IQRF
DPA Framework

Technical Guide

Version v3.02
IQRF OS v4.00D

16. 11. 2017

Table of Contents

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 IQRF plug-in.

2 Basics

DPA protocol uses byte structured messages to communicate at IQMESH network. Every message always contains four mandatory parameters NADR, PNUM, PCMD and HWPID (foursome from now). The message can optionally hold data (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 at 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 request message except the response message is flagged by the most significant bit of PCMD.

All values wider than byte are coded using little-endian style.

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

There are two device types depending on what type of network device it implements. For each device type, there is dedicated IQRF plug-in to upload.

[C] IQMESH Coordinator device
[N] IQMESH Node device

2.2 RF Modes

There is a separate DPA implementation for each of the IQRF RF modes (STD, LP) (as well as for Device types) prepared in the form of IQRF plug-in. Only STD and LP RF modes are supported. It is not possible to mix devices running at different modes at one IQRF MESH network.

2.3 Interfaces

The chosen interface transfers DPA message to/from the connected device. The message consists of the successively stored foursome and optional data.

2.3.1 SPI

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

2.3.2 UART

UART is configured 8 data bits, 1 stop bit, and no parity bit. UART baud rate is specified at HWP 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 actual data byte (applies to 8-bit CRC value too) equals to 0x7e (Flag Sequence) or 0x7d (Control Escape) then it is replaced by two bytes: a 1^{st} byte is 0x7d (Control Escape) and 2^{nd} byte equals to 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 0xFF, the polynomial is $x^8+x^5+x^4+1$. See CRC Calculation for the implementations of CRC algorithm. There is also an online calculator available.

Example

The example shows encoded DPA Request "write bytes 0x7E, 0x7D at the RAM address 0 at node with address 0x2F":

`NADR=0x002F`(Node address)`, PNUM=0x05`(RAM peripheral)`, PCMD=0x01`(RAM write)`, HWPID=0xFFFF, PData={00`(address)`, {7E, 7D}`(bytes to write)

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

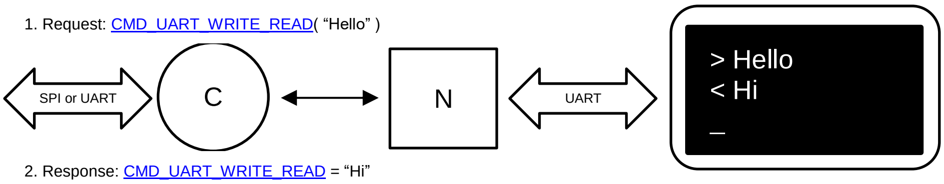
Data in index		0	1	2	3	4	5	6	7		8		CRC		
Data in		0x2f	0x00	0x05	0x01	0xff	0xff	0x00	0x7e		0x7d		0x7e		
Data out index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Data out	0x7e	0x2f	0x00	0x05	0x01	0xff	0xff	0x00	0x7d	0x5e	0x7d	0x5d	0x7d	0x5e	0x7e
Note	*Flag Sequence*	original byte	original byte	original byte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

SPI or UART peripherals differ from SPI or UART interfaces. In general, the peripheral is just byte oriented data channel used to exchange data between the network and external devices while the interface is used to control network device from an external device using DPA messages. In the case of SPI, the external device must be an SPI master as the DPA network device is always an SPI slave.

2.3.3.1 Peripherals

Peripherals are typically used to control an external device connected to the [N] device via SPI or 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” (at this example the terminal automatically answers “Hi” to “Hello”) is read back to the [C] at the corresponding DPA response.



2.4 DPA Plug-in filename

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

HWP-**[device]**-**[rfmode]**-**[interface]**-**[dctr]**-**[version]**-**[date]**.iqrf

Item	Value	Description
[device]	Coordinator	Coordinator device [C]
	Node	Node device [N]
[rfmode]	STD	STD RF mode
	LP	LP RF mode
[interface]	SPI	SPI interface
	UART	UART interface
	<empty>	No interface supported (e.g. [N] at LP RF mode)
[dctr]	7xD	For (DC)TRs of 7xD series
[version]	V*abc*	DPA version *a.bc* (e.g. V*213* stands for version 2.13)
[date]	*yymmdd*	Release date (e.g. *140602* stands for June 2nd, 2014)

2.5 Message parameters

All numbers are in hexadecimal format unless otherwise noted.

Parameter	Value [hex]		Description
NADR [2B]	00 01-EF F0-FB FC FD FE FF 100-FFFF	IQMESH Coordinator IQMESH Node address Reserved Local (over interface) device Reserved IQMESH temporary address IQMESH broadcast address Reserved	Network device address. Although it is 2 bytes wide, the 2B addressing is not supported (a higher byte is ignored).

PNUM [1B]	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E-1F 20-3E 3F 40-7F 80-FF	COORDINATOR NODE OS EEPROM EEEPROM RAM LEDR LEDG SPI IO Thermometer PWM [*] UART FRC Reserved User peripherals Not available Reserved Not available	Peripheral number (0x00 – 0x1F reserved for embedded peripherals) (0x40 – 0x7F reserved for IQRF standard peripherals)
PCMD [1B]	0-3E 3F 40-7F 80-FF	Command value Not available Command value Not available	Command specifying an action to be taken. Actually allowed value range depends on the peripheral type. The most significant bit is reserved for indication of DPA response message.
HWPID [2B]	0000 0001-xxxE xxxF FFFF	Default HW Profile Certified HW Profiles User HW Profiles Reserved	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.
PData [0-56B]	An array of bytes. The maximum length is limited to 56 bytes (decimal).		Optional message data.

[*] Available at Demo version [N] device only. See source code at UserPeripheral-PWM.

2.6 DPA Messages

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

- Master sends DPA request.
- If addressee (NADR) is a (remote) IQMESH Node, not a local over the interface connected device (applies only to coordinator), then:
  - The device immediately sends DPA confirmation back to the interface master.
  - Node processes the DPA message.
- If the DPA message does not have a read-only (can be configured by EnableSPInotificationOnRead) side-effect and the interface is configured for the DPA communication at the node side, then the node sends DPA notification to its SPI master.
  - If the DPA message was not sent using the broadcast address.
    - Node returns DPA response back to coordinator via RF.
    - Coordinator receives the DPA response and re-sends it to the interface master.
- In case of a local device
  - The device processes the DPA request. In this case, the both sender and addressee addresses of the request are equal to 0xFC (local address).
  - The device returns DPA response back to interface master.

2.6.1.1 Interfaces

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] devices must be explicitly enabled at HWP Configuration. 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].

See Examples of the interface usage.

2.6.2 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 HW profile ID in the foursome equals to 0xFFFF (*HWPID_DoNotCheck*). In some scenarios, the request can be asynchronously sent from node to coordinator. Then it is marked as asynchronous the same way as asynchronous DPA Response.

2.6.3 DPA Confirmation

DPA confirmation confirms a reception of DPA request by interface slave to interface master at the coordinator. It consists of the same foursome that was part of the original DPA request plus following 5 additional data bytes. The Confirmation is not returned if the Request is incorrect (e.g. if request NADR is not valid). In this case, 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 node. A hop represents any sending of a packet including sending from the sender as well as from any routing node.
Timeslot length	Timeslot length used to deliver the DPA request to the addressed node. Please note that the timeslot used to deliver the response message from node to coordinator can have a different length.
Hops Response	Number of hops used to deliver the DPA response from the addressed node back to the coordinator. In the case of broadcast, this parameter is 0 as there is no response sent back to the coordinator.

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 RF mode (STD, LP).

PData length [bytes]		**Timeslot length** [ms]	
STD	**LP**	**STD**	**LP**
< 16	< 11	40	80
16 – 39	11 – 33	50	90
> 39	34 – 56	60	100
	> 56		110

This information can be used to implement a precise timing of the control system (master) connected to the coordinator device by the interface in order to prevent data collision (e.g. when another DPA request is sent to the network before a 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 confirmation was received (to be used later at 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 node side then, this time, 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 expected length of data returned within the DPA response or it can be the worst case (e.g. 6 = 60 ms at STD mode). If the Timeslot length from the step 5 equals to the diagnostic long timeslot (20 = 200 ms), then use the same value for the estimated response timeslot length.
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 to: *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, SPI Write & Read, IO Set, OS Sleep, OS Reset). 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 OS Read command returns the shortest and the longest timeslot length.

Example
Next figure shows processing UART Write & Read request. The request is marked Request 1. It writes 5 bytes of data to node [N_n] UART peripheral, waits 20 ms and then reads a number (unknown in advance) of bytes back from UART peripheral. The network is operated at STD mode and 200 ms diagnostic time slot is not used.

After sending **Request 1** to the coordinator [C] the [C] replies by **Confirmation 1**. The confirmation reports *q* hops to deliver a request from [C] to [N_n] with a timeslot of 40 ms and also *r* hops to deliver response back from [N_n] to [C]. After the confirmation is sent the [C] transmits RF packet to the network (1^{st} hop). The packet is received by [N_1] and [N_1] routes the packet further (2^{nd} hop). The routed packet is received by [N_2] as expected. The routing continues. Last but one node [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 node [N_n]. Then [N_{n-1}] makes 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 totally 20 bytes is read which results in the real timeslot of 50 ms to be used to deliver 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 also by [C]. [C] immediately delivers **Response 1** to its interface. In 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.

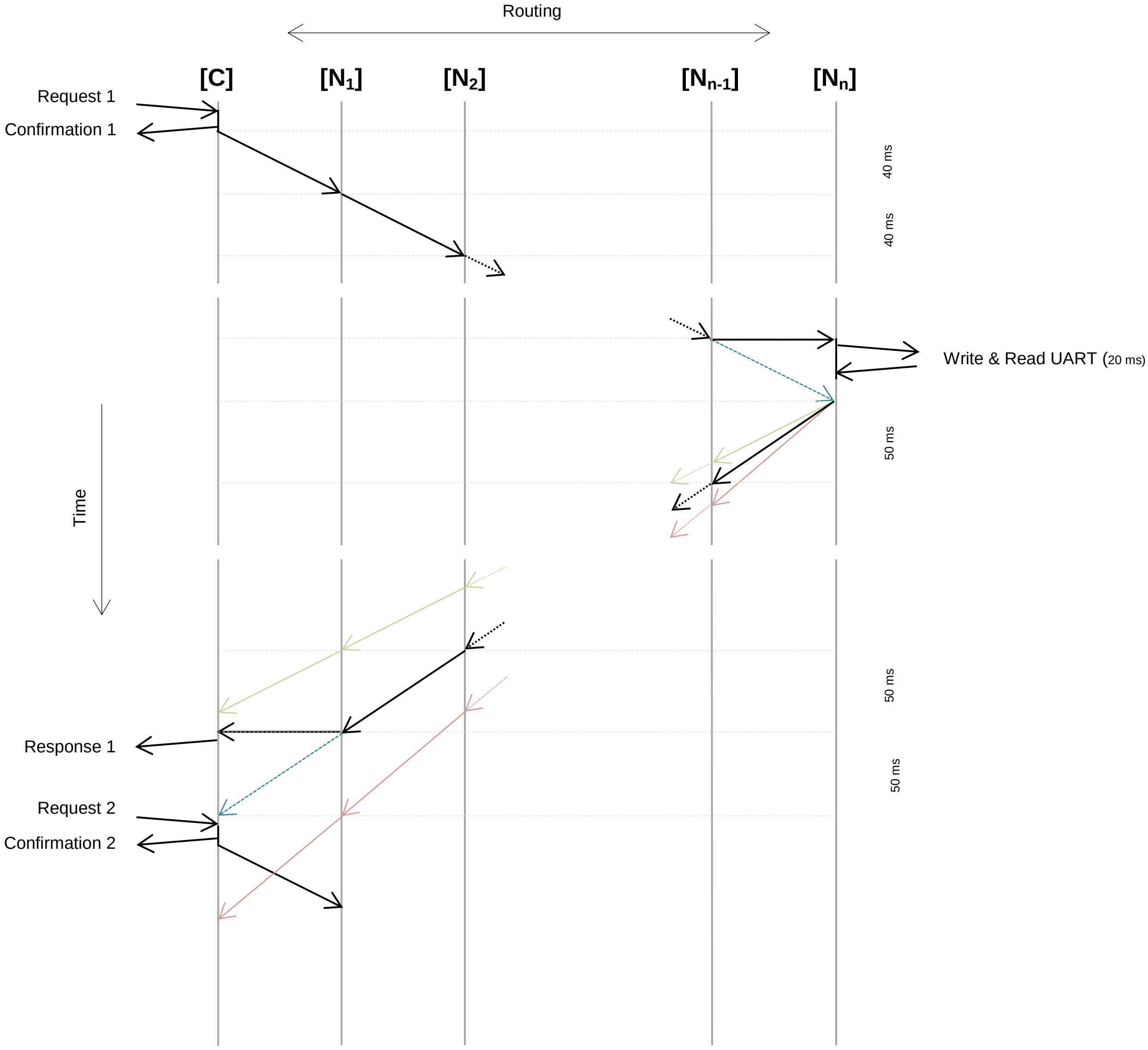
The optimistic response time is:
((q + 1) × 40 ms) + 20 ms + ((r + 1) × 40 ms)

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 the **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 next 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.

2.6.4 DPA Notification

DPA notification notifies a connected master device at the node side that there was a DPA request without a read-only (can be configured by *EnableIFacenotificationOnRead*) side-effect processed by the node. 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 actual HW Profile ID of the device. DPA notification is therefore always 6 bytes long.

DPA notification is issued to the connected master interface when DPA request is sent from the coordinator 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 from predefined FRCs Memory read and Memory read plus 1.

2.6.5 DPA Response

DPA response is an actual answer to the DPA request. DPA response 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 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 to *STATUS_NO_ERROR*), no additional data is present. In the case of a *STATUS_NO_ERROR* response code, the presence of the additional data depends on the DPA response 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 response in the Custom DPA Handler there is sometimes a need to signalize an error response with certain Response Code. The way how to return such response is described at chapter Handle Peripheral Request.

2.6.6 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 coordinator address (0x00FC can be used too)
- PNUM 0x06 Specifies red LED peripheral
- PCMD 0x01 Set LED On command
- DPA Value Coordinator's value

Example 2

Reading 2 bytes from RAM at address 1 of the local node:

- **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 remote IQMESH node 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 node 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 green LED peripheral
- NADR 0x0000 At DPA notification specifies that the Coordinator sent the original request
- DPA Value DPA confirmation: Coordinator's value
  DPA response: 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	PerNr	EmbeddedPers	HWPID	HWPIDver	Flags	UserPer

DpaVer DPA protocol version
- 1st byte: bits 0-6 = minor version, bit 7 = demo 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

PerNr Number of all non-embedded peripherals implemented by Custom DPA Handler. Implemented peripherals are flagged at the UserPer variable-size bitmap array.

EmbeddedPers Bits array (starting from LSb of the 1st byte) specifying which of 32 embedded peripherals are enabled in the HWP Configuration (it is a copy of 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 actually 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 HWP Configuration. The bit value for OS is always set.

HWPID Hardware profile ID, 0x0000 if default.
HWPIDver Hardware profile version, 1st byte = minor version, 2nd byte = major version
Flags Various flags:
- bit 0 STD IQMESH RF Mode supported
- bit 1 LP IQMESH RF Mode supported
- bit 2-7 Reserved

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

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)`,{12,02}`(DpaVer 2.12)`, {02}`(PerNr)`, {E6,06,00,00}`(StdPers)`, {CD,AB}`(HWPID)`, {01,00}`(HWPIDver)`, {41}`(Flags)`, {02,01}`(UserPer)

Coordinator (NADR=0x0000) having 2 user defined peripheral, Hardware profile ID of type 0xABCD (version 0x0001), DPA version 2.12 (not a demo version).
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.TPeripheralInfoAnswer;
```

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 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 request but with most significant bit set to indicate response message
n Number of peripherals the information was returned about.

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];
```

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.

Commands marked *[comdown]* wait for maximum 100 ms to flush output buffers of SPI/UART Peripheral/Interface and then shuts it down. This is to prevent raising HW interrupts or to release OS *bufferCOM* variable that has to be 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` A number of bytes at 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 or its value is above the address limit in case of the DPA demo version.

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 request but with most significant bit set to indicate 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 request but with most significant bit set to indicate 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] devices and it is always enabled there regardless of the configuration settings.

General note: bond state of the node is not synchronized between the node and coordinator. There are separate requests concerning the bonding for node and coordinator.

3.2.1 Peripheral information

PerT PERIPHERAL_TYPE_IQMESH_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.

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 coordinator memory. It actually destroys the network from the coordinator 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 node by the coordinator. There is a maximum approx. 10 s blocking delay when this function is called.

Request

NADR	PNUM	PCMD	HWPID	0	1
NADR	0x00	0x04	?	ReqAddr	Bonding mask

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

Bonding mask See IQRF OS User's and Reference guides (remote bonding, function *bondNewNode*).

Response

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

BondAddr Address of the node newly bonded to the network

DevNr Number of bonded network nodes

Error codes

`ERROR_FAIL`
a. Nonzero ReqAddr is already used.
b. No free address is available when ReqAddr equals to 0.
c. ReqAddr or assigned free address is above the address limit in case of the DPA demo version.
d. Internal call to *bondNewNode* failed.

3.2.6.1 Source code support

```
typedef struct
{
  uns8 ReqAddr;
  uns8 BondingMask;
```

```
} TPerCoordinatorBondNode_Request;

TPerCoordinatorBondNode_Request _DpaMessage.PerCoordinatorBondNode_Request;

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

TPerCoordinatorBondNode_Response _DpaMessage.PerCoordinatorBondNode_Response;
```

3.2.7 Remove bonded node

Removes already bonded node from the list of bonded nodes at coordinator memory.

Request

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

BondAddr	Address of the node 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 node.

3.2.7.1 Source code support

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

TPerCoordinatorRemoveRebondBond_Request
       _DpaMessage.PerCoordinatorRemoveRebondBond_Request;

typedef struct
{
  uns8 DevNr;
} TPerCoordinatorRemoveRebondBond_Response;

TPerCoordinatorRemoveRebondBond_Response
       _DpaMessage.PerCoordinatorRemoveRebondBond_Response;
```

3.2.8 Re-bond node

Puts specified node back to the list of bonded nodes in the coordinator memory.

Request

NADR	PNUM	PCMD	HWPID	0
NADR	0x00	0x06	?	BondAddr

BondAddr	Address of the node to be re-bonded

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x00	0x86	?	0	?	DevNr

DevNr Number of bonded network nodes

Error codes

`ERROR_FAIL` a. BondAddr is already bonded.
b. BondAddr is above the address limit in case of the DPA demo version.

3.2.8.1 Source code support

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

TPerCoordinatorRemoveRebondBond_Request
       _DpaMessage.PerCoordinatorRemoveRebondBond_Request;

typedef struct
{
  uns8 DevNr;
} TPerCoordinatorRemoveRebondBond_Response;

TPerCoordinatorRemoveRebondBond_Response
       _DpaMessage.PerCoordinatorRemoveRebondBond_Response;
```

3.2.9 Discovery

[comdown] Runs IQMESH discovery process. The time when the response is delivered depends highly on the number of network devices, the network topology, 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 Nonzero value specifies maximum node address to be part of the discovery process. This feature allows splitting all node devices 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.
The value of this parameter is ignored at demo version. A value 5 is always used instead.

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.

3.2.9.1 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;
```

3.2.10 Set DPA Param

Sets DPA Param. DPA Param (DPA Parameter) is a one-byte parameter stored at the coordinator RAM that configures network behavior. Default value 0x00 is set upon coordinator reset. The default value can be changed using Autoexec feature.

<table>
<tr><th>Bit</th><th colspan="3">Description</th></tr>
<tr><td rowspan="8">0-1</td><td colspan="3">Specifies which type of DPA Value is returned in every DPA response or DPA confirmation messages:</td></tr>
<tr><td>00</td><td colspan="2">lastRSSI: IQRF OS variable (*). In the case of the [C] device, the value is 0 until some RF packet is received.</td></tr>
<tr><td>01</td><td colspan="2">voltage: Value returned by getSupplyVoltage IQRF OS call (*)</td></tr>
<tr><td rowspan="4">10</td><td colspan="2">system:</td></tr>
<tr><td>bit 0:</td><td>Equals to bit DSMactivated.</td></tr>
<tr><td>bits 1-6:</td><td>Reserved</td></tr>
<tr><td>bit 7:</td><td>(*)</td></tr>
<tr><td>11</td><td colspan="2">user specified DPA Value. See UserDpaValue.</td></tr>
<tr><td rowspan="3">2</td><td colspan="3">If 1, it allows easily diagnosing the network behavior based on following LED activities. Please note that this feature might collide with LED peripheral when used simultaneously giving undesirable effects.</td></tr>
<tr><td colspan="2">Red LED flashes</td><td>When Node or Coordinator receives network message.</td></tr>
<tr><td colspan="2">Green LED flashes</td><td>When Coordinator sends network message or when Node routes network message.</td></tr>
<tr><td>3</td><td colspan="3">If 1, then instead of using ideal timeslot length, a long fixed 200 ms timeslot is used. It allows easier tracking of network behavior.</td></tr>
<tr><td>4-7</td><td colspan="3">Reserved</td></tr>
</table>

(*) The highest 7th bit indicates, that the node, that returned the DPA response, provided a remote pre-bonding to another node. Then Node peripheral commands can be used to find out its module ID and proceed with node authorization using Coordinator peripheral.

DPA Param is transparently sent with every DPA message from the coordinator 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	?	DPA Param

DPA Param   DPA Param to set.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0

NADR	0x00	0x88	?	0	?	DPA Param

DPA Param Previous value

3.2.10.1 Source code support

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

TPerCoordinatorSetDpaParams_Request_Response
       _DpaMessage.PerCoordinatorSetDpaParams_Request_Response;
```

3.2.11 Set Hops

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

Request

NADR	PNUM	PCMD	HWPID	0	1
NADR	0x00	0x09	?	Request Hops	Response Hops

Hops values:

0x00, 0xFF: See a description of the parameter of function *optimizeHops* in the IQRF OS documentation. 0x00 does not make sense for Response Hops parameter.

0x01 – 0xEF: Sets number of hops to the value *Request*/Response*Hops - 1*.
The result of [Discovery data](#) command can be used to find out an optimal number of hops based on destination node logical address or virtual routing number respectively.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0	1
NADR	0x00	0x89	?	0	?	Request Hops	Response Hops

Request/Response Hops Previous values

3.2.11.1 Source code support

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

TPerCoordinatorSetHops_Request_Response
       _DpaMessage.PerCoordinatorSetHops_Request_Response;
```

3.2.12 Discovery data

Allows reading of coordinator internal discovery data. Discovery data can be used for instance for IQMESH network visualization and traffic optimization. Discovery data structure is documented at IQRF OS Operating System User's Guide, Appendix “Coordinator Bonding and Discovery Data”.

Request

NADR	PNUM	PCMD	HWPID	0 … 1
NADR	0x00	0x0A	?	Address

Address Address of the discovery data to read. See IQRF OS documentation for details.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … 47
NADR	0x00	0x8A	?	0	?	Discovery data

DiscoveryData Discovery data read from the coordinator private external EEPROM storage

Error codes
`ERROR_FAIL` Error accessing serial EEPROM chip.

3.2.12.1 Source code support

```
typedef struct
{
  uns16        Addr;
} TPerCoordinatorDiscoveryData_Request;

TPerCoordinatorDiscoveryData_Request _DpaMessage.PerCoordinatorDiscoveryData_Request;

typedef struct
{
  uns8 DiscoveryData[48];
} TPerCoordinatorDiscoveryData_Response;

TPerCoordinatorDiscoveryData_Response
       _DpaMessage.PerCoordinatorDiscoveryData_Response;
```

3.2.13 Backup

This command reads coordinator network information data that can be then restored to another coordinator in order to make a clone of the original coordinator. The backup data structure is not public and it is encrypted (except the very last byte) by an AES-128 algorithm using access password as a key.

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	?	Network data

Network data One block of the coordinator network info data

To read all data blocks just start with Index = 0 and execute the Backup request. Then store received data block from the response. The last of the read data specifies how many data blocks remains 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.

3.2.13.1 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;
```

3.2.14 Restore

The command allows writing previously backed up coordinator network data to the same or another coordinator 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 access password as a key, the access password at the device must be same as the access password at the device that was originally backed up.

The following conditions must be met to make the coordinator backup fully functional:

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

Request

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

NetworkData One block of the coordinator network info data previously obtained by 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. coordinator data used to restore node or vice versa) NetworkData content.

`ERROR_FAIL` Error accessing serial EEPROM chip.

3.2.14.1 Source code support

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

TPerCoordinatorNodeRestore_Request _DpaMessage.PerCoordinatorNodeRestore_Request;
```

3.2.15 Authorize bond

Authorizes previously remotely pre-bonded node. This assigns the node the final network address. See IQRF OS documentation for more information about remote bonding concept.

Request

NADR	PNUM	PCMD	HWPID	0	1 … 4
NADR	0x00	0x0D	?	ReqAddr	MID

ReqAddr See Bond node request. If 0xFF is specified then the pre-bonded node is unbonded and then reset.

MID Module ID of the node to be authorized. Module ID is obtained by calling Read remotely bonded module ID.

Response: see response of Bond node command (except PCMD is 0x8D).

Error codes

`ERROR_FAIL`	a. Nonzero ReqAddr is already used. b. No free address is available when ReqAddr equals to 0. c. ReqAddr or assigned free address is above the address limit in case of the DPA demo version. d. Internal call to *nodeAuthorization* failed.

3.2.15.1 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;
```

3.2.16 Enable remote bonding

Implemented at [C] devices. Has the same behavior as Enable remote bonding except PNUM = 0x00 and PCMD = 0x11.

3.2.17 Read remotely bonded module ID

Implemented at [C] devices. Has the same behavior as Read remotely bonded module ID except PNUM = 0x00 and PCMD = 0x0F.

3.2.18 Clear remotely bonded module ID

Implemented at [C] devices. Has the same behavior as Clear remotely bonded module ID except PNUM = 0x00 and PCMD = 0x10.

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 node is not synchronized between the node and coordinator. There are separated requests for node and coordinator concerning the bonding.

3.3.1 Peripheral information

PerT	`PERIPHERAL_TYPE_IQMESH_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 node 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 Node device 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] The node is marked as unbonded (removed from network) using *removeBond()* IQRF OS function. Bonding state of the node on the coordinator side is not affected at all. Please note, that the node will not receive messages anymore from the network after this command. Therefore this command is often combined with a subsequent Restart command inside one Batch command.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x01	0x01	?

Response

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

3.3.4 Enable remote bonding

Puts node into a mode that provides a remote bonding of up to 7 new nodes. Remote bonding gives the new node temporary network address (0xFE). This process is called pre-bonding. A final logical network address is provided to the node using Authorize bond command. Then the node can be discovered and its virtual routing number is assigned. See IQRF OS documentation for more information about remote bonding concept.

Node stays in the remote bonding mode even if all 7 nodes were pre-bonded. It allows to the already pre-bonded node to be pre-bonded again, pre-bonding of another node is rejected. This gives possibility the new node to try pre-bonding again in the case when it did not receive pre-bonding confirmation after the previous bonding requests. Also, see bit ProvidesRemoteBonding.

Request

NADR	PNUM	PCMD	HWPID	0	1	2 … 5
NADR	0x01	0x04	?	BondingMask	Control	UserData

BondingMask See IQRF OS User's and Reference guides (remote bonding, function *bondNewNode*).
Control bit 0 Enables remote bonding mode. If enabled then previously bonded nodes are forgotten.
bit 1-7 Reserved
UserData Optional data that can be used at Reset Custom DPA Handler event.

Response

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

3.3.4.1 Source code support

```
typedef struct
{
  uns8        BondingMask;
  uns8        Control;
  uns8        UserData[4];
} TPerCoordinatorNodeEnableRemoteBonding_Request;

TPerCoordinatorNodeEnableRemoteBonding_Request
      _DpaMessage.PerCoordinatorNodeEnableRemoteBonding_Request;
```

3.3.5 Read remotely bonded module ID

This command returns module IDs and user data of the remotely pre-bonded nodes. Non-user DPA Values also indicate if any node was pre-bonded. See Set DPA Param and RemoteBondingCount.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x01	0x02	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0 … 3	4 … 7	…	8×(n-1) …	… 8×n-1
NADR	0x01	0x82	?	0	?	MID	UserData		MID	UserData

The response contains a list of MID/UserData pairs of pre-bonded nodes. If no node was pre-bonded no data is returned.

MID Module ID of the remotely pre-bonded node. It can be used later for bonding authorization later. See Authorize bond.
UserData Optional bonding user data specified at Reset Custom DPA Handler event.

3.3.5.1 Source code support

```
typedef struct
{
  uns8 MID[4];
  uns8 UserData[4];
} TPrebondedNode;

typedef struct
{
  TPrebondedNode  PrebondedNodes[ DPA_MAX_DATA_LENGTH / sizeof(TPrebondedNode) ];
} TPerCoordinatorNodeReadRemotelyBondedMID_Response;

TPerCoordinatorNodeReadRemotelyBondedMID_Response
```

```
_DpaMessage.PerCoordinatorNodeReadRemotelyBondedMID_Response;
```

3.3.6 Clear remotely bonded module ID

This call makes a node forget of the nodes that were previously remotely pre-bonded. After calling this command calling of Read remotely bonded module ID returns no data. This command does not affect remote bonding mode enable/disable state.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x01	0x03	?

Response

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

3.3.7 Remove bond address

[sync] The node stays in the IQMESH network (it is not unbonded) but a temporary address 0xFE is assigned to it. This allows to address it (them) or to authorize it later by AuthorizeBond. It is highly recommended to read the device's Module ID before removing bond address to be able to authorize it later.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x01	0x05	?

Response

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

3.3.8 Backup

Same as coordinator Backup except PNUM = 0x01 and PCMD = 0x06.

3.3.9 Restore

Same as coordinator Restore except PNUM = 0x01 and PCMD = 0x07.

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 month of the date of the DPA build, higher nibble contains year above 2010.

Example: Par1=0x31, Par2=4A => 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	8	9	10	11
NADR	0x02	0x80	?	0	?	ModuleID	OSVersion	TR&McuType	OsBuild	Rssi	SupplyVoltage	Flags	SlotLimits

ModuleID,
OSVersion,
TR&McuType,
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 OsBuild for the used DPA version.
bit.1 is 0 if SPI interface is supported; 1 if 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 HWP Configuration. See details of the handling of this erroneous state.
bit.4 is 1 if no interface is supported.
bit.5-7 are reserved.

SlotLimits — Lower nibble stores 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.

3.4.2.1 Source code support

```
typedef struct
{
  uns8        ModuleId[4];
  uns8        OsVersion;
  uns8        McuType;
  uns16       OsBuild;
  uns8        Rssi;
  uns8        SupplyVoltage;
  uns8        Flags;
  uns8        SlotLimits;
} TPerOSRead_Response;

TPerOSRead_Response _DpaMessage.PerOSRead_Response;
```

3.4.3 Reset

[sync] [comdown] Forces (DC)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 (DC)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 HWP configuration

Reads a raw HWP configuration memory. Bit values for Coordinator (bit 0) and Node (bit 1) peripheral stored at HWP configuration are set the same way as at Peripheral enumeration.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x02	0x02	?

Response

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

Checksum	Checksum of the Configuration part.
Configuration	Content the configuration memory block from address 0x01 to 0x1F.
RFPGM	See parameter of *setupRFPGM* IQRF OS function.

This command returns all bytes both from Checksum and Configuration sections being XORed by byte value 0x34 (other bytes are not XORed). The Checksum byte XORed with all Configuration bytes gives 0x5F.

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 HWP configuration

Writes HWP configuration memory. It is a programmer's responsibility to prepare correct configuration block including 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 HWP configuration for details.

Request

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

Checksum	Checksum of the Configuration part. The Checksum byte XORed with all Configuration bytes gives 0x5F.
Configuration	Content the configuration memory block from address 0x01 to 0x1F.
RFPGM	See parameter of *setupRFPGM* IQRF OS function.

Response

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

Example

Following example shows writing RF output power value to the configuration in the Custom DPA Handler code.

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

// Decode configuration
FSR0 = _DpaMessage.Response.PData + sizeof( _DpaMessage.PerOSWriteCfg_Request.Checksum
) + sizeof( _DpaMessage.PerOSWriteCfg_Request.Configuration );
do
{
  setINDF0( *--FSR0 ^ 0x34 );
} while ( FSR0.low8 != ( _DpaMessage.Response.PData & 0xff ) );

// Update checksum
_DpaMessage.PerOSWriteCfg_Request.Checksum ^=
_DpaMessage.PerOSWriteCfg_Request.Configuration[CFGIND_TXPOWER -
sizeof(_DpaMessage.PerOSWriteCfg_Request.Checksum)] ^ txPowerToSet;
// Update TX power
_DpaMessage.PerOSWriteCfg_Request.Configuration[CFGIND_TXPOWER -
sizeof(_DpaMessage.PerOSWriteCfg_Request.Checksum)] = txPowerToSet;

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

3.4.6.1 Source code support

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

TPerOSWriteCfg_Request _DpaMessage.PerOSWriteCfg_Request;
```

3.4.7 Write HWP configuration byte

Writes multiple bytes to the HWP 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 actually performed the configuration write. Please note that the device should be restarted for some configuration changes to take effect. See HWP configuration for details.

Request

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

Address: Address of the item at configuration memory block. The valid address range is 0x01-0x1F for configuration values. Also, address 0x20 is a valid value for RFPGM settings. See parameter of *setupRFPGM* IQRF OS function.

Value: Value of the configuration item to write.

Mask: Specifies bits of the configuration 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 to 0xFF then the whole Value will be written to the configuration byte. For example, when Mask equals to 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 device into RFPGM mode configured at HWP Configuration. The device is reset when RFPGM process is finished. RFPGM runs at same channels (configured at HWP 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.

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

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

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

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 calibration process without performing sleep).
Control	• bit 0	Wake up on PORTB.4 pin negative edge change. See *iqrfSleep* IQRF OS function for more information.
	• bit 1	Runs 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 to run the calibration always before a longer 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 calibration was not already initiated by Time=0 and Control.bit.1=1.
	• bit 2	If set, then if 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 *iqrfSleep* IQRF OS function for more information.
	• bit 4	If set then the unit is 32.768 ms instead of 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.

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 access password stored at Data using *setAccessPassword.* IQRF OS function
	1	Sets user key stored at Data using *setUserKey* IQRF OS function.
	other	Reserved
Data	See Type above.	

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 sender's and addressee's addresses of each embedded request equal to the corresponding addresses of the original Batch DPA request. It is not allowed to embed Batch command itself within series of individual DPA requests. Using neither Run discovery is not allowed inside 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 — Contains more DPA requests to be executed. The format at which the DPA requests are stored is the same as the format of Autoexec DPA requests. See Autoexec 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 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 in addition 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];
} STRUCTATTR 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 actually 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, Autoexec 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 two 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 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 .hex file is doubled so the code starts from address 0x7440 at the .hex file. Please read 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 (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 last 64-byte block. We recommend filling in unused trailing bytes by value 0x34FF in order 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 HWP 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 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 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.

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.
bit 1 Code type:
0 Loads Custom DPA Handler.
1 Loads IQRF plug-in.
bits 2-7 Reserved, must equal to 0.

Address A physical address at external EEPROM memory to load the code image from. The address value is recommended to be a multiple of 64 because it allows more effective writing the code image to the memory.

Length Length of the code image in bytes at the external EEPROM. See text above.

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 text above. Different initial checksum values for both types of upload code ensure that code types cannot be confused.

Response

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

Result bit 0 1 The checksum matches a checksum of a code at the external EEPROM. The code will be loaded if Flags.0=1 was specified at the request.
0 The checksum does not match.
bit 1-7 Unused, equals to 0.

3.4.13.1 Source code support

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

TPerOSLoadCode_Request _DpaMessage.TPerOSLoadCode_Request;
```

3.5 EEPROM

PNUM = 0x03

This peripheral controls internal MCU EEPROM memory.

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 to 192 at [N] device or 64 at [C] respectively.
Par2 Maximum data block length. In the current version of DPA it equals to 55 bytes.

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

3.5.2 Read

Reads data from the memory.

Request

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

Address An address to read data from.
Len 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` 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` 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 present ERROR_FAIL code is returned. Please note that the part of the external EEPROM memory space can be used for Autoexec and/or IO Setup.

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 to 0x80.
Par2	Data block size (equals to 16). The parameter is used by Read & Write commands.

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	Len

Address	A physical address to read data from.
Len	Length of the data to read in bytes. Allowed range is 0-54 bytes. Reading behind 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	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;
} STRUCTATTR 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 the memory.
n	Length of the data to write in bytes. Allowed range is 1-54 bytes. Writing to multiple adjacent 64-byte pages of the EEPROM chip or behind maximum address range by one extended write command is unsupported and undefined. Please see IQRF OS documentation for *eeeWriteData* function details.

Response

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

Error codes

`ERROR_ADDR` 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;
} STRUCTATTR TPerXMemoryRequest;

TPerXMemoryRequest _DpaMessage.XMemoryRequest;
```

3.7 RAM

PNUM = 0x05

This peripheral controls block of internal MCU RAM memory. 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.

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 equals to 48.
Par2 Maximum data block length. In the current version of DPA equals to 48.

3.7.2 Read & Write

See EEPROM.

3.7.2.1 Source code support

```
#pragma rambank = 12
uns8  PeripheralRam[PERIPHERAL_RAM_LENGTH];
```

3.8 SPI (Slave)

PNUM = 0x08

The peripheral is not available at the Coordinator [C] device. The peripheral is not available at [N] devices supporting UART interface too.

The usage of the peripheral is limited at LP mode because the device regularly sleeps 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 details below.

3.8.1 Peripheral information

PerT `PERIPHERAL_TYPE_SPI`
PerTE `PERIPHERAL_TYPE_EXTENDED_READ_WRITE`
Par1 Maximum data block length
Par2 Not used

3.8.2 Write & Read

Writes and/or reads data to/from SPI peripheral. See UART Write & Read which uses the same read & write logic except PNUM = 0x08 and PCMD = 0x00.

3.9 LED

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

Please note that at LP mode the device regularly enters a sleep mode when waiting for a packet so the LED is switched off. To keep 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 constant.
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 Get

Returns a state of the LED.

Request

NADR	PNUM	PCMD	HWPID
NADR	0x06 or 0x07	0x02	?

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x06 or 0x07	0x82	?	0	?	OnOff

OnOff   0x01 when LED is on, 0x00 when LED is off

3.9.4 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.

3.10 IO

PNUM = 0x09

This peripheral controls IO pins of the MCU. Please note that the pins used by an internal IQRF (DC)TR module circuitry cannot be used and their control by this peripheral is blocked. See a corresponding IQRF (DC)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 supported 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 setup weak pull-ups at the pins where available. See datasheet of the PIC MCU for a description of IO ports.

Request

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

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. 0x11=WPUB, 0x14=WPUE (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 an 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 response is sent back to the device that sent the original DPA IO Set request. Therefore, if an invalid port is specified an error code is not returned inside DPA response but the rest of the request execution is skipped.

Request

NADR	PNUM	PCMD	HWPID	0	1	2	…	n × 3	n × 3 + 1	n × 3 + 2
NADR	0x09	0x01	?	$command_0$			…	$command_n$		

triple	There are 2 types of 3-byte commands allowed:
a. Setting an output value
port	Specifies the port to setup 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

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

Setting of 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 supported the 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	?	TempC	Temp16	

TempC Temperature in °C, integer part, not rounded.
See return value of *getTemperature IQRF* OS function. If the temperature sensor is not installed (see HWP Configuration) then the returned value is 0x80 = -128 °C.
Temp16 Complete 12 bit value of the temperature in 1/16 = 0.0625 °C units with 0.5 °C resolution. See *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

PNUM = 0x0B for standard MCU PWM peripheral

The peripheral is available at Demo version, STD mode and at the [N] device only. The source code of the demo version implementation of the PWM peripheral is available among custom DPA handler examples. See *CustomDpaHandler-UserPeripheral-PWM.c*.

3.12.1 Peripheral information

PerT PERIPHERAL_TYPE_PWM
PerTE PERIPHERAL_TYPE_WRITE
Par1 Not used
Par2 Not used

3.12.2 Set

Sets PWM parameters.

Request

NADR	PNUM	PCMD	HWPID	0	1	2
NADR	0x0B	0x00	?	Prescaler	Period	Duty

Prescaler bit <1:0> codes prescaler values at T6CON register:

• 11 = prescaler is 64
• 10 = prescaler is 16
• 01 = prescaler is 4
• 00 = prescaler is 1
bit <5:4> codes two least significant bits of 10bit Duty cycle <1:0>.

Period	Sets the PR6 register for PWM period.
Duty	Eight most significant bits of 10bit duty cycle value <9:2>. It sets the CCPR6 register.

When all 3 parameters equal to 0, PWM is stopped.

Response

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

Error codes

`ERROR_DATA` Invalid Prescaler value.

Example 1

Set PWM for 1 kHz with 50% of duty cycle and prescaler 16:

- **DPA request** (master > slave)
`NADR=0x0001, PNUM=0x0B, PCMD=0x00, HWPID=0xFFFF, PData={0x02,0x7d,0x40}`
- **DPA response** (slave > master)
`NADR=0x0001, PNUM=0x0B, PCMD=0x80, HWPID=0xABCD, PData={0x00}`(No error)

Example 2

Set PWM for 1 kHz with 70% of duty cycle and prescaler 16:

Note: prescaler value is 0x02 = 0b00000010, but the duty cycle value is in this case 0x15E = 0b101011110, the bits<1:0> (0b1010111**10**) are added into Prescaler value (0b00**10**0010 = 0x22) to bits <5:4> and the seven most significant bits (0b**1010111**10) are written into Duty (0b1010111 = 0x57).

- **DPA request** (master > slave)
  `NADR=0x0001, PNUM=0x0B, PCMD=0x00, HWPID=0xFFFF, PData={0x22,0x7d,0x57}`
- **DPA response** (slave > master)
  `NADR=0x0001, PNUM=0x0B, PCMD=0x80, HWPID=0xABCD, PData={0x00}`(No error)

3.12.2.1 Source code support

```
typedef struct
{
  uns8  Prescaler;
  uns8  Period;
  uns8  Duty;
} TPerPwmSet_Request;

TPerPwmSet_Request _DpaMessage.PerPwmSet_Request;
```

3.13 UART

PNUM = 0x0C for embedded UART peripheral

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

The usage of the peripheral is limited at LP mode because the device regularly sleeps 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 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.

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 to 55 bytes.
Par2 Not used

3.13.2 Open

This command opens UART peripheral at 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)

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 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)

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) `{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 )];
} TPerUartSpiWriteRead_Request;

TPerUartSpiWriteRead_Request _DpaMessage.PerUartSpiWriteRead_Request;
```

3.13.1 Clear & Write & Read

Same as Write & Read from above except it clears 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.

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 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. Type of the collected information is specified by a byte called 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) or 2 bytes from up to 31 nodes (having logical addresses 1-31). Type of collected data is specified by 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 – 0xFF	0xE0 – 0xEF	0xF0 – 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 node (1-63) are stored in bytes 1-63 of the output buffer.

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

For more information see IQRF OS manuals. If the node 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.

Request

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

FRC Command	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.

Response

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

Status Return code of the *sendFRC* IQRF OS function. See IQRF OS documentation for more information.

FRC data 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 total 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	?	FRC data

FRC data Remaining FRC data that could not be read by FRC Send command because 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 DPA request at selected nodes only and get the confirmation plus one data bit from selected nodes. Both request and response have the same structure as Send except SelectedNodes field. Also, the length of 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).

Request

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

FRC Command — Specifies data to be collected.

SelectedNodes — Specifies a bitmap with selected nodes. Bit_1 of the 1^{st} byte of the bitmap represents node with address 1, bit_2 of the 1^{st} byte of the bitmap represents node with address 2, …, bit_7 of the 30^{th} 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 data is from 2 to 25 bytes.

Response

See Send DPA response.

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	?	FRCresponseTime

FRCresponseTime — Value corresponding to one of the constants `_FRC_RESPONSE_TIME_??_MS` (see *IQRF-macros.h*) to set maximum time reserved for preparing return FRC value. See IQRF OS documentation for more details.

Response

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

FRCresponseTime — Previous FRCresponseTime value.

3.14.5.1 Source code support

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

TPerFrcSetParams_RequestResponse _DpaMessage.PerFrcSetParams_RequestResponse;
```

3.14.6 Embedded FRC Commands

There are a few embedded FRC commands. The user can implement 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 value 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 commands 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 Prebonding

FRC_Prebonding = 0x00

Collects bits. Gives detail information about the state of pre-bonding. Bit 0 is 1 when a node is accessible; bit1 is 1 if the node provided pre-bonding to a new node. If bit 0 of the 1st user data byte sent with FRC command is set, the remote bonding at node device is also disabled. Subsequently, detail information can be read using Read remotely bonded module ID from the node.

3.14.6.2 UART or SPI data available

FRC_UART_SPI_data = 0x01

Collects bits. Bit 0 is 1 when a node is accessible; bit1 is 1 when there is some data available for reading from UART or SPI peripheral.

3.14.6.3 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.

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 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, FrcValue event is raised to allow setting resulting Bit.1 by the user. The sender address of the embedded DPA request equals to 0x00 (coordinator address) and the addressee addresses is 0xFF (broadcast address).

Returned bits:

bit 0	bit 1	Description
0	0	Node device did not respond to FRC at all.
0	1	HWPID did not match HWPID of the device.
1	x	HWPID matches HWPID of the device. Bit.1 can be set by FrcValue event. At the end, 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}

3.14.6.4 Read temperature

FRC_Temperature = 0x80

Collects bytes. Resulting byte equals to the temperature value read by *getTemperature* IQRF OS method. If resulting temperature is 0°C, which would normally equal to value 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 working temperature range.

3.14.6.5 Acknowledged broadcast - bytes

FRC_AcknowledgedBroadcastBytes = 0x81

Collects bytes. Resulting byte equals normally to the same temperature value as Read temperature command, but if this FRC command is caught by 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 Acknowledged broadcast - bits FRC command does.

3.14.6.6 Memory read

FRC_MemoryRead = 0x82

Collects bytes. A resulting byte is read from the specified memory address after 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 cannot equal to 0 there is also Memory read plus 1 FRC command available.

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 node 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 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 IQRF OS *lastRSSI* variable. Dummy LED Get DPA Request will be executed and then a byte from *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_LEDR = 0x06
CMD = CMD_LED_GET = 0x02
Length = 0 = No data bytes
PData *none*

Example 3

This example reads a lower byte of HWPID version from more nodes at once. Peripheral enumeration DPA Request is executed and result byte is read. Address 0x4A9 points to 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 return supply voltage level using 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*

3.14.6.7 Memory read plus 1

FRC_MemoryReadPlus1 = 0x83

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

3.14.6.8 FRC response time

FRC_FrcResponseTime = 0x84

Collects bytes. This embedded FRC command is used to find out FRC response time of the specified user FRC command. This is useful when a network consists of devices with different hardware profiles implementing the same user FRC command but 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 node from the set of nodes that will receive the specified FRC command. This FRC command actually raises FrcResponseTime event where a user code returns the time. The returned time value equals to the value of the corresponding `_FRC_RESPONSE_TIME_??_MS` constant (see IQRF-macros.h) with the lowest bit set (internally by DPA) in order to prevent returning zero value. If the specified FRC command is not supported (i.e. FrcResponseTime event is not handled) returned value is 0xFF.

FRC user data has the following format:

0	1
FRCcommand	0

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

4 HWP Configuration

HWP (hardware profile) 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 HWP configuration/Write HWP configuration/Write HWP configuration byte commands. There are predefined symbols *CFGIND_???* having the address of each configuration item.

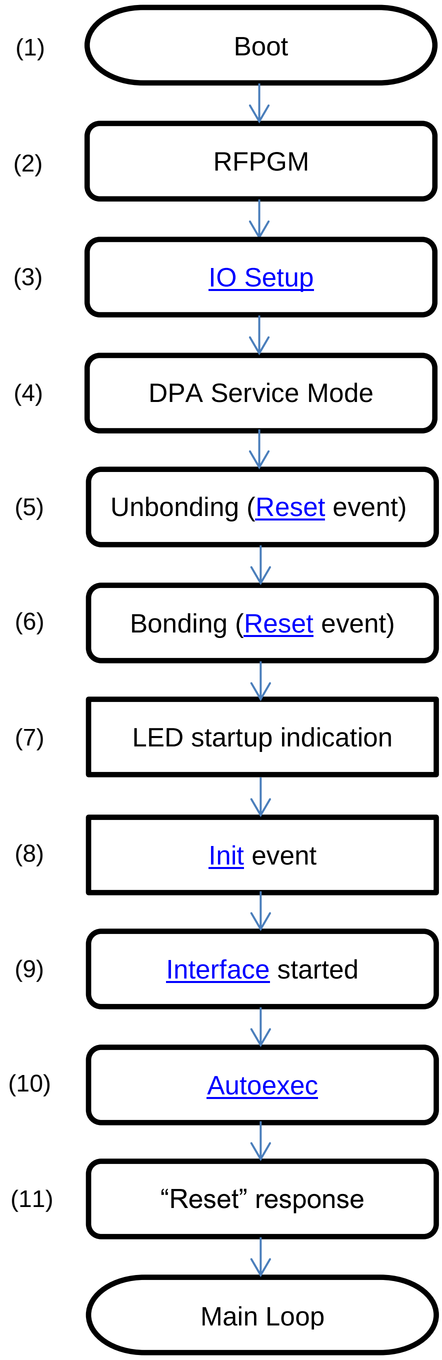
The following table depicts documented configuration items. Other items are reserved. The total size of the configuration block is 32 bytes.

Address	Description
0x00	The checksum of HWP Configuration block. See Write HWP configuration for details.
0x01 [**] **0x02** [**] **0x03** [**] **0x04** [**]	An array of 32 bits. Each bit enables/disables one of the embedded 32 predefined peripherals. Peripheral #0 (Coordinator) is controlled 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).
0x05 [*]	DPA configuration bits:
bit 0	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 Custom DPA Handler chapter.
bit 1	If set, then Node device can be controlled by a local interface. In this case, the same peripheral must not be enabled. This option is not valid for a main network coordinator device [C] and is not supported in LP mode at [N] devices.
bit 2	If set, then DPA Autoexec is run at a later stage of the module boot time.
bit 3	If set, then the Node device does not route packets on the background.
bit 4	If set, then DPA IO Setup is run at an early stage of the module boot time.
bit 5	If set, then device receives also peer-to-peer (non-networking) packets and raises PeerToPeer event.
bits 6-7	Reserved
0x06	Main RF channel A of the optional subordinate network in case the node also plays a role of the coordinator of such network. Valid numbers depend on used RF band.
0x07	Same as above but second B channel.
0x08	RF output power. Valid numbers 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]) and 3. at DpaApiSetRfDefaults API. Use *setRFpower* IQRF OS function to set power at runtime.
0x09 [*]	RF signal filter. Valid numbers 0-64. Also see API variable RxFilter.
0x0A [*]	Timeout for receiving RF packets at LP mode at N device. The unit is cycles (one cycle is 46 ms at LP mode). Greater values save energy but might decrease responsiveness to the master interface DPA Requests and also decrease Idle event calling frequency. Valid numbers are 1-255. See also API variable LPtoutRF.
0x0B [*]	Baud rate of the UART interface if one is used. Uses the same baud rate coding as UART Open (i.e. 0x00 = 1 200 Baud)
0x0C	A nonzero value specifies an alternative DPA service mode channel.
0x11	Main RF channel A of the main network. Valid numbers depend on used RF band. Setting this item does not have an immediate effect at [C] or [N] devices except these moments: 1. at Startup and 2. at DpaApiSetRfDefaults API. Use *setRFchannel* IQRF OS function to change the RF channel at runtime.
0x12	Same as above but second B channel.

[*] The device must be restarted for configuration item change to take effect.
[**] Same as [*] but only in case of SPI and UART embedded peripherals bits.

5 Device Startup

When device **(1)** boots 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 indicated by a repeated long green LED light followed by short red LED flash. RFPGM mode is terminated depending on its configuration. RFPGM mode is fully controlled by IQRF OS.



Next **(3)** IO Setup is executed if one is enabled.

At the very beginning, it is possible to remotely start the device at 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. *DSMactivated* API variable indicates whether DSM was started during device startup. Upon DSM exit, the device is always reset. The device first tries to establish DSM session at the fixed channel number 0* and then it tries an alternative channel optionally specified at HWP configuration. CATS - DPA Service Tool must be set to use the same required channel for the DSM session.

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

Brown-out Reset is disabled now and user interrupt is enabled so Interrupt event can be raised if any interrupt source is enabled from now on.

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

By default, a bonding or a bond removal (unbonding) at node side is initiated and controlled by a “default“ IQRF button connected between ground and `PORTB.4` MCU pin which is normally available at IQRF development tools. The default behavior can be modified by an implementation of Reset event that is raised during bonding and/or unbonding phases. To keep default behavior but with a custom bonding button an event BondingButton can be used.

Already bonded node can be **(5)** unbonded by the following procedure. Switch off the node. Keep pressing the button and switch on the node. Skip optional RFPGM mode depending on its configuration (typically pressed button terminates it). Keep the button pressed. Green LED is then on. After 2 seconds the green LED goes off. Release the button immediately within 0.5 s. Unbonding is then confirmed by red LED being on for 1 second and consequently by the rapid red flashes described above. Such complicated unbonding procedure is needed in order to prevent unwanted unbonding caused by accidental button press after the device is reset.

(6) If the node is not bonded then its red LED rapidly flashes (four times per second). Node waits for the button press. If the button is not pressed within 10s then the node goes into power saving sleep mode and red LED stops flashing. From the sleep mode, the node can be woken up by the button press. By pressing the button a bonding process is initiated. If the button is pressed the node continuously requests bonding (indicated by red LED). If the red LED becomes off and a green LED is lit when the button is still pressed then the node is bonded. If the red LED keeps flashing rapidly after the button is released then the node is not bonded yet and the whole bonding phase repeats.

At this point, [N] devices are bonded and ready to work. This is **(7)** indicated by a short red LED flash. If the device has a temporary network address (0xFE) obtained by remote bonding then the device flashes twice. Devices [C] perform one green LED flash instead when they are ready.

After that, **(8)** Init event is raised and **(9)** Interface is started (in the case of [N] devices only when enabled at HWP Configuration). If the SPI peripheral is enabled in the HWP Configuration, then it is started.

Consequently, an **(10)** Autoexec is executed if one is enabled.

At **(11)** if the interface is enabled (always at the [C] device) the device (being always slave interface) sends the following asynchronous "Reset" DPA response equal (except PCMD) to 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 response of Peripheral enumeration

Then the [C] device checks a 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 interface master is not connected an extra green LED flash is carried out and API variable *IFaceMasterNotConnected* is set to 1.

6 Autoexec

If Autoexec feature is enabled at HWP Configuration, then a series of DPA requests can be executed at the boot time (after Init event) of the device. Both sender and addressee addresses of the requests are equal to 0xFC (local address). DPA requests are stored in the block at the external EEPROM starting from the physical address *AUTOEXEC_EEEPROM_ADDR* = 0x0000. 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 - a 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 Autoexec batch. When executing DPA request a local interface notification is not performed although DPA via the interface is enabled. Other events at the user DPA routine are called as usual. It is not allowed to embed Batch within series of individual DPA requests.

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

Autoexec example:

The following example shows the bytes stored at the Autoexec external EEPROM memory space that will run these 4 actions upon the module reset:
1. Switch the green LED On (PNUM=0x07)
2. Open UART at 9 600 baud rate (PNUM=0x0C)
3. Write hex. bytes [01,02,03,04,05] to the UART (PNUM=0x0C)
4. Write hex. bytes [06,07,08,09,0a] to the RAM at address 0x0A (PNUM=0x05)

Actual bytes stored at serial EEPROM from address 0x0000:

```
   Len   PNUM  PCMD          HWPID   PData
1. 0x05, 0x07, 0x01(LED On),   0xFFFF
2. 0x06, 0x0C, 0x00(UART open), 0xFFFF, 0x03(9 600 Baud)
3. 0x0b, 0x0C, 0x02(UART write), 0xFFFF, 0xFF(no UART read), {0x01, 0x02, 0x03, 0x04,  0x05}(data)
4. 0x0b, 0x05, 0x01(RAM write),  0xFFFF, 0x0a(address), {0x06, 0x07, 0x08, 0x09, 0x0a}(data)
5. 0x00(end of Autoexec)
```

C code to upload Autoexec example to the external EEPROM:

```
#define NO_CUSTOM_DPA_HANDLER

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

#pragma cdata[ __EEESTART + AUTOEXEC_EEEPROM_ADDR ] = \
/* Len PNUM       PCMD                 HWPID       PData */ \
   5,  PNUM_LEDG, CMD_LED_SET_ON,      0xff, 0xff, \
   6,  PNUM_UART, CMD_UART_OPEN,       0xff, 0xff, DpaBaud_9600, \
   11, PNUM_UART, CMD_UART_WRITE_READ, 0xff, 0xff, 0xff, 1, 2, 3, 4, 5, \
   11, PNUM_RAM,  CMD_RAM_WRITE,       0xff, 0xff, 0x0a, 6, 7, 8, 9, 10, \
0
```

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

7 IO Setup

IO Setup feature can be used to setup direction, pull-ups and value of individual IO pins of the MCU at the very beginning of the device startup. It is very similar to Autoexec except only DPA peripheral IO requests are executed in order to make sure the device will always enter DPA Service Mode that can be used to fix an incorrect behavior. Also every request must use HWPID equal to 0xFFFF (*HWPID_DoNotCheck*). IO Setup DPA requests likewise Autoexec ones are stored at external EEPROM memory but, in this case, starting from its physical address *IOSETUP_EEEPROM_ADDR* = 0x0040; the size of the block is 64 bytes (it is located just after Autoexec memory space

Important: Updating Custom DPA Handler code using 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 at 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.

8 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 and much more. If the custom DPA handler is implemented it must be enabled in the HWP configuration in order to receive events.

If the Custom DPA handler is enabled in the HWP Configuration but it was not detected (see point 2. below) then device indicates an error by constant switching on the red LED and by returning *ERROR_MISSING_CUSTOM_DPA_HANDLER* error code to the 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 in order 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 Interrupt event).
3. There is an 864 instruction long block in the MCU flash memory reserved for 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. The whole bank is available.
6. Variables can be also mapped to the RAM bank 12 that equals to the peripheral RAM memory space.
7. Do not use *bufferRF*, *bufferCOM*, and *bufferAUX* at all (except inside events Reset, Init, Idle, and DisableInterrupts). *bufferAUX* can be used at 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 *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 effectively.
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 the coordinator only device [C]. Use DpaTicks being internally driven by TMR6 instead.
14. Do not use IQRF OS functions *start[Long]Delay* and *waitDelay* (except inside events Reset, Init, Idle, FrcValue and DisableInterrupts). 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 same RF settings (see *setTXpower, setRFspeed, setRFband, setRFchannel, setRFmode, set*mode, setNetworkFiltering*, setRouting**, 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 event code. It is required to save their values and restore them at the event exit.
17. Starting from Init event an MCU watchdog timer with 4 s period is enabled. Do not change WDT settings. Also, make sure to call *clrwdt()* if needed in order to prevent WDT reset.
18. If possible, try to avoid executing MCU stack demanding complex requests (e.g. Discovery) from subroutines in order to prevent MCU stack overflow. Such overflow results in HW device reset.
19. Both FSR0 and FSR1 point to the message PData at the Custom DPA Handler entry. This can be used for code optimization.

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

8.1 Handler Example

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

```
// 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 FALSE;

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

// Default Custom DPA Handler header
// (2nd include to implement Code bumper to detect too long code of the handler)
#include "DPAcustomHandler.h"
```

8.2 Events Flow

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

8.2.1 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]

if Autoexec enabled
    Run Autoexec

Send Reset response to Interface
loop
    if 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 standard request
                    else
                        [Handle Peripheral Request]
                    [BeforeSendingDpaResponse]
                Send response to Interface
                [Notification]
                Execute optional [sync] part of request
                [AfterRouting]
            else
                Wait for the previous routing timeout to finish
                Send DPA Confirmation to Interface
                Transmit request packet to the network
                Set routing timeout to the real [C]>[N] plus optimistic [N]>[C] routing

    if packet (typically response) received from the network
        if not system packet
            if not peer to peer packet
                if not same DPA packet 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 standard request
                                else
                                    [Handle Peripheral Request]
                                [BeforeSendingDpaResponse]
                            [Notification]
                            Execute optional [sync] part of request
                            [AfterRouting]
                        else
                            Send received packet to Interface
            else
                if peer to peer packet enabled
                    [PeerToPeer]
        else
            if remote bonding and not [AuthorizePreBonding]
                Pre-bond node
```

```
    else
        [Idle]
endloop
```

8.2.2 Node

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

```
if IO Setup enabled
    Run IO Setup

DPA Service Mode

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

while the node is not bonded
    if not [Reset]
        Default bonding procedure
[Init]

if Autoexec enabled
    Run Autoexec

Send Reset response to Interface
loop
    if 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 standard request
                        else
                            [Handle Peripheral Request]
                        [BeforeSendingDpaResponse]
                    if packet was not broadcasted
                        Wait for [C]>[N] routing to finish
                        Transmit response back to network
                    [Notification]
                    if Interface enabled
                        Send notification to Interface
                    Wait for [C]>[N] routing to finish
                    Execute optional [sync] part of request
                    [AfterRouting]
                else
                    Wait for [C]>[N] routing to finish
                    if not predefined FRC command
                        [FrcValue]
                    Response FRC value
            else
                if peer to peer packet enabled
                    [PeerToPeer]
        else
            if remote bonding and not [AuthorizePreBonding]
                Pre-bond node
    else
        [Idle]

    if local request packet received from enabled Interface
        if not [ReceiveDpaRequest]
            if embedded peripheral
                Execute standard request
            else
                [Handle Peripheral Request]
            [BeforeSendingDpaResponse]
        Send response back to Interface
```

```
        [Notification]
        Execute optional [sync] part of request
        [AfterRouting]
endloop
```

8.2.3 General events

Next chapters show pseudo codes illustrating the logic of raising general events at any device where the described event makes sense.

8.2.3.1 Interrupt

Interrupt event is raised whenever an MCU interrupt occurs.

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

8.2.3.2 Disable Interrupts

Disable interrupts event is raised at Reset, Restart, and Run RFPGM commands.

```
if Run RFPGM
    [Disable Interrupts]
    Device will reset or restart
```

8.2.3.3 Sleep Events

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

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

8.3 Events

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 the illustration purpose only. Please use the C code template and examples distributed with DPA package instead.

8.3.1 Interrupt

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

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

1. The time spent handling this event is critical. If there is no interrupt to handle 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.
   It is desirable that this event is handled with an immediate *return* even if it is not used by the custom handler because the Interrupt event is raised on every MCU interrupt and the “empty” *return* handler ensures the shortest possible interrupt routine response time.
2. Only global variables or local ones marked by *static* keyword 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 variable is usually C*number*cnt.
5. Do not call any OS functions except *setINDFx*. Use direct reading by FSRx or INDFx registers instead of calling obsolete *getINDFx* IQRF OS function.
6. Do not use any OS variables especially for writing access.
7. All above rules apply also to any other function being called from the event handler code, although calling any function from 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;
       T6CON = 0b0.0110.1.00;

       // timerOccured is a global or static variable
       timerOccured = TRUE;

       return Carry;
```

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

8.3.2 Idle

This event is periodically raised when the main loop is waiting for incoming RF (or interface) message to handle. The time spent handling this event is critical. When there is 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 *RFRXpacket* IQRF OS function (used to receive network packets) located in the main DPA loop. In the case when there is full IQMESH network consisting of 239 devices and the long diagnostic timeslot (200 ms) is used, the Idle event might not be called even for 239 × 200 ms = 47.8 s. Even longer 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.

8.3.3 Init

This event is called just before the main loop starts after Reset event i.e. when the [N] is bonded. Also, Enumerate Peripherals is called before this event is raised in order to find out the hardware profile ID. Immediately after the event is processed the Autoexec is executed. 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 1st call of a Reset event.

If variable *NodeWasBonded* is set, then variable *UserBondingData* contains user data passed from the node that provided pre-bonding of the device.

Example

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

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

8.3.4 Notification

This event is called when a DPA request was successfully processed and the DPA response was sent. DPA response (but not original 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 response to.

Example

```
case DpaEvent_Notification:
// Anything was writen to the RAM?
if ( _PNUM == PNUM_RAM && _PCMD == CMD_RAM_WRITE )
{
	if (PeripheralRam[0] == 0xAB)
		LEDR = 1;
	else
		LEDG = 1;

	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.

8.3.5 AfterRouting

[sync] This event is called after the DPA response 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 a 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;
		LEDR = 0;
		LEDG = 0;
	}
```

```
        return Carry;
```

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

8.3.6 BeforeSleep

This event is called before the device goes to the Sleep mode. The code has to 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 given hardware should be set as outputs.

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

This event is not implemented at the device having coordinator functionality i.e. [C] and not in the demo version.

Example

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

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

8.3.7 AfterSleep

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

This event is not implemented at the device having coordinator functionality i.e. [C] not in the demo version.

Example

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

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

8.3.8 Reset

This event is not raised in the demo version. The event is called just after the module was reset. It can be used to handle bonding/unbonding of the node in [N] devices. In this case, the code must return TRUE. If the node is not bonded the handler routine must not finish until the node is bonded. See Init event concerning the initialization options. An interrupt is enabled so the Interrupt event can be already called. [N] devices are set to the node mode by calling *setNodeMode* IQRF OS function before this event is raised.

The event is also used to specify optional Bonding user data (see code example below) using variable *UserBondingData* in [N] devices that are passed during the remote bonding process and can be read by Read remotely bonded module ID. The code should also handle the setting of *NodeWasBonded*.

The Reset event is also once raised at the [C] device for the sake of 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

```
case DpaEvent_Reset:
        if (!doCustomBonding)
        {
```

```
            UserBondingData[0] = 0x12;
            UserBondingData[1] = 0x34;
            UserBondingData[2] = 0x56;
            UserBondingData[3] = 0x78;
            return FALSE;
    }

    if ( amIBonded() )
    {
            if ( unBondCondition )
            {
                    removeBond();
                    _LEDR = 1;
                    waitDelay( 100 );
                    _LEDR = 0;
            }
    }
    else
    {
            while ( !amIBonded() )
            {
                    if ( bondRequestCondition )
                    {
                            UserBondingData[0] = 0x12;
                            UserBondingData[1] = 0x34;
                            UserBondingData[2] = 0x56;
                            UserBondingData[3] = 0x78;
                            bondRequestAdvanced();
                            setWDToff();
                    }
            }

            NodeWasBonded = TRUE;
            bondingUserDataOut = UserBondingData;
    }

    return TRUE;
```

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

8.3.9 Disable Interrupts

This event is not raised in the demo version. The event is called when the device needs all hardware interrupts to be disabled. Such moment occurs at Reset, Restart, and Run RFPGM commands.

Example

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

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

8.3.10 FrcValue

[sync] This event is called whenever the node 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 *_PCMD* variable. FRC data to collect must be stored at *responseFRCvalue* IQRF OS variable. If 2 bytes are collected then the data must be stored at *responseFRCvalue2B* variable instead. If bits are collected then only lowest 2 bits of *responseFRCvalue* are used. Before calling the event both variables are prefilled with value 0x01 or with 0x0001 respectively (except Acknowledged broadcast - bytes).

It is critical that the code will take less than 40 ms at all nodes in order 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 longer time to prepare data for FRC to return.

Important: If the event handler exceeds selected time then the device does not respond via FRC at all thus “returning” 0 value. 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 *DataOutBeforeResponseFRC* IQRF OS variable. This event is implemented at [N] devices only.

Example

```
case DpaEvent_FrcValue:
 {
      // This example is sensitive to the bit FRCommand 0x40
      if ( _PCMD == FRC_USER_BIT_FROM )
      {
       // Return info about providing remote bonding
       if ( ProvidesRemoteBonding )
             // Both bits bit0 and bit1 are set now
             responseFRCvalue.1 = 1;
      }
      // This example is sensitive to the byte FRCommand 0xC0
      else if ( _PCMD == FRC_USER_BYTE_FROM )
      {
       // Just return your logical address as an example
       responseFRCvalue = ntwADDR;
      }
      // This example is sensitive to the byte FRCommand 0xF0
      else if ( _PCMD == FRC_USER_2BYTE_FROM )
      {
       // Return 2 byte value,
       responseFRCvalue2B = Measure2Bytes();
      }

      return Carry;
 }
```

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

8.3.11 *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 the 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 coordinator to find out the longest FRC response time in the network consisting of “unknown” heterogeneous node devices. DPA internally sets the lowest bit of the return value in order to prevent returning zero (equals to `_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.

8.3.12 ReceiveDpaResponse

This event is called when there is a DPA response packet received from the network. If the event handler returns TRUE, then further standard DPA response processing (passing DPA response to the interface master internally by DpaApiSendToSpiMaster) is skipped. The event is raised even when HWPID does not match. At this time, system variables RTTSLO and RTHOPS have valid numbers corresponding to the received response.

This event is implemented at [C] devices but not in the demo version.

Example

```
case DpaEvent_ReceiveDpaResponse:
 {
      // This example just for demonstration purposes consumes any
      // DPA response 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.

8.3.13 IFaceReceive

This event is called when there is a DPA request packet 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 response to the network internally by DpaApiRfTxDpaPacketCoordinator) is skipped. In this case, interface master receives an error DPA response with *ERROR_INTERFACE_CUSTOM_HANDLER* Response Code. The event is raised even when HWPID does not match.

This event is implemented at [C] device but not in the demo version.

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;
 }
```

8.3.14 ReceiveDpaRequest

This event is not raised in the demo version. The event is called when a DPA request (except Get information for more peripherals and Remove bond) is received from the network or from interface master (if applicable). If the event handler returns TRUE, then the request is not passed to the default handling by 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 original DPA request. 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 request from Interface
if ( TX == LOCAL_ADDRESS )
{
	_PCMD |= RESPONSE_FLAG;
	DpaApiSetPeripheralError( ERROR_NADR );
	return TRUE;
}

return FALSE;
```

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

8.3.15 BeforeSendingDpaResponse

This event is not raised in the demo version. 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 event in the way that the error code is returned.

Example

```
case DpaEvent_BeforeSendingDpaResponse:
 // Always adds one more read byte from EEPROM peripheral and sets it to 0x55
 if ( _PNUM == PNUM_EEPROM && _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;
```

8.3.16 PeerToPeer

This event is not raised in the demo version. When peer-to-peer (non-networking) packets are enabled at HWP Configuration then device raises this event when such 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 an optional addressing (*_DpaParams* DPA variable can be misused for this purpose) must be implemented at 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();
  break;
```

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

8.3.17 AuthorizePreBonding

This event is sent whenever there is a request from a node to pre-bond to the network. The event is raised even if the remote bonding is not enabled (see ProvidesRemoteBonding) or if the pre-bonding was already provided (see RemoteBondingCount). This gives the user code the opportunity to monitor all bonding requests in the network. The event handler can decide whether the pre-bonding will be accepted (by returning a FALSE value, which is the default custom DPA handler exit code) or rejected (by returning TRUE). Please note that even when the pre-bonding request is accepted it does not mean that the pre-bonding will be actually executed. The reason might be that the remote bonding is not enabled (see Enable remote bonding and ProvidesRemoteBonding) or another node was already pre-bonded (see RemoteBondingCount) or this node will stay only pre-bonded (not authorized by Authorize bond yet).

There are many options how the event handler can decide whether the request will be accepted or rejected. Usually, the handler decides based on request node MID (variable `BondingNodeMID` can be used) or on bond request used data (variable `UserBondingData` can be used).

Example

```
case DpaEvent_AuthorizePreBonding:
  // Called when remote bonding is enabled and a node requests pre-bonding
```

```
  // We might monitor all bond requests
  LogPreBondEquest( BondingNodeMID );

  // Is the requesting node (MID) trustworthy?
  if ( !isThustworthyMID( BondingNodeMID ) )
       return TRUE;

  // Does the node use the correct PIN being sent as bonding user data?
  if ( !PINmatches( BondingNodeMID ) )
       return TRUE;

  // Allow pre-bonding of this node.
  return FALSE;
```

☼ See example code [*CustomDpaHandler-AutoNetwork.c*]() for more details.

8.3.18 UserDpaValue

This event is raised whenever DPA is internally required to return user defined DPA value in the response. This event is the very last time when it is necessary to fill in [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 = myValue;
  return Carry;
```

8.3.19 BondingButton

This event is called during standard DPA (un)bonding [process]() 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 node does not go into power saving sleep mode during bonding. IQRF OS function *amIBonded* can distinguish between bonding and unbonding.

Example

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

8.3.20 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.

8.3.20.1 Enumerate Peripherals

This DPA request is called 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 custom DPA handler instead of default handler (overriding embedded peripheral).
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;
}
```

8.3.20.2 Get Peripheral Info

If the user code handles user 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;
}
```

8.3.20.3 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, otherwise it must return TRUE.

Please note in the following code how to return an error state. Set PNUM to *PNUM_ERROR_FLAG*, set 1st data byte of the DPA response to the error code, set 2nd byte to the original PNUM and finally specify that the length of the data is equal to 2. The best way is to use 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
        // DpaApiReturnPeripheralError(ERROR_PCMD); is the easiest way
        _DpaMessage.ErrorAnswer.ErrN = ERROR_PCMD;
        goto UserErrorAnswer;
       }
 }

 return TRUE;
}

return FALSE;
```

8.3.20.4 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 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;
```

8.4 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 *userReg0* variable the value of macro *GetDpaEvent()* is undefined.

8.4.1 DpaApiRfTxDpaPacket

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

Available at [N] devices. This function wraps all necessary code to send a DPA message (typically response) from Node to Coordinator. 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 response. 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 coordinator to the interface master in case of a [C] device after it is received.

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

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 coordinator, use value 2 if the message should be terminated at the DPA interface of the same coordinator or at the coordinator above the same coordinator, 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 response. So it is advised to store PID of such response and use a different one then. Please see a 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 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 to the lastRSSI (can be any other value)
DpaApiRfTxDpaPacket( lastRSSI, 1 );
```

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

8.4.2 DpaApiReadConfigByte

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

This function returns HWP configuration value from a given index (address).

Example

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

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

8.4.3 DpaApiSendToIFaceMaster

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

Available at [C] and [N] (at STD mode) devices. The function passes prepared DPA packet (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 actually ignored in the case of 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.

8.4.4 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 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 a number of hops used to deliver the DPA response from the addressed device back to the coordinator. A number of hops used to deliver the DPA response to the addressee and slot length are available at IQRF OS variables *RTHOPS* and *RTTSLOT* respectively. Thus, the same information (Hops, Timeslot length, Hops Response) like 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.

8.4.5 DpaApiLocalRequest

```
void DpaApiLocalRequest()
```

Performs a local DPA request at the embedded peripheral, that even does not have to be enabled in the HWP Configuration. Of course, the peripheral must be implemented. After the function returns a corresponding DPA response 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 even as during 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 destroys value obtained by *GetDpaEvent()* macro.

Example

```
case DpaEvent_Idle:
 if ( IsSleepTime() )
 {
       // 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.

8.4.6 DpaApiReturnPeripheralError

```
DpaApiReturnPeripheralError ( uns8 error )
```

This is actually a macro calling internal API *DpaApiSetPeripheralError(error)* to prepare an error DPA response 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 following lines of code:

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

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.

8.4.7 DpaApiSetRfDefaults

```
void DpaApiSetRfDefaults()
```

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

- RF filter value,
- RF mode bits,
- 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.

8.5 DPA API Variables

The following variables can be used within custom DPA handler routine. The variables marked by *[readonly]* are read-only variables. Writing to these variables will cause incorrect device behavior.

8.5.1 bit ProvidesRemoteBonding

[readonly] Equals to 1 when the device provides remote bonding, see Enable remote bonding.

8.5.2 uns8 RemoteBondingCount

[readonly] Number of pre-bonded nodes.

8.5.3 bit IFaceMasterNotConnected

[readonly] Valid at [C] device. Equals to 1 when master interface device was not connected during device startup.

In the case of SPI interface, it is considered not connected when a Reset DPA response is not read during the startup process.

In the case of UART interface, it is considered not connected when there was any DPA message received by the interface yet.

Please note that this flag might become equal to 0 when a master interface device sends some data to the [C] device later. The variable value is valid after Init event.

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

8.5.4 bit NodeWasBonded

Valid at [N] devices. Is set to 1 during Device startup if the node was newly bonded. It is a programmer's responsibility to set this variable if default bonding mechanism is overridden at Reset event.

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

8.5.5 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.

8.5.6 uns16 DpaTicks

Implemented at [C] device only. The value of this variable is decremented every 10 ms after Init event. The variable is driven by TMR6. The variable can be used for implementation of timing algorithms. As this 2-byte wide variable is modified internally within CPU interrupt routine the whole (both 2 bytes) variable should be accessed (either read or written) only when an interrupt is disabled to ensure an atomic access.

Example

```
case DpaEvent_Idle:
	// Is 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.

8.5.7 uns8 LPtoutRF

Valid at [N] devices and LP mode only. Timeout when receiving RF packets in LP mode. After a device startup, it is filled with a respective value from HWP Configuration at index 0x0A. See that chapter for more details.

8.5.8 uns8 ResetType

Identifies the type of reset (stored at *UserReg0* upon module reset). See Reset chapter at IQRF User's Guide for more information.

8.5.9 bit DSMactivated

Equals to 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.

8.5.10 uns8 UserDpaValue

This variable is used to store user defined DPA value. See Set DPA Param and UserDpaValue.

8.5.11 uns8 NetDepth

[readonly] This variable is used at ReceiveDpaResponse event to find out whether the received 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.

8.5.12 bit LpRxPinTerminate

When set to 1 then at [N] device running at LP mode waiting for packet reception is discontinued by a low level at MCU pin `PORTB.4` regardless of configuration LP timeout value at index 0x0A. See *setRFmode* IQRF OS function for more information. Immediately after the packet reception is discontinued the Idle event is raised. The default value is 0.

8.5.13 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 HWP Configuration at the startup and can be modified at the runtime.

8.6 Examples

Find below a list of all examples. Next chapters describe selected Custom DPA Handler examples in more detail.

CustomDpaHandler-AsyncRequest - Asynchronous request from Node to the coordinator.
CustomDpaHandler-Autobond - Autobonding example.
CustomDpaHandler-AutoNetwork - Node Autonetwork part.
CustomDpaHandler-Bonding - Custom bonding.
CustomDpaHandler-ButtonlessBonding_Node – Button less bonding - Node part.
CustomDpaHandler-ButtonlessBonding_Coordinator - Button less bonding - Coordinator part.
CustomDpaHandler-Coordinator-AutoNetwork-Embedded - Embedded Autonetwork by coordinator.
CustomDpaHandler-Coordinator-AutoNetwork - Autonetwork controlled by the external system.
CustomDpaHandler-Coordinator-FRCandSleep - Regular FRC & sleep controlled by the coordinator.
CustomDpaHandler-Coordinator-PollNodes - Polling data from nodes by the coordinator.
CustomDpaHandler-Coordinator-PulseLEDs - Pulsing LEDs at nodes controlled by the coordinator.
CustomDpaHandler-Coordinator-ReflexGame - Simple reflex game.
CustomDpaHandler-DDC-RE01 - DDC-RE01 demo.
CustomDpaHandler-DDC-SE01 - DDC-SE01 demo.
CustomDpaHandler-DDC-SE01_RE01 - DDC-SE01 and DDC-RE01 demo.
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-MultiResponse - Multiple responses to the one request.
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-SPI - Custom SPI Peripheral.
CustomDpaHandler-Template-OptimizedSwitch - Optimized custom DPA Handler template.
CustomDpaHandler-Template - Custom DPA Handler template.
CustomDpaHandler-Timer - Using PIC HW timer.
CustomDpaHandler-UART - Connecting external device using embedded UART peripheral.
CustomDpaHandler-UartHwRxSwTx - Hooking UART embedded peripheral to free PWM pin.
CustomDpaHandler-UARTrepeater - Sample UART repeater example.
CustomDpaHandler-UserEncryption - AES-128 demonstration.
CustomDpaHandler-UserPeripheral-18B20 - Dallas 18B20 temperature sensor as peripheral.
CustomDpaHandler-UserPeripheral-18B20-Idle - Dallas 18B20 sensor operated at 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 I2C.
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 - Basic user peripheral.
DpaAutoexec - Autoexec demonstration.
DpaIoSetup - IO Setup demonstration.

8.6.1 Bonding

This example shows how to implement a custom (un)bonding procedure inside Reset event. The code actually behaves the exact same way the default (un)bonding procedure does, except the button is assigned to the different MCU GPIO pin and the node is not put to sleep when the button is not pressed for longer time.

→ Self-study tip: Modify the code in the way the node request bonding when the button is pressed only when the node does not sense stronger RF signal thus implementing List-Before-Talk technique. Hint: Use checkRF IQRF OS function to sense RF signal.

8.6.2 AutoNetwork & Coordinator-AutoNetwork-Embedded

These are actually the most complex examples published to date. The Custom DPA Handlers automatically build IQRF network consisting of coordinator device only and new nodes or they allow adding new nodes to the existing network too. Building network means bonding new nodes at their final physical location and then performing network discovery.

Autonetwork process consists of repeated so called waves. In each wave, the process tries to bond as many new nodes as possible and finally performs discovery. As the new nodes are to be added to the network from their final physical location a remote bonding must be used (not a local one). There are two time-separated phases during each wave. Network either works as usual (i.e. typically DPA messages are transmitted) or new nodes (candidates) are remotely bonding to the network. The strict time separation of these two phases is needed in order to prevent RF traffic jamming.

New nodes do not try to (remotely) bond to the network unless they are allowed to. They listen for a special packet to start bonding. When coordinator decides to start a bonding phase then it itself and all already bonded nodes send special non-networking LP packet that announces the bonding phase. The packet is sent at separate time slots by each node (using user DPA peripheral) and coordinator respectively. The packet contains time in which the bonding phase starts and how long it lasts. The packet is received by node candidates so they know when they can start bonding.

Node candidates use *Listen Before Talk* technique before sending bonding request during bonding phase in order to minimize the number of collisions. If they succeed to bond they receive temporary network address (0xFE). If the node is having the temporary address for a too long time (this timeout is specified inside previously described LP packet) then it removes locally its own bond and restarts. It will then try to bond again.

When the bonding phase is over the coordinator collects using Prebonding FRC and Read remotely bonded module ID MIDs of all prebonded new nodes having a temporary network address. Then coordinator authorizes the nodes so they get definitive network address assigned.

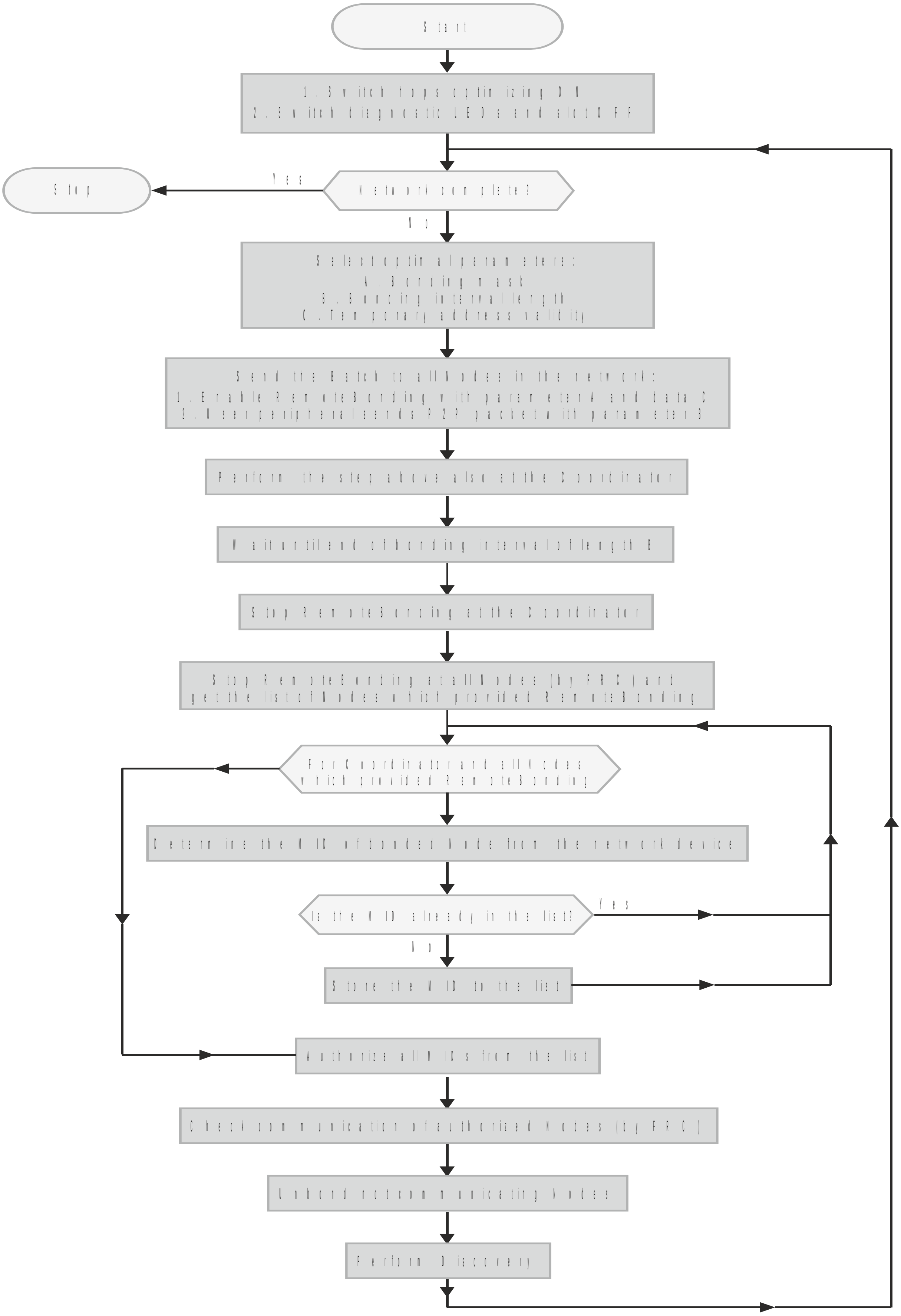
Finally, FRC is used to check if newly bonded nodes are responding. If this is not the case the bond is removed both at coordinator as well as at node side for sure. At the very end, a network discovery is performed.

There are extra features that can be found in the source code. For instance, a system connected to the coordinator device can (dis)approve authorization of new nodes. Also, even prebonding of a new node can be authorized at the Custom DPA Handler code to allow prebonding of authorized nodes only and to avoid penetration of strange nodes.

To run autonetwork at LP mode the node AutoNetwork handler must be compiled with DPA_LP symbol defined. Please see the source code for more details.

→ Self-study tip: Look at the authorization code example conditionally compiled by AUTHORIZE_PRE_BONDING symbol. Use MID to authorize prebonding instead of bonding data. External EEPROM peripheral of the node might store a database of MIDs that will be allowed to prebond.

The following picture depicts the process in more detail.



8.6.3 Coordinator-FRCandSleep

This example shows autonomous coordinator, that 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 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 to 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 coordinator performs delay using DpaTicks API variable including safety gap after both Send and Extra result commands are executed inside the Idle event handler. Also please note a small delay inside Init event to allow 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 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 coordinator's peripheral RAM for passing a set of nodes to return the FRC byte value from. This is useful in case of bigger networks (with the address above 62, see Send).
Hint: You will have to substitute using Send for 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.

8.6.4 FRC-Minimalistic

This is a truly minimalistic code example. It shows literally at only two lines of C code how to implement custom FRC command. Its code is `FRC_USER_BIT_FROM = 0x40`. It returns 2nd bit equal to 1 if 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.

8.6.5 LED-MemoryMapping

The example shows controlling of physical LEDs at DCTR by the peripheral RAM. A custom command byte 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 RAM peripheral that was written to. Modify the code so it will check the actual byte range written to the RAM peripheral and to control the appropriate LED(s) only.
Hint: Use ReceiveDpaRequest event to find out the address and length of data written to the peripheral RAM.

8.6.6 PeripheralMemoryMapping

This example implements the bidirectional mapping of several MCU peripherals to the peripheral RAM. It allows controlling LEDs, reading button's state and reading temperature value. All by means of peripheral RAM.

→ Self-study tip: Currently the example always controls both LEDs or reads buttons & temperature sensor 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.

8.6.7 UserPeripheral-18B20

This example demonstrates connecting the node 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 DDC-SE-01 sensor kit so it is very easy to create a device operating the sensor at the lab.

A deep knowledge of 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 maximum 40 ms FRC response time both Set FRC Params at coordinator side and FRC response time at node side must be used.

8.6.8 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 implementation of user FRC command.

8.6.9 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.

8.6.10 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 Interrupt event.

→ Self-study tip: implement variable UART baud rate when UART is opened.

8.6.11 UserPeripheral-i2c

The example implements user peripheral that returns a value read from a connected I^{2}C device. The code can directly read a temperature value from MCP9802 temperature sensor presented at DDC-SE-01.

A deep knowledge of I^{2}C protocol is necessary to understand the source code in full detail.

→ Self-study tip: Implement user byte FRC command to return value from an I^{2}C device. Pay attention to the maximum FRC response time.

8.6.12 UserPeripheral-PWM

This is actually the copy of the implementation of the embedded PWM peripheral that is available only at demo version only. Use it as a template for your own PWM implementation.

8.6.13 UserPeripheral-SPImaster

This example shows how to connect SPI slave device to the DCTR node so the node behaves as SPI master. IQRF OS SPI support implements only SPI slave side. SPI slave device is controlled using custom command passed to the custom DPA peripheral. See the source code for full details.

→ *Self-study tip: Connect ordinary another DCTR node with DPA SPI peripheral being enabled thus playing the role of SPI slave. Try to communicate bi-directionally between the two nodes.*

9 DPA in Practice

9.1 Network Deployment

This chapter is a kind of a 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 plugin 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 logical addresses from compact address interval at the top of the address space during bonding. This allows to effectively using parameter MaxAddr at Discovery.
2. Download required DPA plug-ins based on RF Mode, Interface, and (DC)TR type used. Upload them to the devices.
3. Get ready your Custom DPA Handlers for all devices. Make sure the handler code states unique HWPID of the device. Some handlers do not have any internal application logic code except stating HWPID but contain Autoexec and/or 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 a 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 correct SPI/UART peripheral/interface.
    g. Enable Autoexec, 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 a normal network operation.
    b. 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 a communication with all devices.
    f. 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 a future reference.
8. Backup 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.

9.2 Over The Air (OTA) upgrade of IQRF OS and DPA

Please follow this checklist to upgrade both IQRF OS and DPA 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-V3xx-xxxxxx.iqrf* at *Source File* group box. The file can be found at the *IQRF Startup Package* at a 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 *Upload* button at the group box *Upload* to upload the selected file to the external EEPROM at all nodes.
1.6. Press *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 *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 *Upload* button at the group box *Upload* to upload the selected file to the external EEPROM at the coordinator.
1.11. Press *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 *Configure TR* button at the *Command* group box. A *TR Configuration* window will open.
2.5. Enable *Custom DPA Handler* at the *HWP* tab and press *Upload*. Press *Try Selected* if the configuration wizard reports 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 *Configure TR* button at the *Command* group box. A *TR Configuration* window will open.
2.9. Enable *Custom DPA Handler* at the *HWP* 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 to *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 RF mode and interface). The file can be found at the *IQRF Startup Package* at a 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 RF mode and interface). The file can be found at the *IQRF Startup Package* at a folder *Development\DPA\OTA_upgrade.*
3.7. Continue according to 1.9.-1.11.

4. Finishing up

4.1. Both IQRF OS and DPA is upgraded. The network is working.
4.2. Refresh and check the network map from the coordinator.
4.3. Follow the chapter 1. to upload back your normal Custom DPA Handlers or follow the chapter 2. to disable Custom DPA Handler at the devices that do not use it.
4.4. Follow the 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.

9.3 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. Next paragraphs describe how to proceed from the programmer's point of view.

9.3.1 Storing Code at External EEPROM

Code image or the IQRF OS change file must be stored in the external EEPROM using series of Extended Write commands. The easiest but not the optimal way is to store the content at the external EEPROM address that is multiple of 64 by the repetitive writing of up to 32 bytes at the beginning and at the middle of 64 bytes long external EEPROM page. The most optimal way is to write as many as possible bytes that one Extended Write request can handle (54 bytes) while the external EEPROM page boundary is not crossed or to use Batch command that contains two Extended Writes. The first write just fills in the rest of the 64 bytes external EEPROM page, the second one writes as many bytes as possible to the beginning of the next page. Find below C# example implementation of the algorithm.

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 a 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 to the external EEPROM make sure that other data are not overwritten. That could be another upload data, handler operation data, Autoexec or IO Setup. A precise planning of the external EEPROM content is recommended.

It is also recommended to plan the whole upload or change process in advance in the way that all required data (active handler, IQRF OS change handler, IQRF plugin DPA, IQRF OS change file) are first stored in the external EEPROM and then used in order 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 at the even small network.

Optimal writing to the external EEPROM:

```
void WriteToEEEPROM ( byte[] writtenBytes, UInt16 eeepromAddress )
{
  // Size of NADR, PNUM, PCMD and HWPID
  const int FoursomeSize = ( 2 + 1 + 1 + 2 ) * sizeof( byte );
  // Maximum number of DPA PData bytes ( minus 8b error code + 8b DpaValue )
  const byte MaxPDataLen = ( 64 - FoursomeSize - 2 * sizeof( byte ) );
  // Maximum number of bytes one XWrite can handle
  const byte MaxEEEPROMXdataLength = MaxPDataLen - sizeof( UInt16 );
  // External EEPROM page size
  const byte EEEPROMpageSize = 64;
  // EEEPROM peripheral number
  const byte PNUM_EEEPROM = 4;
  // EEEPROM XWrite command
  const byte CMD_EEEPROM_XWRITE = 3;
  // HWPID used at this example
  const UInt16 HWPID = 0xFfFf;
  // Batch overhead = length of subcommand + PNUM + PCMD + HWPID + EEEPROM address
  const byte xWriteBatchOverhead = ( 1 + 1 + 1 + 2 + 2 ) * sizeof( byte );
  // Maximum number of bytes that can be written by 2 XWrites stored inside batch
  const byte maxBatchXWriteDataTwice = MaxPDataLen - 2 * xWriteBatchOverhead - 1;

  // Current address
  int writtenBytesAddress = 0;
```

```
  // Length
  int remainsToWrite = writtenBytes.Length;
  // While not everything written
  for ( int written = 0; remainsToWrite != 0; eeepromAddress += (UInt16)written, remainsToWrite -=
(UInt16)written, writtenBytesAddress += written )
  {
    // Remaining bytes at the page or at all, whatever is smaller
    int remainInPageOrTotal = Math.Min( ( eeepromAddress / EEEPROMpageSize + 1 ) * EEEPROMpageSize -
eeepromAddress, remainsToWrite );

    // If remains more than XWrite can write, or more than Batch can write, or remains just what
totally remains
    if ( remainInPageOrTotal >= MaxEEEPROMXdataLength || remainInPageOrTotal > (
maxBatchXWriteDataTwice - 1 ) || remainInPageOrTotal == remainsToWrite )
    {
      // Do one WXrite
      // Do not write more than XWrite can write
      written = Math.Min( MaxEEEPROMXdataLength, remainInPageOrTotal );
      // Data to write by XWrite
      byte[] bytes = new byte[written];
      Array.Copy( writtenBytes, writtenBytesAddress, bytes, 0, written );
      // Execute the XWrite
      EEEPROMxWrite( eeepromAddress, bytes );
    }
    else
    {
      // Do Batch with 2 XWrites inside
      // 1st write length
      int write1 = remainInPageOrTotal;
      // 2nd write length
      int write2 = Math.Min( remainsToWrite, maxBatchXWriteDataTwice ) - write1;
      // Total write equals to the sum of both XWrites
      written = write1 + write2;

      // Build a batch content
      byte[] bytes = new byte[written + 2 * xWriteBatchOverhead + 1];

      // Lengths
      int write2ndOffset = xWriteBatchOverhead + write1;
      bytes[0] = (byte)write2ndOffset;
      bytes[0 + write2ndOffset] = (byte)( xWriteBatchOverhead + write2 );
      // PNUMs
      bytes[1] = bytes[1 + write2ndOffset] = PNUM_EEEPROM;
      // PCMDs
      bytes[2] = bytes[2 + write2ndOffset] = CMD_EEEPROM_XWRITE;
      // HWPIDs
      bytes[3] = bytes[3 + write2ndOffset] = (byte)( HWPID & 0xff );
      bytes[4] = bytes[4 + write2ndOffset] = (byte)( HWPID >> 8 );
      // 1st write address
      bytes[5] = (byte)( eeepromAddress & 0xff );
      bytes[6] = (byte)( eeepromAddress >> 8 );
      // 2nd write address
      bytes[5 + write2ndOffset] = (byte)( ( eeepromAddress + write1 ) & 0xff );
      bytes[6 + write2ndOffset] = (byte)( ( eeepromAddress + write1 ) >> 8 );
      // 1st write data
      Array.Copy( writtenBytes, writtenBytesAddress, bytes, xWriteBatchOverhead, write1 );
      // 2nd write data
      Array.Copy( writtenBytes, writtenBytesAddress + write1, bytes, xWriteBatchOverhead +
write2ndOffset, write2 );
      // Execute the batch
      OSrunBatch( bytes );
    }
  }
}
```

9.3.2 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 byte FRC command Memory read plus 1 to read result of the checksum check from multiple devices instead of individual polling each device one by one. When FRC is used then it is necessary to use Send Selective instead of Send in case of 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 is to be used. Pay special attention when the former or new uploaded handler requires its own data to be stored at the internal and/or external EEPROM. See LoadCode for more details.

9.3.3 Executing IQRF OS Change

Changing IQRF OS version is very similar to the 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 IQRF OS version, the process can also change DPA version at the same time. It implies that the current normally used custom handler must be replaced and then returned back. 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 the item No. 1 by the process described above. Then similarly (to the loading the code) check that the 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 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 the item No. 3 back.

10 Constants

All symbols and constants are defined in header files DPA.h and DPAcustomHandler.h.

10.1 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_SPI             0x08
#define     PNUM_IO              0x09
#define     PNUM_THERMOMETER     0x0A
#define     PNUM_PWM             0x0B
#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
```

10.2 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 response from [N]
STATUS_CONFIRMATION =             0xFF  // Error code used to mark confirmation
```

10.3 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_REBOND_NODE 6
#define     CMD_COORDINATOR_DISCOVERY 7
#define     CMD_COORDINATOR_SET_DPAPARAMS 8
#define     CMD_COORDINATOR_SET_HOPS 9
#define     CMD_COORDINATOR_DISCOVERY_DATA 10
#define     CMD_COORDINATOR_BACKUP 11
#define     CMD_COORDINATOR_RESTORE 12
#define     CMD_COORDINATOR_READ_REMOTELY_BONDED_MID 15
```

```
#define     CMD_COORDINATOR_CLEAR_REMOTELY_BONDED_MID 16
#define     CMD_COORDINATOR_ENABLE_REMOTE_BONDING 17

#define     CMD_NODE_READ 0
#define     CMD_NODE_REMOVE_BOND 1
#define     CMD_NODE_READ_REMOTELY_BONDED_MID 2
#define     CMD_NODE_CLEAR_REMOTELY_BONDED_MID 3
#define     CMD_NODE_ENABLE_REMOTE_BONDING 4
#define     CMD_NODE_REMOVE_BOND_ADDRESS 5
#define     CMD_NODE_BACKUP 6
#define     CMD_NODE_RESTORE 7

#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_RESTART 8
#define     CMD_OS_WRITE_CFG_BYTE 9
#define     CMD_OS_LOAD_CODE 10
#define     CMD_OS_SELECTIVE_BATCH 11
#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_GET 2
#define     CMD_LED_PULSE 3

#define     CMD_SPI_WRITE_READ 0

#define     CMD_IO_DIRECTION 0
#define     CMD_IO_SET    1
#define     CMD_IO_GET    2

#define     CMD_THERMOMETER_READ 0

#define     CMD_PWM_SET 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
```

10.4 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 not defined and implemented yet. See example *CustomDpaHandler-UserPeripheral-ADC.c* for potential implementation.

10.5 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_AuthorizePreBonding        16
#define     DpaEvent_UserDpaValue               17
#define     DpaEvent_FrcResponseTime            18
#define     DpaEvent_BondingButton              19
```

10.6 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
```

10.7 HW Profile IDs

```
HWPID_Default = 0,          // No HW Profile specified
HWPID_DoNotCheck = 0xffff   // Use this type to override HW Profile ID check
```

10.8 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
```

10.9 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    0xFF
```

11 Appendix

11.1 CRC Calculation

The following examples show the implementation of 1-Wire CRC used to protect UART Interface data. Before using the routines do not forget to initialize CRC accumulator variable to the initial value 0xFF.

11.1.1 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 */
}
```

11.1.2 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;
}
```

11.1.3 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;
}
```

11.1.4 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;
```

11.2 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 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.

11.2.1 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;
if ( Carry )
  checksum.low8++;

// Update higher checksum byte
checksum.high8 += checksum.low8;
if ( Carry )
  checksum.high8++;
```

11.2.2 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;
}
```

11.3 Custom DPA Handler Code at .hex File

The following example shows 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.

```
:020000040000FA
...
:08741000AC310024BA31080080
```

```
:10744000640070080A3A031D0A3225002F08403AEA
:10745000031D053220000D1A02322B00B814031050
:02746000080022
: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 to 0xEA3A.

11.4 IQRF OS Change

[sync] IQRF OS version at any DPA device can be upgraded (or downgraded) over the network without having a physical access to the device. It can also optionally change the DPA version at the same time. A special 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 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 LoadCode command.

Important: During the whole process of the IQRF OS change (starting at the time of sending 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 to 0xC05E. The request will not be executed if HWPID equals to 0xFFFF.

The actual IQRF OS change process after the response is received takes several seconds. During the process, the red LED in 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 to 0.

Address — A physical address of the external EEPROM memory block containing the IQRF OS change file. The address value is recommended to be a multiple of 64 because it allows more effective writing the content of the change file to the memory.

Response

NADR	PNUM	PCMD	HWPID	ErrN	DpaValue	0
NADR	0x20	0x80	0xC05E	0	?	Result

Result:

0 All 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 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.

11.4.1 IQRF OS Change File

The IQRF OS change file content should be inspected before the file is stored in external EEPROM in order 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 File version. Currently, only value 0x01 is supported.
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.
First 2 bytes contain DPA version at the same BCD format the enumeration uses.
3rd byte contains the following flags/bits:
0: DPA supports Coordinator
1: DPA supports Node
2: 0=STD mode, 1=LP mode
3: SPI interface
4: UART interface
5-7: unused

Note: all 3 bytes are zero when DPA is not part of the change file.
DPAfrom DPA version to change from. Same format as DPAto.

11.5 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.

11.5.1 W as a temporary variable

FLASH- RAM Speed

When a content of 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;`

11.5.2 Variable access reorder

FLASH- RAM Speed+

Try to group access to the variables from the same bank in order to avoid excess MOVLB instructions. By the way, C compilers are free to reorder statement in order 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`

11.5.3 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();`

11.5.4 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`

11.5.5 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 in order 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 );`

11.5.6 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 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;`

11.5.7 Switch instead of if

FLASH- RAM Speed+

CC5X generates more effective code in 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;`
	`}`

11.5.8 Function call before return

FLASH- RAM Speed+

If the very last function statement is another function (from the same page) call, then CC5X uses effectively *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 )`
	`void Method ()`
	`{`
	`  if ( enable )`

<pre>   enableSPI();
  else
      disableSPI();
}</pre>
	<pre> {
	      enableSPI();
	      // return forces CC5X to emit GOTO before
	      // else instead of CALL
	      return;
	  }
	  else
	      disableSPI();
	}</pre>

11.5.9 *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;
}
```

11.5.10 *Avoiding readFromRAM and getINDFx*

FLASH- RAM Speed+

IQRF OS allows to use *FSR0, *FSR1, INDF0, INDF1 for memory read purposes instead of un-effective and obsolete `readFromRAM()` and `getINDFx()` calls.

```
byte = readFromRAM( &mask );
```

```
FSR0 = (uns16)&mask;
byte = *FSR0;
```

11.5.11 *Advanced C-compiler optimized instructions*

FLASH- RAM Speed+

It is effective 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];
```

11.5.12 *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 )
```

```
uns8 loop = 12;
do
```

`{`
`  // use loop`
`  loop -= 3;`
`}`
	`{`
	`  // use loop`
	`  loop -= 3;`
	`}`
	`while ( loop != 0 );`

11.5.13 Use DECFSZ/INCFSZ

FLASH- RAM Speed+

Loop *do {} while ()* with a condition *--var != 0* or *++var != 0* leads to the effective 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 );`

11.5.14 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`
	`}`

11.5.15 Carry as a variable

FLASH- RAM- Speed+

Sometimes the Carry MCU flag can be carefully and effectively used as a variable.

Also, the following example shows how to compare and store the last value at one step.

```
// Keeps Carry, changes Zero_
#define XorWithAndCopyTo(value,xorWithAndCopyTo) do { \
  W = value; \
  xorWithAndCopyTo ^= W; \
  xorWithAndCopyTo = W; } while(0)

// Compare and copy 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 )
...
```

11.5.16 Limiting variable scope

FLASH RAM- Speed

CC5X is not able to detect a minimal variable scope and therefore to share RAM location between the variables. The latest possible variable declaration plus artificial code blocks will help to save some RAM.

<pre>uns8 temperature;
uns16 capture;
 	
temperature = getTemperature();
bufferCOM[0] = temperature;
 	
captureTicks();
capture = param3;
bufferCOM[1] = capture.low8;
bufferCOM[2] = capture.high8;</pre>
	<pre>{
	  uns8 temperature = getTemperature();
	  bufferCOM[0] = temperature;
	}
	
	{
	  captureTicks();
	  uns16 capture = param3;
	  bufferCOM[1] = capture.low8;
	  bufferCOM[2] = capture.high8;
	}</pre>

11.5.17 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 are not called.

<pre>uns16 Squared (uns8 value)
{
  uns8 tempValue = value;
  uns16 squared = (uns16)value * tempValue;
  return squared;
}</pre>
	<pre>uns16 Squared (uns8 value @ param2)
	{
	  uns8 tempValue @ param3;
	  tempValue = value;
	  uns16 squared @ param4;
	  squared = (uns16)value * tempValue;
	  return squared;
	}</pre>

`param2`, `param3`, `param4` can be used with caution. It is much safer to use user dedicated `userReg0` and `userReg1`.

11.5.18 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.

<pre>void Method (uns8 value)
{
  switch (value)
  {
  case 1:
  case 2:
       _LEDG = 1;
       break;
 	
  case 4:
  case 8:
       _LEDG = 0;
       break;
  }</pre>
	<pre>void Method (uns8 value @ W)
	{
	  switch (value)
	  {
	  case 1:
	  case 2:
	       _LEDG = 1;
	       break;
	
	  case 4:
	  case 8:
	       _LEDG = 0;
	       break;
	  }</pre>

}	}

11.5.19 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 );
}
```

11.5.20 FSRx as 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 effective.

.

```
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 );
```

11.5.21 Using FSRx to copy between buffers and variables

FLASH- RAM- Speed+

It is effective to use FSRx to repeatedly access (copy, compare) content of buffers and variables in order to avoid MOVLBs.

```
RX = bufferRF[0];
RTDT3 = bufferRF[10];
var0 = bufferRF[20];
var1 = bufferRF[30];
var2 = bufferRF[40];
```

```
FSR0 = bufferRF;
// or even better (shorter, but not faster)
setFSR0( _FSR_RF );

RX = FSR0[0];
RTDT3 = FSR0[10];
var0 = FSR0[20];
var1 = FSR0[30];
var2 = FSR0[40];
```

11.5.22 Accessing 16-bit MCU registers

FLASH RAM Speed

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;
```

11.5.23 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();`

11.5.24 Effective is not always effective

FLASH- RAM Speed+

Observe output .lst file when it makes sense.

`counter += value > maxValue;`	`if ( value > maxValue )` `  counter++;`

11.5.25 Assignment also have a value

FLASH- RAM Speed+

This can eliminate extra assignment statements.

`copyMemoryBlock(bufferAUX, bufferRF, 5);`
`DLEN = 5;`
`RFTXpacket();`
	`copyMemoryBlock(bufferAUX,bufferRF,DLEN=5);`
	`RFTXpacket();`

11.5.26 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;
}
```

11.5.27 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 more effective code is generated but the side effect is minimal.

`#define DELAY 1000` `startLongDelay( DELAY );`	`#define DELAY 1024` `startLongDelay( DELAY );`

11.5.28 Equality result

FLASH- RAM Speed+

When a function result is equality of two expressions, then instead of converting comparison result to the Carry (used to return bit result) it is better to return difference and to use Zero_ MCU flag. Carry flag can be even used for smaller/bigger comparison 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 )`
	`       ...`
	`}`

11.5.29 One instruction at 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;`

11.5.30 Utilization of the preloaded W

FLASH- RAM Speed+

CC5X is not able to optimize commutative expressions in order to use already preloaded variable or W register.

`uns8 var1, var2;`
 	
`var1 = 1;`
`if ( var1.0 )`
`{`
`  if ( var2 == var1 )`
`       nop();`
`}`
`else`
`  nop();`
	`uns8 var1, var2;`
	
	`var1 = 1;`
	`if ( var1.0 )`
	`{`
	`  if ( var1 == var2 )`
	`       nop();`
	`}`
	`else`
	`  nop();`

11.5.31 == 1 is more effective than != 1

FLASH- RAM Speed+

A test == 1 is more effective (`DECFSZ`) than a test != 1.

`if ( var1 != 1 )`
`  nop2();`
`else`
`  nop();`
	`if ( var1 == 1 )`
	`  nop();`
	`else`
	`  nop2();`

11.5.32 == 0xFF is more effective than != 0xFF

FLASH- RAM Speed+

A test == 0xFF is more effective (`INCFSZ`) than a test != 0xFF.

`if ( var1 != 0xFF )`
`  nop2();`
`else`
`  nop();`
	`if ( var1 == 0xFF )`
	`  nop();`
	`else`
	`  nop2();`

11.5.33 Expression modification

FLASH- RAM Speed+

Simplifying algebraic expressions can help a compiler to produce more effective code.

`uns8 a, b;`
 	
`if ( a > ( 16 - b ) )`
`      nop();`
	`uns8 a, b;`
	
	`if ( a + b > 16 )`
	`      nop();`

11.5.34 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
…
```

11.5.35 Default is first at switch

FLASH- RAM Speed+

If there is a *default* used inside *switch* then it should be the first “case“ in order 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:`
	`switch ( DLEN )`
	`{`
	`  default:`
	`       return 0;`
	
	`  case 12:`
	`       return 21;`
	
	`  case 34:`

`return 0;` `}`	`return 43;` `}`

11.5.36 Better to return from than after the loop

FLASH- RAM Speed+

It is more effective to return from the function in the middle of the loop then 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();
}
```

11.5.37 Modification instead of storing value

FLASH- RAM Speed+

In special cases, it is better to modify the value of the variable then to assign it as the compiler optimizes to the shorter code. 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;
  else
       if ( condition2 )
         state = STATE_B;
       else
         state = STATE_C;
```

```
#define       STATE_A       0
#define       STATE_B       1
#define       STATE_C       2

  uns8 state = STATE_A;
  if ( !condition1 )
  {
       state += STATE_B - STATE_A;// ++
       if ( condition2 )
         state += STATE_C - STATE_B; // ++
  }
```

11.5.38 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_ )`
	`        ...`

11.5.39 End condition of 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--; // effective
} while ( FSR1H != -1 );
```

11.5.40 Shift for a smart comparison

FLASH- RAM Speed+

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 )
```

11.5.41 Optimized return TRUE/FALSE

FLASH- RAM Speed-

Each `return TRUE` or `return FALSE` actually requires two instructions. If there are more such statements it is more effective to implemented 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();
}
```

11.5.42 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();`

11.5.43 Avoiding MOVLP #2

FLASH- RAM Speed-

If there are repeated calls of some function residing at 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();
```

11.5.44 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 effective code (setting just one bit).

```
memoryLimit = 64;
eeeWriteData( 0 );
```

```
// memoryLimit is zero so the next statement takes 1 instruction
memoryLimit |= 64;
eeeWriteData( 0 );
```

11.5.45 Compare to zero is more effective

FLASH- RAM Speed+

Comparing to a constant zero value is more effective 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 )`

11.5.46 setFSR01

FLASH- RAM Speed-

Registers FSR0 and/or FSR1 can be effectively set to the common buffer addresses by calling IQRF OS *setFSRxy* function. Calling this function takes 2 instructions only. Setting both or one of FSR registers normally takes 8 or 4 instructions respectively.

`FSR0 = &bufferCOM[0];` `FSR1 = &bufferINFO[0];`	`setFSR01( _FSR_COM, _FSR_INFO );`

12 DPA Release Notes

12.1 DPA 3.02

IQRF OS: 4.02D-08B8 (DCTR-7xD)

Changes and enhancements
- Autoexec and IO Setup can use embedded peripherals that are not enabled in the HWP Configuration.

New features
- New API variable RxFilter.

Bug Fixes
- Fixed an issue when during precise sleep the current drawn jumps by a few μA under certain GPIO settings.
- Fixes and enhancements at CustomDpaHandler-AutoNetwork example.

12.2 DPA 3.01

IQRF OS: 4.01D-08B7 (DCTR-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 HWP 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 HWP 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.

12.3 DPA 3.00

IQRF OS: 4.00D-08B1 (DCTR-7xD)
- DCTR-5xD devices are not supported anymore.
- Demo DPA version is not released anymore.
- DPA for [CN] devices 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.
- 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 HWP configuration and Write HWP 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.

12.4 DPA 2.28

IQRF OS: 3.08D-0858/3.08D-0879 (DCTR-5xD/DCTR-7xD)
- This is the ending major DPA release for DCTR-5xD.

Changes and enhancements
- 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. Same applies to the demo version of [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 DCTR-7xD for times above 2 s.

12.5 DPA 2.27

IQRF OS: 3.08D-0858/3.08D-0879 (DCTR-5xD/DCTR-7xD)

Changes and enhancements

- Parameter Mask added to Write HWP configuration byte command.
- Peripheral OS is always enabled regardless of the configuration settings.
- Change of the RF signal filter value at HWP Configuration takes effect after the device is restarted.

New features

- Write HWP configuration byte command can write multiple values including RFPGM settings.

12.6 DPA 2.26

IQRF OS: 3.08D-0858/3.08D-0879 (DCTR-5xD/DCTR-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 HWP 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 extra byte reserved for a future use.

New features

- Command LoadCode also supports loading code from IQRF plug-ins (.iqrf files). This allows e.g. upgrading DPA version over the network.
- Implemented *CustomDpaHandler-ChangeIQRFOS.iqrf* handler for changing 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 UART interface might receive a frame missing starting HDLC flag Sequence byte 0x7e.

12.7 DPA 2.24

IQRF OS: 3.07D-0852/3.07D-0870 (DCTR-5xD/DCTR-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 DCTR-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 DCTR7x devices when during precise sleep the current drawn exceeds approx. 500 µA.
- Fixed an issue when released DPA 2.20+ plugins for DCTR-7xD devices overwrite tailing (above size 736) instructions of Custom DPA Handler. Workaround - upload Custom DPA Handler after DPA plugin, but not in the inverse order.

12.8 DPA 2.23

IQRF OS: 3.07D-0852/3.07D-0870 (DCTR-5xD/DCTR-7xD)

Changes and enhancements

- Header files `DPA.h` can be compiled using 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.

12.9 DPA 2.22

IQRF OS: 3.07D-0852/3.07D-0870 (DCTR-5xD/DCTR-7xD)

New features

- New command Write HWP configuration byte.

Bug fixes

- Minimum required IQRF OS build number checked by OS Read for DCTR7x devices corrected.

12.10 DPA 2.21

IQRF OS: 3.07D-0852/3.07D-0870 (DCTR-5xD/DCTR-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 embedded request is discontinued when one request does not match HWPID.

12.11 DPA 2.20

IQRF OS: 3.07D-0852/3.07D-0870 (DCTR-5xD/DCTR-7xD)

- Support of DCTR-7xD devices.

Changes and enhancements

- DCTR-7xD Custom DPA handler Flash memory block extended to 864 instructions.
- [N] and [CN] devices send “Reset” DPA response when started the same way the [C] already did.
- Read HWP 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.

12.12 DPA 2.13

IQRF OS: 3.06D-0707 (DCTR-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.

12.13 DPA 2.12

IQRF OS: 3.06D-0707 (DCTR-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.

12.14 DPA 2.11

IQRF OS: 3.06D-0707 (DCTR-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%.

12.15 DPA 2.10

IQRF OS: 3.06D-0707 (DCTR-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 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 UserDpaValue variable. It is not transferred via *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 UART interface is used.
- DpaApiRfTxDpaPacketCoordinator now returns a number of hops to deliver DPA response 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 asynchronous or local request 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.

12.16 DPA 2.01

IQRF OS: 3.05D-06B5 (DCTR-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.

12.17 DPA 2.00

IQRF OS: 3.05D-06B5 (DCTR-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 „Reset“ message upon startup, see Device Startup.
- Notification event called even after read-only DPA response.
- 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 demo version only.

13 Document Revisions

171116	DPA v3.02 release
170714	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.
130602	DPA v2.01 release.
130512	DPA v2.00 release.

Sales and Service

Corporate office

IQRF Tech s.r.o., Prumyslova 1275, 506 01 Jicin, Czech Republic, EU
Tel: +420 493 538 125, Fax: +420 493 538 126, www.iqrf.tech
E-mail (commercial matters): sales@iqrf.org

Technology and development

www.iqrf.org
E-mail (technical matters): support@iqrf.org

Partners and distribution

www.iqrf.org/partners

Quality management

ISO 9001 : 2009 certified

Complies with ETSI directives EN 301489-1 V1.9.2:2011, EN 301489-3 V1.6.1:2013, EN 300220-1 V2.4.1:2012, EN 300220-2 V2.4.1:2012 and VO-R/10/05.2014-3.

Complies with directives 2011/65/EU (RoHS) and 2012/19/EU (WEEE).

Trademarks

Legal

On-line support: support@iqrf.org

Smarter Wireless. Simply.