IQRF OS

Operating System

version 2.08

User's Manual

Simple way to smarter wireless solutions

Content

Compatibility........3
OS Principles........4
IQRF OS Architecture........5
Microcontroller........6
Memories........7
  Program memory (Flash)........7
  Data memory (RAM)........7
  Data memory (EEPROM)........8
Control........8
  Operation modes........8
  Real time........8
  Communication control, checking and timeouts........9
  Watchdog........9
  Sleep........9
  Other PIC peripherals........10
  Reset........10
  Temperature measurement and battery check........10
  Debug........10
RF........10
  RF communication........10
  RF networking........10
SPI........11
Services and Functions........13
  RAM access........13
  EEPROM access........13
  Delays........13
  LED functions........14
  Control........14
  Temperature measurement........14
  Power supply measurement........14
  Buffer support........15
  Information services........16
    Module data:........16
    Application data:........16
  RF communication........17
  RF networking........17
  SPI........17
List of OS functions........18
Documentation and Information........19
Sales and Service........20

Compatibility

Document history:

- 081125 First release
- 090501 `_SPIbusy` and `_SPIwrite` flags not documented. Use `getStatusSPI()` instead of `_SPIbusy`.

Operating system version 2.08 is intended for the following IQRF transceiver modules:

- **TR–xxx–11A**
- **TR–xxx–21A v1.02 and v1.03**
- **TR–xxx–31B**

The modules are delivered with IQRF OS allowing realization of common networking device (Node) as well as network Coordinator (software selectable).

IQRF OS versions and history:

Version	Main differences	Release	Status
v1.14	previous generation	Jul 2007	not for new designs
v2.00	• Much more **effective**, **easier** to use, higher **performance** • **Networking** totally reworked. Extended capability. **Complete IQMESH**. • **SPI on background** • **Encoded** network communication • Indirect RAM access • Temperature measurement supported by OS • Supports user application debugging directly by IQRF OS • Many other improvements • **IDE** – complete **development environment** with all SW tools integrated including effective debug tools	Jan 2008	not for new designs
v2.01	• function **wipeBondNR**() added • function **batteryValueOK**() added	Mar 2008	not for new designs
v2.02	• minor SPI bug fixed	May 2008	not for new designs
v2.03	• Improvements: • BufferCOM size increased from 35B to 41B • Number of nodes in one network increased from 128 to 239 • Minor bug in routing fixed	July 2008	not for new designs
v2.04	• **setNetworkFilteringOn**() switches just packet from active network (1/2), non-networking communication ignored • **Wake-up on pin change** under user's control. Default disabled. To eneble, set RBIE = 1 before iqrfSleep(). Not compatible with previous versions (permanently enabled in Sleep up to v2.03).	July 2008	internal release only
v2.05	• higher RF noise immunity • corrected transfer of MPRWx while not routing • several minor bugs not affecting module functionality corrected	Aug 2008	not for new designs
v2.06	• minor change in routing	Aug 2008	not for new designs
v2.07	• bug in the **setLoggingOff**() function fixed • **Wake-up on pin change** improved. To utilize it, the sequence GIE = 0; RBIE = 1; is required just before iqrfSleep().	Sep 2008	not for new designs
v2.08	• broadcast message support added	Oct 2008	current
	• implemented in TR-868-31B modules	Nov 2008	

IQRF transceiver modules allow **upgrade** to current OS version. This service must be done by the manufacturer.

OS Principles

The IQRF system is designed to allow using of RF wireless connection according to user's needs. Transceiver modules contain a microcontroller for controlling the transceiver operations and for executing of user defined functioning. The microcontroller is equipped with a firmware – all required functions are programmed in advance by the manufacturer. The set of such functions is called **operating system** (OS).

OS helps with development of IQRF applications very effectively. It offers software functions prepared in advance for all common user requirements. Thus, it is not necessary to create the whole user program by oneself (using microcontroller instructions and C commands only) but the user adds a user part of software to the OS only. Moreover even this user part can be very simple thanks to the OS. The user works on much higher level of OS functions. He need not solve partial issues, e.g. how to read a data from equipment connected in a standard way and send it via the RF. He can fully concentrate to implementation his own demands.

Some of them need even not to be programmed with executive commands and OS functions but with writing requested values for appropriate OS parameters in source code only.

The user application so called „runs under the operating system" which means that this is invoked from OS, uses OS functions and is (should be) under OS control.

There are two types of OS functions:

- functions intended for using in application programs – block memory copying, writing of a byte or a block to EEPROM, sending/receiving block data via selected interface, starting/stopping of LED blinking, ...
- supportive functions, i.e. tools which support creation and development of the application but do not directly participate in user program, e.g. functions for debugging and uploading the program into the microcontroller.

Operation not supported with OS can be realized by functions located in user program memory area. Typical solutions can be downloaded from the IQRF support site [1]. Specific function can even be written by the user oneself.

OS functions need not run sequentially (next function invoked not until the preceding one is finished) but some operations can run so called "on background" (the function arranges execution of requested operations which run independently and immediately returns the control back to the superior program). In this way more processes can run "simultaneously". Then the program structure is that besides of execution running sequentially "on foreground" up to several tasks on background can be running. IQRF OS allows to run even very complex operation including complete communication protocols on background. This makes real-time programming really easy.

IQRF OS supports communications:

- **RF** (radio), including networks – in **point-to-(multi)point** and **IQMESH** topologies.
- Standard serial **SPI** (slave mode) protocol for connection to peripherals or PC.

Other communications can be realized with a user program (I^2C, UART, ...).

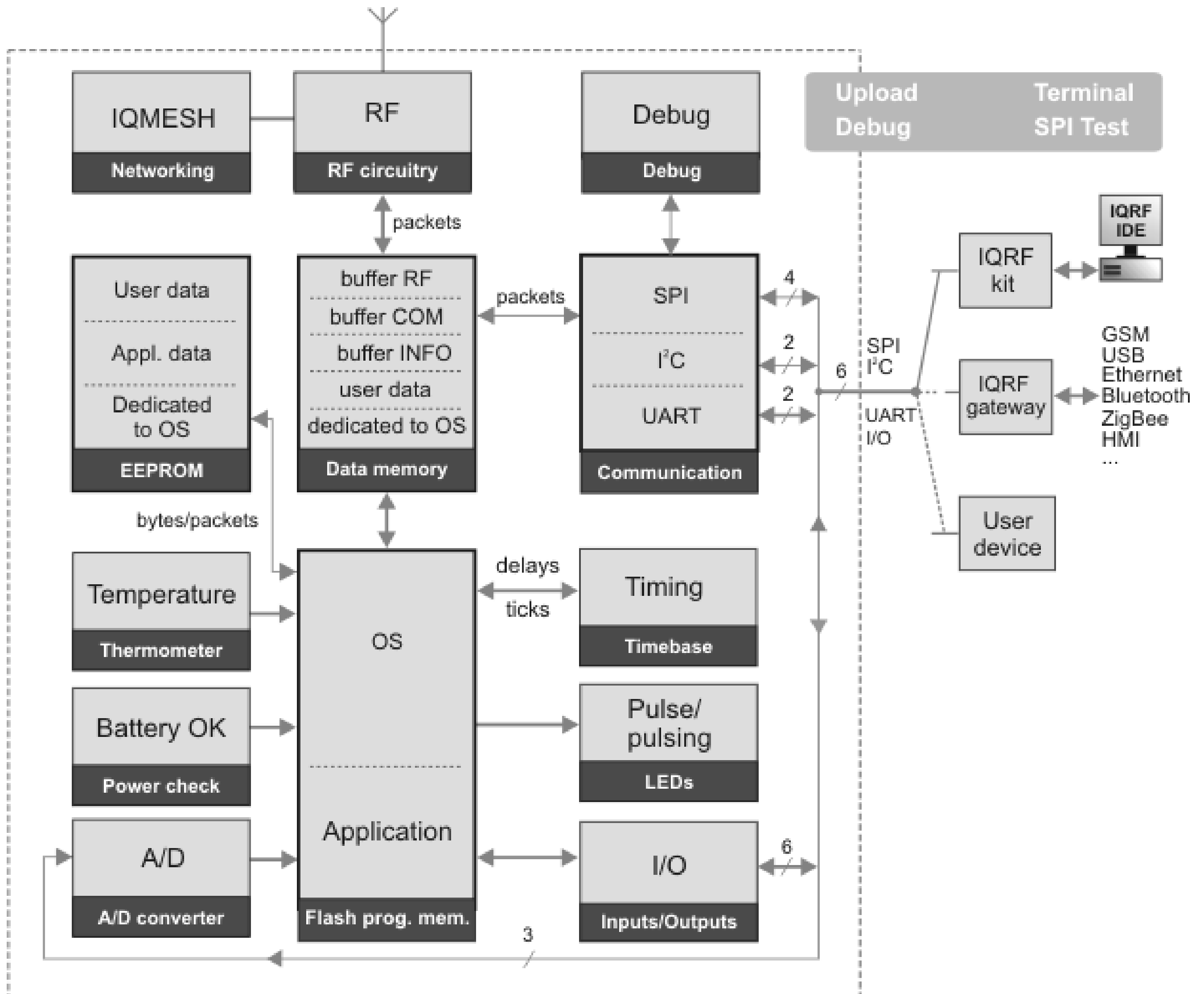
Complex standard communication protocols (**USB**, **Ethernet**, **Bluetooth**, **GSM**, **Zigbee**,...) can be realized using IQRF gateways or development kits.

OS supports low power consumption of IQRF transceivers with the Sleep function when operation of TR module is reduced/stopped until setup time is elapsed or selected inputs are changed.

To increase the reliability the watchdog function is used. This is implemented in microcontroller hardware and controlled via user program – see chapter Control.

IQRF OS Architecture

Hardware of the transceiver module with a microcontroller including the IQRF OS results in architectural model:



Individual blocks are:

- Memories:
  - **program memory** (Flash)
  - **data memory** (RAM)
  - data memory (**EEPROM**)
- Communication interface:
  - **RF** (wireless)
  - **SPI** (standard serial)
- **Temperature** sensor
- Power supply check
- Digital **I/O** (input or input/output). 4 pins are shared with the SPI.
- A/D converter
- **Real time** support: 10 ms interval (tick) generator on background and supporting functions
- IQMESH networking
- Debug: OS support for testing and debugging

Resources partially depend on transceiver module type.

Microcontroller

The IQRF OS version 2.08 is intended for transceivers with the **PIC16LF88** and **PIC16LF88** microcontrollers (8-bit single chip processor by Microchip) – datasheets see [2].

Hardware resources of PIC16LF88 and their utilization in TR-868-11A, 21A and 31B modules with OS:

PIC HW resources		Utilization
Program memory	Flash	512 instructions (TR-xxx-11A/21A) 1024 instructions (TR-xxx-31B)
Data memory	RAM	98 B – user data 140B – communication buffers
Data memory	EEPROM	Node: 160B (+32B application data)
		Coordinator: 0B (+32B application data)
A/D converter (10b)		For temperature measurement, battery check and external analog inputs
HW serial communication	SPI (slave)	supported with OS on background
	I^2C (slave)	realized with user function
	UART	realized with user function
Interrupt		not user available
Brown-out reset		disabled (TR-xxx-11A/21A), selectable (TR-xxx-31B)
Power-up timer		used
Watchdog		timeout 1 ms – 268s (user selectable and programmable)
Oscillator		internal RC, up to 8 MHz (500 ns instruction), user programmable

PIC pins and other details see datasheets of the transceiver modules [3], [4] and [12].

These resources can be under OS supervision and the user should access them in accordance with this manual and possible requirements resulting from hardware construction of the module and OS implementation.

Configuration changes and direct access to some resources by the user are limited or not allowed at all. Serviceability of some resources depends on using some other ones at the same time (some hardware communication modules, pins and memory areas are shared for more functions).

Parts of memories are dedicated to PIC core, peripherals and operating system. Direct access to the EEPROM is not allowed at all, extra OS functions are intended for this. Flash memory is user accessible for uploading the program to the microcontroller using the procedure specified by the manufacturer only. Not dedicated user inputs/outputs, peripherals (e.g. I^2C and UART) and RAM locations can be accessed directly according to user's needs.

Details see datasheets of the transceiver modules [3], [4], [12], PIC16LF88(6) datasheets [2], RAM map [5] and EEPROM map [6]. In doubt, refer to IQRF support by the manufacturer [1].

Memories

For memory purposes the IQRF OS version 2.08 uses internal memories of the microcontroller only.

Individual parts of memories are:
- Dedicated to the microcontroller
- Dedicated to the OS
- Other areas are available for the user

Memories can be under OS supervision and the user should access them in accordance with this manual and possible requirements resulting from hardware construction of the module and OS implementation.

Illegal modification of dedicated memory locations can cause system crash.

There are several header files (with the h extension) delivered with the IQRF system. They are intended for C compiler to provide easy and seamless linking the OS with the user program. Of course, these text files could serve to user's survey concerning memories – but the user should nowise modify them. (The 16F88.h and 16F886.h are based on standard files made by the C compiler manufacturer that is why they contain some irrelevant information to spare.) User's own definitions should be placed to extra user header files.

Refer to RAM map [5] and EEPROM map [6].

Program memory (Flash)

The user can use this as a program memory only. The program remains stored even after power off. Overwriting in not unlimited, number of erase/write cycles is about 100 000.

User program can be uploaded into the TR module using appropriate IQRF development kit, e.g. CK-USB-03 or CK-USB-02 [7] and IQRF IDE servicing program. Codes in standard HEX format or scrambled codes in the IQRF format can be uploaded.

- OS occupies the memory from 0x000 to 0xDFF (TR-xxx-11A and 21A) or to 0x1BFF (TR-xxx-31B)
- Remaining area 0xE00h – 0xFFF (512 machine instructions) for TR-xxx-11A and 21A or 0x1C00h – 0x1FFF (1024 machine instructions) for TR-xxx-31B is available for user program .

User program should begin from address 0xE00 (TR-xxx-11A and 21A) or 0x1C00 (TR-xxx-31B). It is automatically arranged by IQRF header files.

Data memory (RAM)

RAM is a fast memory accessible in a comfortable way. Data is fully under supervision of running program and is lost after power off.

Individual RAM parts:

- dedicated to the **microcontroller** and its **peripherals (**PIC special function registers – SFRs). Direct using is mostly restricted, e.g. the application need not use registers INDF, EECON1, EECON2, EEADR, PIR2, PIE1 and PIE2. PIR1 is partially accessible via special OS function only. In doubt, refer to IQRF support by the manufacturer [1].
- dedicated to **OS**:
  - OS utilizes three basic memory buffers:
    - **buffer RF** (0x110 – 0x14F), 64B – for RF communication
    - **buffer COM** (0xA0 – 0xC8), 41B – for serial communication (especially SPI)
    - **buffer INFO** (0x20 – 0x42), 35B – for OS and user block operations

    Communication buffers are especially intended for transferred data but can be used according user's need in specific cases as well.
  - **buffer networkInfo** (22B) is an area dedicated to network applications.
  - system variables. Only the toutRF register should be directly accessed by the user.
  - OS work variables (not documented, the user need not modify them).
- Remaining area (0x190 – 0x1F1) is **available for the user** (98B). It is the **complete RAM bank 3** excluding the shared area (0x1F0 – 0x1FF) where only two more `userRegx` registers (0x1F0 and 0x1F1) are available. Selection of bank 3 is automatically arranged by the IQRF header files. Refer to the PIC datasheets [2] for information about RAM banking.

Data memory (EEPROM)

EEPROM data remains valid even after power off. Overwriting is not unlimited, number of erase/write cycles is 100 000 min., (typically 1 000 000). EEPROM is especially intended for configuration parameters and data.

Individual EEPROM parts:

- **User data:** 160B from 0x00 to 0x9F (Nodes only). This area is not user available for Coordinators.
- **Application data:** 32 B from 0xA0 to 0xBF (Nodes as well as Coordinators). The user can use this area for his particular needs (especially intended for configuration and similar purposes). It is accessible for reading even via the appINFO() function in a comfortable way. The factory settings string is: `"Hello everybody. IQRF is here! "`
- **Dedicated to OS:** (Node as well as Coordinator)
  Remaining EEPROM area (0xC0 – 0xFF) is dedicated to OS. It is not accessible by the user.

EEPROM access:

- values can be specified in application (source) program to be written to EEPROM while the program is uploaded into the microcontroller. EEPROM address range is 0x2100-0x21FF instead of 0x00-0xFF when using `cdata` and similar C statements (e.g. __EEAPPINFO = 0x21A0).
- The microcontroller can read/write data from/to EEPROM under user program control while the application is running using general OS functions for accessing EEPROM (eeWriteByte, ... – see below). Short addresses (0x00-0xFF) are used in this case.

The user should avoid exceeding the number of erase/write cycles allowed. Note that even some other OS functions (bond, bondRequest, ...) write to EEPROM as well.

Control

Operation modes

The TR modules can work in three modes:

- **programming:** the user program can be uploaded to the TR (including EEPROM content) in this mode. It can be invoked with IQRF IDE or with the button in the programming kit.
- **run:** the TR module executes operation required by the user
- **debug:** execution is stopped and data can be downloaded from the microcontroller and displayed with the IQRF IDE development environment.

- Programming mode can be entered after connection of TR to appropriate IQRF programming kit and invoking the Upload routine of the IQRF IDE [9]. It is indicated with LED blinking for 100 ms with about 2 s period.
- The module is automatically switched to Run mode just after upload finishing. This is not indicated at all, possible indication can be done by the user application.
- Debug mode is fully under control of user program and interactive handling and indication with IQRF IDE.

Real time

OS provides an efficient support for real time applications. It has a generator of time intervals running on background and appropriate functions. Basic interval (elementary OS time interval for timing on background – a “tick”) is 10 ms. Using number of ticks times of appropriate processes (delay, LED blinking, communication timeout check, ...) can be specified, the timebase can be realized, ...

- Capture is another efficient timing tool. It is an independent resettable timer (16-bit counter of ticks) freely running on background. It is suitable especially for working with long periods (up to 655 s).
- OS provides functions even for waiting on foreground.
- Short time intervals for timing on foreground can be derived even from instruction timing. The PIC16LF88(6) is clocked with internal RC oscillator with 8 MHz frequency. Thus, instruction cycle is 500 ns (1 µs for some instructions – see PIC16LF88(6) datasheets [2]).

Note that time precision of TR modules depends on precision of internal RC oscillator – see PIC16LF88(6) datasheets [2]. Microcontrollers are individually calibrated by the manufacturer but despite of this fact the precision and stability are less than for a crystal oscillators. The precision is sufficient for asynchronous communication (UART) with reasonable speed, for clock and calendar functions another suitable method should be used.

Tip: If there is a gateway in the network, it can be used even as a timebase with crystal precision. Time information can be distributed via RF.

Communication control, checking and timeouts

Good programming practice requires to have supervision under times for communication establishing and data transfers. It is convenient for synchronization with the transmitter, securing the program against transmitter failures, etc.

The OS supports the following timeouts and checking:

- **while receiving:** via functions for waiting on background. It is checked whether the requested operation passed during the user setup time. Otherwise attempts for receiving are terminated and receiving function returns control to superordinate program. OS checks the following times:
  - RF packet receiving
  - starting receiving via SPI
  - receiving next word via SPI (i.e. after successfully established communication)
- **while transmitting:** control (e.g. repeated packet transmitting with number of repetitions specified in advance) is fully under user's handling.
- **while receiving and transmitting:** RF and SPI communication is ensured with check „sums“ CRC. Complete SPI packet (not only „significant data“) is ensured with a single byte CRC, the RF packet with more various CRCs.
- Even higher reliability can be achieved with additional **user verification**.

Refer to SPI specification [10] for details.

Watchdog

To increase the reliability, the OS uses the hardware watchdog of the microcontroller. It is a **continuously running** independent timer with a programmable overflow period. It should be used that never overflow during the correct operation. It is accomplished via the `clrwdt()` instruction always executed in time, i.e. before the watchdog overflows. (This function is implemented not in OS but it is the PIC machine instruction supported with the compiler). If an overflow occurs it is regarded as a program execution failure (power supply failure, error in algorithm in application program e.g. after illegal data receiving and so on) and the microcontroller responds with reset. If the failure is not a permanent one, it can lead to system recovering. The watchdog runs even in the Sleep mode (see below). Overflow in Sleep results in wake-up but not the PIC reset.

The watchdog can be enabled/disabled in SW. Overflow period is user selectable (even while the program is running) from 1 ms to 268 ms. Setup registers are WDTCON (WDTPSx and SWDTEN bits) and OPTION (PSA, PS0, PS1 and PS2 bits, remaining bits must be left unchanged by the user) – see PIC16LF88(6) datasheets [2]. Default timeout period is about 4 s.

Sleep

Complete TR module (including the RF circuitry, microcontroller and temperature sensor) can be set in the Standby (Sleep) mode. In this case almost no operation is executed but the power consumption is minimized.

Transition to the Sleep mode:

The Sleep mode is initiated in software using the `iqrfSleep()` function in appropriate location in the source program. Then all TR hardware resources controlled by the OS are automatically suspended: activity of the TR module including the RF circuitry, temperature sensor, microcontroller as well as its peripherals (stopping of timers, disconnecting of internal pull-ups, ...).

Before switching to the Sleep mode:

- Power consumption should be minimized even for hardware resources of TR controlled by the user (PIC pins, possible PIC internal peripherals) and possible external peripherals connected. It must be done in user program. See the TR [3], [4], [12] and PIC [2] datasheets.
- The microcontroller should be configured for for subsequent wake-up if required on pin change:
  - Wake-up on pin change is under user's control, default disabled.
  - To eneble, the sequence `GIE = 0; RBIE = 1;` is required just before iqrfSleep().
  - This is not compatible with previous IQRF OS versions (wake-up on pin change was permanently enabled in Sleep up to v2.03).

Returning to the operating mode (wake-up)**:**

- after watchdog overflow (non-maskable)
- after pin change on some pins (depending on the TR type), when configured as inputs (if enabled)
- after power-off/on

Tip: all wake-up types can be distinguished via the –TO and –PD status flags – see PIC16LF88(6) datasheets [2].

After the wake-up the microcontroller continues with execution the command following the Sleep function.

The user can use Sleep and wake-up without any restriction due to OS, all related microcontroller possibilities can be employed – see PIC16LF88(6) datasheets [2].

Tip: sleep period can be setup via the watchdog timeout period.

Typical sleep power consumption ~2.5 µA can be reached with all peripherals off – see the TR-xxx-31B datasheet [4].

Other PIC peripherals

There are PIC HW resources (I^2C, UART) not supported with the OS but accessible directly via appropriate PIC registers. To allow using them, the specialized function is available to set the appropriate flags in the PIR1 control register. Refer to the datasheet of the microcontroller [2].

Reset

If needed, it is possible to run the application program from the very beginning again. It can be accomplished via the reset() function used in the user program in the given location. It is a real microcontroller reset (the WDT type – see PIC16LF88(6) datasheets [2]), automatically followed with repeated OS and user program initialization.

Temperature measurement and battery check

The IQRF platform supports temperature measurement and power supply check using internal A/D converter of the microcontroller.

Debug

The IQRF platform provides user with an efficient debugging tool.

To enjoy its powerful capabilities, the following configuration should be used: The transceiver module plugged into the CK-USB-02(03) development kit connected to PC via USB with the IQRF IDE development environment [9].

Debug is directly supported by the OS with the *`debug()`* function. This can be included in user program wherever you need to stop program executing and evaluate variables, EEPROM content or RAM registers. After uploading user program into the transceiver module the application is running until the *`debug()`* function is encountered. Then the program stops, the module is switched to the debug mode and data can be downloaded and displayed on the screen.

The module stays in debug mode till the user wishes. Then the application program can continue execution until another *`debug()`* function is encountered and so on. To distinguish individual debug breakpoints the `W` register can be used. See IQRF IDE Help and E06–RAM example [11] for details.

RF

RF communication

OS functions allow powerful and user-friendly control of RF communication. From the user's point of view it is about working with memory only (operations with buffer RF, R/W, ...). IQRF OS automatically provides all needed services:

- at transmission level: HW setup, coding for transmittion, timeouts, ...
  - at packet level: preamble, consistency checking, coding, ...
- at network level: routing, including information about the network and device, filtering, ...

Related memory locations and registers:

uns8	*bufferRF[64]*	*buffer for RF routines, 64B long*
uns8	*DLEN*	*packet length (specify before transmitting, automatically set after receiving)*
uns8	*toutRF*	*timeout for packet receiving (in ticks). Default value is 50 (500 ms).*

RF networking

Supported modes:

- **non-networking**: Two or more devices at the same level, without a network Coordinator. Packets are available for all devices in range and managed by the user program.
- **networking:** Network topology with one Coordinator and number of end devices (Nodes). IQMESH packet transmittion is supported with a lot of additional sophisticated features. Any IQRF TR module or gateway can even work in two independent networks. The communication is possible even between nodes out of RF range each other – using “hops” via other nodes in range (routing). All network communication is coded.

Registers used in networking mode:

uns8	*PIN*	*topology information*
uns8	*RX*	*address of the node the packet is intended to*
uns8	*TX*	*address of transmitting sender (automatically set by `RFTXpacket`())*

SPI

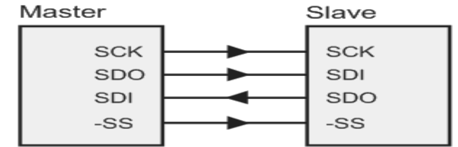
IQRF transceiver modules can communicate with external peripherals via the SPI interface.

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	TR pin	function	
SCK	C6	Serial Clock	issued by master
SDI	C7	Serial Data In	
SDO	C8	Serial Data Out	
–SS	C5	Slave Select	issued by master, active low

The SPI bus with a single slave:



The IQRF transceiver modules can communicate as SPI slaves. Full as well as half duplex is supported. The SPI protocol is implemented in IQRF operating system. Thanks to the state machine architecture the communication is fully synchronous without any timeouts. It is packet oriented and works on OS background. Packets consist of selectable number of bytes (0 to N_{max}). In time constrained cases the communication can be slowed down to work on OS foreground with longer delays between individual bytes. Data stream can even be suspended at all and continue at any time later.

Packet structure:

The master can send two types of packets with the following structure:

Master checks the SPI status of the module:

Master	**SPI_CHECK**
Slave	SPISTAT

Master reads/writes a packet from/to the module:

Master	**SPI_CMD**	PTYPE	DM_1	DM_2	---	$DM_{SPIDLEN}$	CRCM
Slave	SPISTAT	SPISTAT	DS_1	DS_2	---	$DS_{SPIDLEN}$	CRCS

Where:

```
SPI_CHECK = 0x00
SPI_CMD   = 0xF0
```

`SPISTAT`: SPI status of the module

hex value	SPI status
00	SPI not working (disabled by the disableSPI() command)
07	SPI suspended by the stopSPI() command
3F	SPI not ready (buffer full, last CRCM O.K.)
3E	SPI not ready (buffer full, last CRCM error)
40 to 63	SPI data ready. Value - 0x40 = number of bytes to be sent from the slave (1 to N_{max})
80	SPI ready (communication mode)
81	SPI ready (programming mode)
82	SPI ready (debugging mode)
83	SPI not working on background (e.g. during receiving of RF packet) – Slow mode Master should prolong the delay between individual bytes when this status is received – see the T2 parameter in the table above.
FF	SPI not working (HW error)

© 2009 MICRORISC s.r.o.

SPI status of the module is indicated by the IQRF IDE when used together with related IQRF development tools, e.g. CK-USB-02(03):

PTYPE:

b7	b6	b5	b4	b3	b2	b1	b0
CTYPE		SPIDLEN					

CTYPE: communication type
10: full duplex (the master reads/writes from/to the module, bufferCOM changed)
00: half duplex (the master reads from the module, bufferCOM unchanged)

SPIDLEN: data length (from 1 to N_{max})
TR-xxx-xxA: $N_{max} = 35$
TR-xxx-xxB: $N_{max} = 35$

DM: data from the master

DS: data from the slave

$\mathrm{CRCM = SPI_CMD\ xor\ PTYPE\ xor\ DM_1\ xor\ DM_2\ \ ...\ xor\ DM_{SPIDLEN}\ xor\ 0x5F}$

$\mathrm{CRCS = PTYPE\ xor\ DS_1\ xor\ DS_2\ \ ...\ xor\ DS_{SPIDLEN}\ xor\ 0x5F}$

IQRF OS functions related to SPI:

*void **enableSPI**()*	*Enables SPI communication. Hardware of the microcontroller is configured for SPI, related pins can not be used as general I/Os.*
*void **disableSPI**()*	*Disables SPI communication. SPI HW module of the microcontroller is switched off, related pins are general I/Os.*
*void **startSPI**(length)*	*Starts SPI packet transmission. This operates on OS background except of the Slow mode case.*
*void **stopSPI**()*	*Suspends SPI communication. It can continue at any time later (after another **startSPI** command)*
*bit **getStatusSPI**()*	*Updates SPI flags and packet length and checks whether SPI is busy*

SPI flags: available in **param2**

bit 7			bit 4	bit 3	bit 2		bit 0
–	–	–	_SPICRCok	_SPIRX	–	–	–

_SPIRX Something received on SPI.
_SPICRCok The last received SPI CRCM was O.K.

SPI data length: available in param1

Caution: SPI data length and SPI flags are copied to param1 and param2 (parameters used by OS functions) after the getStatusSPI command. They are only valid until another OS command using these parameters is called.

Refer to IQRF SPI Specification [10] for detailed information (especially concerning timing and requirements for SPI master).

See [11] – Example E07–SPI.

Services and Functions

Values between system functions and superordinate program are passed on via parameters. OS uses 4 parameters in total: `param1` (1 B), `param2` (1 B), `param3` (2 B) and `param4` (2 B). Their location in the memory see the RAM map [5]. Individual functions have up to 3 parameters. Some functions use some of these params to return output values. Note that they are valid until another function using the same parameter or the `debug()` function is called by the user.

RAM access

RAM can be accessed either directly (using common C commands) or indirectly. But indirect access using the FSR register is not allowed. Due to security reasons all instructions using FSR are removed during Upload. To avoid unintended behavior all constructions using addressing via FSR should be omitted. Instead of this IQRF provides complete support for indirect (byte oriented) addressing using extra system functions. Before using them, the IRP bank select bit must correspond to the location of the register (IRP=0 for RAM bank 0 and 1, IRP=1 for RAM bank 2 and 3). See the PIC datasheet [2] for details.

`uns8` **`readFromRAM`**`(uns8 adr)`	reads a byte from RAM address `adr`
`void` ***`writeToRAM`*** *`(uns8 adr, value)`*	*writes a byte stored in value to address* `adr`

Examples:

`X = readFromRAM(bufferCOM + 10);`	*one byte from* `bufferCOM` *stored to* `X`
`for (pomi=0;pomi<40;pomi++)`	
` writeToRAM(bufferRF+pomi,pomi);`	*writes 40 bytes to* `bufferRF`

See [11] – Example E06-RAM.

EEPROM access

Direct user EEPROM access (using registers `EECONx`,...) are not allowed, there are extra system functions for read and write. Either 8b word (Byte) or block of 8b words (Data) with specified length can be read/written. For block operation (`eeReadData`/`eeWriteData`) the *`bufferINFO`* array is used.

`uns8` **`eeReadByte`**`(uns8 adr)`	*reads a byte from EEPROM address* `adr`
`void` ***`eeReadData`*** *`(uns8 adr, length)`*	*reads* `length` *bytes from EEPROM address* `adr` *to* `bufferINFO`
`void` ***`eeWriteByte`*** *`(uns8 adr, value)`*	*writes a byte stored in value to EEPROM address* `adr`
`void` ***`eeWriteData`*** *`(uns8 adr, length)`*	*writes* `length` *bytes from* `bufferINFO` *to EEPROM address* `adr`

Examples:

`pomi = eeReadByte(0);`	*// stores 1 byte from address 0 to* `pomi`
`eeReadData(10, 16);`	*// stores 16B from address 10 to* `bufferINFO`
`eeWriteByte(0x10, 55);`	*// writes decimal value 55 to address 0x10*
`eeWriteData(to,10);`	*// writes 10 bytes from* `bufferINFO` *to EEPROM*
	// starting address stored in register `to`

See [11] – Example E04-EEPROM.

Delays

There are several OS functions to simplify control of the application. Waiting on foreground and delays on background are supported.

On foreground:

`void` ***`waitMS`*** *`(ms)`*	*waits specified number of ms*
`void` ***`waitDelay`*** *`(ticks)`*	*waits specified number of ticks*

On background:

`void` ***`startDelay`*** *`(ticks)`*	*starts pause (in ticks) and returns control immediately*
`bit` ***`isDelay`*** *`()`*	`isDelay` *() == 1 if the delay is still in progress*

Moreover, a resettable timer counting ticks on background (Capture) can be used:

`startCapture` *`()`*	*resets counter of ticks*
`captureTicks` *`()`*	*number of ticks (16b) from last startCapture copied to* `param3`
	number of ticks (16b) from last captureTicks copied to `param4`

Tip: *the* `clrwdt()` *instruction should be placed before/after pauses to avoid unintentional watchdog reset.*

See [11] – Example E05-DELAYS.

LED functions

The on-board red LED can be directly controlled on OS foreground using C commands for manipulating the _OUT2 output which is a pin the LED is connected to. The pin must be configured as an output (TRISB.7 = 0) in this case.

Example:

```
_OUT2 = 1;                    // LED on
```

In addition to this, there is a powerful set of functions for efficient manipulation with the LED (including pulsing) on background. Timing is based on ticks ~ 10 ms.

Every pulse consists of ON and OFF intervals. Default: 50 ms (ON) and 200 ms (OFF). Both can be dynamically redefined:

```
void setOnPulsingOUT2(onTime)       set On time (in ticks)
void setOffPulsingOUT2(offTime)     set Off time (in ticks)
```

Once below functions are called requested activity on output OUT2 are initiated and served on background. Functions immediately return control to the application. Background activity will override setting of OUT2 made by the application. Thus, attention should be taken when using direct setup of LED output pin combined with LED functions. Possible background LED routines can overwrite the status set by direct manipulation with the pin.

```
void pulsingOUT2()     starts pulsing on OUT2 output
void pulseOUT2()       one pulse on OUT2 output
void stopOUT2()        stop activity on OUT2 output
```

Examples:

```
pulseOUT2();           // LED blinks once
pulsingOUT2();         // LED starts blinking
stopOUT2();            // stops background activity on OUT2, LED off
```

Control

IQRF OS provides easy and efficient power management. The following function, once called, puts the module into the Sleep mode:

```
void iqrfSleep()       puts the module into power saving mode
```

If needed, it is possible to initialize the application and run the program from the very beginning again:

```
void reset()           forced reset
```

To allow access to some PIC internal peripherals (I^2C, UART), the specialized function is available to set the appropriate flags in the PIR1 control register. See PIC datasheets [2].

```
void setPIR1(value)    set PIR1 value
```

IQRF OS directly supports debugging and testing:

```
void debug()           switch to the debug mode
```

Temperature measurement

Temperature can be measured with on-board temperature sensor and internal A/D converter of the microcontroller.

```
void getTemperature()  sensor output stored to ADRES (A/D destination register, 10b)
```

Temperature (in °C) evaluation: `Ta = ADRES × 75/256 – 50`

See [11] – Example E08–TEMPERATURE and [5] – RAM map.

Power supply measurement

Power supply is measured with internal A/D converter of the microcontroller. Supply voltage depending on the battery condition is used as a reference while the measured input voltage is stable (LED voltage drop).

The A/D converter should be controlled via the user program. OS supports battery check with the function for comparison of the measured value with the constant predefined in EEPROM.

```
uns8 batteryValueOK()  get a value for battery check from EEPROM
```

See [11] – Example E10–BATTERY.

Buffer support

All communication (RF and SPI) of IQRF modules is packet oriented therefore buffer servicing is supported. There are three buffers primarily dedicated to block operation: **bufferRF**, **bufferCOM** and **bufferINFO** (see chapter Data memory and RAM map [5]).

The following routines are intended to provide data transfers between buffers:

`void copyBufferINFO2COM()`	copies `bufferINFO` to `bufferCOM`
`void copyBufferINFO2RF()`	copies `bufferINFO` to `bufferRF`
`void copyBufferRF2COM()`	copies `bufferRF` to `bufferCOM`
`void copyBufferRF2INFO()`	copies `bufferRF` to `bufferINFO`
`void copyBufferCOM2RF()`	copies `bufferCOM` to `bufferRF`
`void copyBufferCOM2INFO()`	copies `bufferCOM` to `bufferINFO`
`void copyMemoryBlock (uns16 from, uns16 to, uns8 length)`	copies `length` bytes from address `from` to address `to` (through the whole memory area)

Following routines provides even more flexibility under buffers:

`void clearBufferINFO()`	clears (zeroes) `bufferINFO`
`bit compareBufferINFO2RF(uns8 length)`	compares `length` bytes of `bufferINFO` and `bufferRF` `return value == 1 if equal`

Examples:

```
clearBufferINFO();                 // clearing the lower half of bufferRF
copyBufferINFO2RF();

if compareINFO2RF(10)              // compare 10B data strings
   myok = 1;

copyMemoryBlock(0x153, 0x1D3,7);   // copy 7B block from address 153h to address 1D3h
```

See [11] – Example E06–RAM.

Information services

Module data:

Every IQRF module contains information about itself. This is accessible via copying to the `bufferInfo` in the following format:

address in `bufferInfo`	7	6	5	4	3	2	1	0
meaning	OS build		PIC type	OS version	Coordinator / Node	serial number		
					Module ID			

Coordinator / Node: reserved for future OS versions. Coordinator / Node is SW selectable in IQRF OS v2.02.

0: Node
1: Coordinator

OS version:

upper nibble (4 b): major version
lower nibble (4 b): minor version

PIC type:

2: PIC16LF88
3: PIC16LF886

OS build: for the manufacturer only

Example (all in hexadecimal):

```
                    [0]  [1]  [2]  [3]  [4]  [5]  [6]  [7]
bufferInfo[0-7] =   1C   10   00   01   21   02   30   03
```

Meaning: Coordinator, Module ID = 0100101C, IQRF OS version 2.01, PIC16LF88, build # 0x0330.

Module ID is displayed with the IQRF IDE development environment.

.

*void **moduleInfo**()* — *Module data to bufferINFO*

Application data:

There is a 32B block in EEPROM (area 0xA0 – 0xBF) dedicated to the user application. It is possible to read data from it just to the `bufferINFO` very effectively with a single instruction only. This area is intended for arbitrary information concerning user application but is especially useful for repeatedly employed (often permanent) data such an identification information to be compared after receiving (with the the `compareBufferINFO2RF` function). This EEPROM area is called the **Application data**.

*void **appInfo**()* — *Application data to bufferINFO*

Examples:

```
appInfo();                          // application data is transferred to bufferINFO
copyBufferINFO2RF();                // … and finally copied to bufferRF

#pragma packedCdataStrings 0        // store application data to EEPROM after compilation
#pragma cdata[__EEAPPINFO] = "Application data, I'm user #01   "

appInfo();                          // dynamic change of application data
bufferINFO[29] = '2';
eeWriteData(__EEAPPINFO+29,1);
```

Refer to memory maps [5] and [6] as well.

RF communication

RF Communication (Coordinator as well as Node):

void ***setTXpower****(level)*	*RF output power settings: 1 (min.) – 7 (max.)*
void ***RFTXpacket****()*	*sends the packet stored in* `bufferRF`
bit ***RFRXpacket****()*	*receives the packet to* `bufferRF`

It is not needed to synchronize packet lengths of transmitter and receiver. The only thing which has to be done by the user is to specify the packet length (`DLEN = ...`) and PIN (if bidirectional communication) before transmitting. The transmitter puts this information to the packet, the receiver reads this information and setup respective length. That is why the original `DLEN` can be overwritten after every receipt and should be updated before transmitting.

The `RFRXpacket()` function returns control to application after successful packet receiving or after the timeout. The user has full control on timing as the timeout can be set in ticks (~10ms) prior to the `RFRXpacket()` function call (`toutRF = ...`).

See [11] – Examples E01–TX, E02–RX, E03–TR and E09–LINK.

RF networking

Bonding (Node only):

bit ***bondRequest****()*	*request for bonding*
void ***removeBond****()*	*request for unbonding (node number remains still reserved)*
bit ***amIBonded****()*	*is the Node bonded?*
void ***wipeBondNR****()*	*reserved node number cancellation (after bond removing)*

Bonding (Coordinator only):

bit ***bondNewNode****()*	*look for bond requesting devices and bonding*
bit ***isBondedNode****(n)*	*is the Node #n bonded?*
bit ***removeBondedNode****(n)*	*unbonding the Node*
bit ***rebondNode****(n)*	*rebonding the Node*
void ***clearAllBonds****(n)*	*clearing of all bonds*

Routing:

void ***setRoutingOn****()*	*device works as a router*
void ***setRoutingOff****()*	*device does not work as a router*

Networking:

void ***setNetworkOne****()*	*networking mode, network 1 selected*
void ***setNetworkTwo****()*	*networking mode, network 2 selected*
void ***setNetworkingOff****()*	*non-networking mode*
void ***setCoordinatorMode****()*	*device works as the Coordinator*
void ***setNodeMode****()*	*device works as the Node*
void ***setNetworkFilteringOn****()*	*packets accepted from current network only*
void ***setNetworkFilteringOff****()*	*packets accepted from both networks*
void ***setLoggingOn****()*	*communication with adjacent nodes logged*
void ***setLoggingOff****()*	*communication with adjacent nodes not logged*
uns8 ***getNetworkParams****()*	*get information about the network*

See [11] – Examples E11–DATACENTER and E12–MEASUREMENT.

SPI

void ***enableSPI****()*	*enables SPI communication*
void ***disableSPI****()*	*disables SPI communication*
void ***startSPI****(length)*	*starts SPI packet transmission*
void ***stopSPI****()*	*suspends SPI communication*
bit ***getStatusSPI****()*	*updates SPI flags and packet length. Busy check.*

See Chapter SPI for details.

List of OS functions

Control	
reset()	initialization of microcontroller, OS and application
iqrfSleep()	enter power saving mode (Sleep)
debug()	enter the debug mode
setTXpower(level)	RF power setting (7 levels)
setPIR1(x)	setting of microcontroller peripheral flags
uns8 **batteryValueOK**()	battery check
Real time	
Active waiting	
waitMS(ms)	active waiting (time in ms)
waitDelay(ticks)	active waiting (time in ticks)
Waiting on background	
startDelay(ticks)	start waiting (time in ticks)
bit **isDelay**()	still waiting
startCapture()	resets counter of ticks
captureTicks()	number of ticks counted
LED indication	
setOnPulsingOUT2(ticks)	LED On time setting (for blinking)
setOffPulsingOUT2(ticks)	LED Off time setting (for blinking)
pulsingOUT2()	LED activation (blinking on background)
pulseOUT2()	single LED pulse (on background)
stopOUT2()	LED off, blinking stopped
RAM	
uns8 **readFromRAM**(addr)	read one byte
writeToRAM(addr,data)	write one byte
EEPROM	
uns8 **eeReadByte**(addr)	read one byte
eeReadData(addr,length)	read a block
eeWriteByte(addr, data)	write one byte
eeWriteData(addr, length)	write a block
Buffer INFO	
moduleInfo()	get info about transceiver module and OS
appInfo()	get info about application
clearBufferINFO()	buffer INFO clearing
Buffers, data blocks	
copyBufferINFO2COM()	copy buffer INFO to buffer COM
copyBufferINFO2RF()	copy buffer INFO to buffer RF
copyBufferRF2COM()	copy buffer RF to buffer COM
copyBufferRF2INFO()	copy buffer RF to buffer INFO
copyBufferCOM2RF()	copy buffer COM to buffer RF
copyBufferCOM2INFO()	copy buffer COM to buffer INFO
bit **compareBufferINFO2RF**(length)	comparison of buffer INFO and buffer RF
copyMemoryBlock (uns16 from, uns16 to, uns8 length)	copy any data block to any position (independent on RAM banking)
SPI	
enableSPI()	SPI communication line activation
disableSPI()	SPI communication line deactivation
startSPI(length)	SPI packet transmission
stopSPI()	SPI stopping
bit **getStatusSPI**()	SPI status, update SPI flags
RF	
RFTXpacket()	send a packet from buffer RF to RF line
bit **RFRXpacket**()	receive a packet from RF line to buffer RF

Networks	
Networking	
setNetworkOne()	network 1 selected
setNetworkTwo()	network 2 selected
setNetworkingOff()	networking disabled
setCoordinatorMode()	device is the Coordinator
setNodeMode()	device is a Node
setNetworkFilteringOn()	packets accepted from current network only
setNetworkFilteringOff()	packets accepted from both networks
setLoggingOn()	communication with adjacent nodes logged
setLoggingOff()	communication with adjacent nodes not logged
uns8 **getNetworkParams**()	get information about the network
Routing	
setRoutingOn()	device works as a router
setRoutingOff()	device does not work as a router
Bonding - Node	
bit **bondRequest**()	request for bonding
removeBond()	request for unbonding
bit **amIBonded**()	is the Node bonded?
wipeBondNR()	reserved node number cancellation (after bond removing)
Bonding - Coordinator	
bit **bondNewNode**()	bonding a Node
removeBondedNode(n)	unbonding a Node
bit **rebondNode**(n)	rebonding a Node
bit **isBondedNode**(n)	is the Node bonded?
clearAllBonds()	clearing of all bonds
Temperature	
getTemperature()	Temperature measurement

Unless otherwise stated, all functions are the `void` type and all their parameters are the `uns8` type.

Documentation and Information

1 **IQRF** support site: www.iq-esupport.com
2 **PIC16LF88(6)** datasheets: http://ww1.microchip.com/downloads/en/DeviceDoc/30487c.pdf and http://ww1.microchip.com/downloads/en/DeviceDoc/41291E.pdf
3 **TR-xxx-21A** Datasheet: www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=88
4 **TR-xxx-11A** Datasheet www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=70
5 **RAM map:** http://www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=92
6 **EEPROM map:** http://www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=93
7 **CK-USB-02** User's manual: www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=74
8 **IQRF** home page: www.iqrf.org
9 **IQRF IDE**: www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=89
10 **SPI** specification: www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=90
11 **Examples** (included in the StartUp Package): www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=96
12 **TR-xxx-31B** Datasheet: www.iq-esupport.com/index.php?dload=Download&_m=downloads&_a=downloadfile&downloaditemid=109

If you need a help or more information please visit IQRF support pages [1] and look into the Knowledgebase or Submit a Ticket with your request. A lot of information is also available on the IQRF home page [8].

Sales and Service

Corporate office:

MICRORISC s.r.o., Delnicka 222, 506 01 Jicin, Czech Republic, EU
Tel: +420 493 538 125, Fax: +420 493 538 126, www.microrisc.com

Partners and distribution:

please visit www.iqrf.org/partners

Quality management:

ISO 9001 : 2000 certified

Trademarks:

The IQRF name and logo are registered trademarks of MICRORISC s.r.o.
PIC, SPI, Microchip, RFM and all other trademarks mentioned herein are property of their respective owners.

Legal:

All information contained in this publication is intended through suggestion only and may be superseded by updates without prior notice. No representation or warranty is given and no liability is assumed by MICRORISC s.r.o. with respect to the accuracy or use of such information.

Without written permission it is not allowed to copy or reproduce this information, even partially.

No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

The IQRF products utilize several patents (CZ, EU, US)

Website	www.iqrf.org
E-mail	sales@iqrf.org
On-line support	http://iq-esupport.com

Simple way to smarter wireless solutions