE to indicated error, otherwise, it must return TRUE.

Example

```
case DpaEvent_DpaRequest:
…
else if ( IsDpaPeripheralInfoRequest() )
{
 // 1st user peripheral
 if ( _PNUM == PNUM_USER )
 {
    _DpaMessage.PeripheralInfoAnswer.PerT = PERIPHERAL_TYPE_LED;
    _DpaMessage.PeripheralInfoAnswer.PerTE = PERIPHERAL_TYPE_EXTENDED_READ_WRITE;
    _DpaMessage.PeripheralInfoAnswer.Par1 = LED_COLOR_UNKNOWN;
 }
 return TRUE;
}
```

9.3.25.3 Handle Peripheral Request

This request is sent whenever there is DPA Request for a peripheral that was not handled by the default DPA code. Typically the code handles requests for user peripherals or overridden embedded peripherals. If the handler does not handle the DPA Request then it must return FALSE to indicated error (then DPA Request contains response code ERROR_PNUM), otherwise, it must return TRUE.

Please note how to return an error state in the following code. Set PNUM to *PNUM_ERROR_FLAG*, set 1st data byte of the DPA Request to the error code, set 2nd byte to the original PNUM, and finally specify that the length of the data is 2. The best way is to use a predefined union member at *_DpaMessage.ErrorAnswer*.

If code saving is not an issue or there are just a few error types returned then it is easier to call DpaApiReturnPeripheralError API to return the error state. Otherwise shared (using *goto*) central error point is advised. Both methods can be seen in the code example below.

Example

```
case DpaEvent_DpaRequest:
…
else if ( IsDpaPeripheralInfoRequest() )
 // …
else
{
 // 1st user peripheral
 if ( _PNUM == PNUM_USER )
 {
  // Test for some data sent
  if ( DpaDataLength == 0 )
  {
   // Return error ERROR_DATA_LEN
   // DpaApiReturnPeripheralError(ERROR_DATA_LEN); is the easiest way
   _DpaMessage.ErrorAnswer.ErrN = ERROR_DATA_LEN;
UserErrorAnswer:
   _DpaMessage.ErrorAnswer.PNUMoriginal = _PNUM;
   _PNUM = PNUM_ERROR_FLAG;
   _DpaDataLength = sizeof( _DpaMessage.ErrorAnswer );
   return TRUE;
  }

  if ( _PCMD == 0 )
  {
```

```
   UseDataCmd0(_DpaMessage.Request.PData[0]);
   _DpaDataLength = 0;
   return TRUE;
  }
  else if ( _PCMD == 1 )
  {
   UseDataCmd1(_DpaMessage.Request.PData[0]);
   _DpaMessage.Response.PData[0] = someDataToReturn;
   _DpaDataLength = 1;
   return TRUE;
  }
  else
  {
   // Return error ERROR_PCMD
   // or DpaApiReturnPeripheralError(ERROR_PCMD); is the easiest way
   _DpaMessage.ErrorAnswer.ErrN = ERROR_PCMD;
   goto UserErrorAnswer;
  }
 }

 return TRUE;
}

return FALSE;
```

9.3.25.4 Alternative Event Processing

There is an optimized macro *IfDpaEnumPeripherals_Else_PeripheralInfo_Else_PeripheralRequest()* that saves a code compared to the previous way when detecting various cases of the event. The macro is the DPA version independent.

```
case DpaEvent_DpaRequest:
  // Called to interpret DPA Request for peripherals
  IfDpaEnumPeripherals_Else_PeripheralInfo_Else_PeripheralRequest()
  {
    // Peripheral enumeration
    ...
    return TRUE;
  }
  else
  {
    // Get information about peripheral
    ...
    return TRUE;
  }

  // Handle peripheral command
  ...
  return TRUE;
```

9.4 DPA API

The following functions can be called from the Custom DPA Handler routine. Please note that after calling an API function or after modification of the *userReg0* variable the value of macro *GetDpaEvent()* is undefined.

When any of the API functions is called more than once it is recommended to call a wrapper function instead, that has the same name but is prefixed by an underscore character. This reduces the size of the compiled code.

9.4.1 DpaApiRfTxDpaPacket

```
void DpaApiRfTxDpaPacket( uns8 dpaValue, uns8 netDepthAndFlags )
```

Available at [N] devices. This function wraps all necessary code to send a DPA message (typically DPA Response) from [N] to [C]. There are only a few global parameters or variables that have to be filled in before the call (see example below). Many other parameters are handled inside the function automatically. The following example shows a typical usage. The parameter *dpaValue* specifies a DpaValue that is returned with the DPA Request. Because the message is asynchronous its response code the highest bit is set (see `STATUS_ASYNC_RESPONSE`).

If the [C] is addressed by `COORDINATOR_ADDRESS = 0x00`, then the DPA packet is sent by the addressed [C] to the interface master after it is received.

If the [C] is addressed by `LOCAL_ADDRESS = 0xFC`, then the DPA packet (request) is executed locally at [C].

The usage of the parameter *netDepthAndFlags* is the following. Lower 7 bits specify net depth. Use value 1 if the message should be terminated at the subordinate [C], use value 2 if the message should be terminated at the DPA interface of the same [C] or the [C] above the same [C], etc. If the most significant bit of *netDepthAndFlags* is set then the message is marked as synchronous otherwise as asynchronous.

Calling DpaApiRfTxDpaPacket is allowed only at Idle and AfterRouting events. The function does not take into account any IQMESH timing requirements (e.g. waiting for the end of the routing process) or possible RF signal collision.

It is important to make sure that the PID of the message differs from the previously sent message from the same device with the same PCMD, otherwise, the message is regarded as a duplicate. Please note, that the previous same message might have been sent as an ordinary DPA Response. So it is advised to store the PID of such a response and use a different one then. Please see the very first statement in the example below.

Example

```
// Generate new packet ID to avoid false detection of duplicate packet
PID = ++pid;
// Number of hops = my VRN
RTHOPS = ntwVRN;
// No DPA Params used
_DpaParams = 0;
// Execute DPA Request at Coordinator
_NADR = LOCAL_ADDRESS;
_NADRhigh = 0;
// We will use an LED peripheral
_PNUM = PNUM_LEDR;
// Pulse the LED
_PCMD = CMD_LED_PULSE;
// HW profile ID
_HWPID = 0x1234;
// Length of the data inside DPA Request message
_DpaDataLength = 0;
// Transmit DPA message with DPA Value equal the lastRSSI (can be any other value)
DpaApiRfTxDpaPacket( lastRSSI, 1 );
```

☼ *See example codes CustomDpaHandler-AsyncRequest.c for more details.*

9.4.2 DpaApiReadConfigByte

```
uns8 DpaApiReadConfigByte( uns8 index )
```

This function returns the TR Configuration value from a given index (address). Calling this function does not modify FSRx registers.

Example

```
setRFchannel( DpaApiReadConfigByte( CFGIND_OS_CHANNEL_2ND ) );
```

☼ *See example codes CustomDpaHandler-AsyncRequest.c for more details.*

9.4.3 DpaApiSendToIFaceMaster

```
void DpaApiSendToIFaceMaster( uns8 dpaValue, uns8 flags )
```

Available at [C] and STD [N] with the interface. The function passes the prepared DPA packet (DPA Response) to the interface master. The function sends the DPA packet marked as asynchronous unless bit flags.0 is set.

The [C] device only:
If the interface master was not previously detected, then the call is ignored in the case of the SPI interface. If there is some older data at the interface bus not being collected by the interface master yet then the function waits until the data is read.

Calling DpaApiSendToIFaceMaster is allowed only at Idle, IFaceReceive, and ReceiveDpaResponse events.

☼ *See example codes CustomDpaHandler-Coordinator-FRCandSleep.c, CustomDpaHandler-Coordinator-PollNodes.c for more details.*

9.4.4 DpaApiRfTxDpaPacketCoordinator

```
uns8 DpaApiRfTxDpaPacketCoordinator()
```

Available at [C] devices only. This function is specially prepared for sending DPA Requests from [C] to the [N] devices in its network. It prepares even more of the requested parameters automatically compared to the DpaApiRfTxDpaPacket function. Last but not least it also takes care of waiting to send another DPA Request until the routing of the previously sent (and received) packet is finished thus minimizing the probability of the network collision. The call initializes NetDepth by value 1.
The function returns the number of hops used to deliver the DPA Request from the addressed device back to the [C]. The number of hops used to deliver the DPA Request to the addressee and slot length is available at IQRF OS variables *RTHOPS* and *RTTSLOT* respectively. Thus, the same information (Hops, Timeslot length, Hops Response) as within DPA Confirmation is available to the developer. See also Set Hops.
Calling DpaApiRfTxDpaPacketCoordinator is allowed only at Idle, AfterRouting, and IFaceReceive events.

Example

```
case DpaEvent_Idle:
 {
   // The following block of code demonstrates autonomous once per 60 s sending
   // of packets if the [C] is not connected to the interface master
   if ( IFaceMasterNotConnected && DpaTicks.15 != 0 )
   {
     // Setup new timer
     GIE = 0;
     DpaTicks = 60 * 100L;
     GIE = 1;

     // DPA Request is broadcasted
     _NADR = BROADCAST_ADDRESS;
     _NADRhigh = 0;
     // Use red LED
     _PNUM = PNUM_LEDR;
     // Make a LED pulse
     _PCMD = CMD_LED_PULSE;
     // HW profile ID
     _HWPID = HWPID_DoNotCheck;
     // This DPA Request has no data
     _DpaDataLength = 0;
     // Send the DPA Request
```

```
    DpaApiRfTxDpaPacketCoordinator();
  }

  return Carry;
}
```

☼ *See example codes CustomDpaHandler-Coordinator-PulseLEDs.c for more details.*

9.4.5 DpaApiLocalRequest

```
void DpaApiLocalRequest()
```

Performs a local DPA Request at the embedded peripheral, which is even not enabled in the TR Configuration. Of course, the peripheral must be implemented. After the function returns, a corresponding DPA Request is available except when the original DPA Request was a Batch. Calling DpaApiLocalRequest is allowed at Init, Idle, AfterRouting, BeforeSleep, AfterSleep, PeerToPeer, and DisableInterrupts events. It can be also called carefully inside the Reset event as during the event the device might not be bonded yet, the interface is not started, etc. When a processed DPA message is not destroyed or used later then the function can be carefully used at ReceiveDpaResponse, IFaceReceive, ReceiveDpaRequest, and BeforeSendingDpaResponse events too. To avoid reentrancy no Custom DPA Handler events (except Interrupt event) are called during local DPA Request processing. This is the reason why performing local DPA Request on custom peripherals do not work. Also, when e.g. Sleep request is executed locally, then events BeforeSleep and AfterSleep are not raised (same applies to e.g. Run RFPGM and Disable Interrupts event). As the DPA Request is executed locally there is no need to fill in `_NADR`, `_NADRhigh`, and `_HWPID` variables, see example below. Please note that this call invalidates the value obtained by *GetDpaEvent()* macro later in the current event handling.

Example

```
case DpaEvent_Idle:
  if ( IsSleepTime )
  {
    IsSleepTime = FALSE;
    // Prepare OS Sleep DPA Request
    _PNUM = PNUM_OS;
    _PCMD = CMD_OS_SLEEP;
    _DpaMessage.PerOSSleep_Request.Time = 123;
    _DpaMessage.PerOSSleep_Request.Control = 0b0010;
    _DpaDataLength = sizeof( TPerOSSleep_Request );
    // Perform local DPA Request
    DpaApiLocalRequest();
    // If no error, pulse the LEDR after wake up
    if ( _PNUM != PNUM_ERROR_FLAG )
      pulseLEDR();
  }
 return Carry;
```

☼ *See example code CustomDpaHandler-Coordinator-FRCandSleep.c for more details.*

9.4.6 DpaApiReturnPeripheralError

```
DpaApiReturnPeripheralError ( uns8 error )
```

This is a macro calling internal API *DpaApiSetPeripheralError(error)* to prepare an error DPA Request from the peripheral DPA Request handling code. Then the macro executes `return TRUE or FALSE`.

This simple statement *DpaApiReturnPeripheralError(ERROR_DATA_LEN)* using the macro is fully equivalent to the following lines of code:

```
_DpaMessage.ErrorAnswer.ErrN = ERROR_DATA_LEN;
_DpaMessage.ErrorAnswer.PNUMoriginal = _PNUM;
_PNUM = PNUM_ERROR_FLAG;
_DpaDataLength = sizeof( _DpaMessage.ErrorAnswer );
```

```
return Carry;
```

The user peripheral can return user error codes. Such code values must lie between `ERROR_USER_FROM` and `ERROR_USER_TO`. See Response Codes.

☼ *See example codes CustomDpaHandler-UserPeripheral.c for more details.*

9.4.7 DpaApiSetRfDefaults

```
void DpaApiSetRfDefaults()
```

Sets the following default RF settings according to the IQRF OS and TR Configurations and a current DPA RF mode:

- RF filter value,
- RF mode,
- RF power value and
- RF channel value.

This function is typically called when some RF setting was altered or when IQRF OS function *wasRFICrestarted* returns TRUE.

9.4.8 DpaApiLocalFrc

```
uns8 DpaApiLocalFrc ( uns8 frcCommand, uns8 replyTxPower )
```

Available at [N] devices only. This function executes a selective Local FRC. Parameter `frcCommand` specifies the FRC command value. It can be an embedded FRC command or a custom one. The parameter `replyTxPower` specifies the RFoutput power used by the addressed [Ns] FRC responses to control RF interference. Addressed [Ns] are selected by individual bits in the 30-byte long bitmap at the *bufferINFO* array (see *SelectedNodes* parameter for more information). Fill in *DataInSendFRC* with the FRC user data according to the actual FRC command used. The function's return value (typically number of responding [Ns]) is the FRC Status (see Send response) and *param2* contains the number of addressed [Ns]. All FRC values collected from the addressed [Ns] are then available *bufferINFO* array.

Local FRC is an extensive stack operation. You can use a macro *DpaApiLocalFrc_StackSaver* instead to prevent a stack overflow. Please note that the return value is available in register W and extra 2 instructions are emitted.

```
DpaApiLocalFrc_StackSaver( FRC_CMD, TX_POWER_MAX );
if ( W != 0 )
{
  …
```

☼ *See example CustomDpaHandler-LocalFRC.c for more details.*

9.4.9 DpaApiCrc8

```
uns8 DpaApiCrc8 ( uns8 crc8, uns8 data )
```

Available at [N] devices only. The function computes and returns a new `crc8` value by applying `data` value. This CRC function uses the same polynomial as UART Interface. Calling this function does not modify FSR0 and FSR1L registers, but modifies the FSR1H register.

9.4.1 DpaApiAggregateFrc

```
void DpaApiAggregateFrc ()
```

Available at [N] devices only. This function is used to initiate FRC value aggregation at the end of the FrcValue event handler. The calling device must be discovered and a feature FRC Aggregation must be enabled from the transceiver manufacturer during transceiver's production.

The FRC aggregation is used to force returning FRC values from devices other than the current one. Other devices do not have to be even alive nor discovered at the time of the FRC request. Typically, this is used to provide data received asynchronously (in non-network mode) from battery sensors (i.e. other devices) that are in sleep mode most of the time. The data received from the sensors are stored and then later aggregated in the case of an FRC request and thus they appear in the Send FRC response in the same way as if the sensor devices were online.

The FRC aggregation consists of the following steps:

1. Catch FrcValue event in the Custom DPA Handler.
2. Check for the supported FRC command(s).
3. Clear aggregation *bufferINFO.*
4. Loop all addressed Nodes.
5. If the addressed node FRC value should be aggregated then store the value at aggregation buffer at the same place the data would appear in the Send FRC response.
6. Call `DpaApiAggregateFrc()`.
7. Exit FrcValue event handler.

☼ *Please see CustomDpaHandler-FrcAggregation.c and CustomDpaHandler-BeamingAggregation.c examples for the implementation details.*

9.4.2 DpaApiSetOTK

```
void DpaApiSetOTK ()
```

Available at [N] devices only. This API call sets 16 bytes long OTK (one-time key) stored at the *bufferRF[0…15]* for the next OTK prebonding.

☼ *See example CustomDpaHandler-OTK-Node.c for more details.*

9.4.3 DpaApiSleep

```
void DpaApiSleep ( uns8 wdtcon )
```

Available at [N] devices only. Executes a controlled sleep with the specified watchdog timer setting. If at TR-7xG the parameter is equal to `DpaApiSleep_WdtOff`, then the watchdog timer is disabled and the device can be woken up e.g. by interrupt-on-change event. Such repeated periods of sleep are typically used for low-power offline activities, e.g. beaming, waiting for sensor measurement, etc. Before the 1st call, a variable FirstDpaApiSleep must be set. After the last call the function DpaApiAfterSleep must be called. Please note the interrupts, integrated temperatures sensor, and external EEPROM are disabled after the call. Brown-Out Reset (BOR) is disabled during the execution and enabled on exit at TR-7xD transceivers.

The following example shows a typical use:

```
case DpaEvent_Idle:
  if ( StartSleepMode() )
  {
    FirstDpaApiSleep = TRUE;
    do {
      DpaApiSleep( WDTCON_1s );
    } while ( !WakeUp() );
    DpaApiAfterSleep();
  }
  break;
```

9.4.4 DpaApiDeepSleep

```
void DpaApiDeepSleep ( uns8 wdtcon )
```

Available only at TR-7xG [N]. Same as DpaApiSleep but the device uses extremely power-saving mode. See Sleep for more details.

9.4.5 DpaApiAfterSleep

```
void DpaApiAfterSleep ()
```

Available at [N] devices only. This function must be called after the last call of DpaApiSleep. The function enables interrupts, integrated temperatures sensor, and external EEPROM. Please note the integrated temperature sensor needs a 300 ms delay after calling this function to return a correct temperature value (*getTemperature IQRF* OS function). If DpaApiDeepSleep was called as the very first sleep and RF must be functional, then *setRFready* must be called.

9.4.6 DpaApiI2Cinit

```
void DpaApiI2Cinit( uns8 frequency )
```

This function initializes the I²C bus at master mode. The SCL line is available at GPIO RC3 and SDA line at RC4 respectively at devices based on D series IQRF transceivers. Make sure the pull-ups are connected to both these lines. The parameter *frequency* specifies the required I²C frequency. The parameter is prepared using the *I2CcomputeFrequency* macro.

The following example shows a typical use of I²C functions:

```
DpaApiI2Cinit( I2CcomputeFrequency( 100000 /* Hz */ ) );

_DpaApiI2Cstart( 0b1001011.0 /* MCP9802 8bit write address */ );
DpaApiI2Cwrite( 0 /* pointer: 0 = temperature */ );
_DpaApiI2Cstop();

_DpaApiI2Cstart( 0b1001011.1 /* MCP9802 8bit read address */ );
int16 temperature;
temperature.high8 = _DpaApiI2Cread( 1 );
temperature.low8 = _DpaApiI2Cread( 0 );
_DpaApiI2Cstop();

DpaApiI2Cshutdown();
```

9.4.7 DpaApiI2Cstart

```
void DpaApiI2Cstart( uns8 address )
```

Sends START signal and writes specified 8bit *address* to the I²C bus.

9.4.8 DpaApiI2Cwrite

```
void DpaApiI2Cwrite( uns8 data )
```

Writes specified 8bit *data* to the I²C bus.

9.4.9 DpaApiI2Cread

```
void DpaApiI2Cread( uns8 ack )
```

Reads 8bit *data* from the I²C bus. If bit `ack.0` is set then ACK is sent, otherwise NACK.

9.4.10 DpaApiI2Cstop

```
void DpaApiI2Cstop()
```

Sends STOP signal to the I²C bus.

9.4.11 DpaApiI2CwaitForACK

```
void DpaApiI2CwaitForACK( uns8 address )
```

Waits till ACK is received from the specified I²C slave device. It internally executes the following code:

```
do {
```

```
      DpaApiI2Cstart( address );
      DpaApiI2CwaitForIdle();
    } while ( ACKSTAT && !I2CwasTimeout );
    DpaApiI2Cstop();
```

9.4.12 DpaApiI2Cshutdown

```
void DpaApiI2Cshutdown()
```

This function disables the I²C bus previously initialized by DpaApiI2Cinit.

9.4.13 DpaApiI2CwaitForIdle

```
void DpaApiI2CwaitForIdle()
```

Waits till the I²C bus is not busy.

9.4.14 DpaApiRandom

```
uns8 DpaApiRandom()
```

Generates next value of Random variable and returns its most significant byte. It is only available on [N] devices.

9.4.15 DpaApiMenu

```
uns8 DpaApiMenu ( uns8 menu, uns8 flags )
```

Available only at TR-7xG [N]. The function is used when handling Beaming DPA Menu. The function should be periodically called in the beaming function to catch opening Beaming DPA Menu by pressed button. The return value is selected menu&menuItem from the menu. The parameter `menu` must be `DMENU_Beaming`. The parameter `flags` is used to customize the menu content by ORing predefined constants:

Enabling menu items:

- `DMENU_Item_Implemented_User1`
- `DMENU_Item_Implemented_User2`
- `DMENU_Item_Implemented_GoBeaming`

Adding confirmation to menu items:

- `DMENU_Item_Confirm_User1`
- `DMENU_Item_Confirm_User2`

Disabling menu items:

- `DMENU_Item_Unimplemented_GoStandby`
- `DMENU_Item_Unimplemented_UnbondAndRestart`
- `DMENU_Item_Unimplemented_UnbondFactorySettingsAndRestart`

Example

```
case DpaEvent_Idle:

  if ( !startBeamingAtIdle )
    break;

  startBeamingAtIdle = FALSE;

  // Beaming loop
  FirstDpaApiSleep = TRUE;
  for ( ;; )
  {
    DoBeamingOncePerMinute();

    DpaApiSleep( WDTCON_1s );
```

```
    uns8 menuAndItem = DpaApiMenu( DMENU_Beaming, DMENU_Item_Implemented_User1 );
    switch ( menuAndItem )
    {
      case MakeDMenuAndItem( DMENU_Beaming, DMENU_Item_User1 ):
        DpaApiMenuIndicateResult( TRUE );
        ExecuteMyUser1MenuItem();
        break;

      case MakeDMenuAndItem( DMENU_Beaming, DMENU_Item_ConnectivityCheck ):
        // Stop beaming sleeps
        DpaApiAfterSleep();
        // Voluntary indication of the connectivity check execution
        pulsingLEDR();
        // Repeaters to test
        clearBufferINFO();
        bufferINFO[ 0 / 8 ] = 0b1111.1110;   // 1...7
        // Do the test and result indication
        DpaApiMenuIndicateResult( DpaApiLocalFrc( FRC_Ping, TX_POWER_MAX ) );
        // Continue regular beaming sleeps
        FirstDpaApiSleep = TRUE;
        break;

      default:
        // Stop beaming sleeps
        DpaApiAfterSleep();
        // Execute menu
        DpaApiMenuExecute( menuAndItem );
        // Continue regular beaming sleeps
        FirstDpaApiSleep = TRUE;
        break;

      case MakeDMenuAndItem( DMENU_Beaming, DMENU_Item_None ):
        // No menu item was selected
        break;
    }
  }

break;
```

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.4.16 DpaApiMenuIndicateResult

```
void DpaApiMenuIndicateResult ( uns8 ok )
```

Available only at TR-7xG [N]. The function is used when handling Beaming DPA Menu. The function must be called for the selected menu items that are optional and implemented in the Custom DPA Handler. If the `ok` parameter is zero the error is indicated, otherwise OK is indicated. See example at DpaApiMenu function.

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.4.17 DpaApiMenuExecute

```
void DpaApiMenuExecute ( uns8 menuAndItem )
```

Available only at TR-7xG [N]. The function is used when handling Beaming DPA Menu. The function must be called for the menu items that are executed by DPA. See example at DpaApiMenu function. The parameter `menuAndItem` must have the value of the return value of DpaApiMenu.

☼ *See example CustomDpaHandler-DpaMenu for more details.*

9.5 DPA API Variables

The following variables can be used within a custom DPA handler routine. The variables marked by *[readonly]* are read-only. Writing to these variables will cause incorrect device behavior.

9.5.1 bit IFaceMasterNotConnected

[readonly] Valid at [C] device. Equals 1 when the master interface device was not connected during device startup.

In the case of the SPI interface, it is considered not connected when a Reset DPA Request is not read during the startup process.

In the case of the UART interface, it is considered not connected when there was no DPA message received by the interface yet.

Please note that this flag might become 0 when a master interface device sends some data to the [C] device later. The variable value is valid after the Init event.

☼ *See example CustomDpaHandler-Coordinator-PulseLEDs.c for more details.*

9.5.2 bit NodeWasBonded

Valid at [N] devices. It is set to 1 during Device startup if the [N] was newly bonded.

☼ *See example CustomDpaHandler-Bonding.c for more details.*

9.5.3 bit EnableIFaceNotificationOnRead

Valid at [N] devices. Setting to 1 enables sending DPA Notification to the interface master even in the case of “read-only” DPA Request. The default value is 0.

9.5.4 uns16 DpaTicks

Implemented at [C] device only. The value of this variable is decremented every 10 ms after the Init event. The variable is driven by TMR6 driven by an internal PIC RC oscillator. The variable can be used for the implementation of timing algorithms. As this 2-byte wide variable is modified internally within the CPU interrupt routine the whole (both 2 bytes) variable should be accessed (either read or written) only when an interrupt is disabled to ensure atomic access.

Example

```
case DpaEvent_Idle:
       // Is the timeout over?
       if ( DpaTicks.15 != 0 )
       {
              // Setup new 10s timeout
              GIE = 0;
              DpaTicks = 10 * 100L;
              GIE = 1;
…
```

☼ *See example codes CustomDpaHandler-Coordinator-PulseLEDs.c for more details.*

9.5.5 uns8 LPtoutRF

Valid at LP [N] devices. Timeout when receiving RF packets in LP-RX mode. After a device startup, the variable is filled with a respective value from TR Configuration at index 0x0A. See that chapter for more details.

9.5.6 uns8 ResetType

Identifies the type of reset (stored at *UserReg0* upon module reset). See the Reset chapter at IQRF User's Guide for more information.

9.5.7 bit DSMactivated

Equals 1 if the device was maintained at DPA Service Mode (see Device Startup) when the device was started last time. The variable is set even when DPA Service Mode was terminated by Reset or Run RFPGM commands. The variable is not set when DPA Service Mode was terminated by Power-on Reset.

9.5.8 uns8 UserDpaValue

This variable is used to store user-defined DPA value. See Set DPA Param and UserDpaValue.

9.5.9 uns8 NetDepth

[readonly] This variable is used at the ReceiveDpaResponse event to find out whether the received DPA Response is intended for (terminated at) the current device (*NetDepth* == 1) or is to be forwarded automatically by DPA to the higher network or interface (*NetDepth* >= 2).

☼ See example codes *CustomDpaHandler-Coordinator-PollNodes.c* for more details.

9.5.10 bit LpRxPinTerminate

When set to 1 then LP [N] device discontinues packet reception when MCU pin `PORTB.4` goes low regardless of configuration LP timeout value at index 0x0A. See the *setRFmode* IQRF OS function for more information. Immediately after the packet reception is discontinued the Idle event is raised. The default value is 0.

9.5.11 uns8 RxFilter

A variable used as a filter parameter of the *checkRF* IQRF OS function call at the main message DPA loop. The variable value is read from the RF signal filter item at TR Configuration at the startup and can be carefully modified at the runtime.

9.5.12 uns16 BondingSleepCountdown

This variable can be modified at the event BondingButton (at TR-7xD) or at the event MenuActivated (at TR-7xG) to adjust the time without a pressed standard bonding button before [N] goes into deep sleep mode during bonding. The variable is internally zeroed when the bonding phase is initiated. The variable counts down and when it reaches zero (after it is pre-decremented) then the deep sleep mode is activated. A countdown unit is approximately 290 ms. When the variable is continuously set to 0 then the device will never activate deep sleep mode. Also setting bit.6 at DPA configuration bits avoids sleeping.

Example #1
The following example sets the time before going to sleep to 4 seconds:

```
// Was the BondingSleepCountdown just initiated?
#ifdef DpaEvent_BondingButton
case DpaEvent_BondingButton:
  if ( BondingSleepCountdown == 0 )
#endif

#ifdef DpaEvent_MenuActivated
case DpaEvent_MenuActivated:
  if ( BondingSleepCountdown == 0 )
#endif

    // Yes, set the requested timeout to 4 seconds
    BondingSleepCountdown = 4000 / BONDING_SLEEP_COUNTDOWN_UNIT;
  break;
```

Example #2
The example disables the bonding button timeout at all:

```
#ifdef DpaEvent_BondingButton
case DpaEvent_BondingButton:
```

```
#endif

#ifdef DpaEvent_MenuActivated
case DpaEvent_MenuActivated:
#endif
  BondingSleepCountdown = 0;
  break;
```

9.5.13 uns16 Random

[readonly] This variable contains a non-zero pseudo-random value. It is updated on every event except Interrupt. It is only available on [N] devices. See also DpaApiRandom.

9.5.14 bit AsyncReqAtCoordinator

Valid at [C] devices. When set to 1 then [C] can execute asynchronous DPA Requests received from [N]. The default value is 0.

9.5.15 bit NonroutedRfTxDpaPacket

When set to 1 then only the very next call of DpaApiRfTxDpaPacket sends a non-routed packet. This feature is used for beaming purposes. It is only available at [N] devices.

9.5.16 uns8 DpaValue

[readonly] DPA value from the received packet or just to be sent to the interface.

9.5.17 uns8 I2Ctimeout

Specifies an optional timeout at the I²C bus. The unit is approximately 0.7 ms. The default zero value specifies no timeout. A variable I2CwasTimeout is set when the I²C bus timeout occurs after calling DpaApiI2Cstart, DpaApiI2Cwrite, DpaApiI2Cread, DpaApiI2Cstop, DpaApiI2CwaitForACK, and DpaApiI2CwaitForIdle.

9.5.18 bit I2CwasTimeout

This indicates that I²C bus timeout occurred. See uns8 I2Ctimeout.

9.5.19 bit FirstDpaApiSleep

See DpaApiSleep.

9.6 Examples

Find below a list of all examples. The next chapters describe selected Custom DPA Handler examples in more detail.

CustomDpaHandler-AsyncRequest - Sending asynchronous DPA Request from [N] to the [C].
CustomDpaHandler-Autobond - Autobonding example.
CustomDpaHandler-BeamingAggregation.c - FRC aggregation of the IQRF Standard Sensor data.
CustomDpaHandler-Bonding - Custom bonding.
CustomDpaHandler-BondingButton - Custom bonding button.
CustomDpaHandler-BondingNoSleep - Bonding without sleep.
CustomDpaHandler-Bridge-SPI - Bridging handler to the external device using SPI.
CustomDpaHandler-Bridge-UART - Bridging handler to the external device using UART.
CustomDpaHandler-Buttons - Handling multiple hardware buttons with individual debouncing.
CustomDpaHandler-Coordinator-FRCandSleep - Regular FRC & sleep controlled by the [C].
CustomDpaHandler-Coordinator-PollNodes - Polling data from Nodes by the [C].
CustomDpaHandler-Coordinator-PulseLEDs - Pulsing LEDs at Nodes controlled by the [C].
CustomDpaHandler-Coordinator-ReflexGame - Simple reflex game.
CustomDpaHandler-CustomIndicate - Customized indication.
CustomDpaHandler-DDC-RE01 - DDC-RE01 demo.
CustomDpaHandler-DDC-SE01 - DDC-SE01 demo.
CustomDpaHandler-DDC-SE01_RE01 - DDC-SE01 and DDC-RE01 demo.
CustomDpaHandler-DpaMenu – DPA Menu demo.

CustomDpaHandler-FrcAggregation.c - FRC aggregation example.
CustomDpaHandler-FRC-Minimalistic - The smallest FRC handler.
CustomDpaHandler-FRC - Custom FRC commands.
CustomDpaHandler-HookDpa - Intercepting DPA Requests and Responses.
CustomDpaHandler-LED-Green-On - Diagnostic „green LED ON“.
CustomDpaHandler-LED-MemoryMapping - Mapping LED to the RAM peripheral.
CustomDpaHandler-LED-Red-On - Diagnostic „red LED ON“.
CustomDpaHandler-LED-UserPeripheral - LED user peripheral.
CustomDpaHandler-LocalFRC - Local FRC Controller and Actuator.
CustomDpaHandler-LocalFRC-Controller - Local FRC Controller.
CustomDpaHandler-MultiResponse - Multiple DPA Responses to the one DPA Request.
CustomDpaHandler-OTK-Node - OTK prebonding example.
CustomDpaHandler-Peer-to-Peer - Peer-to-peer receiver.
CustomDpaHandler-PeripheralMemoryMapping - Mapping MCU peripheral to the RAM peripheral.
CustomDpaHandler-PIRlighting - PIR controlled lighting.
CustomDpaHandler-ScanRSSI - RSSI measurement among Nodes.
CustomDpaHandler-SelfLoadCode.c - The handler switches itself to the other handler.
CustomDpaHandler-SensorBeaming.c - Beaming sensor example.
CustomDpaHandler-SPI - Custom SPI Peripheral.
CustomDpaHandler-Template-OptimizedSwitch - Optimized custom DPA Handler template, all events.
CustomDpaHandler-Template-OptimizedSwitch-Coordinator - Same as above but for Coordinator.
CustomDpaHandler-Template-OptimizedSwitch-Node - Same as above but for Node only.
CustomDpaHandler-Template - Custom DPA Handler template, all events listed.
CustomDpaHandler-Template-Coordinator - Custom DPA Handler template but for Coordinator only.
CustomDpaHandler-Template-Node - Custom DPA Handler template optimized but for Node only.
CustomDpaHandler-Timer - Using PIC HW timer.
CustomDpaHandler-TimerCalibrated - Using calibrated PIC HW timer.
CustomDpaHandler-UART - Connecting an external device using an embedded UART peripheral.
CustomDpaHandler-UARTrepeater - Sample UART repeater example.
CustomDpaHandler-UartHwRxSwTx - Software UART TX at embedded peripheral to free PWM pin.
CustomDpaHandler-UserEncryption - AES-128 demonstration.
CustomDpaHandler-UserPeripheral-18B20 - Dallas 18B20 temperature sensor as peripheral.
CustomDpaHandler-UserPeripheral-18B20-Idle - Dallas 18B20 sensor operated in the background.
CustomDpaHandler-UserPeripheral-18B20-Multiple - Multiple Dallas 18B20 sensors as peripheral.
CustomDpaHandler-UserPeripheral-ADC - ADC user peripheral.
CustomDpaHandler-UserPeripheral-HW-UART - User HW UART peripheral.
CustomDpaHandler-UserPeripheral-I2C - User peripheral connected to I^2C.
CustomDpaHandler-UserPeripheral-I2Cmaster - I^2C master peripheral.
CustomDpaHandler-UserPeripheral-McuTempIndicator - Internal PIC temperature indicator.
CustomDpaHandler-UserPeripheral-PWM - PWM user peripheral.
CustomDpaHandler-UserPeripheral-PWMandTimer - PWM user peripheral together with a timer.
CustomDpaHandler-UserPeripheral-SPImaster - User SPI master peripheral.
CustomDpaHandler-UserPeripheral-SPIslave - User SPI slave peripheral.
CustomDpaHandler-UserPeripheral - Basic user peripheral.
CustomDpaHandler-XLPstandBy - Putting [Ns] into XLP “sleep” mode with RF wake up.
DpaIoSetup - IO Setup demonstration.

9.6.1 Bonding

This example for TR-7xD shows how to implement a custom (un)bonding procedure inside the Reset event. The code behaves the same way the default (un)bonding procedure does, except the button is (might be) assigned to the different MCU GPIO pin and the [N] is not put to sleep when the button is not pressed for a longer time. The example supports three bonding types: Smart Connect, and traditional “button” bonding.

→ Self-study tip: Modify the code in the way the [N] requests bonding when the button is pressed only when the [N] does not sense a stronger RF signal thus implementing the List-Before-Talk technique. Hint: Use checkRF IQRF OS function to sense RF signal.

9.6.2 Coordinator-FRCandSleep

This example shows autonomous [C], which regularly sends a predefined FRC command Acknowledged broadcast - bytes to the network. It might become a seed of a sophisticated battery-powered long-life sensor network.

The FRC command serves two purposes. Firstly it reads the temperature value from onboard temperature sensors at the Nodes, which is its default return FRC value. Secondly, it utilizes the acknowledged broadcast feature to put Nodes in the sleep state after they return the temperature value via FRC. The embedded acknowledged DPA Request in the FRC command is an ordinary Sleep command. The [C] performs delay using the DpaTicks API variable including the safety gap after both Send and Extra result commands are executed inside the Idle event handler. Also please note a small delay inside the Init event to allow the external interface master to boot. This is necessary in the case of IQRF gateways.

→ *Self-study tip: Change sleeping time to 2 minutes.*

→ *Self-study tip: Modify the code to return the last RSSI value instead of temperature.*
Hint: You will have to handle the FrcValue event and Acknowledged broadcast - bytes FRC command code.

→ *Self-study tip: Utilize the Coordinator's peripheral RAM for passing a set of Nodes to return the FRC byte value from. This is useful in the case of bigger networks (with the address above 62, see Send).*
Hint: You will have to substitute using Send to Send Selective.

→ *Self-study tip: Modify the code to return the state of the IQRF button.*
Hint: You will have to substitute using Acknowledged broadcast - bytes for Acknowledged broadcast - bits and add a simple FrcValue event handler.

9.6.3 FRC-Minimalistic

This is a truly minimalistic code example. It shows literally at only two lines of C code how to implement a custom FRC command. Its code is `FRC_USER_BIT_FROM = 0x40`. It returns 2nd bit equals 1 if the IQRF button is pressed, otherwise, it returns 0.

Following code extract shows the key part of the handler:

```
if (GetDpaEvent() == DpaEvent_FrcValue && _PCMD == FRC_USER_BIT_FROM && buttonPressed)
  responseFRCvalue.1 = 1;
```

The code checks:
- for event `DpaEvent_FrcValue`,
- for custom FRC command code `FRC_USER_BIT_FROM` and
- for the button being pressed.

If all conditions are met then it sets the 2nd bit returned by FRC to 1. That's all.

→ *Self-study tip: Modify the code in the way the FRC command returns the bit indicating whether the green LED is switched on or off.*

9.6.4 LED-MemoryMapping

The example shows the controlling of physical LEDs at TR by the peripheral RAM. A custom command byte is written to the 1st or 2nd byte of the RAM peripheral controls the red LED or green red respectively. It allows switching LED on, to switch off, to pulse, or to start pulsing.

→ *Self-study tip: Currently the example always controls both LEDs regardless of the part of the RAM peripheral that was written to. Modify the code so it will check the actual byte range written to the RAM peripheral and control the appropriate LED(s) only.*
Hint: Use the ReceiveDpaRequest event to find out the address and length of data written to the peripheral RAM.

9.6.5 PeripheralMemoryMapping

This example implements the bidirectional mapping of several MCU peripherals to the peripheral RAM. It allows controlling LEDs, reading the button's state, and reading temperature values. It utilizes peripheral RAM.

→ *Self-study tip: Currently the example always controls both LEDs or reads buttons & temperature sensors regardless of the part of RAM peripheral memory space that was written to or read from respectively. Modify the code so it will work with the peripheral(s) that correspond to the peripheral memory range that was read from or written to.*

9.6.6 UserPeripheral-18B20

This example demonstrates connecting the [N] to the 1-Wire device. It might be a starting application to create a sensor network having external temperature sensors.

The example uses a popular temperature sensor Dallas 18B20. The sensor is present at the DDC-SE-01 sensor kit so it is very easy to create a device operating the sensor at the lab.

Deep knowledge of the 1-Wire protocol is necessary to understand the whole source code.

→ *Self-study tip: Modify the code to return the temperature value using the user FRC command.*
Hint: As the 18B20 conversion time exceeds the maximum 40 ms FRC response time both Set FRC Params at [C] side and FRC response time at [N] side must be used.

9.6.7 UserPeripheral-18B20-Idle

This is a more advanced version of the previous UserPeripheral-18B20 example. This version performs a repetitive reading of the temperature value from the 1-Wire sensor at the Idle event in the background so the temperature value is available anytime without any delay. This simplifies the implementation of the user FRC command.

9.6.8 UserPeripheral-ADC

There is no embedded ADC peripheral implemented at DPA. The reason is that there are many diverse requirements (number of channels, channel selection, conversion time, conversion precision, etc.) to the actual ADC peripheral implementation.

This example implements analog to digital conversion from two channels. Intentionally these channels can be driven directly by a photoresistor and a potentiometer and available at DDC-SE-01.

→ *Self-study tip: Implement user two-byte FRC command that will return MSB values from both ADC channels at once.*

9.6.9 UserPeripheral-HW-UART

This example shows how to implement custom HW UART with circular buffers i.e. not using embedded UART peripheral. This is necessary in case the UART must be used when handling custom peripheral or during any event including an Interrupt event.

→ *Self-study tip: implement variable UART baud rate when UART is opened.*

9.6.10 UserPeripheral-i2c

The example implements a user peripheral that returns a value read from a connected I²C device. The code can directly read a temperature value from the MCP9802 temperature sensor presented at DDC-SE-01.

Deep knowledge of the I²C protocol is necessary to understand the source code in full detail.

→ *Self-study tip: Implement user byte FRC command to return value from an I²C device. Pay attention to the maximum FRC response time.*

9.6.11 UserPeripheral-PWM

This is a copy of the implementation of the formerly embedded PWM peripheral that was available only in the demo version. Use it as a template for your own PWM implementation. See also UserPeripheral-PWMandTimer.c.

9.6.12 UserPeripheral-SPImaster

This example shows how to connect the SPI slave device to the TR [N] so the [N] behaves as SPI master. IQRF OS SPI support implements only the SPI slave side. SPI slave device is controlled using a custom command passed to the custom DPA peripheral. See the source code for full details.

→ Self-study tip: Connect ordinary another TR [N] with DPA SPI peripheral being enabled thus playing the role of SPI slave. Try to communicate bi-directionally between the two Nodes.

9.7 Migration Notes to DPA 3.03

Please find below important topics when migrating Custom DPA Handler to DPA 3.03+ from DPA 3.02.

- Custom DPA Handler implementing FRC functionality must be recompiled, because IQRF OS variables for returning FRC values (*responseFRCvalue**) changed their addresses and they currently overlap at the same address.
- If the Custom DPA Handler implements a custom bonding at the Reset event, then please review the handler code according to the Bonding example to correctly support all required types of bonding.
- Do necessary changes according to DPA Release Notes.
- Test your Custom DPA Handler with this DPA release before production.

10 DPA Peer-to-Peer

DPA Peer-to-Peer (DP2P from now) allows communicating with bonded [Ns] at the existing network in non-network mode. Unlike DSM no reset or restart of the [N] is needed. DP2P is useful for inventorying the network (e.g. localization of the [Ns] after autonetwork), device maintenance, remote control, etc. Communication runs on the network A channel and the data is encrypted by an AES-128 algorithm using an access password as a key. DP2P must be enabled in the [N] configuration to work. All packets use the *_DPAF* flag and DPA reserves the right to use this flag exclusively only for DP2P purposes in case of non-networking packets.

The DP2P protocol consists of one DP2P Request packet (it contains DPA Request that is executed at the addressed [Ns]) and then DP2P Response Handshake packets exchanged between the addressed and reachable [Ns] and the device, that sent DP2P Request.

10.1 DP2P Request

The DP2P Request stored at b*ufferRF* has the following structure:

```
typedef struct
{
  uns8  Header[3];  // 0x000000
  uns8  SelectedNodes[30];
  uns8  SlotLength;
  uns8  ResponseTxPower;
  uns8  Reserved;
  uns16 HWPID;
  uns8  PDATA[sizeofBufferRF - (3+30+1+1+1)*sizeof( uns8 ) - (1)*sizeof( uns16 )];
} STRUCTATTR TDP2Prequest;
```

Header	Must consist of 3 zeros.
SelectedNodes	Specifies addressed [Ns] that should send DP2P Response back. See identically named field at Send Selective command. Communication with prebonded [Ns] is not possible.
SlotLength	Specifies the timeslot length for the DP2P Response Handshake. The unit is 10 ms. If the default value 0 is used then the timeslot length to accommodate common DPA Request is chosen (see table below). Other values specify a custom timeslot length when the DPA Request might take a longer time (e.g. UART Write & Read) or by contrast to shorten the timeslot if the DPA Request takes short time and its response is also short (e.g. LED Pulse).
ResponseTxPower	RF output power used to send DP2P Invite and DP2P Response by [Ns]. Valid numbers are 0-7.
HWPID	HWPID of the DPA Request.
PDATA	PData part of the DPA Request.

Also, the following variables must be correctly assigned before sending the DP2P Request:

_PNUM	PNUM of the DPA Request to execute at [Ns].
_PCMD	PCMD of the DPA Request to execute at [Ns].
DLEN	Must equal 64.
PPAR	Length the DPA Request data payload at the PDATA field.

RF mode used to send DP2P Request depends on the network type. Example of sending DP2P Request:

```
// Set RF Mode
if ( "is STD network" )
  setRFmode( _WPE | _RX_STD | _TX_STD | _STDL );
if ( "is STD+LP network" )
  setRFmode( _WPE | _RX_STD | _TX_LP );
```

```
// DP2P Request variable
TDP2Prequest DP2Prequest @ bufferRF;
// Prepare DPA Request
// We will read 10 bytes of EEPROM from address 1
_PNUM = PNUM_EEPROM;
_PCMD = CMD_EEPROM_READ;
_DpaMessage.MemoryRequest.Address = 1;
_DpaMessage.MemoryRequest.ReadWrite.Read.Length = 10;
// DPA request data length
PPAR = sizeof( _DpaMessage.MemoryRequest.Address ) +
       sizeof( _DpaMessage.MemoryRequest.ReadWrite.Read );

// Save the prepared PData of the DPA Request for later copying
copyBufferRF2INFO();
// Clear DP2Prequest (it clears the Header and SelectedNodes fields)
clearBufferRF();
// Move the saved PData of the DPA Request to the correct place at the DP2P Request
memoryOffsetTo = offsetof( TDP2Prequest, PDATA );
copyBufferINFO2RF();

// Select Node #2 to respond
DP2Prequest.SelectedNodes[0].2 = TRUE;
// Select Node #11 to respond
DP2Prequest.SelectedNodes[1].3 = TRUE;
// Use default timeslot length
DP2Prequest.SlotLength= 0;
// Set TX power for the DP2P Response
DP2Prequest.ResponseTxPower = TX_POWER_MAX;
// Any HWPID is OK
DP2Prequest.HWPID = HWPID_DoNotCheck;
// Set the DP2P packet length
DLEN = sizeof( DP2Prequest );
// Use Access Password for encryption
encryptByAccessPassword = TRUE;
// Encrypt the DP2P Request
encryptBufferRF( sizeof( DP2Prequest ) / 16 );
// Set RF Flags
PIN = _DPAF_MASK;
// And finally send DP2P Request
RFTXpacket();
```

10.2 DP2P Response Handshake

DP2P Response Handshake consists of 3 special DP2P packets described below. Every [N] addressed by the *SelectedNodes* field of the DP2P Request has a dedicated timeslot for DP2P Response Handshake after it receives DP2P Request. The index of the timeslot equals the index of the bit corresponding to the [N] in the *SelectedNodes* field. If [N] does not receive the DP2P Request then its timeslot remains empty i.e. unutilized. All following packets are sent using `_TX_STD` without `_STDL` mode.

Note: in the pictures below we interchange the term Timeslot for the term DP2P Response Handshake as it is shorter to write.

	Timeslot #0	*Timeslot #1*		*Last timeslot*
DP2P Request	1st selected [N]	2nd selected [N]	…	The last selected [N]

For instance, if only two [Ns] #2 and #11 from the above example are selected, then [N] #2 uses the 1st timeslot while [N] #11 has the 2nd (last) one.

At the beginning of the timeslot i.e. DP2P Response Handshake, the [N] first sends DP2P Invite, then waits for the DP2P Confirm, and finally sends DP2P Response. Invite and Confirm packets are used to protect the communication against resending previously sniffed DP2P Request packets. Please see the details below and DP2Papp.c and DP2Papp-UART.c examples for the implementation.

DP2P Response Handshake detail:

Timeslot #n-1	*Timeslot #n*			*Timeslot #n+1*
	DP2P Invite	DP2P Confirm	DP2P Response	

The predefined length of the DP2P Response Handshake timeslots is 110 ms (`DP2P_TIMESLOT`). The non-zero field *SlotLength* in the DP2P Request can be used to specify a different length.

10.2.1 DP2P Invite

The [N] sends this packet with a *Rand* field containing random content.

```
// DP2P invite packet.
typedef struct
{
  uns8  Header[3];  // 0x000001
  uns8  NADR;
  uns8  Rand[12];
} STRUCTATTR TDP2Invite;
```

10.2.2 DP2P Confirm

When the device, that sent the DP2P Request, receives the DP2P Invite, it transforms it into DP2P Confirm just by modifying the header and sends it back to the [N] using `_TX_STD` without `_STDL` mode. When [N] receives DP2P Confirm with the same random data as DP2P Invite, it can then safely execute the DPA Request and then reply with DPA DP2P Response at the end of its timeslot.

```
typedef struct
{
  uns8  Header[3];  // 0x000003
  uns8  NADR;
  uns8  Rand[12];
} STRUCTATTR TDP2Confirm;
```

10.2.3 DP2P Response

The DP2P Response has the following structure:

```
typedef struct
{
  uns8  Header[3];  // 0xFfFfFf
  uns8  NADR;
  uns8  PDATA[DPA_MAX_DATA_LENGTH];
} STRUCTATTR TDP2Presponse;
```

Header	Consist of 3 bytes 0xFF. The Header is used for packet validation purposes.
NADR	Address of the responding [N].
PDATA	PData part of the DPA Response.

Also, the following variables are received:

_PNUM	PNUM of the original DPA request.
_PCMD	PCMD of the original DPA request with the most significant bit set to indicate DPA Response.
PPAR	Equals effective packet length (i.e. length to the PData payload plus length of the *Header* and *NADR* fields).
DLEN	Equals effective packet length (PPAR) rounded up to the 16-byte block (to enable AES-128 decryption).

11 DPA in Practice

Refer to the following chapters for useful DPA procedures.

11.1 Network Deployment

This chapter is a kind of checklist to go through when deploying the IQMESH network with DPA. Please note, that some steps might not be obligatory as they are already fulfilled (e.g. installed devices are already preloaded with DPA plug-in and Custom DPA Handler). We suppose IQRF IDE is used as a tool.

1. Plan your network in terms of size, the number of (non-)routing devices, etc. If non-routing devices are present then it is recommended to assign them the logical addresses from the compact address interval at the top of the address space during bonding. This allows us to effectively use the parameter MaxAddr at Discovery.
2. Download required DPA plug-ins based on Interface, and TR type used. Upload them to the devices.
3. Get ready your Custom DPA Handlers for all devices. Make sure the handler code states the unique HWPID of the device. Some handlers do not have any internal application logic code except stating HWPID but also contain IO Setup. Upload the handlers to the devices.
4. Configure the devices:
    a. The configuration very often differs between [C] and [N]s and even between various [N]s.
    b. Start with a default configuration offered by IQRF IDE.
    c. We recommend setting a unique access password for each network.
    d. Do frequency planning, i.e. set the working channel that is not used and jammed.
    e. Enable all needed peripherals (do not forget to enable FRC at [C] and disable it at [N]s).
    f. Make sure to enable the correct SPI/UART peripheral/interface.
    g. Enable IO Setup, Custom DPA Handler, disable routing, etc. as needed.
5. Bond [N]s to the [C]. This process depends on the used devices as it might be implemented differently at every handler. Also, Autonetwork is available. In general, the process is somehow initiated at [C] and [N] sides (e.g. by pressing a button). Sometimes devices are bonded before their physical installation, sometimes at the final place. Before the bonding of the new network, it is recommended to execute Clear all bonds at [C]. Of course, [N]s must not be already bonded before bonding. Also, CATS from IQRF IDE can be used for (un)bonding.
6. Run Discovery after all devices are successfully bonded and installed:
    a. Use a lower RF output power than the one used during normal network operation.
    b. The duration of the discovery process depends on the network size and its topology. In the case of complicated networks, it might take 1 hour.
    c. In the case of a homogeneous network, it is not always necessary to discover all devices (e.g. 95 from out of 100 might be OK) but all devices must be accessible.
    d. When the network contains non-routing devices then all routers must be discovered.
    e. After the discovery is finished, test communication with all devices.
    f. The Discovery result (number of discovered devices, the number of zones, parents) varies at the time because of an actual RF environment.
    g. Discovery must be repeated every time the topology (new, removed, and/or moved router) and/or RF conditions (e.g. a new RF obstacle) change.
    h. Note: discovery is an integral part of the Autonetwork feature.
7. Enumerate the network and save information (IQRF OS and DPA versions, configuration, etc.) into separate files for future reference.
8. Back up the network data from all devices ([C] and [N]s). The backup is required for an optional future cloning of the damaged device.
9. To protect your device from unauthorized CATS access you can set your own access password.

11.2 Over The Air (OTA) upgrade of IQRF OS and DPA

Please follow this checklist to upgrade both IQRF OS and DPA with TR-7xD over the air using the IQRF IDE. IQRF IDE uses public DPA commands described in this document to accomplish the upgrade. Select *All* at *Tools/Options/Environment Options/IQMESH Network Manager/Log background DPA communication* to see the commands at *Terminal Log* panel.

1 Uploading a special OTA Custom DPA Handler to the Coordinator and all Nodes

1.1 Go to *Tools / IQMESH Network Manager / Control / Upload* at IQRF IDE.
1.2 Browse a file *CustomDpaHandler-ChangeIQRFOS-7xD-Vvvv-yymmdd.iqrf* at *Source File* group box. The file can be found at the *IQRF Startup Package* of the IQRF OS target version in the folder *Development\DPA\OTA_upgrade*.
1.3 Set *External EEPROM Address* to *0x800* at the *Upload* group box.
1.4 Select *All Nodes* and set *HWPID* to *0xFFFF* at the *Destination Device* group box.
1.5 Press the *Upload* button at the group box *Upload* to upload the selected file to the external EEPROM at all Nodes.
1.6 Press the *Verify* button to check the uploaded file integrity.
1.7 Upload and verify the file to the Nodes that report an integrity error until no error is reported.
1.8 Press the *Load* button to write the handler from EEPROM to the flash memory at all Nodes.
1.9 Select *Coordinator* at the *Destination Device* group box.
1.10 Press the *Upload* button at the group box *Upload* to upload the selected file to the external EEPROM at the Coordinator.
1.11 Press the *Verify* button to check the uploaded file integrity and then *Load* to write it to the flash memory at the Coordinator.

2 Enabling the special OTA Custom DPA Handler at the Coordinator and Nodes

2.1 Go to *Tools / IQMESH Network Manager / Control / TR Config*.
2.2 Uncheck the *Source File* group box if it is checked.
2.3 Select *All Nodes* and set *HWPID* to *0xFFFF* at the *Destination Device* group box.
2.4 Press the *Configure TR* button at the *Command* group box. A *TR Configuration* window will open.
2.5 Enable *Custom DPA Handler* at the *DPA* tab and press *Upload*. Press *Try Selected* if the configuration wizard reports an error writing configuration to some Nodes. *Close* the configuration window.
2.6 Press *Restart* at the *Command* group box to restart all Nodes.
2.7 Select *Coordinator* at the *Destination Device* group box.
2.8 Press the *Configure TR* button at the *Command* group box. A *TR Configuration* window will open.
2.9 Enable *Custom DPA Handler* at the *DPA* tab and press *Upload*. *Close* the configuration window.
2.10 Press *Restart* at the *Command* group box to restart the Coordinator.
2.11 Refresh a table at the *Table View* tab and check that an HWPID of all network members equals *0xC05E*.

3 Uploading a change file to the Coordinator and all Nodes.

3.1 Go to *Tools / IQMESH Network Manager / Control / Upload* at IQRF IDE.
3.2 Browse a file *ChangeOS-TR7x-ooo(oooo)-nnn(nnnn)-Vooo+Node+xxx-Vnnn+Node+xxx.bin* (*ooo* specifies original IQRF OS and DPA version while *nnn* specifies new IQRF OS and DPA version respectively; *xxx* specifies required interface). The file can be found at the *IQRF Startup Package* in the folder *Development\DPA\OTA_upgrade.*
3.3 Set *External EEPROM Address* to *0x800* at the *Upload* group box.
3.4 Select *All Nodes* at the *Destination Device* group box. The *HWPID* is set to *0xC05E* automatically.
3.5 Continue according to 1.5.-1.8.
3.6 Browse a file *ChangeOS-TR7x-ooo(oooo)-nnn(nnnn)-Vooo+Coordinator+xxx-Vnnn+Coordinator+xxx.bin* (*ooo* specifies original IQRF OS and DPA version while *nnn* specifies new IQRF OS and DPA version respectively; *xxx* specifies required interface). The file can be found at the *IQRF Startup Package* in the folder *Development\DPA\OTA_upgrade.*
3.7 Continue according to 1.9.-1.11.

4 Finishing up

4.1 Both IQRF OS and DPA are upgraded. The network is working.
4.2 Refresh and check the network map from the Coordinator.
4.3 Follow chapter 1 to upload back your normal Custom DPA Handlers or follow chapter 2. to disable Custom DPA Handler on the devices that do not use it.
4.4 Follow chapter 2 to set an Access password and/or User Key at the TR Configuration of all devices if they were upgraded from IQRF OS 3.0x.
4.5 Enumerate the network, check it, and save the enumeration.
4.6 Backup the network and save the backup file.

11.3 Code Upload

DPA supports uploading executable code to the devices as well as upgrading IQRF OS at the devices over the network without a need to connect the device to the HW programmer. In general, the code image or the IQRF OS change file must be first stored in the external EEPROM at the device and then a corresponding DPA Request does the job. The next paragraphs describe how to proceed from the programmer's point of view.

11.3.1 Storing Code at External EEPROM

Code image or the IQRF OS change file must be stored in the external EEPROM using a series of Extended Write commands.

When Custom DPA Handler should be uploaded using LoadCode command then a .hex file containing the handler code must be stored in the external EEPROM. See LoadCode and Custom DPA Handler Code at .hex File for more details about decoding the file.

When IQRF plug-in containing an e.g. newer version of DPA or IQRF OS patch is to be loaded then the content of the .iqrf file has to be stored in the external EEPROM. See LoadCode for more details about decoding the file.

If IQRF OS change is to be executed then a special handler must be active and the corresponding .bin file containing the IQRF OS change data must be stored in the external EEPROM. See IQRF OS Change for more details.

When storing the data in the external EEPROM make sure that other data are not overwritten. That could be another upload data, handler operation data, or IO Setup. Precise planning of the external EEPROM content is recommended.

It is also recommended to plan the whole upload or change process in the way that all required data (active handler, IQRF OS change handler, IQRF plug-in DPA, IQRF OS change file) are first stored in the external EEPROM and then used to minimize the code upload time. Some of the items at the external EEPROM may take up to a few kilobytes and it takes a considerable time to store them in the even small network.

11.3.2 Executing Code Upload

Once the content of the .hex or .iqrf file is stored in the external EEPROM then the request LoadCode can be executed at the device to load the code. We recommend first running the command to check the checksum of the data at the external EEPROM only to make sure the code upload will not later fail. In case more devices are to load the code, it is useful to use the byte FRC command Memory read plus 1 to read the result of the checksum check from multiple devices instead of individually polling each device one by one. When FRC is used then it is necessary to use Send Selective instead of Send in case of a larger network. When all devices have the correct data at external EEPROM ready then finally the request LoadCode can be fully executed to perform the desired code upload. To run the request at selected devices only then specific HWPID or Acknowledged broadcast - bits with Send Selective are to be used. Pay special attention when the former or new uploaded handler requires its data to be stored at the internal and/or external EEPROM. See LoadCode for more details.

11.3.3 Executing IQRF OS Change

Changing the IQRF OS version is very similar to loading the code described above. The difference is that a special custom DPA handler must be used. See IQRF OS Change for more details. Apart from changing only the IQRF OS version, the process can also change the DPA version at the same time. It implies that the current normally used custom handler must be replaced and then returned. We recommend storing these items in the external EEPROM first before the IQRF OS change is performed:

1. Image of the special handler *CustomDpaHandler-ChangeIQRFOS.iqrf*,
2. IQRF OS change file and
3. Image of the normally used custom DPA handler.

First, upload the special handler from item No. 1 by the process described above. Then similarly (to load the code) check that item No. 2 from the above list is correctly stored in the external EEPROM. In this

case, use a command of the custom peripheral implemented at the special handler for the check. Again the FRC can be used to verify the content at more devices in one stroke. When the content is OK then run the command again to perform the real IQRF OS change. When the change is finished then Memory read plus 1 can be used to check the IQRF OS version or the build number (checking the lower byte of the build number is enough) from more devices at one go. Finally, return the normally used custom DPA handler stored at item No. 3 back.

12 Constants

All symbols and constants are defined in header files DPA.h and DPAcustomHandler.h.

12.1 Peripheral Numbers

```
#define    PNUM_COORDINATOR 0x00
#define    PNUM_NODE        0x01
#define    PNUM_OS          0x02
#define    PNUM_EEPROM      0x03
#define    PNUM_EEEPROM     0x04
#define    PNUM_RAM         0x05
#define    PNUM_LEDR        0x06
#define    PNUM_LEDG        0x07
#define    PNUM_IO          0x09
#define    PNUM_THERMOMETER 0x0A
#define    PNUM_UART        0x0C
#define    PNUM_FRC         0x0D

#define    PNUM_USER        0x20    // Number of the 1st user peripheral
#define    PNUM_USER_MAX    0x3E    // Number of the last user peripheral
#define    PNUM_MAX         0x7F    // Maximum peripheral number
#define    PNUM_ERROR_FLAG  0xFE
```

12.2 Response Codes

```
STATUS_NO_ERROR =             0, // No error
ERROR_FAIL =                  1, // General fail
ERROR_PCMD =                  2, // Incorrect PCMD
ERROR_PNUM =                  3, // Incorrect PNUM or PCMD
ERROR_ADDR =                  4, // Incorrect Address
ERROR_DATA_LEN =              5, // Incorrect Data length
ERROR_DATA =                  6, // Incorrect Data
ERROR_HWPID =                 7, // Incorrect HW Profile ID used
ERROR_NADR =                  8, // Incorrect NADR
ERROR_IFACE_CUSTOM_HANDLER = 9, // Data from interface consumed by Custom DPA Handler
ERROR_MISSING_CUSTOM_DPA_HANDLER = 10, // Custom DPA Handler is missing

ERROR_USER_FROM =             0x20, // Beginning of the user code error interval
ERROR_USER_TO =               0x3F, // End of the user error code interval
STATUS_RESERVED_FLAG =        0x40, // Bit/flag reserved for a future use
STATUS_ASYNC_RESPONSE =       0x80, // Bit to flag asynchronous DPA Response from [N]

STATUS_CONFIRMATION =         0xFF  // Error code used to mark DPA Confirmation
```

12.3 DPA Commands

```
#define    CMD_COORDINATOR_ADDR_INFO 0
#define    CMD_COORDINATOR_DISCOVERED_DEVICES 1
#define    CMD_COORDINATOR_BONDED_DEVICES 2
#define    CMD_COORDINATOR_CLEAR_ALL_BONDS 3
#define    CMD_COORDINATOR_BOND_NODE 4
#define    CMD_COORDINATOR_REMOVE_BOND 5
#define    CMD_COORDINATOR_DISCOVERY 7
#define    CMD_COORDINATOR_SET_DPAPARAMS 8
#define    CMD_COORDINATOR_SET_HOPS 9
#define    CMD_COORDINATOR_BACKUP 11
#define    CMD_COORDINATOR_RESTORE 12
#define    CMD_COORDINATOR_AUTHORIZE_BOND 13
#define    CMD_COORDINATOR_SMART_CONNECT 18
#define    CMD_COORDINATOR_SET_MID 19
```

```
#define    CMD_NODE_READ 0
#define    CMD_NODE_REMOVE_BOND 1
#define    CMD_NODE_BACKUP 6
#define    CMD_NODE_RESTORE 7
#define    CMD_NODE_VALIDATE_BONDS 8

#define    CMD_OS_READ 0
#define    CMD_OS_RESET 1
#define    CMD_OS_READ_CFG 2
#define    CMD_OS_RFPGM 3
#define    CMD_OS_SLEEP 4
#define    CMD_OS_BATCH 5
#define    CMD_OS_SET_SECURITY 6
#define    CMD_OS_INDICATE 7
#define    CMD_OS_RESTART 8
#define    CMD_OS_WRITE_CFG_BYTE 9
#define    CMD_OS_LOAD_CODE 10
#define    CMD_OS_SELECTIVE_BATCH 11
#define    CMD_OS_TEST_RF_SIGNAL 12
#define    CMD_OS_FACTORY_SETTINGS 13
#define    CMD_OS_WRITE_CFG 15

#define    CMD_RAM_READ 0
#define    CMD_RAM_WRITE 1

#define    CMD_EEPROM_READ CMD_RAM_READ
#define    CMD_EEPROM_WRITE CMD_RAM_WRITE

#define    CMD_EEEPROM_XREAD ( CMD_RAM_READ + 2 )
#define    CMD_EEEPROM_XWRITE ( CMD_RAM_WRITE + 2 )

#define    CMD_LED_SET_OFF 0
#define    CMD_LED_SET_ON 1
#define    CMD_LED_PULSE 3
#define    CMD_LED_FLASHING 4

#define    CMD_IO_DIRECTION 0
#define    CMD_IO_SET    1
#define    CMD_IO_GET    2

#define    CMD_THERMOMETER_READ 0

#define    CMD_UART_OPEN 0
#define    CMD_UART_CLOSE 1
#define    CMD_UART_WRITE_READ 2
#define    CMD_UART_CLEAR_WRITE_READ 3

#define    CMD_FRC_SEND 0
#define    CMD_FRC_EXTRARESULT 1
#define    CMD_FRC_SEND_SELECTIVE 2
#define    CMD_FRC_SET_PARAMS 3

#define    CMD_GET_PER_INFO 0x3f
```

12.4 Peripheral Types

```
PERIPHERAL_TYPE_DUMMY = 0x00,
PERIPHERAL_TYPE_COORDINATOR = 0x01,
PERIPHERAL_TYPE_NODE = 0x02,
PERIPHERAL_TYPE_OS = 0x03,
PERIPHERAL_TYPE_EEPROM = 0x04,
```

```
PERIPHERAL_TYPE_BLOCK_EEPROM = 0x05,
PERIPHERAL_TYPE_RAM = 0x06,
PERIPHERAL_TYPE_LED = 0x07,
PERIPHERAL_TYPE_SPI = 0x08,
PERIPHERAL_TYPE_IO = 0x09,
PERIPHERAL_TYPE_UART = 0x0a,
PERIPHERAL_TYPE_THERMOMETER = 0x0b,
PERIPHERAL_TYPE_ADC = 0x0c, (*)
PERIPHERAL_TYPE_PWM = 0x0d,
PERIPHERAL_TYPE_FRC = 0x0e,

PERIPHERAL_TYPE_USER_AREA = 0x80
```

(*) Embedded peripheral of this type is not defined and implemented yet. See example *CustomDpaHandler-UserPeripheral-ADC.c* for potential implementation.

12.5 Custom DPA Handler Events

```
#define    DpaEvent_DpaRequest               0
#define    DpaEvent_Interrupt                1
#define    DpaEvent_Idle                     2
#define    DpaEvent_Init                     3
#define    DpaEvent_Notification             4
#define    DpaEvent_AfterRouting             5
#define    DpaEvent_BeforeSleep              6
#define    DpaEvent_AfterSleep               7
#define    DpaEvent_Reset                    8
#define    DpaEvent_DisableInterrupts        9
#define    DpaEvent_FrcValue                10
#define    DpaEvent_ReceiveDpaResponse      11
#define    DpaEvent_IFaceReceive            12
#define    DpaEvent_ReceiveDpaRequest       13
#define    DpaEvent_BeforeSendingDpaResponse 14
#define    DpaEvent_PeerToPeer              15
#define    DpaEvent_UserDpaValue            17
#define    DpaEvent_FrcResponseTime         18
#define    DpaEvent_BondingButton           19
#define    DpaEvent_Indicate                20
#define    DpaEvent_VerifyLocalFrc          21
#define    DpaEvent_MenuActivated           22
#define    DpaEvent_MenuItemSelected        23
#define    DpaEvent_MenuItemFinalize        24
#define    DpaEvent_InStandby               25
```

12.6 Extended Peripheral Characteristic

```
PERIPHERAL_TYPE_EXTENDED_DEFAULT    = 0b00,
PERIPHERAL_TYPE_EXTENDED_READ       = 0b01,
PERIPHERAL_TYPE_EXTENDED_WRITE      = 0b10,
PERIPHERAL_TYPE_EXTENDED_READ_WRITE = PERIPHERAL_TYPE_EXTENDED_READ |
                                      PERIPHERAL_TYPE_EXTENDED_WRITE
```

12.7 HW Profile IDs

```
HWPID_Default = 0,          // No HW Profile specified
HWPID_DoNotCheck = 0xffff   // Use this type to override HW Profile ID check
```

12.8 Baud rates

```
DpaBaud_1200   = 0x00,
```

```
DpaBaud_2400   = 0x01,
DpaBaud_4800   = 0x02,
DpaBaud_9600   = 0x03,
DpaBaud_19200  = 0x04,
DpaBaud_38400  = 0x05,
DpaBaud_57600  = 0x06,
DpaBaud_115200 = 0x07,
DpaBaud_230400 = 0x08
```

12.9 User FRC Codes

```
#define    FRC_USER_BIT_FROM   0x40
#define    FRC_USER_BIT_TO     0x7F
#define    FRC_USER_BYTE_FROM  0xC0
#define    FRC_USER_BYTE_TO    0xDF
#define    FRC_USER_2BYTE_FROM 0xF0
#define    FRC_USER_2BYTE_TO   0xF7
#define    FRC_USER_4BYTE_FROM 0xFC
#define    FRC_USER_4BYTE_TO   0xFF
```

13 Appendix

13.1 CRC Calculation

The following examples show the implementation of the 1-Wire CRC used to protect UART Interface data. Before using the routines do not forget to initialize the CRC accumulator variable to the initial value 0xFF.

13.1.1 CC5X Compiler

```
// One Wire CRC
static uns8 OneWireCrc;

// Updates crc at OneWireCrc variable, parameter value is an input data byte
void UpdateOneWireCrc( uns8 value @ W )
{
  OneWireCrc ^= value;
#pragma update_RP 0 /* OFF */
  value = 0;
  if ( OneWireCrc.7 )
       value ^= 0x8c;       // 0x8C is reverse polynomial representation
  if ( OneWireCrc.6 )       // (normal is 0x31)
       value ^= 0x46;
  if ( OneWireCrc.5 )
       value ^= 0x23;
  if ( OneWireCrc.4 )
       value ^= 0x9d;
  if ( OneWireCrc.3 )
       value ^= 0xc2;
  if ( OneWireCrc.2 )
       value ^= 0x61;       // …
  if ( OneWireCrc.1 )       // 1 instruction
       value ^= 0xbc;       // 1 instruction
  if ( OneWireCrc.0 )       // 1 instruction
       value ^= 0x5e;       // 1 instruction
  OneWireCrc = value;       // 1 instruction
#pragma update_RP 1 /* ON */
}
```

13.1.2 C#

```
/// <summary>
/// Computes 1-Wire CRC
/// </summary>
/// <param name="value">Input data byte</param>
/// <param name="crc">Updated CRC</param>
static void UpdateOneWireCrc ( byte value, ref byte crc )
{
  for ( int bitLoop = 8; bitLoop != 0; --bitLoop, value >>= 1 )
    if ( ( ( crc ^ value ) & 0x01 ) != 0 )
      crc = (byte)( ( crc >> 1 ) ^ 0x8C );
    else
      crc >>= 1;
}
```

13.1.3 Java

```
/**
 * Returns new value of CRC.
 * @param crc current value of CRC
 * @param value input data byte
 * @return updated value of CRC
 */
```

```
static short updateCRC(short crc, short value) {
    for ( int bitLoop = 8; bitLoop != 0; --bitLoop, value >>= 1 ) {
        if ( ( ( crc ^ value ) & 0x01 ) != 0 ) {
            crc = (short)( ( crc >> 1 ) ^ 0x8C );
        } else {
            crc >>= 1;
        }
    }
    return crc;
}
```

13.1.4 Pascal/Delphi

```
/// <summary>
/// Computes 1-Wire CRC
/// </summary>
/// <param name="value">Input data byte</param>
/// <param name="crc">Updated CRC</param>
procedure UpdateOneWireCrc ( value: byte; var crc: byte );
var
  bitLoop: integer;
begin
  for bitLoop := 8 downto 1 do begin
    if ( ( ( crc xor value ) and $01 ) <> 0 ) then
      crc := ( crc shr 1 ) xor $8C
    else
      crc := crc shr 1;
    value := value shr 1;
  end;
end;
```

13.2 One's Complement Fletcher-16 Checksum Calculation

The following examples show the implementation of one's complement Fletcher-16 checksum used to check code uploaded by the LoadCode command.

Please note that the one's complement adding implementation does not use a well-known "modulo 255" algorithm that requires more code but it makes use of "carry technique" that unlikely does not avoid one's complement negative zero value 0xFF.

13.2.1 CC5X Compiler

```
// Initialize One's Complement Fletcher Checksum
uns16 checksum = "initial value";

...

// Loop through all data bytes, each stored at oneByte

// Update lower checksum byte
checksum.low8 += oneByte;
checksum.low8 += Carry;
// Update higher checksum byte
checksum.high8 += checksum.low8;
checksum.high8 += Carry
```

13.2.2 C#

```
public static UInt16 FletcherChecksum ( byte[] bytes )
{
  // Initialize One's Complement Fletcher Checksum
  UInt16 checkSum = "initial value";
```

```
  // Loop through all data bytes, each stored at oneByte
  foreach ( byte oneByte in bytes )
  {
    // Update lower checksum byte
    int tempL = checkSum & 0xff;
    tempL += oneByte;
    if ( ( tempL & 0x100 ) != 0 )
      tempL++;

    // Update higher checksum byte
    int tempH = checkSum >> 8;
    tempH += tempL & 0xff;
    if ( ( tempH & 0x100 ) != 0 )
      tempH++;

    checkSum = (UInt16)( ( tempL & 0xff ) | ( tempH & 0xff ) << 8 );
  }

  return checkSum;
}
```

13.2.3 Python

```
# One's Complement Fletcher-16 Checksum Calculation
def fletcher_checksum(init_value: int, bytes: list):
    # Initialize One's Complement Fletcher Checksum
    checksum = init_value
    # Loop through all data bytes, each stored at oneByte
    for one_byte in bytes:
        # Update lower checksum byte
        temp_low = checksum & 0xff
        temp_low += one_byte
        if (temp_low & 0x100) != 0:
            temp_low += 1

        # Update higher checksum byte
        temp_high = checksum >> 8
        temp_high += temp_low & 0xff
        if (temp_high & 0x100) != 0:
            temp_high += 1

        checksum = ((temp_low & 0xff) | (temp_high & 0xff) << 8)
    return checksum
```

13.3 Custom DPA Handler Code at .hex File

The following example shows the principles of obtaining the code for Custom DPA Handler to be stored at external EEPROM and to be later loaded into MCU flash memory and executed.

Below is the piece of output .lst file of the compiled FRC-Minimalistic Custom DPA Handler example. The code is located from the mandatory starting address 0x3A20 and in this example ends at address 0x3A30.

```
 ; bit CustomDpaHandler()
 ; {
 ;   // Handler presence mark
 ;   clrwdt();
3A20 0064           CLRWDT
 ;
 ;   // Return 1 if IQRF button is pressed
 ;   if (GetDpaEvent() == DpaEvent_FrcValue && _PCMD == FRC_USER_BIT_FROM && buttonPressed)
3A21 0870           MOVF  userReg0,W
```

```
3A22 3A0A           XORLW 0x0A
3A23 1D03           BTFSS 0x03,Zero_
3A24 320A           BRA   m001
3A25 0025           MOVLB 0x05
3A26 082F           MOVF  PCMD,W
3A27 3A40           XORLW 0x40
3A28 1D03           BTFSS 0x03,Zero_
3A29 3205           BRA   m001
3A2A 0020           MOVLB 0x00
3A2B 1A0D           BTFSC PORTB,4
3A2C 3202           BRA   m001
 ;     responseFRCvalue.1 = 1;
3A2D 002B           MOVLB 0x0B
3A2E 14B8           BSF   responseFRCvalue,1
 ;
 ;   return FALSE;
3A2F 1003   m001    BCF   0x03,Carry
3A30 0008           RETURN
 ; }
```

The portion of the corresponding .hex file stores the **code** bytes from the double address 0x7440 = 2 × 0x3A20 to 0x7460 = 2 × 0x3A30. The first two digits at the line specify the byte count, two zeros after the address specify the record type.

```
:020000040000FA
...
:08741000AC310024BA31080080
:1074400006400700 80A3A031D0A3225002F08403AEA
:1074500000031D053220000D1A02322B00B8140310 50
:0274600000800022
:027AFE0008007E
...
```

The exact code size is 2 × (0x3A30 - 0x3A20 + 1) = 34 bytes. The length of the code stored at external EEPROM must be multiple of 64 so, in our example, the stored size is 64 = 0x40 bytes. If the unused 30 bytes (64 - 34) bytes of the 64-byte block are filled in with zeros then the Fletcher-16 checksum equals 0xEA3A.

13.4 IQRF OS Change

[sync] IQRF OS version at any DPA device can be upgraded (or downgraded) over the network without having physical access to the device. It can also optionally change the DPA version at the same time. A specially prepared Custom DPA Handler named *CustomDpaHandler-ChangeIQRFOS.iqrf* must be used. The handler can be found at the IQRF Startup Package. Upload the handler to the device using the LoadCode command. Before that store an IQRF OS change file (e.g. *ChangeOS-TR7x-308(0873)-308(0874).bin*) at the external EEPROM using a series of Extended Write commands. The file can be found at the IQRF Startup Package too. Then execute a below-described DPA Request at the custom peripheral implemented at the special uploaded handler. After the IQRF OS change is successfully finished the device is reset and you can upload your previously used handler back again using the LoadCode command.

Important: During the whole process of the IQRF OS change (starting at the time of sending the below-described request) do not interrupt the power supply of the module and do not reset the module otherwise it would interrupt the process and irreversible damage the module. Make sure all batteries and accumulators powering modules are fully charged before the IQRF OS change is initiated.

Request

Please note that for security reasons the request requires explicitly specifying HWPID of the special IQRF OS Change handler equal 0xC05E. The request will not be executed if HWPID equals 0xFFFF.

The actual IQRF OS change process after the response is received takes several seconds. During the process, the red LED is on. At the end of the process, the device is reset and the red LED goes off.

NADR	PNUM	PCMD	HWPID	0	1 … 2
NADR	0x20	0x00	0xC05E	Flags	Address

Flags bit 0 Action:
0 Checks all required conditions without performing IQRF OS change.
1 Same as above plus performs IQRF OS change.
bits 1-7 Reserved, must equal 0.

Address A physical address of the external EEPROM memory block containing the IQRF OS change file.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x20	0x80	0xC05E	0	?	Result

Result:

0 All the required conditions are met. IQRF OS change will be performed if Flags.0=1 was specified at the request.

3 Old IQRF OS is not present (old checksum does not match) at the module. IQRF OS change is not possible.

4 The content of the IQRF OS change file stored in the external EEPROM is not valid. IQRF OS change is not possible.

7 IQRF OS change file stored in the external EEPROM has an unsupported version. IQRF OS change is not possible.

13.4.1 IQRF OS Change File

The IQRF OS change file content should be inspected before the file is stored in external EEPROM to find out the versions of IQRF OSs (and optionally DPA) it changes between and to check the file consistency.

File format

0 … 1	2 … 3	4	5	6	7 … 8	9 … 10	11 … 13	14 … 16	17 … Length + 3
Checksum	Length	Version	OsVerTo	OsVerFrom	OsBuildTo	OsBuildFrom	DPAto	DPAfrom	Undocumented

Checksum Fletcher-16 Checksum of the file content starting from the 3rd field Version. The initial checksum value is 0x0000.

Length Length of the file content starting from the 3rd field Version, so the total file length is Length + 4.

Version bit 0-6 File version. Only the values of 0x01 (TR-7xD) and 0x02 (TR-7xG) are supported.
bit 7 Undocumented

OsVerTo IQRF OS version the file changes to. See *moduleInfo* IQRF OS function for more details.

OsVerFrom IQRF OS version the file changes from. See *moduleInfo* IQRF OS function for more details.

OsBuildTo IQRF OS build number the file changes from. See *moduleInfo* IQRF OS function for more details.

OsBuildFrom IQRF OS build number the file changes from. See *moduleInfo* IQRF OS function for more details.

DPAto 3 bytes specifying DPA version to optionally change to.
The first 2 bytes contain the DPA version in the same BCD format the enumeration uses.
3rd byte contains the following flags/bits:
0: DPA implements [C].
1: DPA implements [N].

2: 0=supports both STD and STD+LP networks, 1=supports STD+LP network.
3: SPI interface
4: UART interface
5-7: unused

Note: all 3 bytes are zero when DPA is not updated by the change file.

DPAfrom The DPA version to change from. Same format as DPAto.

13.5 Code Optimization

If the implemented algorithm is already optimal enough and there is still a need to optimize the code in terms of minimizing code size, increasing execution speed, or minimizing memory footprint, an optimization technique could be used. The following chapters describe a few of them. Some techniques are general and some of them are very specific for the CC5X compiler, IQRF ecosystem, or the MICROCHIP PIC MCU. Some techniques are straightforward, some more complex. It is advisable to consult the generated code at the output .lst file in any case.

13.5.1 W as a temporary variable

FLASH- RAM Speed

When the content of the W register is preserved, it can be used as a temporary variable.

```
if ( byte & mask )
  bufferCOM[ 0 ]  = 0xAB;
else
  bufferCOM[ 0 ]  = 0xCD;
```

```
if ( byte & mask )
  W = 0xAB;
else
  W = 0xCD;
bufferCOM[ 0 ] = W;
```

13.5.2 Variable access reorder

FLASH- RAM Speed+

Try to group access to the variables from the same bank to avoid excess MOVLB instructions. By the way, C compilers by definition are free to reorder statements to optimize generated code.

```
uns8 savedTX;
...
RTHOPS = 0xFF; // @bank5 !=
TX = savedTX;  // @bank11 != @bank5 ==
RTDEF = 2;     // @bank5
```

```
uns8 savedTX;
...
TX = savedTX;  // @bank11 != @bank5 ==
RTHOPS = 0xFF; // @bank5 ==
RTDEF = 2;     // @bank5
```

13.5.3 Variable access decomposition

FLASH- RAM Speed+

CC5X is not able to reorder hidden access to the bytes the wider variables consist of so it generates excess MOVLB instructions.

```
bank11 uns16 v11;
bank12 uns16 v12;

if ( v11 == v12 )
  nop();
```

```
bank11 uns16 v11;
bank12 uns16 v12;

if ( v11.low8 == v12.low8 && v12.high8 ==
v11.high8 )
  nop();
```

13.5.4 Explicit MOVLB omitting

FLASH- RAM Speed+

Under certain circumstances and CC5X settings (*-bu* command line option) the CC5X generates excess MOVLB instructions. Using `#pragma updateBank` MOVLB can be suppressed. It is recommended to study .lst files.

```
if ( byte > 0x04 )
  byte = 0;
byte *= 2;
```

```
if ( byte > 0x04 )
  byte = 0;
#pragma updateBank 0
byte *= 2;
#pragma updateBank 1
```

13.5.5 Direct function parameter usage

FLASH- RAM- Speed+

It is advisable to use a variable that maps exactly the fixed-function parameter (when available or when intentionally implemented to save RAM) at a function call to avoid useless data moves between the variable and the respective parameters. For instance, `startLongDelay` maps a parameter `ticks` to the `param3` system variable.

```
uns16 delay;
delay = (uns16)RTDT0 * RTDT1;
startLongDelay( delay );
```

```
uns16 delay @ param3;
delay = (uns16)RTDT0 * RTDT1;
startLongDelay( delay );
```

13.5.6 Avoiding else

FLASH- RAM Speed-

By avoiding *else* branch it is possible to avoid skipping out of the *if* branch. This "*else* before *if* move" is possible only when it does not bring any unwanted side effects and when the slower execution does not matter. It is also better when the original *else* branch code is a faster one and the *if* branch code is less frequent.

```
if ( checkValue( value ) )
  byte |= mask;
else
  byte &= ~mask;
```

```
byte |= mask;
if ( !checkValue( value ) )
  byte &= ~mask;
```

```
bufferCOM[ 0 ]  = 0xCD;
if ( value.1 )
  bufferCOM[ 0 ]  = 0xAB;
```

```
W = 0xCD;
if ( value.1 )
  W = 0xAB;

bufferCOM[ 0 ] = W;
```

13.5.7 Switch instead of if

FLASH- RAM Speed+

CC5X generates more efficient code in the case of the *switch* when an expression value is compared to the more than usually 2 constant values.

```
if ( byte == 1 || byte == 3 )
  _LEDR = 1;
else if ( byte == 7 || byte == 13 )
  _LEDR = 0;
```

```
switch( byte )
{
case 1:
case 3:
  _LEDG = 1;
  break;

case 7:
case 13:
  _LEDG = 0;
  break;
}
```

13.5.8 Function call before return

FLASH- RAM Speed+

If the very last function statement is another function (from the same page) call, then CC5X uses efficiently *goto* instead of *call+return*. It is faster, shorter, and consumes less MCU stack.

```
void Method ()
{
  disableSPI();
  variable = 0;
}
```

```
void Method ()
{
  variable = 0;
  disableSPI();
}
```

```
void Method ()
{
  if ( enable )
       enableSPI();
  else
       disableSPI();
}
```

```
void Method ()
{
  if ( enable )
  {
       enableSPI();
       // return forces CC5X to emit BRA/GOTO
before        // else instead of CALL
       return;
  }
  else
       disableSPI();
}
```

13.5.9 Using goto to avoid redundant code

FLASH- RAM Speed-

CC5X is not able to detect and merge the same tailing code from more blocks that terminate at the same point. *goto* statement will help.

```
switch ( byte )
{
default:
  return TRUE;

case 0:
  variable = 0xbb;
  err = TRUE;
  disableSPI();
  return FALSE;

case 1:
  variable = 0xaa;
  err = TRUE;
  disableSPI();
  return FALSE;
}
```

```
switch ( byte )
{
default:
  return TRUE;

case 0:
  variable = 0xbb;
  goto LABEL;

case 1:
  variable = 0xaa;
LABEL:
  err = TRUE;
  disableSPI();
  return FALSE;
}
```

13.5.10 Avoiding readFromRAM and getINDFx

FLASH- RAM Speed+

IQRF OS allows to use *FSR0, *FSR1, INDF0, INDF1 for memory read purposes instead of inefficient and obsolete `readFromRAM()` and `getINDFx()` calls.

```
byte = readFromRAM( &mask );
```

```
FSR0 = (uns16)&mask;
byte = *FSR0;
```

13.5.11 Advanced C-compiler optimized instructions

FLASH- RAM Speed+

It is efficient to use C-compiler optimized instruction, e.g. *MOVIW*.

```
byte = INDF0; // = *FSR0
FSR0++;
mask = INDF0; // = *FSR0
FSR0 -= 5;
var = INDF0; // = *FSR0
```

```
byte = *FSR0++;
mask = INDF0; // or = *FSR0
var = FSR0[-5];
```

13.5.12 do {} while () is preferred

FLASH- RAM Speed+

If possible *do {} while ()* should be used instead of *while( ){}* or *for(;;) {}* because a jump from the end of the loop is not needed and the condition is evaluated one less time.

```
uns8 loop = 12;
while ( loop != 0 )
{
  // use loop
  loop -= 3;
}
```

```
uns8 loop = 12;
do
{
  // use loop
  loop -= 3;
}
while ( loop != 0 );
```

13.5.13 Use DECFSZ/INCFSZ

FLASH- RAM Speed+

Loop *do {} while ()* with a condition *--var != 0* or *++var != 0* leads to the efficient compilation using *DECFSZ* respectively *INCFSZ* instructions.

```
uns8 loop = 0;
do
{
  // execute loop body
}
while ( ++loop != 10 );
```

```
uns8 loop = 10;
do
{
  // execute loop body
}
while ( --loop != 0 );
```

13.5.14 Widening function parameter

FLASH- RAM- Speed+

Sometimes it is necessary to extend the function parameter size.

```
void Method ( uns8 value )
{
  uns16 var16;
  var16.high8 = 0;
  var16.low8 = value;

  var16 *= 3;
  // use var16
}
```

```
uns16 var16;

void Method ( uns8 value @ var16 )
{
  var16.high8 = 0;

  var16 *= 3;
  // use var16
}
```

13.5.15 Carry as a variable

FLASH- RAM- Speed+

Sometimes the Carry MCU flag can be carefully and efficiently used as a variable.

Also, the following example shows how to compare and store the last value in one step.

```
// Keeps Carry, changes Zero_
#define XorWithAndCopyTo(value,xorWithAndCopyTo) do { \
  W = value; \
  xorWithAndCopyTo ^= W; \
  xorWithAndCopyTo = W; } while(0)

// Compare and copy the last values of PID, TX, and PCMD to detect duplicate packets
Carry = FALSE;

XorWithAndCopyTo( PID, lastPID );
if ( !Zero_ )
  Carry = TRUE;

XorWithAndCopyTo( TX, lastTX );
```

```
if ( !Zero_ )
  Carry = TRUE;

XorWithAndCopyTo( _PCMD, lastPCMD );
if ( !Zero_ )
  Carry = TRUE;

// At least one of 3 parameters must be different to use the packet
if ( !Carry )
...
```

13.5.16 Limiting variable scope

FLASH RAM- Speed

CC5X is not able to detect a minimal variable scope and therefore to effectively share RAM location between the variables. The latest possible variable declaration plus artificial code blocks will help to save some RAM.

```
uns8 temperature;
uns16 capture;

temperature = getTemperature();
bufferCOM[0] = temperature;

captureTicks();
capture = param3;
bufferCOM[1] = capture.low8;
bufferCOM[2] = capture.high8;
```

```
{
  uns8 temperature = getTemperature();
  bufferCOM[0] = temperature;
}

{
  captureTicks();
  uns16 capture = param3;
  bufferCOM[1] = capture.low8;
  bufferCOM[2] = capture.high8;
}
```

13.5.17 Using IQRF variables

FLASH- RAM- Speed+

When there is no risk of memory conflict then IQRF OS variables and function parameters can be used to save RAM and to avoid MOVLBs as these variables reside at the share core RAM area. Such variables can be used when no IQRF functions are not called.

```
uns16 Squared ( uns8 value )
{
  uns8 tempValue = value;
  uns16 squared = (uns16)value *
tempValue;
  return squared;
}
```

```
uns16 Squared ( uns8 value @ param2  )
{
  uns8 tempValue @ param3;
  tempValue = value;
  uns16 squared @ param4;
  squared = (uns16)value * tempValue;
  return squared;
}
```

`param2`, `param3`, `param4` can be used with caution. It is much safer to use user dedicated `userReg0` and `userReg1`.

13.5.18 Parameter mapped to W

FLASH- RAM- Speed+

When the content of the W register is not modified then the very last function parameter can be mapped to it.

```
void Method ( uns8 value )
{
  switch ( value )
  {
  case 1:
```

```
void Method ( uns8 value @ W )
{
  switch ( value )
  {
  case 1:
```

```
  case 2:
       _LEDG = 1;
       break;

  case 4:
  case 8:
       _LEDG = 0;
       break;
  }
}
```

```
  case 2:
       _LEDG = 1;
       break;

  case 4:
  case 8:
       _LEDG = 0;
       break;
  }
}
```

13.5.19 Pointer parameters mapped to FSRx

FLASH- RAM- Speed+

When a function pointer parameter is later used as FSRx, then it is better to directly map this parameter to FSRx.

```
void ZeroMemory (uns16 from, uns8 length)
{
  FSR0 = from;
  do
  {
       setINDF0( 0 );
       FSR0++;
  }
  while ( --length != 0 );
}
```

```
void ZeroMemory (uns16 from@FSR0, uns8
length)
{
  do
  {
       setINDF0( 0 );
       FSR0++;
  }
  while ( -- length != 0 );
}
```

13.5.20 FSRx as a 16-bit variable

FLASH- RAM- Speed+

When FSRx content is preserved then it can be used as a general 16-bit variable to save RAM and avoid MOVLBs. Also because of *ADDFSR* instruction adding/subtracting small constant numbers is very efficient.

.

```
uns16 loop16 = 1000;
uns8 var8;
do
{
  var8 = getTemperature();
  // use loop16 and var8
  loop16 -= 5;
} while ( loop16 != 0 );
```

```
FSR0 = 1000;
uns8 var8 @ FSR1L;
do
{
  var8 = getTemperature();
  // use FSR0 and var8
  FSR0 -= 5;
} while ( FSR0 != 0 );
```

13.5.21 Using FSRx to copy between buffers and variables

FLASH- RAM- Speed+

It is efficient to use FSRx to repeatedly access (copy, compare) the content of buffers and variables to avoid MOVLBs.

```
 RX = bufferRF[0];
 RTDT3 = bufferRF[10];
 var0 = bufferRF[20];
 var1 = bufferRF[30];
```

```
 FSR0 = bufferRF;
 // or even better (shorter, but not faster)
 setFSR0( _FSR_RF );

 RX = FSR0[0];
 RTDT3 = FSR0[10];
 var0 = FSR0[20];
 var1 = FSR0[30];
```

13.5.22 Accessing 16-bit MCU registers

FLASH RAM Speed

The undocumented CC5X (parenthesis) trick can be used to map to the byte pair of the 16-bit MCU variable without warning.

```
  CCPR2L = 0x34;
  CCPR2H = 0x12;
```

```
  uns16 CCPR2 @ ( &CCPR2L );
  CCPR2 = 0x1234;
```

13.5.23 Using IQRF OS offset and limit variables

FLASH- RAM Speed

There are predefined IQRF OS variables that can optimize various copy functions.

```
copyMemoryBlock( bufferRF + 5, bufferINFO + 2,
3);
```

```
memoryOffsetFrom = 5;
memoryOffsetTo = 2;
memoryLimit = 3;
copyBufferRF2INFO();
```

13.5.24 Effective is not always efficient

FLASH- RAM Speed+

Observe the output .lst file when it makes sense.

```
counter += value > maxValue;
```

```
if ( value > maxValue )
  counter++;
```

13.5.25 The assignment also has a value

FLASH- RAM Speed+

This can eliminate extra assignment statements.

```
copyMemoryBlock(bufferAUX, bufferRF,
5);
DLEN = 5;
RFTXpacket();
```

```
copyMemoryBlock(bufferAUX,bufferRF,DLEN=5);
RFTXpacket();
```

13.5.26 Interval detection optimization

FLASH- RAM- Speed+

```
uns8 GetRfRxFilter  ( uns8 rxFilter )
{
  if ( rxFilter < 20 )
       return _FLT_5;

  if ( rxFilter < 35 )
       return _FLT_20;

  if ( rxFilter < 50 )
       return _FLT_35;
  else
       return _FLT_50;
}
```

```
uns8 GetRfRxFilter  ( uns8 rxFilter @ W )
{
  W -= 20;
  if ( !Carry )
       return _FLT_5;
  W -= 35 - 20;
  if ( !Carry )
       return _FLT_20;
  W -= 50 - 35;
  if ( !Carry )
       return _FLT_35;
  else
       return _FLT_50;
}
```

13.5.27 Optimized constants

FLASH- RAM Speed+

It is advisable to use constants, which generate smaller code. In the following example, the lower byte of the constant is 0, therefore a more efficient code is generated but the side effect is minimal.

```
#define DELAY 1000
startLongDelay( DELAY );
```

```
#define DELAY 1024
startLongDelay( DELAY );
```

13.5.28 Equality result

FLASH- RAM Speed+

When a function result is equality of two expressions, then instead of converting the comparison result to the Carry (used to return bit result) it is better to return the difference and to use the Zero_ MCU flag. Carry flag can be even used for smaller/bigger comparisons too.

```
uns8 var1, var2;

bit AreSame ()
{
  return var1 == var2;
}

void APPLICATION  ( void )
{
  if ( AreSame() )
       ...
  else if ( var2 > var1 )
       ...

}
```

```
uns8 var1, var2;

uns8 AreSame ()
{
  return var1 - var2;
}

void APPLICATION  ( void )
{
  AreSame();
  if ( Zero_ )
       ...
  else if ( !Carry )
       ...
}
```

13.5.29 One instruction at the if branch

FLASH- RAM Speed+

If the whole *if* branch is just one instruction long, then a *goto* instruction can be avoided.

```
RandomValue = lsr( RandomValue );
if ( Carry )
  RandomValue ^= 0b10111000;
```

```
RandomValue = lsr( RandomValue );
W = 0b10111000;
if ( Carry )
  RandomValue ^= W; // 1 instruction
```

```
if ( OERR )
{
  CREN = 0;
  CREN = 1;
}
```

```
if ( OERR )
  CREN = 0;

CREN = 1;
```

13.5.30 Utilization of the preloaded W

FLASH- RAM Speed+

CC5X is not able to optimize commutative expressions to use the already preloaded variable or W register.

```
uns8 var1, var2;

var1 = 1;
if ( var1.0 )
{
  if ( var2 == var1 )
       nop();
}
else
```

```
uns8 var1, var2;

var1 = 1;
if ( var1.0 )
{
  if ( var1 == var2 )
       nop();
}
else
```

nop();	nop();

13.5.31 == 1 is more efficient than != 1

FLASH- RAM Speed+

A test == 1 is more efficient (DECFSZ) than a test != 1.

```
if ( var1 != 1 )
  nop2();
else
  nop();
```

```
if ( var1 == 1 )
  nop();
else
  nop2();
```

13.5.32 == 0xFF is more efficient than != 0xFF

FLASH- RAM Speed+

A test == 0xFF is more efficient (INCFSZ) than a test != 0xFF.

```
if ( var1 != 0xFF )
  nop2();
else
  nop();
```

```
if ( var1 == 0xFF )
  nop();
else
  nop2();
```

13.5.33 Expression modification

FLASH- RAM Speed+

Simplifying algebraic expressions can help the CC5X compiler to produce more efficient code.

```
uns8 a, b;

if ( a > ( 16 - b ) )
       nop();
```

```
uns8 a, b;

if ( a + b > 16 )
       nop();
```

13.5.34 Computed goto with a table limit

FLASH- RAM- Speed+

```
void  Table ( uns8 index @ W )
{
#define MAX   2

  // Is index @ W > MAX?
  index = MAX - index;
  if ( !Carry )
       return;      // Above table limit

  skip( index );   // Reverse order because of previous subtraction
#pragma computedGoto 1
  goto _label2;    // or e.g. return 0xEF
  goto _label1;    // or e.g. return 0xCD
  goto _label0;    // or e.g. return 0xAB
#pragma computedGoto 0

label0: // If the last used label is the 1st one then one goto instruction is
avoided
...
```

13.5.35 Default is first at switch

FLASH- RAM Speed+

If there is a *default* used inside the *switch* then it should be the first “case“to avoid internal “goto default” instruction. It might in some cases produce shorter and faster code.

```
switch ( DLEN )
{
  case 12:
       return 21;

  case 34:
       return 43;

  default:
       return 0;
}
```

```
switch ( DLEN )
{
  default:
       return 0;

  case 12:
       return 21;

  case 34:
       return 43;
}
```

13.5.36 Better to return from than after the loop

FLASH- RAM Speed+

It is more efficient to return from the function in the middle of the loop than to exit the loop then return so internal “goto to the return” can be avoided.

```
void function ()
{
  uns8 loop;
  for ( loop = 10; --loop != 0; )
  {
       nop2();
       nop2();
  }
}
```

```
void function ()
{
  uns8 loop;
  for ( loop = 10;; )
  {
       if ( --loop == 0 )
         return;

       nop2();
       nop2();
  }
}
```

The same applies to the return from the function itself.

```
void Function()
{
  if ( condition1 )
  {
       nop();
       if ( condition2 )
       {
         nop();
       }
  }
}
```

```
void Function()
{
  if ( !condition1 )
       return;

  nop();

  if ( !condition2 )
       return;

  nop();
}
```

13.5.37 Modification instead of storing the value

FLASH- RAM Speed+

In special cases, it is better to modify the value of the variable than to assign it as the compiler optimizes to the shorter code. The compiler just increments the value in the example below.

```
#define       STATE_A       0
#define       STATE_B       1
#define       STATE_C       2

  uns8 state;
  if ( condition1 )
    state = STATE_A;
```

```
#define       STATE_A       0
#define       STATE_B       1
#define       STATE_C       2

  uns8 state = STATE_A;
  if ( !condition1 )
  {
```

```
  else
    if ( condition2 )
       state = STATE_B;
    else
       state = STATE_C;
```

```
    state += STATE_B - STATE_A;   // ++
    if ( condition2 )
      state += STATE_C - STATE_B; // ++
  }
```

13.5.38 Assignment compares to 0

FLASH- RAM Speed+

Copying among variables often compares them to zero too (because of MOVF instruction).

```
uns8 variable = *FSR0++;
if ( variable == 0 )
       ...
```

```
uns8 variable = *FSR0++;
if ( Zero_ )
       ...
```

13.5.39 End condition of the 16-bit loop variable

FLASH- RAM Speed+

Sometimes this can be optimized.

```
uns16 var16 = 12345;
do
{
  var16--;
} while ( var16 != -1 );
```

```
uns16 var16 = 12345;
do
{
  var16--;
} while ( var16.high8 != -1 );
// or
FSR1 = 12345;
do
{
  FSR1--; // efficient
} while ( FSR1H != -1 );
```

13.5.40 Shift for a smart comparison

FLASH- RAM Speed+

A comparison of small numbers can be optimized by a shift.

```
uns8 upCount;

if ( upCount > 1 )
// or
if ( upCount >= 2 )
```

```
uns8 upCount;

W = upCount >> 1;
if ( W != 0 )
```

13.5.41 Optimized return TRUE/FALSE

FLASH- RAM Speed-

Each `return TRUE` or `return FALSE` requires two instructions. If there are more such statements it is more efficient to implement a function to just return TRUE or FALSE and to return their value. This leads just to one goto instruction.

```
bit MyFunction()
{
  // Do something
  if ( condition )
       return FALSE;
  // Do something
  return TRUE;
}
```

```
bit returnTRUE()
{
  return TRUE;
}

bit returnFALSE()
{
  return FALSE;
}
```

	bit MyFunction()
	{
	  // Do something
	  if (condition)
	  return returnFALSE();
	  // Do something
	  return returnTRUE();
	}

13.5.42 Avoiding MOVLP #1

FLASH- RAM Speed+

Try to group, if possible, calls of functions from the same Flash page.

copyBufferRF2INFO();
callingAnotherPageFunction();
eeeWriteData(0);
	copyBufferRF2INFO();
	eeeWriteData(0);
	callingAnotherPageFunction();

13.5.43 Avoiding MOVLP #2

FLASH- RAM Speed-

If there are repeated calls of some function residing on another page, then create a function at the current page that calls this function.

#pragma origin __EXTENDED_FLASH
 	
...
  pulseLEDG();
  // Do something
  pulseLEDG();
  // Do something
  pulseLEDG();
  // Do something
  pulseLEDG();
	#pragma origin __EXTENDED_FLASH
	
	  void pulseLEDGfromExtendedFlash()
	  {
	  pulseLEDG();
	  }
	
	...
	  pulseLEDGfromExtendedFlash();
	  // Do something
	  pulseLEDGfromExtendedFlash();
	  // Do something
	  pulseLEDGfromExtendedFlash();
	  // Do something
	  pulseLEDGfromExtendedFlash();

13.5.44 Setting zeroed variables

FLASH- RAM Speed+

When it is for sure the variable is already zero the new value can be ORed in and it might lead to the more efficient code (setting just one bit).

memoryLimit = 64;
eeeWriteData(0);
	// memoryLimit is zero so the next statement takes 1 instruction
	memoryLimit |= 64;
	eeeWriteData(0);

13.5.45 Compare to zero is more efficient

FLASH- RAM Speed+

Comparing to a constant zero value is more efficient than to the other constant numbers. The “~” operator takes one instruction as well as moving variable value to the working W register in the less efficient code.

if ((address & 7) == 7)	if ((~address & 7) == 0)

13.5.46 setFSR01

FLASH- RAM Speed-

Registers FSR0 and/or FSR1 can be efficiently set to the common buffer addresses by calling IQRF OS *setFSRxy* function. Calling this function takes 2 instructions only. Setting both or one of the FSR registers normally takes 8 or 4 instructions respectively.

```
FSR0 = &bufferCOM[0];
FSR1 = &bufferINFO[0];
```

```
setFSR01( _FSR_COM, _FSR_INFO );
```

13.5.47 Pointer arithmetic

FLASH- RAM Speed+

When a variable in RAM is traditionally addressed, the variable content could not lie in more than one PIC RAM bank. Therefore, when a pointer value to a variable is calculated the higher byte of the pointer is never changed and the calculation can be done only on the lower byte.

```
  uns8 indexAtBufferINFO;
...
  setFSR0( _FSR_INFO );
  FSR0 += indexAtBufferINFO;
```

```
  uns8 indexAtBufferINFO;
...
  setFSR0( _FSR_INFO );
  FSR0L += indexAtBufferINFO;
```

Note: The above code might not work when Linear data memory (0x2---) or Program memory (0x8---- … 0xFFFF) is indirectly addressed. It always works with the Traditional data addressing (0x0---) only.

Please note that when a constant value (from -32 to +31) is added to an FRSx register, then the calculation should be always done with the whole register as the optimal PIC ADDFSR instruction is generated.

```
  setFSR0( _FSR_INFO );
  FSR0 += 12;
```

13.5.48 Circular buffer index increment

FLASH- RAM Speed+

When for instance a circular buffer index is incremented and the buffer length is a power of two the buffer index can be incremented a better way.

```
index = (index + 1) %
BUFFER_LENGTH;
```

```
#if 0 != (BUFFER_LENGTH & (BUFFER_LENGTH - 1))
#error BUFFER_LENGTH is not power of 2
#endif

index++;
index &= ~BUFFER_LENGTH;
```

14 DPA Release Notes

Detailed release notes for each DPA version can be found in the following chapters.

14.1 DPA 4.32

IQRF OS: 4.06D-08D8/4.06G-090F (TR-7xD/TR-7xG)

Bug Fixes
- Fixed an issue with unbonded TR-7xG transceivers when the return code of the first Reset event has not been processed.
- Fixed an issue where TR-7xG transmitters were excessively internally calling SetRfDefaults in their main loop.

14.2 DPA 4.31

IQRF OS: 4.06D-08D8/4.06G-090F (TR-7xD/TR-7xG)

Changes and enhancements
- Renamed the `McuType` field of `TPerOSRead_Response` to `TrType`.
- DpaApi[Deep]Sleep with parameter `DpaApiSleep_WdtOff` causes that the watchdog timer is disabled.
- Unused user data from FRC Send and FRC Send Selective is null when received by the node.

New features
- Opening the DPA Menu can be disabled in MenuActivated event.

Bug Fixes
- Fixed an issue where DPA Confirmation did not return 0 hops for the DPA response if the node address was from the interval 0xF0-0xFE. Address 0xFF behaved correctly.
- Fixed an issue where EEEPROM did not work on TR-7xG transceivers equipped with a temperature sensor when the transceiver woke up after Sleep, DpaApiSleep, Standby, or after bonding timeout expired.
- Fixed an issue where the read-only EEEPROM area could be written to when the address range exceeded the writeable/readable area boundary.

14.3 DPA 4.30

IQRF OS: 4.06D-08D8/4.06G-090F (TR-7xD/TR-7xG)

Changes and enhancements
- Event BondingButton was discarded at TR-7xG transceivers.

New features
- DPA Menu at TR-7xG transceivers.
- New events MenuActivated, MenuItemSelected, and MenuItemFinalize at TR-7xG transceivers.
- New API functions DpaApiMenu, DpaApiMenuIndicateResult, and DpaApiMenuExecute at TR-7xG transceivers.

14.4 DPA 4.18

IQRF OS: 4.06D-08D8/4.06G-090F (TR-7xD/TR-7xG)

Bug Fixes
- Fixed an issue when UART Interface did not work at TR-7xG.

14.5 DPA 4.17

IQRF OS: 4.06D-08D8/4.06G-090F (TR-7xD/TR-7xG)

Changes and enhancements
- IO Peripheral was discarded at [C] devices.
- IO Setup was discarded at [C] devices.
- Reduced stack usage by one level in the event Idle.
- Reduced stack usage by one level when calling DpaApiLocalFrc.
- One extra stack level was reduced with the new DpaApiLocalFrc_StackSaver macro compared to the standard DpaApiLocalFrc call.
- Bit Flags.0 at Read response now signalizes an insufficient IQRF OS version but not IQRF OS build.
- InitPHY described at Read TR Configuration.
- Brown-Out Reset (BOR) is enabled when DPA starts at TR-7xD transceivers.
- Brown-Out Reset (BOR) is disabled when entering and re-enabled when exiting Sleep and DpaApiSleep at TR-7xD transceivers.

New features
- Support of IQRF OS 4.05G i.e., TR-7xG transceivers.
- LoadCode can upload IQRF OS change .bin file at TR-7xG transceivers, thus special *CustomDpaHandler-ChangeIQRFOS* handler is not needed anymore.
- OTK bonding available at Bond Node command.
- New API function DpaApiRandom.

Bug Fixes
- Fixed an issue when returned FRC values for 2B and 4B Local FRCs were incorrect.
- Fixed an issue where when [N] bonded to a non-preconfigured network channel (factory default 52) without restarting [N], calling DpaApiSetRfDefaults would cause the originally configured channel to be set, causing [N] to stop communicating. DpaApiSetRfDefaults can be called explicitly in the Custom DPA Handler, internally called in the DPA when the Spirit1 RF chip is locked, or called at the end of a DP2P session.

14.6 DPA 4.16

IQRF OS: 4.04D-08D5 (TR-7xD), 4.05D-08D7 (TR-7xD)

Changes and enhancements
- Command OS Read indicates whether the FRC Aggregation feature is enabled.
- Former DpaApiLocalFrc bug fix workaround (DPA 4.15) is not needed anymore.
- The structure of Smart Connect request parameters was changed. See also the bug fixes section below.
- Default SmartConnect bonding is executed with a stricter RF RX filter by value +4.

New features
- New API function DpaApiAggregateFrc for the FRC aggregation.
- New API function DpaApiSetOTK for the OTK prebonding.
- New API functions DpaApiSleep and DpaApiAfterSleep.
- New DPA API variable FirstDpaApiSleep.
- New API functions DpaApiI2Cinit, DpaApiI2Cstart, DpaApiI2Cwrite, DpaApiI2Cread, DpaApiI2Cstop, DpaApiI2CwaitForACK, DpaApiI2Cshutdown, and DpaApiI2CwaitForIdle for I²C bus communication. Wrapper functions are available to decrease the code size when the original API function is called more than once.
- New DPA API variables I2Ctimeout and I2CwasTimeout for I²C bus communication.
- New Custom DPA Handler templates targeted at [C] and [N] devices.
- New DPA API variable DpaValue.

Bug Fixes

- Fixed an issue when `UserData` parameter at Smart Connect was not passed into `hostUserDataReceived` IQRF OS variable in the bonded [N]. The workaround for the previous DPA versions is to store 4B UserData data into former parameters `VirtualDeviceAddress+reserver[0…2]`.
- Fixed an issue when DpaApiRfTxDpaPacketCoordinator did not transmit the network packet at the AfterRouting event handler when IQRF OS coordinator mode was not restored after executing certain non-networking commands (e.g. bonding). The workaround is to call *setCoordinatorMode* before calling DpaApiRfTxDpaPacketCoordinator.
- Fixed an issue when a 4-byte longer packet length was returned at ReceiveDpaRequest for non-routed (`!_ROUTEF`) packets.
- Fixed an issue when commands Smart Connect and Set MID worked with PNUM of any available embedded peripheral when Coordinator peripheral is available.

14.7 DPA 4.15

IQRF OS: 4.04D-08D5 (TR-7xD)

Changes and enhancements
- SPI (Slave) peripheral was discarded.
- SPI Interface at [N] was discarded.
- Autoexec was discarded.
- The parameter of the Set FRC Params command was extended.

New features
- New API function DpaApiLocalFrc for the execution of the local FRC.
- New event VerifyLocalFrc to verify a received local FRC command.
- New API function DpaApiCrc8 to compute DPA UART Interface compatible CRC.
- New API variable NonroutedRfTxDpaPacket to force TX of a non-routed packet.

Bug Fixes
- Please note a bug fix workaround in DpaApiLocalFrc.

14.8 DPA 4.14

IQRF OS: 4.03D-08C8 (TR-7xD)

Changes and enhancements
- Command Authorize bond can now authorize up to 11 [Ns].
- Smart Connect now directly supports prebonding to the overlapping networks.

New features
- New embedded FRC command Prebonded memory compare.

14.9 DPA 4.13

IQRF OS: 4.03D-08C8 (TR-7xD)

Changes and enhancements
- Command Remove bonded Node also removes the [N] from the list of the discovered [Ns].

New features
- New command Indicate for device indication.
- New event Indicate for custom device indication.
- New API variable AsyncReqAtCoordinator.

Bug Fixes
- Fixed an issue when under certain circumstances the internal call of *bondNewNode* inside Bond Node fails to bond. The workarounds for the previous DPA versions are:

1. Restart the [C] immediately before Bond Node.
2. Use the Custom DPA Handler *CustomDpaHandler-BondNewNode-Workaround* uploaded and enabled at [C] to reset RTHOPS before *bondNewNode* is called. The handler is available at https://www.iqrf.org/dpa.

14.10 DPA 4.12

IQRF OS: 4.03D-08C8 (TR-7xD)

Changes and enhancements
- Command OS Read indicates whether IQRF OS is changed from the originally manufactured version.

Bug Fixes
- Fixed an issue introduced with DPA 3.03 when RF filter, RF power, and RF channel values were continuously set (but not only at startup) according to the actual values at TR Configuration.

14.11 DPA 4.11

IQRF OS: 4.03D-08C8 (TR-7xD)

Changes and enhancements
- When UART Peripheral is enabled at the [N] configuration then the UART is automatically opened shortly before the Init event is raised so the Autoexec is not needed anymore for opening the UART. The UART baud rate is also set in the configuration.
- Restore command does not restore physical settings (e.g. RF band, thermometer sensor presence) of the backed-up transceiver. The settings are now maintained.

Bug Fixes
- Fixed an issue introduced at DPA V4.10 when "Routing off" is enabled at the Configuration and later disabled, then IQRF MESH routing is not working anyway. The workaround for DPA V4.10 was to call setRoutingOn() at the Init event.

14.12 DPA 4.10

IQRF OS: 4.03D-08C8 (TR-7xD)

Changes and enhancements
- Command OS Read was extended.
- Discarded embedded FRC command *UART or SPI data available* (FRC_UART_SPI_data).
- Discarded diagnostic routing LED indication from Set DPA Param.
- Discarded 200 ms long diagnostics timeslot from Set DPA Param.

New features
- New command Factory Settings.
- Factory settings can be applied during the [N] startup by pressing the button.
- DPA Peer-to-peer (DP2P) communication protocol.
- New Random API variable.

14.13 DPA 4.03

IQRF OS: 4.03D-08C8 (TR-7xD)

New features
- Newly documented command RAM Read Any.

Bug Fixes

- Fixed an issue introduced at DPA V4.02 when any of MemoryRead+1 FRC commands also executed a DPA Request from *bufferAUX* (e.g. stored from the previous FRC Acknowledged Broadcast).

14.14 DPA 4.02

IQRF OS: 4.03D-08C8 (TR-7xD)

New features
- New FRC command Memory read 4 bytes.

Bug Fixes
- Fixed an issue when custom bonding of the STD [N] implemented in the Reset event did not store valid RF mode. Therefore the [N] did receive and transmit well.

14.15 DPA 4.01

IQRF OS: 4.03D-08C8 (TR-7xD)

New features
- New command Test RF Signal.

Bug Fixes
- Fixed an issue when Remove bond did not call Disable Interrupts event. This might cause [N] not to restart. [N] then had to be restarted by turning off and on.
- Fixed an issue when the HDLC UART interface packet receiver might get out of sync when the HDLC Flag Sequence byte (0x7e) is not received. The receiver then had to be restarted by turning the device off and on.
- Fixed an issue when the Autonetwork V2 did not unbond unresponsive [Ns] at the end of each wave. It might result in duplicate addresses in the network.

14.16 DPA 4.00

IQRF OS: 4.03D-08C8 (TR-7xD)

Changes and enhancements
- DPA plug-ins for [C] support both STD+LP and STD networks, thus the DPA Plug-in filename format was changed and the former specific STD and LP plug-ins for [C] are not released anymore. New configuration bit.7 at TR Configuration byte index 0x05 selects the network type the [C] controls.
- DPA plug-ins for [Ns] support either both or just STD+LP networks, thus the DPA Plug-in filename format was changed.
- When the [N] supports SPI or UART Interface then neither SPI nor UART embedded peripherals are supported.
- Supported Interface type at [N] is controlled by the upload of the appropriate DPA Plug-in, therefore bit.1 at TR Configuration byte index 0x05 was discarded.
- Remote bonding support removal (Autonetwork takes over) results in discarding:
  - Commands: `CMD_COORDINATOR_READ_REMOTELY_BONDED_MID`, `CMD_COORDINATOR_CLEAR_REMOTELY_BONDED_MID`, `CMD_COORDINATOR_ENABLE_REMOTE_BONDING`, `CMD_NODE_READ_REMOTELY_BONDED_MID`, `CMD_NODE_CLEAR_REMOTELY_BONDED_MID`, and `CMD_NODE_ENABLE_REMOTE_BONDING`.
  - DPA API variables: `ProvidesRemoteBonding`, and `RemoteBondingCount`.
  - Embedded FRCs: `FRC_Prebonding`.
  - Events: `DpaEvent_AuthorizePreBonding`.
- Parameter BondingMask at Bond Node command renamed to BondingTestRetries with the same meaning as in Smart Connect.
- FRC peripheral is always enabled at [C] and disabled at [N] respectively regardless of the configuration settings.

- The Remove bond command also restarts the [N] (except in DSM).
- The meaning of the DPA value bit.7 was changed.
- Discarded commands: `CMD_COORDINATOR_REBOND_NODE`, `CMD_COORDINATOR_DISCOVERY_DATA`, and `CMD_NODE_REMOVE_BOND_ADDRESS`.
- When the [N] is successfully bonded using the button then the green LED is lit for 0.5 seconds but does not wait for the button to be released anymore.

New features
- A new bit at Peripheral enumeration flags indicates the actual IQMESH network type.
- FRC command `FRC_Ping` replaces the value of the former `FRC_Prebonding` command.
- FRC command `FRC_SupplyVoltage`.

14.17 DPA 3.04

IQRF OS: 4.03D-08C8 (TR-7xD)

Bug Fixes
- Fixed an issue when the Send and Send Selective commands of the FRC peripheral passed only 20 bytes of the UserData parameter. The bug was introduced with DPA 3.03.

14.18 DPA 3.03

IQRF OS: 4.03D-08C8 (TR-7xD)
- Please see also Migration Notes.

Changes and enhancements
- Command OS Read reads IBK (Individual Bonding Key) too.
- Default bonding supports Smart Connect.
- The default bonding sleep timeout was extended to 5 hours.
- Read TR Configuration does not XOR values by 0x34 anymore.
- Write TR Configuration does not require checksum to be precomputed anymore.
- EEEPROM Extended Write can write over two adjacent 64-byte pages of the EEPROM chip at once.
- Command Discovery data is marked as obsolete and will be removed in a future release. Use more powerful EEEPROM Extended Read instead.
- Peripheral PWM, which was formerly available only in the Demo version, was finally depreciated.
- Discarded command CMD_LED_GET at LED peripherals.
- DPA Service Mode (DSM) operates with a fixed RX filter of value 5 to be independent of potentially too high filter value at TR Configuration. DSM keeps using full TX power of value 7.
- The meaning of EEEPROM information enumeration parameters changed.
- Timeslot lengths updated for the current IQRF OS version. LP DPA got faster.
- Peripheral RAM buffer *PeripheralRam* is now allocated at the fixed address at bank #12 for sure.
- Compiler CC5X V3.7A is required for compiling the Custom DPA Handlers.

New features
- New command Smart Connect.
- New command Validate bonds.
- New command LED Flashing.
- New command Set MID.
- 4 bytes FRC.
- Two new embedded FRC commands for the Autonetwork V2. Please see Prebonded alive and PrebondedMemoryReadPlus1.
- New embedded FRC command Test RF Signal.
- New API variable BondingSleepCountdown.

Bug Fixes
- Fixed an issue when the DPA for Node without DPA interface support at standard RF mode (HWP-Node-STD-7xD-V302-171116.iqrf or HWP-Node-STD-7xD-V301-170814.iqrf) did not

initialize enabled SPI Peripheral.
→ The workaround was to call *enableSPI()* at DpaEvent_Init event.

- Fixed an issue when the DPA for Coordinator with a UART interface (HWP-Coordinator-STD-UART-7xD-V302-171116.iqrf or HWP-Coordinator-LP-UART-7xD-V302-171116.iqrf) did not shutdown the UART interface before Discovery, Reset, Restart, Run RFPGM, and LoadCode commands are executed. This might cause malfunctioning in case of discovery or missing DPA Response in other cases.
  → The workaround was to enable the Node interface at the TR Configuration although the device is the Coordinator.

14.19 DPA 3.02

IQRF OS: 4.02D-08B8 (TR-7xD)

Changes and enhancements
- Autoexec and IO Setup can use embedded peripherals that are not enabled in the TR Configuration.

New features
- New API variable RxFilter.

Bug Fixes
- Fixed an issue when during precise sleep the drawn current jumps by a few µA under certain GPIO settings.
- Fixes and enhancements at CustomDpaHandler-AutoNetwork example.

14.20 DPA 3.01

IQRF OS: 4.01D-08B7 (TR-7xD)
- Generated DPA version for Node at STD mode without Interface support. The name is "HWP-Node-STD-7xD-V*abc-yymmdd*.iqrf".

Changes and enhancements
- With the introduction of standard IQRF peripherals, former Standard peripherals have been renamed to Embedded peripherals. Field StandardPer has been renamed to EmbeddedPers.
- DpaApiRfTxDpaPacket allows specifying a synchronous or asynchronous message.
- ReceiveDpaRequest is not raised at Remove bond command.
- Response values of Read Temperature have been changed from unsigned to signed integers.
- DpaApiLocalRequest can send a request to the peripheral that is not enabled in the TR Configuration.
- PIC HW UART peripheral interrupts can be handled at the Custom DPA Handler Interrupt event unless the DPA UART peripheral is not open or DPA UART Interface is not used. Formerly they could be handled if the DPA UART peripheral was not enabled in the TR Configuration or DPA UART Interface was not used.
- Both UART Peripheral and Interface now support 230 400 Baud rate.
- A flag indicating a missing Custom DPA Handler was documented at OS Read command.
- A flag indicating that no Interface is supported was introduced at OS Read command.
- The word “General” removed from the DPA plug-in filename.

New features
- Event BondingButton allows a simple redefining of the default (un)bonding button thus saving a considerable amount (around 90 instructions) of the handler code.
- Command Selective Batch allows selecting Nodes that will execute a broadcast request.
- Command Clear & Write & Read that unlike Write & Read clears UART RX buffer at first.
- Macro *IfDpaEnumPeripherals_Else_PeripheralInfo_Else_PeripheralRequest()* compared to *IsDpaEnumPeripheralsRequest()* and *IsDpaPeripheralInfoRequest()* saves some handler code (up to 10 instructions).
- Both FSR0 and FSR1 point to the message PData at the Custom DPA Handler entry.

14.21 DPA 3.00

IQRF OS: 4.00D-08B1 (TR-7xD)
- TR-5xD devices are not supported anymore.
- The demo DPA version is not released anymore.
- DPA for the [CN] device is not released anymore.

Changes and enhancements
- User peripherals do not have to be numbered consequently starting from number 0x20.
- Enumeration response extended by a bitmap specifying implemented user peripheral.
- The interval of allowed PCMD values extended.
- Bonding UserData extended from 2 to 4 bytes at Enable remote bonding and Read remotely bonded module ID.
- Remote bonding can bond up to 7 Nodes. See also Read remotely bonded module ID and RemoteBondingCount.
- MID at Authorize bond extended from 2 to 4 bytes to avoid MID collisions.
- The discovery data address extended to 2 bytes and not multiplied by 16 anymore.
- The meaning of Par1 changed at EEEPROM enumeration.
- The unlimited address range of Extended Read.
- The address range of Extended Write limited to the lower 16 kB of EEEPROM only.
- Changed addresses of Autoexec and IO Setup at EEEPROM.
- IO Setup size extended from 32 to 64 bytes.
- Send FRC returns data from one more extra Node in the case of 1B and 2B FRC commands.
- Slot timing updated according to IQRF OS 4.00.
- Backup and Restore data length increased and AES-128 encrypted using an access password.
- DSM protected and encrypted by an AES-128 using an access password.
- FRC command value is accessible at *_PCMD* variable.
- CustomDpaHandler-ChangeIQRFOS.iqrf HWPID changed.
- The response that is sent when the device is started is marked by the new asynchronous flag.
- Usage of Write TR Configuration and Write TR Configuration byte inside Batch is not limited.
- Command OS Read additionally returns the shortest and the longest timeslot length.
- New parameter at DpaApiSendToIFaceMaster to specify asynchronous packets.
- Discarded commands:
  - `CMD_OS_SET_MID` (irrelevant at IQRF OS 4.00)
  - `CMD_OS_SET_USEC` (unused at current DSM)
  - `CMD_EEEPROM_READ` (use Extended Read instead)
  - `CMD_EEEPROM_WRITE` (use Extended Write instead)

New features
- Command Set Security.
- Deep sleep feature at Sleep.
- DPA API function DpaApiSetRfDefaults.
- IQRF OS Change process can also change the DPA version at the same time.

14.22 DPA 2.28

IQRF OS: 3.08D-0858/3.08D-0879 (TR-5xD/TR-7xD)
- This is the ending major DPA release for TR-5xD.

Changes and enhancements
- The maximum data block length for EEPROM peripheral extended from 32 to 55 bytes.

Bug Fixes
- Fixed an issue when more LP mode [N] devices restarted at the same time caused some of them to delay their start by approximately 2 seconds.

- Fixed an issue when the demo DPA version [C] device responded with `ERROR_NADR` when the broadcast address or the temporary address was specified in the request. The same applies to the demo version of the [CN] device at Bridge command.
- Fixed an issue when the PWM peripheral or the corresponding *CustomDpaHandler-UserPeripheral-PWM.c* example generated unwanted output glitch when PWM parameters were set.
- Improved Sleep accuracy at TR-7xD for times above 2 s.

14.23 DPA 2.27

IQRF OS: 3.08D-0858/3.08D-0879 (TR-5xD/TR-7xD)

Changes and enhancements
- Parameter Mask added to Write TR Configuration byte command.
- Peripheral OS is always enabled regardless of the configuration settings.
- Change of the RF signal filter value at TR Configuration takes effect after the device is restarted.

New features
- Write TR Configuration byte command can write multiple values including RFPGM settings.

14.24 DPA 2.26

IQRF OS: 3.08D-0858/3.08D-0879 (TR-5xD/TR-7xD)

Changes and enhancements
- The size of both read and write peripheral UART Write & Read circular buffers extended from 32 to 64 bytes. A maximum number of bytes transferred by this command extended from 32 to 55 bytes.
- Initial checksum value at LoadCode when loading Custom DPA Handler changed from 0x0000 to 0x0001.
- If Custom DPA Handler is enabled at the TR Configuration but it is missing (not loaded in the Flash memory) then a response return code *ERROR_MISSING_CUSTOM_DPA_HANDLER* is not returned anymore when explicitly a peripheral OS is used. The request to the OS peripheral is executed.
- Set FRC Params now returns previous values.
- Read OS now returns an extra byte reserved for future use.

New features
- Command LoadCode also supports loading code from IQRF plug-ins (.iqrf files). It allows e.g. upgrading the DPA version over the network.
- Implemented *CustomDpaHandler-ChangeIQRFOS.iqrf* handler for changing the IQRF OS version over the network.
- Autonetwork examples support LP mode.

Bug fixes
- Fixed an issue when new commands Extended Read and Extended Write undesirably modified first 3 bytes of peripheral RAM memory space.
- Fixed an issue when the UART interface might receive a frame missing starting HDLC flag Sequence byte 0x7e.

14.25 DPA 2.24

IQRF OS: 3.07D-0852/3.07D-0870 (TR-5xD/TR-7xD)

Changes and enhancements
- Command Discovery data returns 48 bytes instead of formerly 16 bytes.

New features

- New commands Extended Read and Extended Write to access 16 kB of TR-7xD external EEPROM memory.
- New command LoadCode for loading Custom DPA Handler code from external EEPROM into MCU Flash memory.

Bug fixes
- Fixed an issue at TR-7x devices when during precise sleep the current drawn exceeds approx. 500 µA.
- Fixed an issue when released DPA 2.20+ plug-ins for TR-7xD devices overwrite tailing (above size 736) instructions of Custom DPA Handler.
  → Workaround - upload Custom DPA Handler after DPA plug-in, but not in the inverse order.

14.26 DPA 2.23

IQRF OS: 3.07D-0852/3.07D-0870 (TR-5xD/TR-7xD)

Changes and enhancements
- Header files `DPA.h` can be compiled using the GCC compiler in order to help to interface with other frameworks.

Bug fixes
- Fixed an issue introduced at DPA V2.22 when commands Set FRC Params and UART Write & Read accept only no data.

14.27 DPA 2.22

IQRF OS: 3.07D-0852/3.07D-0870 (TR-5xD/TR-7xD)

New features
- New command Write TR Configuration byte.

Bug fixes
- Minimum required IQRF OS build number checked by OS Read for TR-7x devices corrected.

14.28 DPA 2.21

IQRF OS: 3.07D-0852/3.07D-0870 (TR-5xD/TR-7xD)

Changes and enhancements
- IQRF button used e.g. for bonding redefined to GPIO pin `PORTB.4` only.

New features
- Sleep command optionally supports 32.768 ms time unit.
- LpRxPinTerminate API variable allows interrupting LP packet reception by a pin change.

Bug fixes
- Fixed an issue introduced at DPA 2.20 when Batch, Autoexec or IO Setup execution of the embedded request is discontinued when one request does not match HWPID.

14.29 DPA 2.20

IQRF OS: 3.07D-0852/3.07D-0870 (TR-5xD/TR-7xD)
- Support of TR-7xD devices.

Changes and enhancements
- TR-7xD Custom DPA handler Flash memory block extended to 864 instructions.
- [N] and [CN] devices send “Reset” DPA Request when started the same way the [C] already did.
- Read TR request configuration documented and returned checksum updated.

- Bridge response improved.
- DPA API variable *LP_XLP_toutRF* renamed to *LPtoutRF*
- EEEPROM peripheral allows reading and writing of a variable number of bytes.

New features
- 2 byte FRC commands.
- Selective FRC.
- Peer2peer packets.
- Alternative DSM channel.
- New commands Restart, Send Selective, Set FRC Params.
- New predefined FRC commands Memory read, Memory read plus 1, FRC response time.
- New events FrcResponseTime, UserDpaValue, AuthorizePreBonding, PeerToPeer.

Bug fixes
- Fixed an issue when a precise sleep calibration caused exceptionally a shorter time at the very next sleep session.

14.30 DPA 2.13

IQRF OS: 3.06D-0707 (TR-5xD)

Bug fixes
- Fixed an issue when a precise sleep calibration (a part of OS/Sleep request) caused exceptionally an endless sleep of the device.

14.31 DPA 2.12

IQRF OS: 3.06D-0707 (TR-5xD)

Bug fixes
- Fixed an issue when PWM peripheral disabled [N] and [CN] devices until (WDT)reset is executed.
- Fixed an issue when DpaEvent_Interrupt executed *clrdwt()* as the 1st statement at the Custom DPA Handler (i.e. obligatory Handler presence mark) thus causing WDT being cleared every time when an interrupt was raised.

14.32 DPA 2.11

IQRF OS: 3.06D-0707 (TR-5xD)

Bug fixes
- Fixed an issue when a module startup time was significantly delayed in case of a strong service channel jamming.
- DpaTicks variable "frequency" fixed, it was slower by +0.8 %.

14.33 DPA 2.10

IQRF OS: 3.06D-0707 (TR-5xD)

Changes and enhancements
- Foursome parameters NAdr, PNum, PCmd capitalized to NADR, PNUM, and PCMD.
- Foursome parameter HwProfile renamed to HWPID.
- Updated timing recommendation, see DPA Confirmation.
- *DpaEvent_None* event renamed to DpaEvent_DpaRequest.
- CMD_OS_SLEEP - Control bit 0 and bit 3 functionality enhanced and changed.
- Brown-out Reset disabled after the device starts.
- Extra 32 bytes added to both EEPROM and EEEPROM peripherals.
- IQRF OS variable *DataOutBeforeResponseFRC* type changed from *uns16* to *uns8[30]*.

- System DPA value bit 0 returns value of *DSMactivated* variable.
- DpaApiSendToIFaceMaster has a new parameter.
- User DPA Value is stored at the UserDpaValue variable. It is not transferred via the *userReg0* variable at the *Idle* event only anymore.
- Set Hops does not limit the number of hops to the VRN of the addressed and discovered Node anymore.
- UART interface uses more sophisticated 8-bit CRC instead of simple XOR checksum to protect data.
- DpaApiSendToIFaceMaster works even when *IFaceMasterNotConnected* is set in the case when the UART interface is used.
- DpaApiRfTxDpaPacketCoordinator now returns a number of hops to deliver DPA Request back to the Coordinator.

New features
- Full low-power (LP) support (i.e. bonding, Discovery, and FRC).
- FRC Acknowledged Broadcast.
- Custom DPA Handler auto-detection.
- IO Setup (early Autoexec).
- Extra 32 bytes memory space added to EEPROM and external EEPROM peripherals.

Bug fixes
- Fixed an issue when NADR did not contain original sender address at (1.) DpaEvent_Notification event at the [C] device or (2.) inside the Batch request.
- Fixed an issue when NADR did not contain recipient address at DpaEvent_DpaRequest event when DPA Request was part of Batch (or Autoexec) request.
- Fixed an issue with the [C] device where the asynchronous or local requests might not be executed (because of internal HWPID variable was not initialized) until enumeration of [C] peripherals was performed.
- Fixed an issue where at CMD_OS_SLEEP wake up on pin did not work when the calibration was initiated too (always the 1st time the CMD_OS_SLEEP was requested).
- Fixed an issue when using CMD_IO_SET as a part of Autoexec or CMD_OS_BATCH might cause device malfunction.
- Flushing internal buffers of SPI or UART before calling IQRF OS functions that use shared bufferCOM or when the device is going to sleep or reset.
- Improved disabling/enabling SPI/UART peripherals/interfaces before calling IQRF OS functions that use shared bufferCOM or when the device is going to sleep or reset.

14.34 DPA 2.01

IQRF OS: 3.05D-06B5 (TR-5xD)

Bug fixes
- Fixed an issue of DpaApiLocalRequest() API call to allow Custom DPA Handler Interrupt event (now only this event is enabled during the call) to be raised. Missing Interrupt event might cause deadlock resulting in WDT reset.
- Fixed an issue where custom peripheral did not return an error (PNum was not set to PNUM_ERROR_FLAG) at [C] and [CN] devices.

14.35 DPA 2.00

IQRF OS: 3.05D-06B5 (TR-5xD)

Changes and enhancements
- Every DPA Request/Response contains a new 2B HWPID parameter, see General message parameters.
- Changes of parameters or response results of the following commands, services or API: `CMD_COORDINATOR_DISCOVERY`, `CMD_COORDINATOR_BACKUP`, `CMD_COORDINATOR_RESTORE`,

CMD_NODE_ENABLE_REMOTE_BONDING, CMD_NODE_READ, CMD_OS_READ_CFG, CMD_OS_READ, CMD_OS_BATCH, CMD_UART_OPEN, Peripheral enumeration, Autoexec, *DpaApiRfTxDpaPacket*.

- The [C] device sends a „Reset“ message upon startup, see Device Startup.
- Notification event called even after read-only DPA Request.
- Custom DPA Handler location and reserved Flash memory size changed and events renumbered. Custom DPA Handler must be recompiled and uploaded.
- Custom DPA Handler must use case *DpaEvent*_None: instead of the default:
- Event *DpaEvent_Async* renamed to DpaEvent_AfterRouting.
- A Node can address the Coordinator by *COORDINATOR_ADDRESS* or *LOCAL_ADDRESS*. See DpaApiRfTxDpaPacket.
- Changed LED indication style of the forbidden address upon Node startup at demo mode.
- Embedded LED peripherals are not limited to the demo version only.

15 Document Revisions

241205	DPA v4.32 release
240417	DPA v4.31 release
230307	DPA v4.30 release
221005	DPA v4.18 release
220224	DPA v4.17 release
210818	DPA v4.16 release
200903	DPA v4.15 release
200403	DPA v4.14 release
200227	DPA v4.13 release
200109	DPA v4.12 release
191211	DPA v4.11 release
191009	DPA v4.10 release
190612	DPA v4.03 release
190603	DPA v4.02 release
190307	DPA v4.01 release
190110	DPA v4.00 release
181130	DPA v3.04 release
181025	DPA v3.03 release
171116	DPA v3.02 release
170814	DPA v3.01 release
170314	DPA v3.00 release
160912	DPA v2.28 release
160414	DPA v2.27 release
160303	DPA v2.26 release
151201	DPA v2.24 release
151023	DPA v2.23 release
151008	DPA v2.22 release
150903	DPA v2.21 release
150805	DPA v2.20 release
150130	DPA v2.13 release
150115	DPA v2.12 release
141119	DPA v2.11 release
141105	DPA v2.10 release
140602	DPA v2.01 release
140512	DPA v2.00 release

16 Acknowledgement

This project has been made possible with a government grant by means of the Ministry of Industry and Trade of Czech Republic in the TRIO program.

17 Sales and Service

Corporate office, technology and development

MICRORISC s.r.o.
Prumyslova 1275, 506 01 Jicin, Czech Republic, EU
Tel: +420 493 538 125, www.microrisc.com
E-mail (commercial matters): sales@microrisc.com

Support

www.iqrf.org
E-mail (technical matters): support@iqrf.org

Partners and distribution

www.iqrf.org/partners

Quality management

ISO 9001 : 2016 certified

Trademarks

Legal

On-line support: support@iqrf.org