# RadiaCode API Documentation (Detailed Specification)

> **IMPORTANT NOTE**: This document describes the communication protocol for RadiaCode radiation detection devices as implemented in the RadiaCode Arduino library. The protocol is based on commands that access Virtual Special Function Registers (VSFRs) and Virtual Strings (VS) to control device functionality.

## Table of Contents



**Version: 1.0.0**

# History

| Date | Driver Version | Doc. Version | Description |
|---|---|---|---|
| 26/09/2025 | 1.0.0 | 1.0.0 | Initial Release, tested with RadiaCode 103 and firmware v4.14 |

# Overview

This document provides a comprehensive and detailed specification of the communication protocol for RadiaCode radiation detection devices. The API enables device configuration, measurement control, and data acquisition through the Bluetooth Low Energy (BLE) protocol.

# Connection Specifications

## Service UUID

- Primary Service UUID: `0000ff10-0000-1000-8000-00805f9b34fb`

## Characteristic UUIDs

1. **Command Characteristic**

   - UUID: `0000ff11-0000-1000-8000-00805f9b34fb`
   - Properties: Write, Write Without Response
   - Purpose: Send commands to the device
   - Maximum Payload Size: 20 bytes (BLE standard MTU)

2. **Response Characteristic**

   - UUID: `0000ff12-0000-1000-8000-00805f9b34fb`
   - Properties: Notify
   - Purpose: Receive responses and notifications from the device
   - Maximum Payload Size: 20 bytes (BLE standard MTU)

## Device Discovery

- Advertising Name Format: "RadiaCode-XXXX" (where XXXX represents the last 4 digits of the device serial number)
- Advertising Service UUID: `0000ff10-0000-1000-8000-00805f9b34fb` (partial UUID advertised)

# Protocol Fundamentals

## Byte Order

- All multi-byte values use **little-endian** byte order unless explicitly stated otherwise
- Example: The value 0x1234 is transmitted as [0x34, 0x12]

## Data Alignment

- No padding or alignment is required between fields
- Fields are packed contiguously in the order specified

## Character Encoding

- All string data uses UTF-8 encoding
- Strings are null-terminated unless specified otherwise

# Message Structure

## Request Format

All commands follow this standardized structure:

```
[Request Length (4 bytes, LE)] [Request Type (2 bytes, LE)] [Spare Byte (1 byte)] [Sequence Number (1 byte)] [Payload (variable)]
```

- **Request Length** (4 bytes, little-endian): Total size of the command payload (header + data)
- **Request Type** (2 bytes, little-endian): Command identifier (e.g., 0x0005 for GET_STATUS)
- **Spare Byte** (1 byte): Reserved byte, always 0x00
- **Sequence Number** (1 byte): Command tracking number (0x80 + sequence counter, wraps after 31)
- **Payload** (variable): Command-specific data (may be empty for simple commands)

## Response Format

All responses follow this standardized structure:

```
[Response Type (2 bytes, LE)] [Spare Byte (1 byte)] [Sequence Number (1 byte)] [Response Data (variable)]
```

- **Response Type** (2 bytes, little-endian): Command identifier (matches the request)
- **Spare Byte** (1 byte): Reserved byte, always 0x00
- **Sequence Number** (1 byte): Matches the request sequence number
- **Response Data** (variable): Command-specific response data
    - For success: Command-specific payload data
    - For errors: Error codes or status information

## Fragmentation

For requests or responses exceeding the BLE MTU size (typically 20 bytes):

1. The complete message is sent across multiple BLE packets
2. The underlying Bluetooth transport layer handles packet reassembly automatically
3. The application layer receives the complete, reassembled message

### Sequence Number Management

- Sequence numbers start at 0x80 and increment by 1 for each command
- Valid range: 0x80 to 0x9F (32 values total)
- After 0x9F, wraps back to 0x80
- Used for matching requests with their corresponding responses
- Helps detect lost or duplicate packets

### Example Message Flow

**Simple command with no payload (GET_STATUS):**

```
Request:
[04][00][00][00]      // Request length: 4 bytes
[05][00][00][84]      // GET_STATUS (0x0005), spare: 0x00, sequence: 0x84

Response:
[05][00][00][84]      // GET_STATUS (0x0005), spare: 0x00, sequence: 0x84
[02][00][00][00]      // Status flags: 0x00000002
```

**Command with payload (WR_VIRT_SFR):**

```
Request:
[0C][00][00][00]      // Request length: 12 bytes
[25][08][00][85]      // WR_VIRT_SFR (0x0825), spare: 0x00, sequence: 0x85
[04][05][00][00]      // VSFR ID: 0x00000504 (DEVICE_TIME)
```

```
[00][00][00][00]      // Value: 0x00000000

Response:
[25][08][00][85]      // WR_VIRT_SFR (0x0825), spare: 0x00, sequence: 0x85
[01][00][00][00]      // Return code: 1 (success)
```

## Command Reference

### GET_STATUS (0x0005)

**Purpose**: Retrieve device status information including operating state, detector status, and more

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000004)
- Request Type: 2 bytes (little-endian) - command GET_STATUS (0x0005)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0005)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Status Flags: 4 bytes (little-endian) - device status bitfield
    - Meaning of individual Bits is not determined yet

**Example:**

```
Request (hexadecimal):
[04][00][00][00]      // Request Length: 4 bytes
[05][00][00][84]      // Request Type: 0x0005 (GET_STATUS), spare byte: 0x00, sequence: 0x84

Response (hexadecimal):
[05][00][00][84]      // Response Type: 0x0005, spare byte: 0x00, sequence: 0x84 (matches request)
[02][00][00][00]      // Status Flags: 0x00000002 (bit 1 set)
```

**Implementation Notes:**

- This command returns a simple 4-byte status value directly in the response
- The actual device status (battery level, temperature, etc.) is available through specific VSFRs
- The RadiaCode library `deviceStatus()` method returns these flags as a 4-byte value
- Status flags interpretation may vary depending on device firmware version
- This is one of the simpler commands with no payload in the request

## SET_EXCHANGE (0x0007)

**Purpose**: Configure communication parameters and initialize the data exchange session

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000008)
- Request Type: 2 bytes (little-endian) - command SET_EXCHANGE (0x0007)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- Payload: 4 bytes (little-endian) - communication parameters (0xFF12FF01)
    - Byte 0: Protocol version - (0x01)
    - Byte 1: Exchange flags - (0xFF)
    - Byte 2: Parameter 1 - (0x12)
    - Byte 3: Parameter 2 - (0xFF)

**Response Format:**

- Request Type: 2 bytes (little-endian) - matches request (0x0007)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Payload: Variable Number Of Bytes - device response parameters

**Example:**

```
Request (hexadecimal):
[08][00][00][00]      // Request length: 8 bytes
[07][00][00][80]      // Request type: 0x0007 (SET_EXCHANGE), spare byte: 0x00, sequence: 0x80
[01][FF][12][FF]      // Payload: communication parameters

Response (hexadecimal):
[07][00][00][80]      // Response Type: 0x0007, spare byte: 0x00, sequence: 0x80 (matches request)
[05][01][DA][DA][20][10][00][10][01][02][00][00]  // Response payload
```

**Implementation Notes:**

- This command is typically sent at the beginning of a session to establish communication parameters
- The RadiaCode library sends this automatically during connection initialization
- The request length field indicates the total size of the command payload (header + data)
- The sequence number is used for command tracking and should be incremented for each command
- This command must be sent before other commands to ensure proper communication

### GET_VERSION (0x000A)

**Purpose**: Retrieve bootloader and target firmware version information

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000004)
- Request Type: 2 bytes (little-endian) - command GET_VERSION (0x000A)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x000A)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Boot Minor Version: 2 bytes (little-endian) - Minor version of the bootloader
- Boot Major Version: 2 bytes (little-endian) - Major version of the bootloader
- Boot Date Length: 1 byte - length of boot date string
- Boot Date String: Variable length (UTF-8, not null-terminated) - Date of bootloader firmware
- Target Minor Version: 2 bytes (little-endian) - Minor version of the target firmware
- Target Major Version: 2 bytes (little-endian) - Major version of the target firmware
- Target Date Length: 1 byte - length of target date string
- Target Date String: Variable length (UTF-8, null-terminated) - Date of target firmware

**Example:**

```
Request (hexadecimal):
[04][00][00][00]      // Request length: 4 bytes
[0A][00][00][83]      // Request type: 0x000A (GET_VERSION), spare byte: 0x00, sequence: 0x83

Response (hexadecimal):
[0A][00][00][83]      // Response type: 0x000A, spare byte: 0x00, sequence: 0x83 (matches request)
```

```
[00][00][04][00]      // Boot Minor: 0, Boot Major: 4
[14]                  // Boot date length: 20 bytes
[46][65][62][20][20][36][20][32][30][32][33][20][31][35][3A][34][39][3A][31][34] // "Feb  6 2023 15:49:14"
[0E][00][04][00]      // Target Minor: 14, Target Major: 4
[15]                  // Target date length: 21 bytes (incl. null terminator)
[4A][75][6C][20][20][37][20][32][30][32][35][20][31][31][3A][32][30][3A][33][30][00] // "Jul  7 2025 11:20:30"
```

**Implementation Notes:**

- The library parses this into a tuple containing major/minor versions and date strings
- The `fwVersion()` method returns a tuple containing the values for boot_major, boot_minor, boot_date, target_major, target_minor and target_date
- Both version numbers are represented as 16-bit values in little-endian format

## GET_SERIAL (0x000B)

**Purpose**: Retrieve device hardware serial number

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000004)
- Request Type: 2 bytes (little-endian) - command GET_SERIAL (0x000B)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x000B)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (matches request)
- Serial Length: 4 bytes (little-endian) - length of the serial number in bytes (must be multiple of 4)
- Serial Groups: Variable length - Series of 32-bit values representing the serial number
    - Each 32-bit value is stored in little-endian format
    - The serial number is displayed as hexadecimal groups separated by hyphens

**Example:**

```
Request (hexadecimal):
[04][00][00][00]      // Request length: 4 bytes
[0B][00][00][86]      // Request type: 0x000B (GET_SERIAL), spare byte: 0x00, sequence: 0x86

Response (hexadecimal):
[0B][00][00][86]      // Response type: 0x000B, spare byte: 0x00, sequence: 0x86 (matches request)
[0C][00][00][00]      // Serial length: 12 bytes (3 groups)
[78][56][34][12]      // First group: 0x12345678
[BC][9A][F0][DE]      // Second group: 0xDEF09ABC
[34][12][CD][AB]      // Third group: 0xABCD1234
```

**Formatted Serial Number Output:**

```
12345678-DEF09ABC-ABCD1234
```

**Implementation Notes:**

- The library `hwSerialNumber()` method formats this as hyphen-separated hexadecimal groups
- Each 4-byte group is displayed as an 8-character hexadecimal value
- The serial length must be a multiple of 4 bytes for proper parsing
- This is different from the device identifier accessible via `serialNumber()` which uses VS::SERIAL_NUMBER (0x08)
- The response includes the serial length directly, followed immediately by the serial data groups

## FW_IMAGE_GET_INFO (0x0012)

**Purpose**: Retrieve firmware image information and version details

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000004)
- Request Type: 2 bytes (little-endian) - command FW_IMAGE_GET_INFO (0x0012)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0012)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte (matches request)
- Firmware Signature: 4 bytes (little-endian) - unique firmware identifier
- Image Information String: Variable bytes - null-terminated string containing firmware details
    - Format typically includes version, build date, and device model information

**Example:**

```
Request (hexadecimal):
[04][00][00][00]      // Request length: 4 bytes
[12][00][00][94]      // Request type: 0x0012 (FW_IMAGE_GET_INFO), spare byte: 0x00, sequence: 0x94

Response (hexadecimal):
[12][00][00][94]      // Response type: 0x0012, spare byte: 0x00, sequence: 0x94 (matches request)
[78][56][34][12]      // Firmware signature: 0x12345678
[52][43][31][30][31][2D][76][31][2E][33][2E][34][2D][32][30][32][34][30][33][31][35][2D][72][65][6C][65][61][73][65][00]
// "RC101-v1.3.4-20240315-release" + null terminator
```

**Implementation Notes:**

- This command is used to identify the firmware version and characteristics
- The firmware signature is a unique identifier for the specific firmware image
- The image information string typically includes version, date, and device model
- This command is often used during connection to verify device compatibility
- No payload is required in the request - it's a simple information query
- The response variable length depends on the firmware information string content
- Firmware signature can be used to verify authentic firmware images

## FW_SIGNATURE (0x0101)

**Purpose**: Retrieve firmware signature, filename and ID string information

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000004)
- Request Type: 2 bytes (little-endian) - command FW_SIGNATURE (0x0101)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0101)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Firmware Signature: 4 bytes (little-endian)
- Filename Length: 1 byte - length of filename string
- Filename String: Variable length (UTF-8, not null-terminated)
- ID Length: 1 byte - length of ID string
- ID String: Variable length (UTF-8, not null-terminated)
- Unknown Length: 1 byte - length of unknown string
- Unknown String: Variable length (UTF-8, not null-terminated)

**Example:**

```
Request (hexadecimal):
[04][00][00][00]      // Request length: 4 bytes
[01][01][00][83]      // Request type: 0x0101 (FW_SIGNATURE), spare byte: 0x00, sequence: 0x83

Response (hexadecimal):
[01][01][00][83]      // Response type: 0x0101, spare byte: 0x00, sequence: 0x83 (matches request)
[AD][42][E7][5A]      // Firmware Signature: 0x45AE742AD
```

```
[0A]                  // Filename length: 10 bytes
[72][63][2D][31][30][33][2E][62][69][6E]  // Filename String: "rc-103.bin"
[10]                  // ID length: 16 bytes
[52][61][64][69][61][43][6F][64][65][20][52][43][2D][31][30][33]  // ID String: "RadiaCode RC-103"
[01]                  // Unknown length
[30]                  // Unknown string
```

**Implementation Notes:**

- The library parses this into a tuple containing firmware signature, filename and ID string
- The `fwSignature()` method returns a string in the format `Signature: signature_value, FileName="filename.bin", IdString="RadiaCode RC-1xx"`
- The Firmware signature is represented as 32-bit value in little-endian format

## RD_FLASH (0x081C)

**Purpose**: Read data directly from device flash memory

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x0000000C)
- Request Type: 2 bytes (little-endian) - command RD_FLASH (0x081C)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- Address: 4 bytes (little-endian) - starting flash memory address to read from
- Size: 4 bytes (little-endian) - number of bytes to read

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x081C)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Address: 4 bytes (little-endian) - address that was read from (echoed from request)
- Size: 4 bytes (little-endian) - number of bytes read (echoed from request)
- Data: Variable bytes - raw flash memory content

**Example:** Reading 16 bytes from flash address 0x08020000

```
Request (hexadecimal):
[0C][00][00][00]      // Request length: 12 bytes
[1C][08][00][92]      // Request type: 0x081C (RD_FLASH), spare byte: 0x00, sequence: 0x92
[00][00][02][08]      // Address: 0x08020000
[10][00][00][00]      // Size: 16 bytes

Response (hexadecimal):
[1C][08][00][92]      // Response type: 0x081C, spare byte: 0x00, sequence: 0x92 (matches request)
[00][00][02][08]      // Address: 0x08020000
```

```
[10][00][00][00]      // Size: 16 bytes
[XX][XX]...[XX][XX]   // 16 bytes of flash content
```

**Implementation Notes:**

- This command provides direct access to device memory and should be used with caution
- Flash memory organization is device-specific and not documented for general use
- The library typically uses higher-level methods to access device data rather than direct flash access
- This command is mainly used for firmware updates, diagnostics, or advanced customization
- Maximum read size is typically limited to 256 bytes per request due to BLE packet limitations
- Reading beyond valid memory regions will result in an error response
- Flash memory contains calibration data, configuration parameters, and firmware
- The response echoes the address and size before providing the actual data
- Payload: Requested flash memory data

## RD_VIRT_SFR (0x0824)

**Purpose**: Read a Virtual Special Function Register (VSFR) value from the device

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000008)
- Request Type: 2 bytes (little-endian) - command RD_VIRT_SFR (0x0824)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- VSFR ID: 4 bytes (little-endian) - ID of the register to read (filled-up with 0x0000)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0824)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- VSFR Value: 4 bytes (little-endian) - value of the requested VSFR
  - Interpretation depends on the specific VSFR:
    - For most values, this is a 32-bit integer or float
    - For bitfields, each bit has a specific meaning
    - For fixed-point values, a specific scaling factor applies

**Example 1:** Reading temperature (VSFR::TEMP_degC = 0x00008024)

```
Request (hexadecimal):
[08][00][00][00]      // Request length: 8 bytes
[24][08][00][87]      // Request type: 0x0824 (RD_VIRT_SFR), spare byte: 0x00, sequence: 0x87
[24][80][00][00]      // VSFR ID: 0x00008024 (TEMP_degC)

Response (hexadecimal):
[24][08][00][87]      // Response type: 0x0824, spare byte: 0x00, sequence: 0x87 (matches request)
[00][00][C8][41]      // Value: 0x41C80000 (25.0 in IEEE-754 float format)
```

**Example 2:** Reading device control flags (VSFR::DEVICE_CTRL = 0x00000500)

```
Request (hexadecimal):
[08][00][00][00]      // Request length: 8 bytes
[24][08][00][88]      // Request type: 0x0824 (RD_VIRT_SFR), spare byte: 0x00, sequence: 0x88
[00][05][00][00]      // VSFR ID: 0x00000500 (DEVICE_CTRL)

Response (hexadecimal):
[24][08][00][88]      // Response type: 0x0824, spare byte: 0x00, sequence: 0x88 (matches request)
[05][00][00][00]      // Value: 0x00000005 (bits 0 and 2 set)
                      // Meaning: device on (bit 0), sound on (bit 2)
```

**Implementation Notes:**

- The library handles the conversion between binary data and appropriate types
- For floating point values, the binary representation is IEEE-754 single precision
- VSFR IDs are 4-byte values that uniquely identify each register in the device
- The request payload size is always 8 bytes (4-byte VSFR ID)
- The response always contains a 4-byte value, but interpretation varies by VSFR type
- See the Virtual Special Function Registers section for a complete list of VSFRs and their meanings
- Individual VSFR reads are less efficient than batch reads using RD_VIRT_SFR_BATCH

### WR_VIRT_SFR (0x0825)

**Purpose**: Write a value to a Virtual Special Function Register (VSFR)

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x0000000C)
- Request Type: 2 bytes (little-endian) - command WR_VIRT_SFR (0x0825)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- VSFR ID: 4 bytes (little-endian) - register identifier (e.g., DEVICE_TIME = 0x00000504) (filled-up with 0x0000)
- Value: 4 bytes (little-endian) - value to write to the register

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0825)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Return Code: 4 bytes (little-endian) - 1 for success, other values for failure

**Payload Details:** Value interpretation depends on the specific VSFR:

- For most registers, 32-bit integer or IEEE-754 float value
- Write-only registers may trigger specific actions when written
- Some registers may only accept specific values or value ranges
- Writing to read-only registers will result in an error response
- Bitfield registers allow controlling multiple settings with a single write

**Example 1:** Setting device timestamp (VSFR::DEVICE_TIME = 0x00000504) to 0x00000000

```
Request (hexadecimal):
[0C][00][00][00]      // Request length: 12 bytes
[25][08][00][82]      // Request type: 0x0825 (WR_VIRT_SFR), spare byte: 0x00, sequence: 0x82
```

```
[04][05][00][00]      // VSFR ID: 0x00000504 (DEVICE_TIME)
[00][00][00][00]      // Value: 0x00000000

Response (hexadecimal):
[25][08][00][82]      // Response type: 0x0825, spare byte: 0x00, sequence: 0x82 (matches request)
[01][00][00][00]      // Return code: 1 (success)
```

**Example 2:** Setting display brightness (VSFR::DISP_BRT = 0x00000511) to 5

```
Request (hexadecimal):
[0C][00][00][00]      // Request length: 12 bytes
[25][08][00][83]      // Request type: 0x0825 (WR_VIRT_SFR), spare byte: 0x00, sequence: 0x83
[11][05][00][00]      // VSFR ID: 0x00000511 (DISP_BRT)
[05][00][00][00]      // Value: 0x00000005

Response (hexadecimal):
[25][08][00][83]      // Response type: 0x0825, spare byte: 0x00, sequence: 0x83 (matches request)
[01][00][00][00]      // Return code: 1 (success)
```

**Implementation Notes:**

- The library method `writeRequest()` handles writing values to VSFRs
- Write operations that fail will typically return result code 0 instead of 1
- For setters in the library API (like `setDisplayBrightness()`), the conversion from user-friendly values to appropriate register values is handled automatically
- VSFR IDs are 4-byte values that uniquely identify each register in the device
- Value interpretation depends on the specific VSFR (integer, float, bitfield, etc.)
- Writing to read-only registers will result in an error response
- Some registers may only accept specific values or value ranges

## RD_VIRT_STRING (0x0826)

**Purpose**: Read a Virtual String (VS) from the device

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x00000008)
- Request Type: 2 bytes (little-endian) - command RD_VIRT_STRING (0x0826)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- VS ID: 4 bytes (little-endian) - ID of the virtual string to read (filled-up each with 0x0000)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0826)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Return Code: 4 bytes (little-endian) - 1 for success, other values for failure
- String Length: 4 bytes (little-endian) - Length of the string data in bytes
- String Data: Variable length - The actual string data
  - For text strings: UTF-8 encoded text
  - For binary data (like spectrums): Raw binary format specific to the VS ID

**Example 1:** Reading device configuration (VS::CONFIGURATION = 0x02)

```
Request (hexadecimal):
[08][00][00][00]      // Request length: 8 bytes
[26][08][00][84]      // Request type: 0x0826 (RD_VIRT_STRING), spare byte: 0x00, sequence: 0x84
[02][00][00][00]      // VS ID: 0x02 (CONFIGURATION)

Response (hexadecimal):
[26][08][00][84]      // Response type: 0x0826, spare byte: 0x00, sequence: 0x84 (matches request)
[01][00][00][00]      // Return code: 1 (success)
```

```
[B8][0C][00][00]      // String length: 3256 bytes (0x0CB8)
[5B][44][65][76][69][63][65][50][61][72][61][6D][73][5D][0A] // "[DeviceParams]\n..."
...                   // Configuration data continues
```

**Example 2:** Reading device serial number (VS::SERIAL_NUMBER = 0x08)

```
Request (hexadecimal):
[08][00][00][00]      // Request length: 8 bytes
[26][08][00][85]      // Request type: 0x0826 (RD_VIRT_STRING), spare byte: 0x00, sequence: 0x85
[08][00][00][00]      // VS ID: 0x08 (SERIAL_NUMBER)

Response (hexadecimal):
[26][08][00][85]      // Response type: 0x0826, spare byte: 0x00, sequence: 0x85 (matches request)
[01][00][00][00]      // Return code: 1 (success)
[0D][00][00][00]      // String length: 13 bytes
[52][43][2D][31][30][33][2D][30][30][39][33][31][39] // "RC-103-009319"
```

**Implementation Notes:**

- The library method `readRequest()` handles reading virtual strings using RD_VIRT_STRING
- Return code 1 indicates successful read, other values indicate errors
- String responses may or may not be null-terminated depending on the VS type
- For binary data (like spectrum data), additional parsing logic is required to interpret the response
- The response includes both a return code and the actual string length for validation
- Large responses may be fragmented across multiple BLE packets

## WR_VIRT_STRING (0x0827)

**Purpose**: Write data to a Virtual String (VS) on the device

**Request Format:**

- Request Length: Variable bytes (little-endian) - total payload size (minimum 0x00000010 for Request type + VS ID + data length + data)
- Request Type: 2 bytes (little-endian) - command WR_VIRT_STRING (0x0827)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- VS ID: 4 bytes (little-endian) - ID of the virtual string to write (filled-up with 0x0000)
- Data Length: 4 bytes (little-endian) - length of the data to write
- Data: Variable bytes - the actual data to write
  - For text strings: UTF-8 encoded text (often null-terminated)
  - For binary data: Raw binary format specific to the VS ID

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x0827)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Result Code: 4 bytes (little-endian) - 1 for success, 0 or other values for failure

**Example:** Setting energy calibration values (VS::ENERGY_CALIB = 0x202)

```
Request (hexadecimal):
[18][00][00][00]      // Request length: 24 bytes
[27][08][00][87]      // Request type: 0x0827 (WR_VIRT_STRING), spare byte: 0x00, sequence: 0x87
[02][02][00][00]      // VS ID: 0x0202 (ENERGY_CALIB)
[0C][00][00][00]      // Data length: 12 bytes
[09][16][AD][3F]      // a0 = 1.352235 (IEEE-754)
[B9][70][18][40]      // a1 = 2.381880 (IEEE-754)
[C5][73][B6][39]      // a2 = 0.000348 (IEEE-754)
```

```
Response (hexadecimal):
[27][08][00][87]      // Response type: 0x0827, spare byte: 0x00, sequence: 0x87 (matches request)
[01][00][00][00]      // Result: 1 (success)
```

**Implementation Notes:**

- The library handles packing the data into the appropriate format based on the VS ID
- Each VS has a specific data format requirement (text strings, binary data, etc.)
- The request length includes Request type + VS ID (4 bytes) + data length field (4 bytes) + actual data
- Success is indicated by result code 1; failure is indicated by 0 or other values
- The RadiaCode library `writeRequest()` method abstracts this command for VS operations
- Virtual Strings provide a uniform interface for accessing device configuration and data
- Some VS IDs are read-only and will return an error if written to
- For structured data (like calibration parameters), the format must match exactly what the device expects
- The library method `execute(COMMAND::WR_VIRT_STRING, ...)` handles writing virtual strings
- Binary data must follow the specific structure expected by the VS
- Configuration data must include all required fields in the correct format

## RD_VIRT_SFR_BATCH (0x082A)

**Purpose**: Read multiple Virtual Special Function Registers (VSFRs) in a single request

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size
- Request Type: 2 bytes (little-endian) - command RD_VIRT_SFR_BATCH (0x082A)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- Number of VSFRs: 4 bytes (little-endian) - Count of VSFRs to read
- VSFR IDs: 4 bytes per ID (little-endian) - Array of VSFR IDs to read (filled-up each with 0x0000)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x082A)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Valid Flags: 4 bytes (little-endian) - Bitmask indicating which VSFRs are valid
- VSFR Values: 4 bytes per value (little-endian) - Array of values in the same order as requested
  - For integer values: 32-bit integer
  - For floating point: IEEE-754 single precision float

**Example:** Reading temperature and sound control in one request

```
Request (hexadecimal):
[10][00][00][00]      // Request length: 16 bytes
[2A][08][00][86]      // Request type: 0x082A (RD_VIRT_SFR_BATCH), spare byte: 0x00, sequence: 0x86
[02][00][00][00]      // Number of VSFRs: 2
[20][05][00][00]      // VSFR ID: 0x0520 (SOUND_CTRL)
[24][80][00][00]      // VSFR ID: 0x8024 (TEMP_degC)

Response (hexadecimal):
```

```
[2A][08][00][86]      // Response type: 0x082A, spare byte: 0x00, sequence: 0x86 (matches request)
[03][00][00][00]      // Valid flags: 0x03 (binary 3, both VSFRs are valid)
[03][00][00][00]      // Value: 0x00000003 (Button + Click sounds are on)
[00][00][C8][41]      // Value: 0x41C80000 (25.0°C in IEEE-754)
```

**Implementation Notes:**

- The `batchReadVSFRs()` method in the library handles reading multiple VSFRs efficiently
- The response includes valid flags as a bitmask indicating which VSFRs were successfully read
- Valid flags value should equal (2^n - 1) where n is the number of requested VSFRs
- All values are returned as 32-bit values (floating point or integer) based on the VSFR type
- Reading multiple VSFRs in a single request is much more efficient than individual reads
- The response values are returned in the same order as the requested IDs
- If any VSFR ID is invalid, the corresponding bit in valid flags will be 0

### WR_VIRT_SFR_BATCH (0x082B)

**Purpose**: Write to multiple Virtual Special Function Registers (VSFRs) in a single request

**Request Format:**

- Request Length: Variable bytes (little-endian) - total payload size
- Request Type: 2 bytes (little-endian) - command WR_VIRT_SFR_BATCH (0x082B)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- VSFR Count: 4 bytes (little-endian) - number of VSFRs to write
- VSFR IDs: count * 4 bytes (little-endian) - array of VSFR identifiers (filled-up each with 0x0000)
- VSFR Values: count * 4 bytes (little-endian) - array of values (same order as IDs)

**Response Format:**

- Response Type: 2 bytes (little-endian) - matches request (0x082B)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Result Flags: 4 bytes (little-endian) - bitfield indicating success for each VSFR

**Example:** Configuring display and sound settings in one request

```
Request (hexadecimal):
[18][00][00][00]      // Request length: 24 bytes
[2B][08][00][91]      // Request type: 0x082B (WR_VIRT_SFR_BATCH), spare byte: 0x00, sequence: 0x91
[01][00][00][00]      // VSFR Count: 1 (LE)
[11][05][00][00]      // VSFR ID: 0x0511 (DISP_BRT)
[20][05][00][00]      // VSFR ID: 0x0520 (SOUND_CTRL)
[09][00][00][00]      // Value: 0x00000009 (9 decimal - full brightness)
[03][00][00][00]      // Value: 0x00000003 (Button + Click sounds are on)

Response (hexadecimal):
```

```
[2B][08][00][91]      // Response type: 0x082B, spare byte: 0x00, sequence: 0x91 (matches request)
[03][00][00][00]      // Result flags: 0x00000003 (bit 0+1 set - both writes successful)
```

**Implementation Notes:**

- All values are written in a single atomic operation
- If any value is invalid or write-protected, the corresponding bit in result flags will be 0
- The method `setAlarmLimits()` uses this command for efficient batch configuration
- Writing multiple VSFRs in a batch is more efficient than individual writes
- This command is particularly useful for configuring multiple related settings at once
- The VSFR IDs and values are sent as separate arrays, not interleaved pairs
- Result flags use bit positions to indicate success/failure for each VSFR (bit 0 = first VSFR, bit 1 = second VSFR, etc.)
- For complete success, result flags should equal (2^count - 1)

SET_TIME (0x0A04)

**Purpose**: Set the device real-time clock

**Request Format:**

- Request Length: 4 bytes (little-endian) - total payload size (0x0000000C)
- Request Type: 2 bytes (little-endian) - command SET_TIME (0x0A04)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - (0x80 + sequence counter)
- Payload: 8 bytes - Time data structure
  - Day: 1 byte - Day of month (1-31)
  - Month: 1 byte - Month (1-12)
  - Year: 1 byte - Years since 2000 (e.g., 25 (0x19) for 2025)
  - Reserved: 1 byte - (0x00)
  - Second: 1 byte - Seconds (0-59)
  - Minute: 1 byte - Minutes (0-59)
  - Hour: 1 byte - Hours (0-23)
  - Reserved: 1 byte - (0x00)

**Response Format:**

- Request Type: 2 bytes (little-endian) - matches request (0x0A04)
- Spare Byte: 1 byte - (0x00)
- Sequence Number: 1 byte - matches request
- Payload: None (acknowledgment only)

**Example:** Setting the time to July 25, 2025, 12:30:45

```
Request (hexadecimal):
[0C][00][00][00]      // Request length: 12 bytes
[04][0A][00][93]      // Request type: 0x0A04 (SET_TIME), spare byte: 0x00, sequence: 0x93
```

```
[19]                  // Day: 25
[07]                  // Month: 7 (July)
[19]                  // Year: 25 (2025)
[00]                  // Reserved
[2D]                  // Second: 45
[1E]                  // Minute: 30
[0C]                  // Hour: 12
[00]                  // Reserved

Response (hexadecimal):
[04][0A][00][93]      // Request type: 0x0A04, spare byte: 0x00, sequence: 0x93 (matches request)
```

**Implementation Notes:**

- The library method `setLocalTime()` handles setting the device time with appropriate parameters
- Time is stored in local time format, not as a Unix timestamp
- This command sets the device's internal real-time clock used for timestamping measurements
- The library handles conversion from calendar date/time to the required binary format
- After setting the time, you can verify it by reading the DEVICE_TIME VSFR (0x0504)

# Virtual Special Function Registers (VSFR)

VSFRs are 32-bit registers used to control various aspects of the device. They are accessed using the RD_VIRT_SFR, WR_VIRT_SFR, RD_VIRT_SFR_BATCH, and WR_VIRT_SFR_BATCH commands.

## Device Control VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x0500 | DEVICE_CTRL | Device control register (bitfield) | R/W | Integer |
| 0x0502 | DEVICE_LANG | Device language setting | R/W | Integer |
| 0x0503 | DEVICE_ON | Device power state (1=on, 0=off) | R/W | Integer |
| 0x0504 | DEVICE_TIME | Device current timestamp | R/W | Integer |

## Display Control VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x0510 | DISP_CTRL | Display control register (bitfield) | R/W | Integer |
| 0x0511 | DISP_BRT | Display brightness (0=0% ... 9=100%) | R/W | Integer |
| 0x0513 | DISP_OFF_TIME | Display auto-off time in seconds | R/W | Integer |
| 0x0515 | DISP_DIR | Display direction (0=auto, 1=right, 2=left) | R/W | Integer |

## Audio Control VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x0520 | SOUND_CTRL | Sound control register (bitfield) | R/W | Integer |
| 0x0522 | SOUND_ON | Sound enable state (1=on, 0=off) | R/W | Integer |

## Vibration Control VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x0530 | VIBRO_CTRL | Vibration control register (bitfield) | R/W | Integer |
| 0x0531 | VIBRO_ON | Vibration enable state (1=on, 0=off) | R/W | Integer |

## Alarm and Measurement Mode VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x05E0 | ALARM_MODE | Alarm mode settings(1=once, 0=continuously) | R/W | Integer |

## Alarm and Dose Rate VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x8000 | DR_LEV1_uR_h | Dose rate level 1 alarm threshold (μR/h) | R/W | Float |
| 0x8001 | DR_LEV2_uR_h | Dose rate level 2 alarm threshold (μR/h) | R/W | Float |
| 0x8004 | DS_UNITS | Dose units setting (1=Sievert, 0=Roentgen) | R/W | Integer |
| 0x8007 | DOSE_RESET | Dose reset control | R/W | Integer |
| 0x8008 | CR_LEV1_cp10s | Count rate level 1 alarm (counts per 10s) | R/W | Float |
| 0x8009 | CR_LEV2_cp10s | Count rate level 2 alarm (counts per 10s) | R/W | Float |

## Calibration VSFRs

| ID | Name | Description | Access | Value Type |
| --- | --- | --- | --- | --- |
| 0x8010 | CHN_TO_keV_A0 | Channel to keV conversion coefficient A0 | R/W | Float |
| 0x8011 | CHN_TO_keV_A1 | Channel to keV conversion coefficient A1 | R/W | Float |

| ID | Name | Description | Access | Value Type |
|---|---|---|---|---|
| 0x8012 | CHN_TO_keV_A2 | Channel to keV conversion coefficient A2 | R/W | Float |
| 0x8013 | CR_UNITS | Count rate units setting (1=CPM, 0=CPS) | R/W | Integer |
| 0x8014 | DS_LEV1_uR | Dose level 1 alarm threshold (μR) | R/W | Float |
| 0x8015 | DS_LEV2_uR | Dose level 2 alarm threshold (μR) | R/W | Float |

### Temperature VSFRs

| ID | Name | Description | Access | Value Type |
|---|---|---|---|---|
| 0x8016 | TEMP_UNITS | Temperature units setting (1=°F, 0=°C) | R/W | Integer |

### Measurement Data VSFRs

| ID | Name | Description | Access | Value Type |
|---|---|---|---|---|
| 0x8024 | TEMP_degC | Device temperature | R | Float |

### Temperature Calibration VSFRs

| ID | Name | Description | Access | Value Type |
|---|---|---|---|---|
| 0x8033 | RAW_TEMP_degC | Raw temperature reading (°C) | R | Float |
| 0x8034 | TEMP_UP_degC | Upper temperature calibration point (°C) | R/W | Float |
| 0x8035 | TEMP_DN_degC | Lower temperature calibration point (°C) | R/W | Float |

# Virtual Strings (VS)

Virtual Strings (VS) are variable-length data structures accessed using the RD_VIRT_STRING and WR_VIRT_STRING commands. Each VS has a specific format based on its purpose and may contain text, binary data, or structured information.

## Configuration and Device Information VS

| ID | Name | Description | Access | Format |
|---|---|---|---|---|
| 0x02 | CONFIGURATION | Device configuration data | R/W | Binary configuration structure |
| 0x08 | SERIAL_NUMBER | Device serial number | R | UTF-8 text string |
| 0x0F | TEXT_MESSAGE | Text message for device display | R/W | UTF-8 text string |

## Memory and Debug VS

| ID | Name | Description | Access | Format |
|---|---|---|---|---|
| 0x100 | DATA_BUF | Data buffer | R | Binary structure array |
| 0x101 | SFR_FILE | Special function register file | R/W | Binary SFR data |

## Spectrum and Measurement VS

| ID | Name | Description | Access | Format |
|---|---|---|---|---|
| 0x200 | SPECTRUM | Current spectrum data | R | Binary spectrum structure |
| 0x202 | ENERGY_CALIB | Energy calibration data | R/W | Binary calibration parameters |
| 0x205 | SPEC_ACCUM | Accumulated spectrum | R | Binary spectrum structure |

## VS Data Formats

**SPECTRUM (0x200), SPEC_ACCUM (0x205)**

Spectrum data has the following structure:

```
[Spectrum Data Length (4 bytes) LE]
[Duration (4 bytes) LE]
[a0 (4 bytes) LE][a1 (4 bytes) LE][a2 (4 bytes) LE]
[Encoded Channel Count Data (variable number bytes)]
```

- Duration: Measurement duration in seconds
- a0, a1, a2: Energy calibration coefficients (32-bit IEEE float)

**ENERGY_CALIB (0x202)**

Energy calibration data has the following structure:

```
[Calibration Data Length (4 bytes) LE]
[a0 (4 bytes) LE][a1 (4 bytes) LE][a2 (4 bytes) LE][reserved (4 bytes) LE]
```

- a0, a1, a2: Calibration coefficients as IEEE-754 32-bit floats
- Energy (keV) = a0 + a1*channel* + *a2*channel^2

**DATA_BUF (0x100)**

Contains an array of measurement data points with timestamps:

```
[Record Length (4 bytes) LE]
[Record Type (3 bytes): Record Sequence Number (1 byte), Record EID (1 byte), Record GID (1 byte)]
[Record Timestamp Offset (4 bytes) LE]
```

```
[Record Data fields...]
...repeat for each record...
```

- Record types include real-time data (EID = 0x00, GID = 0x00), raw data (EID = 0x00, GID = 0x01), dose rate data (EID = 0x00, GID = 0x02), and rare data (EID = 0x00, GID = 0x03)
- Each record type has a different structure for its data fields

### Spectrum Data Format (VS 0x0200)

The SPECTRUM virtual string contains the current radiation spectrum data. The encoded format is: tbd

**Example for 1024 channels:** tbd

### Energy Calibration Format (VS 0x0202)

The ENERGY_CALIB virtual string contains the energy calibration polynomial coefficients. The format is:

**Coefficients (16 bytes)**:

- Four 32-bit IEEE-754 floating point values (little-endian)
- Representing coefficients a0, a1, a2 in the calibration polynomial:

  ```
  Energy(keV) = a0 + a1*Channel + a2*Channel²
  ```

# Common VSFR Access Examples

## Setting Display Brightness

To set the display brightness to 100%:

```
// Command to write DISP_BRT (0x0511) = 9
[0C][00][00][00]      // Command length: 12 bytes
[25][08][00][96]      // Command ID 0x0825 (WR_VIRT_SFR), Spare byte, Sequence Number
[11][05][00][00]      // VSFR ID: 0x0511 (DISP_BRT)
[09][00][00][00]      // Value: 0x00000009 (9 decimal)

// Response (success)
[25][08][00][96]      // Command ID 0x0825
[01][00][00][00]      // Result: 1 (success)
```

## Reading Temperature

To read the current device temperature:

```
// Command to read multiple VSFRs
[0C][00][00][00]      // Command length: 12 bytes
[2A][08][00][99]      // Command ID 0x082A (RD_VIRT_SFR_BATCH), Spare byte, Sequence Number
[01][00][00][00]      // Number of VSFRs: 1
[24][80][00][00]      // VSFR ID 1: 0x8024 (TEMP_degC)

// Response (example values)
[2A][08][00][99]      // Command ID 0x082A, Spare byte, Sequence Number
[01][00][00][00]      // Valid flags
[00][00][BC][41]      // Value: 0x41BC0000 (23.5°C in IEEE-754)
```

### Reading Spectrum Data

To retrieve the current spectrum:

```
// Command to read SPECTRUM (0x200)
[08][00][00][00]      // Command length: 8 bytes
[26][08][00][83]      // Command ID 0x0826 (RD_VIRT_STRING), Spare byte, Sequence Number
[00][02][00][00]      // VS ID: 0x0200 (SPECTRUM)

// Response (simplified - actual response would be much larger)
[26][08][00][83]      // Command ID 0x0826, Spare byte, Sequence Number
[01][00][00][00]      // Valid flags
[3C][00][00][00]      // Duration: 60 s (1 minute)
[09][16][AD][3F]      // a0: 1.352235
[B9][70][18][40]      // a1: 2.381880
[C5][73][B6][39]      // a2: 0.000348
[Encoded Channel data]  // Variable
```

### Setting Energy Calibration

To set the energy calibration coefficients:

```
Request (hexadecimal):
[18][00][00][00]      // Request length: 24 bytes
[27][08][00][87]      // Request type: 0x0827 (WR_VIRT_STRING), spare byte: 0x00, sequence: 0x87
[02][02][00][00]      // VS ID: 0x0202 (ENERGY_CALIB)
[0C][00][00][00]      // Data length: 12 bytes
[09][16][AD][3F]      // a0 = 1.352235 (IEEE-754)
[B9][70][18][40]      // a1 = 2.381880 (IEEE-754)
[C5][73][B6][39]      // a2 = 0.000348 (IEEE-754)

Response (hexadecimal):
```

```
[27][08][00][87]      // Response type: 0x0827, spare byte: 0x00, sequence: 0x87 (matches request)
[01][00][00][00]      // Result: 1 (success)
```

# Data Type Specifications

## Integers

- All multi-byte integers use **little-endian** byte order
- 16-bit values: Low byte first, high byte second
- 32-bit values: Lowest byte first, highest byte last
- Example: 0x12345678 is transmitted as [0x78, 0x56, 0x34, 0x12]

## Floating Point Values

- All floating point values use IEEE-754 32-bit single precision format
- Byte order is **little-endian**
- Format: [sign (1 bit)][exponent (8 bits)][mantissa (23 bits)]
- Example: The decimal value 1.5 is represented as 0x3FC00000, transmitted as [0x00, 0x00, 0xC0, 0x3F]

## Strings

- All strings use UTF-8 encoding
- Strings are null-terminated (0x00 byte at end)
- Maximum string length is 255 bytes (including null terminator)

## Bit Fields

When bit fields are used within a byte:

- Bit 0 refers to the least significant bit
- Bit 7 refers to the most significant bit

# Error Handling

## Error Codes

- `0x00`: Success (not an error)
- `0x01`: Invalid command (unrecognized Command ID)
- `0x02`: Invalid parameter (parameter out of range or incorrect format)
- `0x03`: Device busy (cannot process command at this time)
- `0x04`: Hardware error (internal device problem)
- `0x05`: Not implemented (command recognized but not supported)
- `0x06`: Timeout (operation took too long)
- `0xFF`: Unknown error (unspecified failure)

## Common Error Conditions

- Invalid command ID
- Invalid VSFR or VS ID
- Permission denied (attempting to write to read-only registers)
- Value out of range
- Device busy or in an incompatible state

# Connection Management

## Connection Establishment

1. Client scans for devices advertising the RadiaCode service UUID 0000ff10-0000-1000-8000-00805f9b34fb
2. Client establishes connection with desired device
3. Client discovers service and characteristic UUIDs
4. Client enables notifications on the Response Characteristic
5. Send SET_EXCHANGE command to establish communication parameters
6. Device is ready for commands

## Disconnection Procedure

1. Simply close the BLE connection
2. Device will timeout and return to advertising mode if no commands are received for a period

## Reconnection Logic

The protocol implements reconnection logic with exponential backoff for handling connection failures:

- Initial delay: 500ms
- Maximum delay: 30 seconds
- Backoff factor: 1.5

## Connection Maintenance

- Device will disconnect if no commands are received within 60 seconds
- Client should implement a keepalive mechanism (e.g., periodic battery status requests)

## MTU Size and Fragmentation

- Default BLE MTU size is 20 bytes
- Commands exceeding this size must be fragmented

- Maximum theoretical MTU is 512 bytes, but actual values depend on hardware

# Implementation Notes

## Endianness Considerations

This protocol uses consistently little-endian byte ordering for all multi-byte values. When implementing on big-endian systems, byte swapping will be necessary.

## Handling Large Responses

For responses exceeding the BLE MTU size:

1. Negotiate a larger MTU if supported by both client and device
2. Otherwise, implement proper fragmentation and reassembly logic

## Thread Safety

When implementing this protocol in a multi-threaded environment:

1. Maintain a command queue to avoid sending overlapping commands
2. Implement request/response pairing to match responses with their commands
3. Use timeouts to handle cases where responses are lost

## Power Considerations

- Minimize the frequency of polling commands in battery-powered applications
- Consider using the continuous acquisition mode with longer intervals between packets
- Implement proper disconnect handling to allow the device to enter low-power modes

## Actual Command IDs

The actual supported command IDs used in the source code (`RadiaCodeTypes.h`) are:

```
// Command identifiers
enum COMMAND
```

```
{
    GET_STATUS           = 0x0005,
    SET_EXCHANGE         = 0x0007,
    GET_VERSION          = 0x000A,
    GET_SERIAL           = 0x000B,
    FW_SIGNATURE         = 0x0101,
    WR_VIRT_SFR          = 0x0825,
    RD_VIRT_STRING       = 0x0826,
    WR_VIRT_STRING       = 0x0827,
    RD_VIRT_SFR_BATCH    = 0x082A,
    WR_VIRT_SFR_BATCH    = 0x082B,
    SET_TIME             = 0x0A04
};
```

## Virtual Special Function Registers (VSFR)

VSFRs are 16-bit registers used to control various aspects of the device. They are accessed using the RD_VIRT_SFR (0x0824) and WR_VIRT_SFR (0x0825) commands.

**Key VSFRs**

```
// Virtual Special Function Register identifiers
enum VSFR {
  DEVICE_CTRL = 0x0500,
  DEVICE_LANG = 0x0502,
  DEVICE_ON   = 0x0503,
  DEVICE_TIME = 0x0504,
  // ... many more VSFRs defined in RadiaCodeTypes.h
};
```

## Virtual Strings (VS)

Virtual Strings are variable-length data structures accessed using the RD_VIRT_STRING (0x0826) and WR_VIRT_STRING (0x0827) commands. The actual supported virtual strings used in the source code (`RadiaCodeTypes.h`) are:

```
// Virtual String identifiers
enum VS
{
    CONFIGURATION   = 2,
    SERIAL_NUMBER   = 8,
    TEXT_MESSAGE    = 0xF,
    DATA_BUF        = 0x100,
    SFR_FILE        = 0x101,
    SPECTRUM        = 0x200,
    ENERGY_CALIB    = 0x202,
    SPEC_ACCUM      = 0x205
};
```

## API Reference

This section provides a comprehensive reference of the RadiaCode library's public API methods. These methods allow you to communicate with RadiaCode radiation detection devices and access their functionality from your Arduino projects.

### General methods

`const char* getDriverVersion(void)`

**Purpose:** Returns the driver version string.
**Parameters:** None
**Return Value:** Char pointer containing the driver version string in semantic versioning format (e.g., "1.1.0").
**Example:**

```
#include <RadiaCode.h>

// Char pointer to version string
const char* version;

// Get driver version
version = getDriverVersion();
Serial.printf("Driver version: %s\n", getDriverVersion());
```

**Notes:** This returns the version of the driver itself, not the device firmware version.

`float spectrumChannelToEnergy(int channel_number, float a0, float a1, float a2)`

**Purpose:** Helper function which returns the channel energy in keV.
**Parameters:**

- `int channel_number`: channel number [0 .. 1023]
- `float a0`: energy calibration coefficient a0

- `float a1`: energy calibration coefficient a1
- `float a2`: energy calibration coefficient a2

**Return Value:** Float value containing the channel energy in keV.
**Example:**

```
#include <RadiaCode.h>

int i;
float energy;

// Get the channel energy in keV of channel i
energy = spectrumChannelToEnergy(i, 1.352235f, 2.381880f, 0.000348f);
Serial.printf("Energy [keV] of channel %d: %f\n", i, energy);
```

**Notes:**

- This is just a helper function used for converting a channel number into a channel energy.
- The formula `channel energy (keV) = a0 + a1*channel + a2*channel^2` is used internally for the conversion.

## Initialization and Connection methods

### `RadiaCode(const char* bluetooth_mac, bool ignore_firmware_compatibility_check)`

**Purpose:** Constructor that initializes communication with a RadiaCode device over Bluetooth.
**Parameters:**

- `const char* bluetooth_mac`: Char pointer to the Bluetooth MAC (e.g. "52:43:06:70:24:67") of the connected RadiaCode device.
- `bool ignore_firmware_compatibility_check`: Ignore checking the firmware compatibility. Default: false

**Return Value:** None (constructor)
**Example:**

```
#include <RadiaCode.h>

// Replace with your device's MAC address
const char* bluetoothMac = "52:43:06:70:24:67";

// Create RadiaCode instance using Bluetooth (used always in following example code)
RadiaCode* radiacode;
radiacode = new RadiaCode(bluetoothMac);
```

**Notes:** The constructor automatically initializes the exchange protocol with the device.

### Device Information methods

`uint32_t deviceStatus(void)`

**Purpose:** Returns the raw status flags from the device.
**Parameters:** None
**Return Value:** uint32_t containing the device status flags.
**Example:**

```
#include <RadiaCode.h>

// Get the device status flags
uint32_t statusFlags = radiacode->deviceStatus();
Serial.print("Status flags (hex): 0x");
Serial.println(statusFlags, HEX);
```

**Notes:** The status flags represent various device conditions. The assigned meanings of the specific bits is not yet defined.

`String fwSignature(void)`

**Purpose:** Returns the firmware signature including the FileName and IdString from the device.
**Parameters:** None
**Return Value:** String containing firmware signature.
**Example:**

```
#include <RadiaCode.h>

String signature;

// Get the firmware signature string
signature = radiacode->fwSignature();
Serial.print("Firmware signature: ");
Serial.println(signature);
```

**Notes:** The signature string e.g. has following format: `Signature: 5AE742AD, FileName="rc-103.bin", IdString="RadiaCode RC-103".`

`std::tuple<int, int, String, int, int, String> fwVersion(void)`

**Purpose:** Returns the bootloader and firmware version from the device.
**Parameters:** None
**Return Value:** Tuple containing bootloader and firmware version (major-, minor- and patch-version) as integers, including the release date of the versions as Strings.
**Example:**

```
#include <RadiaCode.h>

int boot_major, boot_minor, target_major, target_minor;
String boot_date, target_date;

// Get bootloader and firmware version with date
auto version = radiacode->fwVersion();
boot_major = std::get<0>(version);
```

```
boot_minor = std::get<1>(version);
boot_date = std::get<2>(version);
target_major = std::get<3>(version);
target_minor = std::get<4>(version);
target_date = std::get<5>(version);

// Print bootloader version and date
Serial.print(boot_major);
Serial.print(".");
Serial.printf("%02d", boot_minor);
Serial.print(" (");
Serial.print(boot_date);
Serial.println(")");

// Print firmware version and date
Serial.print(target_major);
Serial.print(".");
Serial.printf("%02d", target_minor);
Serial.print(" (");
Serial.print(target_date);
Serial.println(")");
```

**Notes:** The date string e.g. has following format: `Jul 7 2025 11:20:30`

### `String hwSerialNumber(void)`

**Purpose:** Returns the hardware serial number (a more complex identifier) of the connected device.
**Parameters:** None
**Return Value:** String containing the hardware serial number as hyphenated hexadecimal groups.
**Example:**

```
#include <RadiaCode.h>
```

```
String hwSerial = radiacode->hwSerialNumber();
Serial.print("Hardware serial number: ");
Serial.println(hwSerial);
```

**Notes:** The hardware serial number is typically displayed as hexadecimal groups separated by hyphens, e.g., "00374129-35385012-20973021".

### `String serialNumber(void)`

**Purpose:** Returns the serial number of the connected RadiaCode device as a string.
**Parameters:** None
**Return Value:** String containing the device serial number (e.g., "RC-103-123456").
**Example:**

```
#include <RadiaCode.h>

String serial = radiacode->serialNumber();
Serial.print("Device serial number: ");
Serial.println(serial);
```

**Notes:** This retrieves the user-friendly serial number string from the device.

### `String configuration(void)`

**Purpose:** Returns the configuration of the connected RadiaCode device as a string.
**Parameters:** None
**Return Value:** String containing the device configuration.
**Example:**

```
#include <RadiaCode.h>
```

```
String config = radiacode->configuration();
Serial.print("Configuration: ");
Serial.println(config);
```

**Notes:** One possible part of the configuration string is e.g. the spectrum format version (e.g., "SpecFormatVersion="). This function is also called by the constructor to store the spectrum format version globaly.

### `String textMessage(void)`

**Purpose:** Returns the text message of the connected RadiaCode device as a string.
**Parameters:** None
**Return Value:** String containing the text message.
**Example:**

```
#include <RadiaCode.h>

String text = radiacode->textMessage();
Serial.print("Text message: ");
Serial.println(text);
```

**Notes:** This retrieves a text message string from the device (e.g., "-147, BLE: connection timeout expired -68, BLE: client connected")

### `String commands(void)`

**Purpose:** Returns the SFR file content of the connected RadiaCode device as a string.
**Parameters:** None
**Return Value:** String containing the SFR file content.
**Example:**

```
#include <RadiaCode.h>

String cmd = radiacode->commands();
Serial.print("SFR file content: ");
Serial.println(cmd);
```

**Notes:** This retrieves the SFR file content from the device (e.g., tbd)

### Time and configuration methods

**`void setLocalTime(uint8_t day, uint8_t month, uint16_t year, uint8_t second, uint8_t minute, uint8_t hour)`**

**Purpose:** Sets the device's internal clock.
**Parameters:**

- `uint8_t day`: Day of month (1-31)
- `uint8_t month`: Month (1-12)
- `uint16_t year`: Full year (e.g., 2023)
- `uint8_t second`: Seconds (0-59)
- `uint8_t minute`: Minutes (0-59)
- `uint8_t hour`: Hours (0-23, 24-hour format)

**Return Value:** None
**Example:**

```
#include <RadiaCode.h>

// Set device time to March 15, 2023, 14:30:45
radiacode->setLocalTime(15, 3, 2023, 45, 30, 14);
```

**Notes:** The device uses this time for timestamping measurements and logs. This function is also called by the constructor to set the current time in the device.

`void deviceTime(uint32_t v)`

**Purpose:** Sets the device's internal timer counter.
**Parameters:**

- `uint32_t v`: 4 byte value

**Return Value:** None
**Example:**

```
#include <RadiaCode.h>

// Reset device timer counter to 0
radiacode->deviceTime(0);
```

**Notes:** The device uses this internal timer counter for measurements. This function is also called by the constructor to reset the internal timer counter in the device. The exact purpose of this internal timer counter needs tbd.

### Data acquisition methods

`std::vector<DataItem*> dataBuf(void)`

**Purpose:** Read the device's data buffer. The data buffer contains each time when read an accumulated number of items containing new data.
**Parameters:** None
**Return Value:** Vector containg data items (std::vector<DataItem*>) of different item types:

- `DataItemType type`: realtime data type (TYPE_REAL_TIME_DATA)
  - `float count_rate`: count rate in CPS or CPM
  - `float dose_rate`: dose rate (multiplied with a factor of 10000 gets μSv/h)
- `DataItemType type`: raw data type (TYPE_RAW_DATA)
  - `float count_rate`: count rate in CPS or CPM
  - `float dose_rate`: dose rate (multiplied with a factor of 10000 gets μSv/h)

- `DataItemType type`: rare data type (TYPE_RARE_DATA)
    - `float temperature`: ambient temperature in °C or °F
    - `float charge_level`: charge level of the battery in percent

**Example:**

```
#include <RadiaCode.h>

static float countRate = 0.0f;
static float doseRate = 0.0f;
static float temperature = 0.0f;
static float batteryLevel = 0.0f;

// Read the data buffer
auto data = radiacode->dataBuf();

// Process data buffer
for (size_t i = 0; i < data.size(); i++)
{
    DataItem* item = data[i];
    if (item != nullptr)
    {
        switch (item->type)
        {
            case TYPE_REAL_TIME_DATA:
            {
                RealTimeData* rtData = (RealTimeData*)item;
                if (rtData != nullptr)
                {
                    countRate = rtData->count_rate;
                    doseRate = rtData->dose_rate;
                }
                break;
            }
            case TYPE_RAW_DATA:
```

```
            {
                RawData* rawData = (RawData*)item;
                if (rawData != nullptr)
                {
                    // Use dose rate from raw data if real-time data doesn't have it
                    if (doseRate == 0.0f && rawData->dose_rate > 0.0f)
                    {
                        doseRate = rawData->dose_rate;
                    }
                }
                break;
            }
            case TYPE_RARE_DATA:
            {
                RareData* rareData = (RareData*)item;
                if (rareData != nullptr)
                {
                    temperature = rareData->temperature;
                    batteryLevel = rareData->charge_level;
                }
                break;
            }
            case TYPE_DOSE_RATE_DB:
            default:
                break;
        }
    }
}

// Clean up data objects
for (size_t i = 0; i < data.size(); i++)
{
    if (data[i] != nullptr)
    {
        delete data[i];
        data[i] = nullptr;
```

```
    }
}
data.clear();

Serial.print("Count rate: ");
Serial.print(countRate);
Serial.println(" CPS");

Serial.print("Dose rate: ");
Serial.print(doseRate * 10000.0f);
Serial.println(" µSv/h");

Serial.print("Temperature: ");
Serial.print(temperature);
Serial.println(" °C");

Serial.print("Battery level: ");
Serial.print(batteryLevel);
Serial.println("%");
```

`Spectrum spectrum(void)`

**Purpose:** Reads the current spectrum data from the device.
**Parameters:** None
**Return Value:** Spectrum object containing the spectrum data:

- `uint32_t duration_sec`: duration in seconds since spectrumReset()
- `float a0`: energy calibration coefficient a0 (from spectrum header)
- `float a1`: energy calibration coefficient a1 (from spectrum header)
- `float a2`: energy calibration coefficient a2 (from spectrum header)
- `static uint32_t shared_counts[MAX_CHANNELS]`: number of counts of each channel (in total 1024 channels).

**Example:**

```
#include <RadiaCode.h>

Spectrum spectrum = radiacode->spectrum();
int channelCount = spectrum.size();

if (channelCount > 0)
{
    Serial.print("Duration: ");
    Serial.print(spectrum.duration_sec);
    Serial.println(" seconds");

    Serial.print("Channels: ");
    Serial.println(channelCount);

    // Find maximum count for scaling
    maxCount = 0;
    for (int i = 0; i < channelCount; i++)
    {
        uint32_t count = spectrum.at(i);
        if (count > maxCount)
        {
            maxCount = count;
        }
    }

    Serial.print("Maximum count: ");
    Serial.println(maxCount);

    // Print table header
    Serial.println("\nChannel\tEnergy (keV)\tCounts\tGraph");
    Serial.println("---------------------------------------");

    // Print spectrum data (every 10th channel to keep output manageable)
    for (int i = 0; i < channelCount; i += 10)
    {
```

```
        // Use the existing spectrumChannelToEnergy function from the library
        // a0, a1, a2 can be retrieved from spectrum header or via energyCalib() API
        float energy = spectrumChannelToEnergy(i, a0, a1, a2);
        uint32_t counts = spectrum.at(i);

        Serial.print(i);
        Serial.print("\t");
        Serial.print(energy, 2);
        Serial.print("\t\t");
        Serial.print(counts);
        Serial.print("\t");

        // Print simple ASCII graph
        int barLength = (maxCount > 0) ? (counts * 250) / maxCount : 0;
        for (int j = 0; j < barLength; j++)
        {
            Serial.print("#");
        }
        Serial.println();
    }
}
```

**Notes:**

- The format depends on the firmware version, which can be checked with `getSpectrumFormatVersion()`
- Processing the channel data requires understanding the specific format version

### `Spectrum spectrumAccum(void)`

**Purpose:** Reads the accumulated spectrum data from the device.
**Parameters:** None
**Return Value:** Spectrum object containing the spectrum data:

- `uint32_t duration_sec`: duration in seconds since spectrumReset()

- `float a0`: energy calibration coefficient a0 (from spectrum header)
- `float a1`: energy calibration coefficient a1 (from spectrum header)
- `float a2`: energy calibration coefficient a2 (from spectrum header)
- `static uint32_t shared_counts[MAX_CHANNELS]`: number of counts of each channel (in total 1024 channels).

**Example:**

```
#include <RadiaCode.h>

Spectrum spectrum = radiacode->spectrumAccum();

// ... see example of spectrum() API
```

**Notes:**

- The format depends on the firmware version, which can be checked with `getSpectrumFormatVersion()`
- Processing the channel data requires understanding the specific format version

### Reset methods

`void doseReset(void)`

**Purpose:** Resets the accumulated dose to 0 [Sv or R].
**Parameters:** None
**Return Value:** None
**Example:**

```
#include <RadiaCode.h>

// Reset the dose to 0
radiacode->doseReset();
```

`void spectrumReset(void)`

**Purpose:** Resets the accumulated spectrum to 0. The count values of each channel are set to 0.
**Parameters:** None
**Return Value:** None
**Example:**

```
#include <RadiaCode.h>

// Reset the spectrum to 0
radiacode->spectrumReset();
```

## Calibration methods

`std::vector<float> energyCalib(void)`

**Purpose:** Read the device's energy calibration coefficients. The energy coefficients are used to convert the channel number into the channel enery.
**Parameters:** None
**Return Value:** Vector containg the 3 energy coefficients:

- `float a0`: energy calibration coefficient a0 (from the device)
- `float a1`: energy calibration coefficient a1 (from the device)
- `float a2`: energy calibration coefficient a2 (from the device)

**Example:**

```
#include <RadiaCode.h>

std::vector<float> calib;
calib = radiacode->energyCalib();

if (calib.size() == 3)
```

```
{
    Serial.println("Energy calibration coefficients:");
    Serial.printf("a0: %.6f a1: %.6f a2: %.6f\n", calib[0], calib[1], calib[2]);
}
```

**Notes:**

- Each channel number [0..1023] is converted into the channel energy by following formula: channel energy (keV) = a0 + a1*channel* + *a2*channel^2

### `void setEnergyCalib(float a0, float a1, float a2)`

**Purpose:** Stores new energy calibration coefficients in the device's. The energy coefficients are used to convert the channel number into the channel enery.
**Parameters:** 3 energy coefficients:

- `float a0`: energy calibration coefficient a0
- `float a1`: energy calibration coefficient a1
- `float a2`: energy calibration coefficient a2

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

radiacode->setEnergyCalib(1.352235f, 2.381880f, 0.000348f);
```

**Notes:**

- Each channel number [0..1023] is converted into the channel energy by following formula: channel energy (keV) = a0 + a1*channel* + *a2*channel^2
- **Attention**: use this API carefully, as the energy coefficients are overwritten by the new ones and you can decalibrate your device. Use the tool `calibration.py` in the tools folder, which helps you to determine the new energy coefficients a0, a1 and a2.

## Debug methods

`uint8_t getSpectrumFormatVersion(void)`

**Purpose:** Read the device's version of the spectrum format.
**Parameters:** None
**Return Value:** uint8_t containg the spectrum format version.
**Example:**

```
#include <RadiaCode.h>

uint8_t version;

version = radiacode->getSpectrumFormatVersion();
Serial.print("Detected spectrum format version: ");
Serial.println(version);
```

**Notes:**

- The spectrum format version is read from the configuration string during the constructor and stored to be read with this API.

## Device settings methods

`void setLanguage(const char* lang)`

**Purpose:** Sets the device interface language.
**Parameters:** The language:

- `const char* lang`: Language code string. Currently only English "en" and Russian "ru" are supported.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

radiacode->setLanguage("en");
Serial.println("Language set to English.");
```

**Notes:** The supported languages depend on the device firmware.

`void setDeviceOn(bool on)`

**Purpose:** Turns the device on or off.
**Parameters:** The power state:

- `bool on`: true to turn the device on, false to turn it off.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

radiacode->setDeviceOn(false);
Serial.println("Device turned off.");
```

**Notes:** When turned off, the device enters a low-power state but can still respond to commands.

`void setSoundOn(bool on)`

**Purpose:** Enables or disables sound on the device.
**Parameters:** The main sound state:

- `bool on`: true to enable sound, false to disable it.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// Sound On/Off (main switch)
radiacode->setSoundOn(true);
Serial.println("Sound enabled.");
```

**Notes:** This controls all sound output from the device, including alarms and button feedback.

`void setVibroOn(bool on)`

**Purpose:** Enables or disables vibration feedback on the device.
**Parameters:** The main vibration state:

- `bool on`: true to enable vibration, false to disable it.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// Vibration On/Off (main switch)
radiacode->setVibroOn(true);
Serial.println("Vibration enabled.");
```

**Notes:** This controls the vibration motor for tactile feedback during alarms and interactions.

`void setLightOn(bool on)`

**Purpose:** Controls the device's backlight or display illumination.
**Parameters:** The main light state:

- `bool on`: true to turn the light on, false to turn it off.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// Light On/Off (main switch)
radiacode->setLightOn(true);
Serial.println("Light enabled.");
```

**Notes:** This controls the display backlighting or other illumination features.

### `void setDeviceCtrl(DEV_CTRL ctrl_flags)`

**Purpose:** Sets detailed device control flags.
**Parameters:** The device control flags are organized as a bit field:

- `DEV_CTRL::PWR`: Power the device (0x00000001)
- `DEV_CTRL::SOUND`: Enable sound (0x00000004)
- `DEV_CTRL::LIGHT`: Enable light (0x00000008)
- `DEV_CTRL::VIBRO`: Enable vibration (0x00000010)

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>
```

```
radiacode->setDeviceCtrl((DEV_CTRL)(DEV_CTRL::PWR | DEV_CTRL::LIGHT | DEV_CTRL::SOUND));
Serial.println("Power, Light, Sound enabled, Vibration disabled.");
```

**Notes:** This controls the power state, sound output, vibration feedback and the illumination features in one command.

`void setSoundCtrl(CTRL ctrl_flags)`

**Purpose:** Sets detailed sound control flags.
**Parameters:** The sound control flags are organized as a bit field:

- `CTRL::BUTTONS`: Button press sounds (0x00000001)
- `CTRL::CLICKS`: Click sounds for radiation events (0x00000002)
- `CTRL::DOSE_RATE_ALARM_1`: Dose rate alarm 1 sounds (0x00000004)
- `CTRL::DOSE_RATE_ALARM_2`: Dose rate alarm 2 sounds (0x00000008)
- `CTRL::DOSE_RATE_OUT_OF_SCALE`: Dose rate out of scale sounds (0x00000010)
- `CTRL::DOSE_ALARM_1`: Dose alarm 1 sounds (0x00000020)
- `CTRL::DOSE_ALARM_2`: Dose alarm 2 sounds (0x00000040)
- `CTRL::DOSE_OUT_OF_SCALE`: Dose out of scale sounds (0x00000080)
- `CTRL::CONNECTION`: Connection notification sounds (0x00000100)
- `CTRL::POWER`: Power notification sounds (0x00000200)
- `CTRL::COUNT_RATE_ALARM_1`: Count rate alarm 1 sounds (0x00000400)
- `CTRL::COUNT_RATE_ALARM_2`: Count rate alarm 2 sounds (0x00000800)
- `CTRL::COUNT_RATE_OUT_OF_SCALE`: Count rate out of scale sounds (0x00001000)

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// enable only button and click sounds
```

```
radiacode->setSoundCtrl((CTRL)(CTRL::BUTTONS | CTRL::CLICKS));
Serial.println("Sound control flags set.");
```

**Notes:** This controls all sound types in one command.

`void setVibroCtrl(CTRL ctrl_flags)`

**Purpose:** Sets detailed vibration control flags.
**Parameters:** The vibration control flags are organized as a bit field:

- `CTRL::BUTTONS`: Button press vibrations (0x00000001)
- `CTRL::CLICKS`: Not supported! (0x00000002)
- `CTRL::DOSE_RATE_ALARM_1`: Dose rate alarm 1 vibrations (0x00000004)
- `CTRL::DOSE_RATE_ALARM_2`: Dose rate alarm 2 vibrations (0x00000008)
- `CTRL::DOSE_RATE_OUT_OF_SCALE`: Dose rate out of scale vibrations (0x00000010)
- `CTRL::DOSE_ALARM_1`: Dose alarm 1 vibrations (0x00000020)
- `CTRL::DOSE_ALARM_2`: Dose alarm 2 vibrations (0x00000040)
- `CTRL::DOSE_OUT_OF_SCALE`: Dose out of scale vibrations (0x00000080)
- `CTRL::CONNECTION`: Not supported! (0x00000100)
- `CTRL::POWER`: Not supported! (0x00000200)
- `CTRL::COUNT_RATE_ALARM_1`: Count rate alarm 1 vibrations (0x00000400)
- `CTRL::COUNT_RATE_ALARM_2`: Count rate alarm 2 vibrations (0x00000800)
- `CTRL::COUNT_RATE_OUT_OF_SCALE`: Count rate out of scale vibrations (0x00001000)

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// enable only button vibrations
```

```
radiacode->setVibroCtrl((CTRL)(CTRL::BUTTONS));
Serial.println("Vibration control flags set.");
```

**Notes:**

- Uses the same CTRL enumeration as setSoundCtrl, but applies to vibration feedback.
- Vibration on click events, connection events and power events is not supported.

### `void setDisplayCtrl(DISPLAY_CTRL ctrl_flags)`

**Purpose:** Sets detailed display control flags.
**Parameters:** The display control flags are organized as a bit field:

- `DISPLAY_CTRL::BACKLT_OFF`: Backlight disabled (0x00000000)
- `DISPLAY_CTRL::BACKLT_ON_BY_BUTTON`: Backlight enabled when pressing any button for the time set with setDisplayOffTime() (0x00000004)
- `DISPLAY_CTRL::BACKLT_ON_AUTO`: Backlight enabled when pressing any button and depending on the ambient illumination (0x00000008)

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// enable automatic backlight control
radiacode->setDisplayCtrl(DISPLAY_CTRL::BACKLT_ON_AUTO);
Serial.println("Display control set to automatic backlight control.");
```

**Notes:** This controls all backlight types in one command.

### `void setDisplayOffTime(uint8_t seconds)`

**Purpose:** Sets the display auto-off timeout.
**Parameters:**

- `uint8_t seconds`: Time in seconds before the display automatically turns off. Only limited number of values is allowed (see notes).

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set the display auto-off timout to 10 seconds
radiacode->setDisplayOffTime(10);
Serial.println("Display off time set to 10 seconds.");
```

**Notes:** Only values of 5, 10, 15 and 30 seconds are supported. Other values are rejected.

`void setDisplayBrightness(uint8_t brightness)`

**Purpose:** Sets the display brightness level.
**Parameters:**

- `uint8_t brightness`: Brightness level from 0 (off) to 9 (maximum).

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set brightness level of the display to 5
radiacode->setDisplayBrightness(5);
Serial.println("Display brightness set to 5.");
```

**Notes:** Values bigger than 9 are rejected.

`void setDisplayDirection(DisplayDirection direction)`

**Purpose:** Sets the display orientation.
**Parameters:** The display orientation type is an enumeration:

- `DisplayDirection::AUTO`: Automatic orientation where the device detects the orientation depending on the built-in accelerometer.
- `DisplayDirection::RIGHT`: Buttons can be controlled with the right hand.
- `DisplayDirection::LEFT`: Buttons can be controlled with the left hand.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set automatic orientation
radiacode->setDisplayDirection(DisplayDirection::AUTO);
Serial.println("Display direction set to AUTO.");
```

`void setMeasurementUnit(MeasurementUnits unit)`

**Purpose:** Sets the measurement unit.
**Parameters:** The measurement unit type is an enumeration:

- `MeasurementUnits::ROENTGEN`: Unit is Roentgen [R].
- `MeasurementUnits::SIEVERT`: Unit is Sievert [Sv].

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set measurement unit to Sievert
radiacode->setMeasurementUnit(MeasurementUnits::SIEVERT);
Serial.println("Measurement unit set to Sievert.");
```

`void setCountRateUnit(CountRateUnits unit)`

**Purpose:** Sets the count rate unit.
**Parameters:** The count rate unit type is an enumeration:

- `CountRateUnits::CPS`: Unit is counts per second [cps].
- `CountRateUnits::CPM`: Unit is counts per minute [cpm].

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set count rate unit to CPS
radiacode->setCountRateUnit(CountRateUnits::CPS);
Serial.println("Count rate unit set to CPS.");
```

`void setTemperatureUnit(TemperatureUnits unit)`

**Purpose:** Sets the temperature unit.
**Parameters:** The temperature unit type is an enumeration:

- `TemperatureUnits::CELSIUS`: Unit is degrees Celcius [°C].
- `TemperatureUnits::FAHRENHEIT`: Unit is degrees Fahrenheit [°F].

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set temperature unit to degrees Celsius
radiacode->setTemperatureUnit(TemperatureUnits::CELSIUS);
Serial.println("Temperature unit set to degrees Celsius.");
```

## Alarm methods

`void setAlarmSignalMode(AlarmSignalMode mode)`

**Purpose:** Sets the alarm signal mode.
**Parameters:** The alarm signal mode type is an enumeration:

- `AlarmSignalMode::CONTINUOUSLY`: The audio alarm sounds continuously when there is an alarm.
- `AlarmSignalMode::ONCE`: the audio alarm sounds once when there is an alarm.

**Return Value:** None

**Example:**

```
#include <RadiaCode.h>

// set alarm signal mode to once
radiacode->setAlarmSignalMode(AlarmSignalMode::ONCE);
Serial.println("Alarm signal mode set to once.");
```

`AlarmLimits getAlarmLimits(void)`

**Purpose:** Gets the current alarm threshold settings from the device.
**Parameters:** None

**Return Value:** AlarmLimits structure containing following alarm threshold values:

- `float l1_count_rate`: Level 1 for count rate alarm threshold [cpm; cps].
- `float l2_count_rate`: Level 2 for count rate alarm threshold [cpm; cps].
- `String count_unit`: String for count rate unit ["cpm"; "cps"].
- `float l1_dose_rate`: Level 1 for dose rate alarm threshold [μR/h; μSv/h].
- `float l2_dose_rate`: Level 2 for dose rate alarm threshold [μR/h; μSv/h].
- `float l1_dose`: Level 1 for accumulated dose alarm threshold [R; Sv].
- `float l2_dose`: Level 2 for accumulated dose alarm threshold [R; Sv].
- `String dose_unit`: String for dose unit ["R"; "Sv"].

**Example:**

```
#include <RadiaCode.h>

AlarmLimits limits;

// read all alarm limits
limits = radiacode->getAlarmLimits();

Serial.print("l1_count_rate: ");
Serial.println(limits.l1_count_rate);
Serial.print("l2_count_rate: ");
Serial.println(limits.l2_count_rate);
Serial.print("count_unit: ");
Serial.println(limits.count_unit);
Serial.print("l1_dose_rate: ");
Serial.println(limits.l1_dose_rate);
Serial.print("l2_dose_rate: ");
Serial.println(limits.l2_dose_rate);
Serial.print("l1_dose: ");
```

```
  Serial.println(limits.l1_dose);
  Serial.print("l2_dose: ");
  Serial.println(limits.l2_dose);
  Serial.print("dose_unit: ");
  Serial.println(limits.dose_unit);
```

**bool setAlarmLimits(float l1_count_rate, float l2_count_rate, float l1_dose_rate, float l2_dose_rate, float l1_dose, float l2_dose, bool dose_unit_sv, bool count_unit_cpm)**

**Purpose:** Sets alarm thresholds for various radiation measurements.
**Parameters:**

- `float l1_count_rate`: Level 1 count rate alarm threshold [cpm; cps] (<0.0f to keep current value)
- `float l2_count_rate`: Level 2 count rate alarm threshold [cpm; cps] (<0.0f to keep current value)
- `float l1_dose_rate`: Level 1 dose rate alarm threshold [μR/h; μSv/h] (<0.0f to keep current value)
- `float l2_dose_rate`: Level 2 dose rate alarm threshold [μR/h; μSv/h] (<0.0f to keep current value)
- `float l1_dose`: Level 1 accumulated dose alarm threshold [R; Sv] (<0.0f to keep current value)
- `float l2_dose`: Level 2 accumulated dose alarm threshold [R; Sv] (<0.0f to keep current value)
- `bool dose_unit_sv`: true to use Sievert unit (Sv), false to use Roentgen unit (R)
- `bool count_unit_cpm`: true to use counts per minute (cpm), false to use counts per second (cps)

**Return Value:** Boolean, true if all alarm limits (at least one) were set successfully, else false.
**Example:**

```
#include <RadiaCode.h>

bool result;

result = radiacode->setAlarmLimits(20.0f, 60.0f, 0.4f, 1.2f, 9.99f, 9.99f, true, false);
if (!result)
{
    Serial.println("Failed to set alarm limits!");
```

```
    }
    else
    {
        Serial.println("Alarm limits set.");
    }
```

**Notes:**

- Pass value less than 0.0f for any threshold parameter you don't want to change
- The units parameters (dose_unit_sv and count_unit_cpm) always get set, regardless of other values
- Level 2 alarms should typically be set higher than Level 1 alarms
- All changes are made in a single transaction for consistency

### Direct sensor reading methods

#### `float getTemperature(void)`

**Purpose:** Gets the current device temperature.
**Parameters:** None
**Return Value:** Float value containing the temperature in degrees Celsius [°C].
**Example:**

```
#include <RadiaCode.h>

float temperature;

temperature = radiacode->getTemperature();
Serial.print("Temperature: ");
Serial.print(temperature);
Serial.println(" °C");
```

**Notes:** This is a convenience method that reads the TEMP_degC VSFR and properly converts it to a float.