IQRF SPI

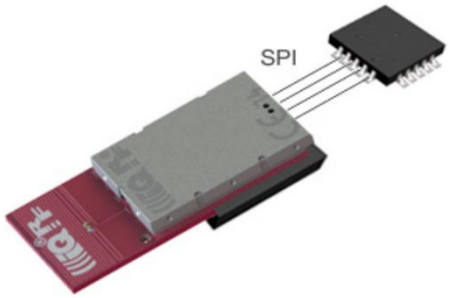
Technical guide

For TR-7xD

For IQRF OS v4.02 or higher

This document describes **SPI** implementation in IQRF TR transceivers. It is primarily intended for **application engineers** implementing one's own **SPI masters** connected to TR transceivers. E.g., it enables to write a one's own PC control program with features similar to ones used in IQRF IDE.

For other readers just the chapters *IQRF SPI purpose* and *SPI in TR transceivers* may be useful.



1 IQRF SPI purpose

SPI serves as an essential wired serial communication for IQRF platform.

It can be used for:

- General communication with TR transceiver
- **Handling** the application contents of the **memories** inside the TR (TR configuration, application program, plug-ins, and data in Flash and both EEPROM memories). Primarily, the contents can be **uploaded** into TR, but can also be **read** from TR, except application program which is allowed to be **verified** only, and plug-ins which are allowed neither to be read nor verified.

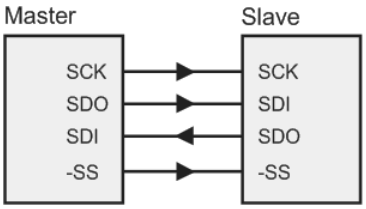
  For short, all such handling TR memories in programming mode is called **TR Upload** throughout this document.

2 SPI general overview

SPI™ (Serial Peripheral Interface, introduced by Motorola) is a standard serial four wire synchronous data bus that can operate in full duplex. Devices communicate in Master / Slave mode with a single master initiating data frames. Multiple Slave devices are allowed with individual Slave select lines.

The SPI bus specifies four logic signals:

SPI signal	Function	Implementation at TR	
		TR pin	Direction
SCK	Serial Clock	C6 / Q6	Issued by Master
SDI	Serial Data In	C7 / Q7	To TR
SDO	Serial Data Out	C8 / Q8	From TR
–SS	Slave Select	C5 / Q9	Issued by Master



The SPI bus with a single Slave.

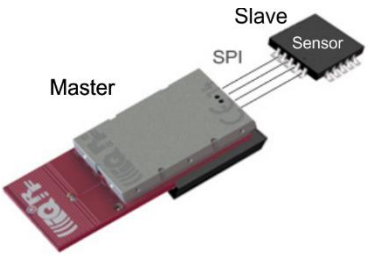
2.1 SPI at TR transceivers

The MCU inside the TR is equipped with generic **HW module** for serial communication (MSSP), among others implementing SPI Master/Slave.

Note: Since SPI shares HW resources (dedicated pins and the MSSP module), it is not possible to use them for I^2C or UART when SPI is used. However, if necessary, such as serial communications can alternatively be implemented in SW (without MSSP and using different pins), thus without the limitation above.

SPI at TR can work in Master or Slave mode as follows:

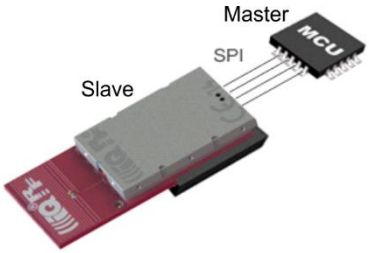
Master



One (or more) SPI slave device(s) (such as sensors or memories) can directly be connected to TR transceiver. However, SPI Master mode is **not supported by** IQRF OS and the communication must be implemented in other layers:

- **Under DPA:** Example `UserPeripheral-SPImaster` implements SPI Master as one of common user DPA peripherals. See *IQRF DPA Framework Technical guide* [3], chapter *UserPeripheral-SPImaster*.
- **Under IQRF OS:** `SPI_MASTER` is available as one of IQRF Advanced examples in C. See *IQRF Startup package* [4], folder *Examples/IQRF_OS/Advanced_examples*.

Slave



On the contrary, SPI Slave mode gains massive **support from IQRF OS**:

- IQRF OS supports TR transceiver working as SPI slave, full or half duplex.
  Simplifiedly speaking, SPI slave implemented in IQRF observes the SPI standard, but with the only exception: Due to the highest priority of RF communication in OS foreground, the minimal space between individual SPI bytes must be longer than the standard one. See chapter *Timing*.
- SPI is implemented as a state machine and runs in **OS background**.
- `bufferCOM` is a buffer dedicated to SPI communication. It enables an easy data exchange between the Master and the Slave.
- OS provides the set of powerful SPI **functions**: `enableSPI()`, `disableSPI()`, `startSPI(x)`, `restartSPI()`, `stopSPI()` and `getStatusSPI()`. See *IQRF OS Reference guide* [2], chapter *SPI*.
- OS implements a very simple proprietary **protocol** above the basic layer described in chapters *Data transfer* and *Timing*. Refer to chapters *SPI status*, *Packet structure*, *SPI in practice* and *Examples*.
- OS supports a set of **commands** (to be sent via this protocol) implementing Upload TR memories in **programming mode**. See chapters *Programming mode* and *TR upload*.

Implementation possibilities:

- **Under DPA:** There are two ways absolutely different on principle:
  - **DPA Peripheral** is intended to exchange data between the TR and a peripheral (in this case via SPI). Refer to *IQRF DPA Framework Technical guide* [3], chapter *Peripherals*.
  - **DPA Interface** is intended to control the TR in wireless Mesh network from a higher system (in this case via SPI). It is a channel communicating via the DPA protocol (transferring DPA messages containing so called DPA foursome and optional data). The DPA protocol corresponds to the `DM` and `DS` bytes of IQRF SPI protocol, see chapter *Packet structure*. Refer to *IQRF DPA Framework Technical guide* [3], chapter *Interfaces*. DPA Interface is typically used at network Coordinator. However, using at network Nodes is also possible.
- **Under IQRF OS:** Example *External SPI Master* shows the implementation of the Master in C language. See *IQRF Startup package* [4], folder *Examples/Miscellaneous*.

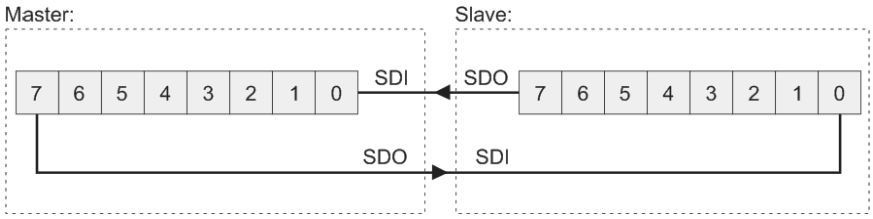
The rest of this document refers to SPI Slave implemented in IQRF OS.

3 SPI communication

3.1 Data transfer

SPI communication is fully synchronous without any timeouts. It is packet oriented and works in OS background. Packets consist of selectable number of bytes. IQRF SPI allows packets up to 64 B.

Each data byte is transferred using two internal shift registers to form a circular buffer, MSb first. After the registers have been shifted (rotated 8 bits to the left), the Master and Slave have swapped their register values.



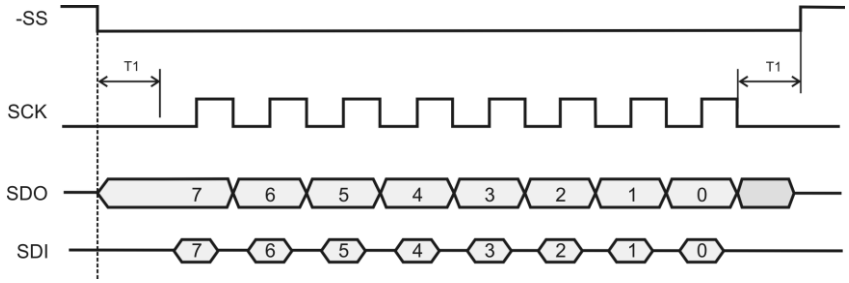
If the Slave is configured to receive data, the received byte is copied to the `bufferCOM` then. More bytes can be transferred by this repeated process in single packet and stored starting from `bufferCOM[0]`.

The TR module operates according the following Slave specification:

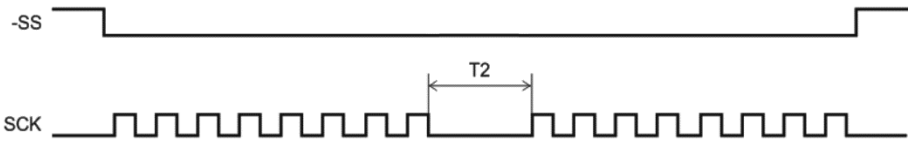
Slave select	Active low
Idle clock polarity	Low
Clock edge	Output data from TR on rising SCK edge

3.2 Timing (from the Slave point of view)

To start the communication, the Master pulls the Slave select low. Then the Master must wait for at least the T1 period before starting to issue clock cycles. During each SPI clock cycle, a duplex data bit can be transferred to complete a full duplex byte transmission in 8 clocks.



Then the Master must wait for at least the T2 period before sending the next byte.



SCK	SPI clock	250 kHz	Max.
T1	–SS to SCK (falling edge), SCK (falling edge) to –SS	5 µs	Min.
T2	Delay between bytes	30 µs For non-networking RF communication 150 µs For networking RF communication	Min.

The **delay between bytes** (T2) is very important parameter for IQRF SPI communication. Due to RF communication and possible peripheral processes running in background (e.g. writing to EEPROM), proper value depends on the application. RF communication has higher priority than SPI (RF interrupts should be served first), is time-demanding (depending on RF duty cycle) and runs in OS foreground. That is why RF RX has the strongest impact to SPI throughput. Additionally, it may strongly depend even on RF noise. All IQRF standard USB devices (CK-USB-04A, GW-USB-06, …) use T2 = 150 µs.

After transferring the last byte, -SS must still be kept low at least the T1 period (after the last SCK falling edge).

For detailed information about SPI implementation in the MCU see MCU datasheet [5], chapter *MSSP module*.

3.3 SPI status

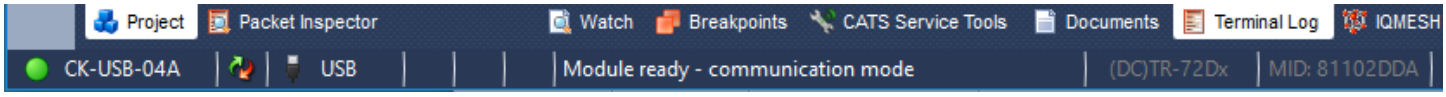
IQRF SPI is implemented as a state machine. Thus, TR stays always in one of given states and can be polled for this state at any time:

- at the Slave using the `getStatusSPI()` function. See the *IQRF OS Reference guide* [2].
- at the Master using the `SPI_CHECK` command sent to the Slave. For answers see the table below (SPI status).

After the `SPI_CHECK` command sent from the Master to the Slave via SPI the following answer can be returned:

Hex value	SPI status (`SPISTAT`)
00 or FF	**SPI not active** (**HW error**, **SPI disabled** by the `disableSPI()` command etc.). The default state is 00.
07	**SPI suspended** (**stopped** by the `stopSPI()` command)
3F	**SPI not ready** (buffer full, last **CRCM O.K.**). Data in `bufferCOM` is protected against overwriting by next transmission from the Master. `startSPI(0)` can be used at the Slave to enable continuing the transmission.
3E	**SPI not ready** (buffer full, last **CRCM error**). `startSPI(0)` can be used to recover.
40 to 79	**SPI data ready** to be read from the Slave • For 0x41 to 0x7F: Data length = this SPI status value - 0x40 • For 0x40: Data length = 64 Examples: • After `startSPI(10)` the TR returns status `0x4A` until SPI transfer starts. It means that 10 B data is ready in `bufferCOM` to be sent from the Slave. • After `startSPI(41)` the TR returns status `0x69`. • After `startSPI(64)` the TR returns status `0x40`.
80	**SPI ready** – **Communication** mode
81	**SPI ready** – **Programming** mode
82	**SPI ready** – **Debugging** mode

The SPI status of TR is indicated by the IQRF IDE when used with related IQRF development tools (e.g. CK-USB-04A):



3.4 Packet structure

Two types of SPI packets are defined: `SPI_CHECK` and `SPI_CMD`.

`SPI_CHECK`: The Master checks the SPI status of TR:

Master:	**SPI_CHECK**
Slave:	SPISTAT

`SPI_CMD`: The Master reads / writes a packet from / to TR:

Master:	**SPI_CMD**	PTYPE	DM1	DM2	. . .	$DM_{SPIDLEN}$	CRCM
Slave:	SPISTAT	SPISTAT	DS1	DS2	. . .	$DS_{SPIDLEN}$	CRCS

Where:

`SPI_CHECK` `0x00` SPI check. Available anytime.

`SPI_CMD` `0xF0` Data read / write (exchanged between the Master and `bufferCOM` at the Slave). Available in all SPI ready modes. See *Example 1* and *Example 3*.

`0xFA` The same as `0xF0`. Intended for DPA. Available in Communication mode only. For DPA data read use `0xF0`, see above.

`0xF5` Get TR Module Info. Available in Communication mode only. 16 B is returned, but only first 8 B is implemented in OS. For Module Info format, refer to the *IQRF OS Reference guide* [2], function `moduleInfo()` and *Example 2*.

`0xF3` Write data to EEPROM. Write other TR configuration parameters. Available in Programming mode only. See *TR memories handling*.

`0xF2` Read data from EEPROM. Read other TR configuration parameters. Available in Programming mode only.

`0xF6` Write data to EEEPROM or Flash. Read data from EEEPROM. Write HWP configuration. Available in Programming mode only.

`0xFC` Verify data in Flash. Read HWP configuration. Available in Programming mode only.

`0xF9` Upload from proprietary secured `.IQRF` file provided by IQRF producer. Available in Programming mode only.

`SPISTAT` SPI status (see the *table* above)

`PTYPE`:

b7	b6	b5	b4	b3	b2	b1	b0
CTYPE	SPIDLEN						

`SPIDLEN` SPI packet data length (from 1 to 64)

`CTYPE` Communication type (regarding data transfer directions)

b7 = 1: `bufferCOM` changed (`DM` → `DS` or `DM` ↔ `DS`)

b7 = 0: `bufferCOM` unchanged (`DS` → `DM`)

Typical applications often use just unidirectional transmissions (`DM` → `DS` and `DS` → `DM`) but in fact, physical communication is always full duplex:

- `DM` ↔ `DS`: The Master sends useful data (to `bufferCOM`) and the Slave sends useful data (from `bufferCOM`).
- `DM` → `DS`: The Master sends useful data (to `bufferCOM`).

  The Slave sends dummy data (currently meaningless data from `bufferCOM`) which should be ignored by the Master.
- `DS` → `DM`: The Master sends dummy data (e.g. zeroes, see example below) just to generate clocks.

  The Slave sends useful data (from `bufferCOM`) and ignores incoming data (ignoring is arranged by OS).

DM	Data from the Master
DS	Data from the Slave
CRCM	SPI_CMD xor PTYPE xor DM1 xor DM2 ... xor $DM_{SPIDLEN}$ xor 0x5F CRCM must be calculated and sent by the Master. OS automatically verifies received CRCM and: • Returns 0x3F (O.K.) or 0x3E (Error) to the Master as an answer to the SPI_CHECK command. • Updates the SPICRCok flag for the Slave after the getStatusSPI().
CRCS	PTYPE xor DS1 xor DS2 ... xor $DS_{SPIDLEN}$ xor 0x5F CRCS is automatically calculated and included to outgoing packet by OS and should be verified by the Master.

Tip: IQRF IDE *SPI Test* provides a powerful tool for SPI communication testing, including CRC calculation and checking.

3.5 SPI in practice

All SPI communication is under the Master's control. The Slave can not initiate the communication at all. It can only offer one's own state to the Master including a possible request to send data or to refuse the transfer when it is not convenient for the Slave. Therefore, the Master should watch the state of the Slave continuously and response in corresponding way (respect the Slave requirements, read offered data, calculate `CRCM` and check `CRCS`, verify successful reception by the Slave, repeat unsuccessful transfers etc.)

The Master should periodically send `SPI_CHECK` to get TR status. A good practice for the Master is to send `SPI_CHECK` every 10 ms. (This is a period used e.g. by IQRF IDE.)

If the Slave requests to send data, the Master should generate the "reading" packet for it. If omitted, the Slave still stays in the state `0x40` to `0x7F` and can be changed e.g. by `startSPI(x)`.

If the Master needs to send data, it should check first whether the Slave is ready to receive, especially whether has processed previous data correctly (so that the Slave is in the communication mode again, without reporting an error). If omitted, the SPI packet will be lost and the `bufferCOM` will not be overwritten. To refuse SPI transfer by the Slave for some reason required by the application, the `stopSPI()` function is intended. If the Slave is not ready (SPI is stopped or disabled by the Slave or the `bufferCOM` is full), it is useless to send packets by the Master.

Stopped SPI communication can also be restarted by `startSPI()` to continue. Thus, current packet can be completed later on.

Useful data can be exchanged at the same time (full duplex) but it is not usual in typical applications.

The Slave should also take into consideration its own state which can be checked by `getStatusSPI()`. Return value and output flags are optimized for user application requirements that is why they differ from the SPI status format sent to the Master.

If a communication error occurs during reading the `bufferCOM`, the reading can immediately be repeated. The data still remains valid in the `bufferCOM` (unless it is overwritten by the user application). See *Example 3*.

If the `CRCM`/`CRCS` check is not sufficient for given application, for further security increase, the user should include one's own additional verification in user data.

See *IQRF OS User's guide* [6], *IQRF OS Reference guide* [2], Application examples included in the *IQRF Startup package* [4], folder *Examples*, and www.iqrf.org.

3.6 Examples

Configuration:

- TR-7xDx transceiver plugged in CK-USB-04A
- The `E07-SPI` example uploaded in TR. See *IQRF Startup package* [4], folder *Examples/IQRF_OS/Basic_examples.*
- The IQRF IDE with *SPI Test* running on PC which CK is connected to.

Values "`From Master`" should be typed into the *Data to send* field and values "`From Slave`" can be watched in the *Terminal Log* window.

3.6.1 Example 1

Sending data between Master and Slave

```
From Master: 00  // SPI_CHECK
From Slave:  80  // SPI_STATUS – TR is in communication mode
.
.                // Master periodically sends SPI_CHECK to get TR status
.

From Master: 00  // SPI_CHECK
From Slave:  80  // SPI_STATUS – TR stays in communication mode

From Master: F0.81.69.47.00              // Now the Master requires to send data "i"
                 // F0 81: SPI_CMD (PTYPE = 81: DM → DS, SPIDLEN = 1)
                 // 69: "i" in ASCII
                 // 47: CRCM (F0 xor 81 xor 69 xor 5F)
                 // 00: SPI_CHECK added to get SPI_STATUS: CRCM OK or Error
From Slave:  80.80.30.EE.3F              // 5 B answer for 5 B issued by Master
                 // 80 80: 2 x SPI_STATUS – TR is in communication mode
                 // 30: dummy - current (undefined) content of bufferCOM
                 // EE: checksum (81 xor 30 xor 5F)
                 // 3F: SPI_STATUS: CRCM is OK

From Master: 00  // SPI_CHECK
From Slave:  4A  // SPI_STATUS - TR offers 10 B in bufferCOM

From Master: F0.0A.00.00.00.00.00.00.00.00.A5.00
                 // F0 0A: SPI_CMD (PTYPE = 0A: DS → DM, SPIDLEN = 10)
                 // 10 x 00: dummy (to generate master clock only)
                 // A5: checksum (F0 xor 0A xor 00 ... xor 00 xor 5F)
                 // 00: SPI_CHECK
From Slave:  4A.4A.30.31.32.33.34.35.36.37.38.39.54.3F
                 // 4A 4A: 2 x SPI status (ready to sent 10 B)
                 // 30 ... 39: data from bufferCOM (10 B)
                 // 54: checksum (0A xor 30 ... xor 39 xor 5F)
                 // 3F: SPI_STATUS: CRCM is OK

From Master: 00  // SPI_CHECK
From Slave:  00  // SPI_STATUS - TR is in communication mode

.
.                // Master periodically sends SPI_CHECK to get TR status
.
```

3.6.2 Example 2

Reading Module info from TR via SPI

```
From Master: 00     // SPI_CHECK
From Slave:  80     // SPI_STATUS – TR is in communication mode

From Master: F5.10.00.00.00.00.00.00.00.00.00.00.00.00.00.00.BA.00
                    // F5: command get Module Info
                    // 10: PTYPE – read 16 B
                    // 16 x 00: dummy
                    // BA: CRCM
                    // 00: SPI_CHECK added to get SPI_STATUS (CRCM OK or Error)
From Slave: 80.80.81.00.2B.E1.37.24.41.07.00.00.00.00.00.00.00.00.51.3F
                    // 80 80: 2 x SPI_STATUS – TR is in communication mode
                    // 81.00.2B.E1: Module ID = 81002BE1
                    // 37: IQRF OS version 3.07D
                    // 24: TR type = TR-72D, FCC not certified, MCU type = PIC16LF1938
                    // 41.07: IQRF OS build 0x0741
                    // Next 8 B are not defined
                    // 51: CRCS
                    // 3F: SPI_STATUS: CRCM is OK

From Master: 00     // SPI_CHECK
From Slave:  80     // SPI_STATUS – TR is in communication mode
```

3.6.3 Example 3

Repeated reading after communication failure

```
From Master: 00     // SPI_CHECK
From Slave:  80     // SPI_STATUS – TR is in communication mode

From Master: F0.81.69.47.00 // Write "i" (0x69 in ASCII)
From Slave:  80.80.30.EE.3F // See Example 1

From Master: 00     // SPI_CHECK
From Slave:  4A     // SPI_STATUS – TR offers 10 B in bufferCOM

From Master: F0.0A.00.00.00.00.00.00.00.00.00.A4.00
                    // Reading with error in CRCM (A4 instead of A5)
From Slave:  4A.4A.30.31.32.33.34.35.36.37.38.39.54.3E
                    // Trailing 3E indicates wrong CRCM
...                 // Send SPI_CHECK until TR is in communication mode again

From Master: 00     // SPI_CHECK
From Slave:  80     // SPI_STATUS – TR is in communication mode

From Master: F0.0A.00.00.00.00.00.00.00.00.00.A5.00
                    // Repeated reading with correct CRCM (It is not needed to
                    // write "i" again. Data remains valid in bufferCOM)
From Slave:  80.80.30.31.32.33.34.35.36.37.38.39.54.3F
                    // Trailing 3F indicates correct CRCM

From Master: 00     // SPI_CHECK
From Slave:  80     // SPI_STATUS – TR is in communication mode
```

4 TR Upload

TR **Upload** (sometimes called TR ***Programming*** as well) means writing a user application or configuration into TR transceiver. Similar approaches can also be applied for reading or verification of uploaded data.

The term ***Upload*** is simplifiedly used for handling contents of **all TR memories** in **programming mode** throughout this document.

4.1 Programming mode

Upload and all other handling TR memories are available when TR is in programming mode, either wired (**PGM**) or wireless (**RFPGM**).

The Upload procedure should be performed as follows:

1. Reset TR transceiver

- Set all SPI interface pins at the Master to low levels.
- Switch TR power supply Off
- Wait 300 ms
- Switch TR power supply On
- Set the idle levels to SPI interface pins at the Master, see the *Slave specification*.

2. Activate the programming mode

Copy the level on the SDO pin to the SDI pin for the first 400 ms after the TR is switched on. TR transceiver generates a determinate data sequence on the SDO pin. If this is copied to the SDI pin, TR is switched to the programming mode. Such copying can be implemented by SW e.g. as follows:

```
while (is_delay_400ms)
{
  if (SDO_level) SDI_level = 1;
  else           SDI_level = 0;
}
```

3. Wait until TR transceiver is in programming mode

Repeat polling for the TR state (`SPI_CHECK`). If TR starts to respond with 0x81 the programming mode is initiated.

4. Perform required operations

Perform TR upload and/or other required tasks according to chapter *TR memories handling*.

5. Terminate the programming mode

After sending the last packet according to point 4 above, it is necessary to wait for end of SPI cycle by polling the TR state (`SPI_CHECK`). If TR transceiver starts to respond with 0x81, reset the TR transceiver according to the point 1 above and the newly uploaded application will start.

4.2 TR memories handling

Of course, all SPI_CMD packets introduced in this chapter must comply with the specification described in chapter *Packet structure*. Thus, packets sent from the Master should also contain PTYPE and CRCM which are not mentioned below for simplicity.

Caution: All addresses in .HEX files generated by CC5X compiler are multiplied by two.

Therefore, when handling with memories according to all the commands described below, all addresses read from a .HEX file must be divided by two first.

All HEX addresses mentioned below mean the resulting addresses (thus divided by two already). See the *Example* in chapter *Write data to EEEPROM*.

4.2.1 EEPROM

Write data to EEPROM

Writing into EEPROM inside the MCU is possible in individual independent data blocks with up to 32 B.

- Write data to EEPROM
  SPI_CMD = 0x**F3**
  DM1 = Address in EEPROM (0 – 255)
  DM2 = Number of data bytes to be written (1 – 32). Addresses must be in allowed range (DM1 + DM2 ≤ 256).
  DM3 to DMx = Data to be written (x = 2+DM2)
- Repeat SPI state checking until the operation is completed (SPI_CHECK = 0x81). Then it is possible to continue (send a next SPI packet).

Read data from EEPROM

Reading from EEPROM inside the MCU is possible in individual independent data blocks with up to 32 B.

- Store data block from EEPROM into bufferCOM[0-31]. Complete 32 B block is always stored.
  SPI_CMD = 0x**F2**
  DM1 = Address in EEPROM (0 – 255)
  DM2 = 0 (Must be cleared. Reserved for future use.)
- Repeat SPI state checking until (SPI_CHECK = 0x60). Then 32 B data is ready in bufferCOM to be sent.
- Read the data from bufferCOM
  SPI_CMD = 0xF0
  DM1 = Number of bytes to be transferred (1 to 32)
  DM2 to DMx = 0x00 (dummy), x = 1 + DM1
- DS1 to DS_{DM1} contains data read from EEPROM
- Repeat SPI state checking until the operation is completed (SPI_CHECK = 0x81). Then it is possible to continue (send a next SPI packet).

4.2.2 EEEPROM

Write data to EEEPROM

Writing into EEEPROM (serial) is possible in individual independent data blocks with fixed length 32 B.

- Write data to EEEPROM
  `SPI_CMD = 0x`**`F6`**
  `DM1` = Index address in EEEPROM, lower byte
  `DM2` = Index address in EEEPROM, upper byte
    - Index address must be in allowed range 0x0000 to 0x1FF.
    - Index addresses access individual 32 B blocks. E.g., index 0x0000 addresses EEEPROM from physical address 0x0000 to 0x001F, 0x0001 from 0x0020 to 0x003F, …, 0x01FF from 0x3FE0 to 0x3FFF.
    - CC5X compiler generates virtual addresses from physical addresses in EEEPROM by adding the offset 0x0200. Address 0x0200 in HEX file corresponds to physical address 0x0000 in EEEPROM etc.
    - Thus, `index address = (HEX address – 0x0200) div 0x20`.
    - Example:
      - A record in .HEX file: `:10044000100011001200`...
      - HEX address = 0x0440 div 2 = 0x0220
      - Index address = (HEX address – offset 0x0200) div 0x20 = (0x0220 – 0x0200) div 0x20 = 1
      - `DM1` = 0x01, `DM2` = 0x00
      - This addresses the block from 0x0020 to 0x003F.

  `DM3` to `DM34` = Data to be written (always 32 B)
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

Read data from EEEPROM

Reading from EEEPROM (serial) is possible in individual independent data blocks with up to 32 B.

- Store data block from EEEPROM into `bufferCOM[0-31]`. Complete 32 B block is always stored.
  `SPI_CMD = 0x`**`F6`**
  `DM1` = Index address in EEEPROM, lower byte
  `DM2` = Index address in EEEPROM, upper byte
    - Index address must be in allowed range 0x0400 to 0x5FF.
    - Index addresses access individual 32 B blocks. E.g., index 0x0400 addresses EEEPROM from physical address 0x0000 to 0x001F, 0x0401 from 0x0020 to 0x003F, …, 0x05FF from 0x3FE0 to 0x3FFF.
    - Thus, `index address = (EEEPROM address div 0x20) + 0x0400`.
- Repeat SPI state checking until (`SPI_CHECK = 0x60`). Then 32 B data is ready in `bufferCOM` to be sent.
- Read data from `bufferCOM`
  `SPI_CMD = 0xF0`
  `DM1` = Number of bytes to be transferred (1 – 32)
  `DM2` to `DMx = 0x00` (dummy), `x = 1+DM1`
- `DS1` to DS_{DM1} contains data read from EEEPROM
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

4.2.3 Flash memory

Write data to Flash

Writing into Flash is only possible in individual independent data blocks with fixed length **16 words** (16 machine instructions), starting from addresses modulo 16 (0x3A00, 0x3A10, 0x3A20, ...), each word by two 8 b `DMx` values, LSB first. However, the Flash memory cells must be precleared before writing. **Preclearing** is only possible in blocks with fixed length **32 words** (32 machine instructions), starting from virtual addresses modulo 32 (0x3A00, 0x3A20, …). It is performed at given block automatically whenever writing into Flash is requested. **All 'unused' positions** (without meaningful values requested by the application) **throughout the entire precleared block** of 32 words must be filled with **`0xFF 0x34`** values.

Tip: To place a code at a specific location in Flash, the `#pragma origin ADDRESS` directive for C compiler is intended.

- Write data to Flash:
  `SPI_CMD = 0x`**`F6`**
  `DM1 =` Virtual address in Flash, lower byte
  `DM2 =` Virtual address in Flash, upper byte.
  The address must be modulo 16 in allowed range (e.g. `3A00 – 3FFF` for standard Flash).
  `DM3 to DM34 =` Data (32 B, 16 instructions)
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

Verify data in Flash

Verification is based on reading the individual words from the Flash but with XORed lower and upper bytes of each word. It is possible in individual independent data blocks with up to 32 words (32 machine instructions). Thus, one verification cycle spans up to two Flash writing cycles.

- Store data block from Flash into `bufferCOM[0-31]`. Complete 32 B block is always stored.
  `SPI_CMD = 0xFC`
  `DM1 =` Virtual address in Flash, lower byte
  `DM2 =` Virtual address in Flash, upper byte
  The address must be modulo 32 (0x3A00, 0x3A20, …) in allowed range (e.g. `3A00 – 3FFF`) for standard Flash).
- Repeat SPI state checking until (`SPI_CHECK = 0x60`). Then 32 B data is ready in `bufferCOM` to be sent.
- Read data from `bufferCOM`
  `SPI_CMD = 0xF0`
  `DM1 =` Number of bytes to be transferred (1 – 32)
  `DM2 to DMx = 0x00` (dummy), `x = 1+DM1`
- `DS1` to DS_{DM1} contains XORed data words read out from the Flash.
  `word.low8[address] XOR word.high8[address], word.low8[address+1] XOR word.high8[address+1], ...`
  `... XOR word.low8[address+31] XOR word.high8[address+31]`
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

Write IQRF plug-in to Flash

IQRF OS plug-in can be uploaded into Flash memory by sending the raw data from given `.IQRF` file. It uses a line-oriented text format. Lines initiated with the '`#`' character must be ignored. All other lines are intended as data to be uploaded as it is (without any changes). Every line corresponds to one SPI packet. Every two subsequent characters represent one `DMx` data byte in hexadecimal. Number of bytes in a line may vary up to 32.

- Write plug-in data to Flash
  `SPI_CMD = 0x`**`F9`**
  `DM1 to DMx =` Data from the line of the `.IQRF` file
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).
- Repeat this two steps line by line until all lines are uploaded.

More plug-ins can be uploaded in a single TR at the same time in this way.

4.2.4 TR configuration

TR configuration means an obligatory predefined basic TR setup to be uploaded into EEPROM and Flash. It is restored after every TR power on or reset and can be changed (except of the RF band switching) during application executing.

TR configuration consists of **OS**, **HWP** and **Security** parts.

HWP part should be uploaded even when DPA approach is not used. However, in this case only the RF channels (specified in HWP configuration bytes 6 and 7, see to *IQRF DPA Framework Technical guide* [3], chapter *HWP configuration*) are read from it, but other parameters (e.g. RF output power) must be selected in application program.

TR configuration can be created, modified, uploaded into TR and read from TR interactively in IQRF IDE. It is also possible to export / import the configuration to / from a .**TRCNFG** file. See IQRF IDE Help for details.

Caution: The method described in this chapter is the only way to upload / download TR configuration and must strictly be observed. Direct access to memory areas dedicated to the configuration by the procedures intended for general access to Flash or EEPROM (described above) are not possible.

Write OS part of TR configuration

Select **RF band**

```
SPI_CMD = 0xF3
DM1 = 0xC0
DM2 = 0x01
DM3 = 0x00      868 MHz
      0x01      916 MHz
      0x02      433 MHz
```

Note: RF band must be set properly even if it is the only possibility, e.g. for 433 MHz transceivers or TR-7xDA-IL (not supporting RF band switching).

Setup **RFPGM**

```
SPI_CMD = 0xF3
DM1 = 0xC1
DM2 = 0x01
DM3 = x         For meaning see IQRF OS Reference guide [2], function setupRFPGM(x).
```

Read OS part of TR configuration

- Read other TR configuration parameters into `bufferCOM[0-31]`. Complete 32 B block is always stored.

```
SPI_CMD = 0xF2
DM1 = 0xC0
DM2 = 0
```

- Repeat SPI state checking until (`SPI_CHECK = 0x60`). Then 32 B data is ready in `bufferCOM` to be sent.
- Reading the data from `bufferCOM`

```
SPI_CMD = 0xF0
DM1 = 0x20          32 B to be transferred
DM2 to DM33 = 0x00  Dummy
```

- `DS1` to `DS32` TR configuration data

```
DS1     RF Band
DS2     RFPGM setup
DS3-32  Reserved
```

- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

Write HWP part of TR configuration

HWP configuration (32 B) can be uploaded into TR using the write procedure repeated two times (to send two SPI packets, each containing 16 B data) as follows:

- Write lower 16 B of HWP configuration to Flash:
  `SPI_CMD = 0x`**F6**
  `DM1 = 0xC0` (virtual address of HWP configuration origin in Flash, lower byte)
  `DM2 = 0x37` (virtual address of HWP configuration origin in Flash, upper byte)
  `DM3` to `DM34 =` Data (32 B) to be written. It must be in format `0x34dd`, where `dd` is HWP configuration byte.
    - For meaning of individual `dd[i]` values (`i = 0` to `31`), refer to *IQRF DPA Framework Technical guide* [3], chapter *HWP Configuration*. Reserved items (not defined at current DPA version) must be cleared (`dd[i] = 0`).
    - HWP configuration address 0x00 corresponds to virtual address 0x37C0 in Flash memory etc.
    - `dd[0]` = `dd[1] XOR dd[2] XOR … XOR dd[31] XOR 0x5F` is a checksum of HWP configuration array. Refer to *IQRF DPA Framework Technical guide* [3], chapter *Write HWP Configuration*.
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`).
- Write upper 16 B of HWP configuration to Flash:
  `SPI_CMD = 0x`**F6**
  `DM1 = 0xD0` (virtual address of upper half of HWP configuration in Flash, lower byte)
  `DM2 = 0x37` (virtual address of upper half of HWP configuration in Flash, upper byte)
  `DM3` to `DM34 =` Data (32 B) to be written. It must be in format `0x34dd`, see above.
    - `dd[30]` is dedicated to configuration version which is used by IQRF IDE for checks but not defined within IQRF DPA Framework. E.g. value 0x03 represents DPA v3.0.
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

Read HWP part of TR configuration

HWP configuration can be read from TR in one SPI packet containing a 32 B data block using the following read procedure. Thus, one reading cycle spans up to both HWP configuration writing cycles.

- Read HWP configuration from Flash into `bufferCOM[0-31]`. Complete 32 B block is always stored.
  `SPI_CMD = 0x`**FC**
  `DM1 = 0xC0` (virtual address of HWP configuration origin in Flash, lower byte)
  `DM2 = 0x37` (virtual address of HWP configuration origin in Flash, upper byte)
    - HWP configuration address 0x00 corresponds to virtual address 0x37C0 in Flash memory etc.
- Repeat SPI state checking until (`SPI_CHECK = 0x60`). Then 32 B data is ready in `bufferCOM` to be sent.
- Read data from `bufferCOM`
  `SPI_CMD = 0x`**F0**
  `DM1 = 32` Number of bytes to be transferred
  `DM2` to `DM33 = 0x00` Dummy
- `DS1` to `DS32` contains XORed data read from the words at the HWP configuration area:
  `DS[x] = HWP_configuration_word.low8[x] XOR HWP_configuration_word.high8[x]`, (`x = 1 – 32`).
  Factual configuration byte should be extracted form this as follows: `dd[x] = DSx xor 0x34`.
  For meaning of individual values Refer to *IQRF DPA Framework Technical guide* [3], chapter *Write HWP Configuration*.
- Repeat SPI state checking until the operation is completed (`SPI_CHECK = 0x81`). Then it is possible to continue (send a next SPI packet).

Write Security part of TR configuration

Specify **Access password**

`SPI_CMD = 0x`**F3**
`DM1 = 0xD0`
`DM2 = 0x10`
`DM3 to DM18 = password[0-15]` See *IQRF OS Reference guide* [2], function *setAccessPassword*.

Specify **User key**

`SPI_CMD = 0x`**F3**
`DM1 = 0xD1`
`DM2 = 0x10`
`DM3-DM18 = key[0-15]` See *IQRF OS Reference guide* [2], function *setUserKey*.

5 Migration notes

5.1 Migration from IQRF OS v3.0xD or v4.00D to v4.02D

Features	OS v3.0xD	OS v4.02D
TR upload	Not publicly documented	Publicly documented
TR configuration upload	Not supported	Supported

5.2 Migration from IQRF OS v3.0xD or v4.00D to v4.01D

Do not use OS v4.01D at all due to a serious bug in RFPGM (wireless upload).

5.3 Migration from TR-5xD to TR-7xD

Features	TR-5xD	TR-7xD
–SS to SCK (falling edge), SCK (falling edge) to –SS (period T1)	T1 = 10 µs	T1 = 5 µs
Min. delay between bytes (period T2)	T2 = from 100 µs to 700 µs	T2 = 30 µs (non-networking RF)
		T2 = 150 µs (networking RF)
Slave select deactivating between bytes (period T3 *)	Necessary (T3 = min. 20 µs)	Not necessary (T3 = 0)

See chapter *Timing*.

* Refer to the *IQRF SPI Technical guide for TR-5xD* [7].

6 Documentation and information

1 IQRF CDC Technical guide – Specification of CDC implementation in IQRF transceivers
2 IQRF OS Reference guide – Description of OS functions and macros
3 IQRF DPA Framework Technical guide – DPA description
4 IQRF Startup package – Complete SW bundle to start with IQRF
5 MCU datasheet – For MCU inside the TR
6 IQRF OS User's guide – Description of IQRF OS
7 SPI Technical guide for TR-5xD – Specification of SPI implementation in TR-5xD transceivers

If you need a help or more information, contact IQRF support.

7 Document revision

- 171009 *Caution* added to chapter *TR memories handling*. Example added to chapter *Write data to EEEPROM*. A typo bug in the `.TRCNFG` file name fixed.
- 170823 Updated for IQRF OS v4.02D.
- 170810 For TR-7xD and IQRF OS v4.01 or higher. TR Upload added.
- 160715 A bug (`Nmax`) in the table on page 4 fixed.
- 151106 Min. delay between SPI bytes changed for networking RF communication.
- 150608 First release for (DC)TR-7xD

8 Sales and Service

8.1 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

8.2 Technology and development

www.iqrf.org

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

8.3 Partners and distribution

www.iqrf.org/partners

8.4 Quality management

ISO 9001 : 2009 certified

8.5 Trademarks

The IQRF name and logo are registered trademarks of IQRF Tech s.r.o.

PIC, SPI, Microchip and all other trademarks mentioned herein are property of their respective owners.

8.6 Legal

On-line support: support@iqrf.org
