microHAM Device Protocol

8. December 2006

Table of Contents

Introduction

This document describes the serial communication protocol used to upgrade, configure and control microHAM devices by a computer. This applies only to serial devices connected via RS232 (non USB). Currently these are micro BAND DECODER and micro STACK MAX.
As a part of this protocol here are included some related themes, such as firmware file format or the detailed description of the configuration parameters.

The document corresponds to versions listed bellow.

general device bootloader	`cbl v2.1, v3.0`
firmware file format	`v2.0`
micro Band Decoder firmware	`uBD v4.5`
micro Stack Max firmware	`uSM v2.7`
microHAM Device Configurator	`uconf v4.1`

Any details can be consulted with Martin by e-mail: bujdak@microham.com.

MicroHAM device software

MicroHAM device software contains two parts, the application firmware and the bootloader. The bootloader is common for all device types. A customer cannot change it. The application firmware can be upgraded using *microHAM device protocol*.
The internal processor flag BSB determines what software part is started at the device power up. If BSB is zero, the application firmware will be started and if BSB is non-zero, the bootloader will be started. This non-zero state is used during firmware upgrade because an incomplete application firmware cannot be started.

All applications store their configuration in EEPROM. It is possible to access the configuration from a computer through *microHAM device protocol*.

Note: MicroHAM supplies a standard tool the **microHAM Device Configurator** running under MS Windows that allows to upgrade and configure all microHAM serial devices.

Communication protocol

Devices are connected to the computer through RS232 interface (COM port). The computer plays the role of the master and the device of the slave. Communication is organized to talks. The computer sends a query packet to the device and waits to an answer packet from it. The device can return an error answer to any query instead of the expected answer.

Serial port settings: 19200 bps, 8 bit data, no parity, 1 stop bit

General packet format

All packets starts with the command prefix 0xEE followed by the *Command*. Commands from ranges of 0xC1-0xCF and 0xA1-0xAF are reserved for the bootloader. Other commands are dedicated for the applications. The next byte Length determines the length of Content only. It can be a zero for some packets. The packet is finished by a check sum the 16-bits arithmetic sum of Command, Length and of all bytes of Content.

The code **0xEE** is never used as a command and it must be **duplicated** inside the packet. Both the computer and the band decoder must duplicate all transmitted 0xEE bytes inside the Length, Content or CheckSum of packet. Leading prefix 0xEE before the command must not be duplicated. And vice-versa when a couple of 0xEE bytes is received, it must be placed to the receive buffer as the single byte 0xEE. When the single byte 0xEE is received, it must be interpreted as the start of a new packet and the following byte as a Command.

Note: All multi byte parameters (word, dword) are little endian, where LSB is first (Intel order), if not other stated.

***General query**, from computer to device*

```
byte   0xEE
byte   QueryCommand
byte   Length
byte   Content[Length]
word   CheckSum
```

***General answer**, from device to computer*

```
byte   0xEE
byte   AnswerCommand
byte   Length
byte   Content[Length]
word   CheckSum
```

***General error answer**, from device to computer*

```
byte   0xEE
byte   ErrorAnswerCommand
byte   0x00
word   CheckSum
```

List of commands

Special bytes

0xEE	*COMMAND_PREFIX*	special byte: command prefix

Bootloader commands

0xC4	*CBL_WR_EEPROM*	query: write data block to EEPROM
0xC3	*CBL_GET_VER*	query: get versions and status
0xC2	*CBL_END_PROG*	query: end of programming
0xC1	*CBL_WR_FLASH*	query: write data block to flash
0xAF	*CBL_CHECKSUM_ER*	error answer: checksum error
0xAE	*CBL_UNDEF_COM*	error answer: undefined command
0xAD	*CBL_WR_NOT_AUTH*	error answer: data block not authorized
0xAC	*CBL_WR_VERIF_FAULT*	error answer: write data verification fault
0xAB	*CBL_WR_FAULT*	error answer: write data fault
0xAA	*CBL_WRONG_LENGTH*	error answer: wrong length for this command
0xA9	*CBL_LOW_SECUR*	error answer: low security level
0xA8	*CBL_WR_PROT_AREA*	error answer: write attempt to protected area
0xA4	*CBL_WR_EEPROM_OK*	answer: write data block to EEPROM OK
0xA3	*CBL_GET_VER_ANSWER*	answer: versions and status
0xA2	*CBL_END_PROG_OK*	answer: end of programming OK
0xA1	*CBL_WR_FLASH_OK*	answer: write data block to flash OK

Common application commands

0xD3	*GET_VER*	query: get versions and status
0xD2	*RESTART_APPL*	query: restart application
0xD1	*WRITE_CONF*	query: write configuration
0xD0	*READ_CONF*	query: read configuration
0xC0	*CBL_START_BOOT*	query: start bootloader
0xBF	*CHECKSUM_ER*	error answer: checksum fault
0xBE	*UNDEF_COM*	error answer: undefined command
0xBD	*WRITE_VERIF_FAULT*	error answer: write configuration verification fault
0xB3	*GET_VER_ANSWER*	answer: versions and status
0xB2	*RESTART_APPL_OK*	answer: restart application OK
0xB1	*WRITE_CONF_OK*	answer: write configuration OK
0xB0	*READ_CONF_ANSWER*	answer: configuration
0xA0	*CBL_START_BOOT_OK*	answer: start bootloader OK

Band Decoder specific commands

0xD4	*END_OF_PC2CPU*	query: end of configuration mode
0xB4	*END_OF_PC2CPU_OK*	answer: end of configuration mode OK

Stack Max specific commands

0xD6	*USM_GET_STATUS*	query: get stack status
0xD5	*USM_EVENT*	query: stack event
0xB6	*USM_GET_STATUS_ANSWER*	answer: stack status
0xB5	*USM_EVENT_OK*	answer: stack event OK

Bootloader protocol

This part of the protocol is supported by the bootloader only and provides the way to upgrade the device's firmware. To use the right technique see the section *How to upgrade the firmware*.

Get version and status

After this query the bootloader returns extended *version information*.
Last four bytes are special hardware registers. Only BSB is important for upgrade process. It's meaning is described in chapter *MicroHAM devices*.

Query: get versions and status

```
byte   0xEE
byte   0xC3 (CBL_GET_VER)
byte   0x00
word   0x00C3
```

Answer: versions and status

```
byte   0xEE
byte   0xA3 (CBL_GET_VER_ANSWER)
byte   0x11
byte   cbl_ver_minor
byte   cbl_ver_major
byte   product_type
byte   hardware_version
byte   mechanical_version
word   serial_number
byte   reserved    ; always 0xFF
byte   appl_product_type
byte   appl_min_hardware_version
byte   appl_min_mechanical_version
byte   appl_ver_minor
byte   appl_ver_major
byte   HSB
byte   SBV
byte   BSB
byte   SSB
word   CheckSum
```

List of possible error answer commands

```
0xAE CBL_UNDEF_COM                undefined command
0xAA CBL_WRONG_LENGTH             wrong length for this command
0xAF CBL_CHECKSUM_ER              checksum error
```

Write data block to flash

After this query bootloader set BSB and writes a block of data to flash memory. The block has a format of the *firmware file* flash data block.

Query: write data block to flash

```
  byte   0xEE
  byte   0xC1 (CBL_WR_FLASH)
  byte   0x86
  byte   FlashDataBlockContent[0x86]
  word   CheckSum
```

Answer: write data block to flash OK

```
  byte   0xEE
  byte   0xA1 (CBL_WR_FLASH_OK)
  byte   0x00
  word   0x00A1
```

List of possible error answer commands

```
  0xAE CBL_UNDEF_COM                 undefined command
  0xAA CBL_WRONG_LENGTH              wrong length for this command
  0xAF CBL_CHECKSUM_ER               checksum error
  0xAD CBL_WR_NOT_AUTH               data block not authorized
  0xAC CBL_WR_VERIF_FAULT            write data block verification fault
  0xAB CBL_WR_FAULT                  write data block fault
  0xA9 CBL_LOW_SECUR                 low security level
  0xA8 CBL_WR_PROT_AREA              write attempt to protected area
```

Write data block to EEPROM

After this query bootloader writes a block of data to EEPROM memory. The block has a format of the *firmware file* EEPROM data block.
Note: The bootloader supports the writing to EEPROM since v2.0.

Query: write data block to eeprom

```
byte   0xEE
byte   0xC4 (CBL_WR_EEPROM)
byte   Length
byte   EepromDataBlockContent[Length]
word   CheckSum
```

Answer: write data block to eeprom OK

```
byte   0xEE
byte   0xA4 (CBL_WR_EEPROM_OK)
byte   0x00
word   0x00A4
```

List of possible error answer commands

```
0xAE CBL_UNDEF_COM                undefined command
0xAA CBL_WRONG_LENGTH             wrong length for this command
0xAF CBL_CHECKSUM_ER              checksum error
0xAC CBL_WR_VERIF_FAULT           write data verification fault
0xAB CBL_WR_FAULT                 write data fault
```

End of programming

After this query the bootloader clears BSB and restarts the application. This query must be used only if the whole application firmware was successfully written to the flash memory.

Query: end of programming

```
byte   0xEE
byte   0xC2 (CBL_END_PROG)
byte   0x00
word   0x00C2
```

Answer: end of programming OK

```
byte   0xEE
byte   0xA2 (CBL_END_PROG_OK)
byte   0x00
word   0x00A2
```

List of possible error answer commands

```
0xAE CBL_UNDEF_COM                undefined command
0xAA CBL_WRONG_LENGTH             wrong length for this command
0xAF CBL_CHECKSUM_ER              checksum error
```

Application protocol

Following packets are supported by all applications, but not by the bootloader. These packets provide the way to configure and on-line control the devices. To use the right technique see the sections *How to configure the device* and *How to on-line control*.

Get version

After this query the device returns the *version information*.

Query: get version

```
byte   0xEE
byte   0xD3 (GET_VER)
byte   0x00
word   0x00D3
```

Answer: version

```
byte   0xEE
byte   0xB3 (GET_VER_ANSWER)
byte   0x0D
byte   cbl_ver_minor
byte   cbl_ver_major
byte   product_type
byte   hardware_version
byte   mechanical_version
word   serial_number
byte   reserved    ; always 0xFF
byte   appl_product_type
byte   appl_min_hardware_version
byte   appl_min_mechanical_version
byte   appl_ver_minor
byte   appl_ver_major
word   CheckSum
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                  checksum fault
0xBE UNDEF_COM                    undefined command
```

Restart application

After this query device restarts the application. This is needed after the configuration is changed.

Query: restart application

```
byte   0xEE
byte   0xD2 (RESTART_APPL)
byte   0x00
word   0x00D2
```

Answer: restart application OK

```
byte   0xEE
byte   0xB2 (RESTART_APPL_OK)
byte   0x00
word   0x00B2
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                   checksum fault
0xBE UNDEF_COM                     undefined command
```

Write configuration

After this query the device writes a data block of specified size to EEPROM starting at specified address.

Query: write configuration

```
byte   0xEE
byte   0xD1 (WRITE_CONF)
byte   size + 2
word   eeprom_address
byte   data[size]    ; see eeprom memory map
word   CheckSum
```

Answer: write configuration OK

```
byte   0xEE
byte   0xB1 (WRITE_CONF_OK)
byte   0x00
word   0x00B1
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                  checksum fault
0xBE UNDEF_COM                    undefined command
0xBD WRITE_VERIF_FAULT            write configuration verification
                                  fault
```

Read configuration

After this query the device returns EEPROM content of specified size from the specified address.

Query: read configuration

```
byte   0xEE
byte   0xD0 (READ_CONF)
byte   0x03
word   eeprom_address
byte   size
word   CheckSum
```

Answer: configuration

```
byte   0xEE
byte   0xB0 (READ_CONF_ANSWER)
byte   size + 2
word   eeprom_address
byte   data[size]    ; see eeprom memory map
word   CheckSum
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                  checksum fault
0xBE UNDEF_COM                    undefined command
```

Start bootloader

After this query the device starts the bootloader.

Query: start bootloader

```
byte   0xEE
byte   0xC0 (CBL_START_BOOT)
byte   0x00
word   0x00C0
```

Answer: start bootloader OK

```
byte   0xEE
byte   0xA0 (CBL_START_BOOT_OK)
byte   0x00
word   0x00A0
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                   checksum fault
0xBE UNDEF_COM                     undefined command
```

But in the case, when instead of application the bootloader is active, the bootloader returns the answer 0xAE (CBL_UNDEF_COM) or other general error answer command.

Interrogation (Band Decoder only)

This non-standard packet needs to be sent from computer to Band Decoder before any communication within this protocol. It switches Band Decoder to the communication mode. In this mode the band data decoding is intercepted and the communication with computer is allowed. The next packet must follow this interrogation up to 100 milliseconds. If no packet is received from the computer, Band Decoder will switch to the monitoring mode. If once some packet from computer is received within this time, the communication mode can be finished by packet *End of configuration mode* or it timeouts after no packet was received during interval of 3000 milliseconds.

Band Decoder accepts interrogation sequence only in polling mode. If Band Decoder is in the monitoring mode, the interrogation is ignored. In this case the computer must wait at least 10 seconds to the monitoring mode timeouts. In both case no answer from the device is sent.

See also the section *How to communicate with Band Decoder*.

Interrogation sequence

```
byte   0xFF
byte   0xFF
byte   0xFF
byte   0xFF
byte   0xFF
byte   0xFF
byte   0xFF
byte   0xFF
byte   'm'
byte   'i'
byte   'c'
byte   'r'
byte   'o'
byte   'H'
byte   'A'
byte   'M'
byte   'm'
byte   'i'
byte   'c'
byte   'r'
byte   'o'
byte   'H'
byte   'A'
byte   'M'
```

No answer

End of configuration mode (Band Decoder only)

After this query the Band Decoder switches from the communication mode to the polling mode.

Query: end of configuration mode

```
byte   0xEE
byte   0xD4 (END_OF_PC2CPU)
byte   0x00
word   0x00D4
```

Answer: end of configuration mode OK

```
byte   0xEE
byte   0xB4 (END_OF_PC2CPU_OK)
byte   0x00
word   0x00B4
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                   checksum fault
0xBE UNDEF_COM                     undefined command
```

Get stack status (Stack Max only)

After this query the stack max returns a status record.

Query: get stack status

```
byte   0xEE
byte   0xD6 (USM_GET_STATUS)
byte   0x00
word   0x00D6
```

Answer: stack status

```
byte   0xEE
byte   0xB6 (USM_GET_STATUS_ANSWER)
byte   0x08
byte   status_aux
byte   status_bop_index
byte   status_rx
byte   status_tx
byte   status_flags
byte   led_shadow
byte   mix_shadow
byte   out_shadow
word   CheckSum
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                 checksum fault
0xBE UNDEF_COM                   undefined command
```

Status Description

variable	*description*
byte status_aux	aux status and special bits
bit 7	split is active
bits 6-4	unused
bit 3	antenna 4 connected to AUX (subradio)
bit 2	antenna 3 connected to AUX (subradio)
bit 1	antenna 2 connected to AUX (subradio)
bit 0	antenna 1 connected to AUX (subradio)
byte status_bop_index	indices to list of BOP combinations
bits 7-4	index to BOP list for TX state
bits 3-0	index to BOP list for RX state
byte status_rx	antenna selection for RX state (used also for TX if split is inactive)
bit 7	antenna 4 has opposite phase
bit 6	antenna 3 has opposite phase
bit 5	antenna 2 has opposite phase
bit 4	antenna 1 has opposite phase
bit 3	antenna 4 is selected
bit 2	antenna 3 is selected
bit 1	antenna 2 is selected
bit 0	antenna 1 is selected
byte status_tx	antenna selection for TX state (used only if split is active)
bits 7-0	format is the same as status_rx
byte status_flags	aux status and special bits
bits 7-6	unused
bit 5	control of INH via RS232 is enabled
bit 4	control of PTT via RS232 is enabled
bit 3	unused
bit 2	current status of PTT
bit 1	status_aux was changed during active PTT
bit 0	status was changed during active PTT
byte led_shadow	status of dichromatic LEDs under antenna buttons
bit 7	green LED 3
bit 6	red LED 3
bit 5	green LED 2
bit 4	red LED 2
bit 3	green LED 1
bit 2	red LED 1
bit 1	red LED 4
bit 0	green LED 4
byte mix_shadow	status of monochromatic LEDs under special buttons
bits 7-3	unused
bit 2	green LED AUX
bit 1	yellow LED BOP
bit 0	red LED T/R
byte out_shadow	status of outputs
bit 7	out 7
bit 6	out 6
bit 5	out 5
bit 4	out 4
bit 3	out 3
bit 2	out 2
bit 1	out 1
bit 0	out 0

Stack event (Stack Max only)

After this query the stack max respond to event.

Query: stack event

```
byte   0xEE
byte   0xD5 (USM_EVENT)
byte   0x01 + numb_of_params
byte   event_id                 ; event type (see the list bellow)
byte   event_parameter_1        ; parameters (depend on event type)
byte   event_parameter_2
byte   event_parameter_3
byte   event_parameter_4
word   CheckSum
```

Answer: stack event OK

```
byte   0xEE
byte   0xB5 (USM_EVENT_OK)
byte   0x00
word   0x00B5
```

List of possible error answer commands

```
0xBF CHECKSUM_ER                  checksum fault
0xBE UNDEF_COM                    undefined command
```

Description of events

Stack Max recognize several event types. One of them, *button_event*, simulate using of buttons on front panel. This event is sufficient to fully control stack max. Its using is universal for any configuration. Other events affects internal state directly and their using highly depends on configuration. Events have up to four parameters of byte type. Here is the full list:

event_id	*event (parameters)*
0x01	store_status(index)
0x02	retrieve_status(index)
0x03	set_antennas(antennas)
0x04	toggle_antennas(antennas)
0x05	cancel_tr_split()
0x06	set_tr_split()
0x07	toggle_tr_split()
0x08	cancel_bop()
0x09	set_bop(bop_index)
0x0A	set_next_bop()
0x0B	cancel_aux()
0x0C	set_aux(aux)
0x0D	set_next_aux()
0x0E	set_status(aux, bop_index, rx, tx)
0x0F	button_event(buttons, buttons_down, buttons_held, buttons_early_up)
0x10	enable_ptt_232()
0x11	disable_ptt_232()
0x12	enable_inh_232()
0x13	disable_inh_232()

store_status(index)

Four bytes of internal status (status_aux, status_bop_index, status_rx, status_tx) can be stored/retrieved to/from one of four memories.This event will store them to memory with specified *index*. There are only 4 memories. Only two least significant bits are taken into account.

retrieve_status(index)

Internal status previously stored by store_status will be retrieved from memory with specified *index*. Only two least significant bits are taken into account.

set_antennas(antennas)

Antenna selection will be changed according to four least significant bit of parameter *antennas.* Set bit means that antenna will be selected and cleared bit vice-versa.
Antennas disabled in configuration or selected for AUX cannot be selected. Also some other restriction are applied. These restrictions are specific for selected “stack type”.
If split state is active than this event will apply only to status_rx (if PTT is inactive) or status_tx (if PTT is active).

toggle_antennas(antennas)

Antenna selection will be changed according to four least significant bit of parameter *antennas.* Set bit means that antenna selection will be inverted and cleared bit means that antenna selection will not be changed.
Antennas disabled in configuration or selected for AUX cannot be selected. Also some other restriction are applied. These restrictions are specific for selected “stack type”.

cancel_tr_split()

Split state will be unactivated.

set_tr_split()

Split state will be activated if it is enabled in configuration.

toggle_tr_split()

Split state will be inverted. Split can be activated only if it is enabled in configuration.

cancel_bop()

If some BOP selection is active, this will be canceled and last “no BOP” selection will be retrieved. Valid only for real stacks. If other “stack type” is selected this event is ignored.

set_bop(bop_index)

Some BOP selection from bop_list in configuration will be selected. Valid only for real stacks. If other “stack type” is selected this event is ignored. Parameter *bop_index* must be less than bop_list_length, otherwise event is ignored. If BOP selection cannot be realized due to aux selection event is ignored too.

set_next_bop()

Next BOP selection from bop_list will be selected. The same restriction are valid as for set_bop.

cancel_aux()

If some AUX selection is active, this will be canceled and last “no AUX” selection will be retrieved.

set_aux(aux)

AUX selection of some antenna will be activated according to four least significant bit of parameter *aux*. Aux selection for specified antenna must be enabled in configuration.

set_next_aux()

AUX selection of next enabled antenna will be activated.

set_status(aux, bop_index, rx, tx)

Parameters will be copied to internal status variables: status_aux, status_bop_index, status_rx, status_tx.

button_event(buttons, buttons_down, buttons_held, buttons_early_up)

This event simulate handling of buttons on Stack Max front panel. All parameters are bit masks where bits are assigned to front panel buttons in the following order (starting with most significant bit): “1”, “2”, “3”, “4”, “T/R”, “BOP”, “AUX”. Least significant bit is ignored.
buttons = current state of buttons (1=pressed, 0=released)
buttons_down (1=button has been just pressed)
buttons_held (1=600ms just elapsed since button was pressed)
buttons_early_up (1=button has been released before 600 ms)

enable_ptt_232() This will enable PTT control via RTS.

disable_ptt_232() This will disable PTT control via RTS.

enable_inh_232() This will enable sending of INH signal on CTS.

disable_inh_232() This will disable sending of INH signal on CTS.

HOWTOs

This section describes the techniques how to communicate with microHAM devices.
The techniques to get the device version, configure the device and upgrade the device firmware, are basically the same for all devices. Some small differences are summarized separately for each device.
The technique to on-line control the device is device specific. Actually some devices cannot be on-line controlled and the configuration fully determines their operation.

How to communicate with Band Decoder?

Band Decoder uses the *application protocol* like other microHAM devices. But there are some differences.
- The communication must be started by *interrogation sequence* to switch Band Decoder to *configuration mode*.
- The interval between the interrogation sequence and the first packet may not exceed 100 milliseconds.
- The interval between packets may not exceed 3000 milliseconds.
- The communication should be finished by query *end of configuration mode* to speed up the switching back to the polling mode.

To get the version information, configure or upgrade the firmware see the sections

How to get the version information?
How to upgrade the firmware?
How to configure the device?

Band decoder communication modes

Band Decoder runs in one of the four modes.

The **polling mode** (CPU2RIG_POLLING) is intended to situation when no logging software is running on the computer.
Band Decoder regularly sends polling commands at the rig baud rate to the rig and receives and decodes the answer commands from the rig. These data doesn't pass to the computer. At the same time Band Decoder detects if any signal from the computer (although single pulse) appears on the line. If yes, Band decoder switches to configurator detection mode.

The **configurator detection mode** (CONFIGURATOR_DETECTION) is the temporary mode used when Band Decoder must resolve if the data from the computer comes from the configurator or from the logging software. Accordingly Band Decoder switches to the corresponding mode.
The data from the computer passes to the rig and data from the rig passes to the computer without influence in this mode. At the same time Band Decoder tries to detect the *interrogation sequence* at 19200 bps in the data stream that comes from the computer. If the interrogation is detected in the limit of 100 milliseconds, Band decoder switches to the configuration mode. If no interrogation is detected within this limit, Band decoder switches to the monitoring mode.

The **monitoring mode** (PC2RIG_MONITORING) is intended to situation when some logging software running on the computer communicates with the rig.
The data from the computer passes to the rig and data from the rig passes to the computer without influence in this mode. At the same time Band Decoder tries to detect some answers from the rig at the rig baud rate to catch the operating frequency. If these answers regularly come from the rig, Band Decoder remains in this mode. If Band Decoder cannot detect any rig answer within the time of 10 seconds, it switches back to the polling mode.
Band decoder does not listen the computer during this mode. It means that the attempt to start communication mode with the interrogation sequence will be unsuccessful and sending of data must be suspended for 10 seconds to this mode timeouts.

The **configuration mode** (PC2CPU_COMMUNICATION) is intended to communication with the configurator that runs on the computer.
Band Decoder communicates with the computer at 19200 bps using the *application protocol*. The rig is completely cut off from this communication. This mode finishes after timeout of 3000 milliseconds without packet detection or after receiving the query *end of configuration mode*.

How to communicate with Stack Max?

Stack Max fully conforms the *application protocol*.

To get the version information, configure or upgrade the firmware see the sections

How to get the version information?
How to upgrade the firmware?
How to configure the device?

To on-line control Stack Max see the section

How to on-line control the Stack Max?

How to get the version information?

To get version information from the device perform the following steps.

- *Open the serial port*.
- Send *interrogation* immediately followed by the query *get version*.
  Interrogation is needed to switch Band Decoder to communication mode. Other devices ignore it.
- If the *expected answer* is received,
  retrieve version information from it and if the device is Band Decoder send the query *end of configuration mode*, close the port and terminate.
- Else if the device sends the error answer *CBL_UNDEF_COM*,
  it means the bootloader is running instead of the application firmware. In this case send the query *get version and status*. If the *expected answer* is received, retrieve version information from it, close the port and terminate.
- Else if some error answer or no answer is received,
  retry query or close the port and terminate.

How to upgrade the firmware?

To write the configuration to device perform the following steps.
It requires having the right version of *microHAM firmware file* that can be downloaded from microHAM web page (http://www.microham.com/downloads.html).

- Open *firmware file* and test its format.
- *Open the serial port*.
- Send *interrogation* immediately followed by the query *start bootloader*.
  Interrogation is needed to switch Band Decoder to communication mode. Other devices ignore it.
- If the device sends the error answer *CBL_UNDEF_COM*,
  it means the bootloader is already running. Ignore this error and continue. If other error is received, retry query or close the port and terminate.
- Wait 200 milliseconds.
- Send the query *get version and status*.
- If error is received, retry query or close the port and terminate.
- If the *expected answer* is received,
  retrieve version information from it. If not, close the port and terminate.
- Now step by step read and process blocks from *firmware file*. If some error occurs, retry query or terminate.
  - Ignore all *comment blocks*.
  - Use *version specification block* to check the compatibility with device.
  - Download all *flash data blocks* using the query *write data block to flash*.
  - Download all *EEPROM data blocks*, if there are any, using the query *write data block to EEPROM*.
- If all flash data blocks were successfully downloaded, send the query *end of programming*.
  Be careful. This query clears BSB flag. It means the device starts the application firmware at power up. If the firmware was not correctly downloaded, it can cause that malfunction firmware will not be able to start the bootloader and download the right firmware.
- Close the port.

How to configure the device?

To read the configuration from the device perform the following steps.

- *Open the serial port*.
- Send *interrogation* immediately followed by the query *get version*.
  Interrogation is needed to switch Band Decoder to communication mode. Other devices ignore it.
- If the *expected answer* is received,
  retrieve *version information* from it and check if the device type is what you expect. If not, close the port and terminate. If the device sends the error answer *CBL_UNDEF_COM*, it means the bootloader is running instead of the application firmware. In this case configuration cannot be read, close the port and terminate. If some other error answer or no answer is received, retry query or close the port and terminate.
- Send the query *read configuration*.
  Set parameters `eeprom_address` and `size` at need. See the chapter *Configuration*.
- If the *expected answer* is received,
  retrieve *the configuration* from it. If some error answer or no answer is received, retry query or close the port and terminate.
- If the device is Band Decoder, send the query *end of configuration mode*.
- Close the port.

To write the configuration to device perform the following steps.

- *Open the serial port*.
- Send *interrogation* immediately followed by the query *get version*.
  Interrogation is needed to switch Band Decoder to communication mode. Other devices ignore it.
- If the *expected answer* is received,
  retrieve *version information* from it and check if the device type is what you expect. If not, close the port and terminate. If the device sends the error answer *CBL_UNDEF_COM*, it means the bootloader is running instead of the application firmware. In this case configuration cannot be modified, close the port and terminate. If some other error answer or no answer is received, retry query or close the port and terminate.
- Prepare *the configuration* and send it in the query *write configuration*.
- If the *expected answer* is received,
  Send the query *restart application* to restart the application firmware with the new configuration.
- Else if some error answer or no answer is received,
  retry query or close the port and terminate.
- Close the port.

How to on-line control the Stack Max?

To control Stack Max remotely from computer follow these steps.

- *Open the serial port*.

Optional:

- If required, send query *get version*.
- If the *expected answer* is received, retrieve version information from it.
- Else if the device sends the error answer *CBL_UNDEF_COM*, it means the bootloader is running instead of firmware. In this case the firmware need to be uploaded. Close port and terminate.
- Else if some error answer or no answer is received,
  retry query or close the port and terminate.

Optional:

- If required, send the query *read configuration*.
  Set parameters `eeprom_address` and `size` at need. See the chapter *Configuration*.
- If the *expected answer* is received, retrieve *the configuration* from it. If some error answer or no answer is received, retry query or close the port and terminate.

- Periodically poll the Stack Max by query *get stack status*.
- If the *expected answer* is received, retrieve internal state and state of outputs and LEDs.
- Else if the device sends the error answer *CBL_UNDEF_COM*, it means the bootloader is running instead of firmware. In this case the firmware need to be uploaded. Close port and terminate.
- Else if some error answer or no answer is received,
  retry query or close the port and terminate.
- To control Stack Max use query *stack event*.

- At the end close the port.

See also examples below.

Stack Max remote control using button_event

There are several approaches how to utilize Stack event command to control Stack Max. The most universal one is using of `button_event` that is independent on configuration.
Stack Max has seven buttons on front panel and recognizes four types of “button events”:

1. button just pressed
2. button held at least 600 ms
3. button released before 600 ms
4. button released later than 600 ms (no action anytime)

What action is linked to these events depends on configuration. Generally the 4th event doesn't invoke any action at any circumstances. (Hence this event is not included in protocol.)
Here are some examples how a control program can simulate these events by sending button_event.

Short pressing of button "1"

```
Query:   EE D5 05 0F 80 80 00 00 E9 01
         button "1" has been just pressed
Answer:  EE B5 00 B5 00
         ok
Query:   EE D5 05 0F 00 00 00 80 69 01
         button "1" has been released before the period of 600 ms was elapsed since it was
         pressed
Answer:  EE B5 00 B5 00
         ok
```

Long pressing of button "BOP"

```
Query:   EE D5 05 0F 04 04 00 00 F1 00
         button "BOP" has been just pressed
Answer:  EE B5 00 B5 00
         ok
Query:   EE D5 05 0F 04 00 04 00 F1 00
         period of 600 ms was elapsed since button "BOP" had been pressed
Answer:  EE B5 00 B5 00
         ok
```

Stack Max remote control using set_status

Alternative approach to control Stack Max is using of `set_status` event. This also allows the full control of Stack Max. How status variables are interpreted by Stack Max depends on configuration. In the following examples we assume that Stack Max was properly configured to device “4 ANTENNA SWITCH” with all antennas enabled and the firmware version is at least v2.7.

Select ANT2

```
Query:   EE D5 05 0E 00 00 02 02 EC 00
         set split off; select ANT2
Answer:  EE B5 00 B5 00
         ok
```

Select ANT2 for RX and ANT3 for TX

```
Query:   EE D5 05 0E 80 00 02 04 6E 01
         set split on; select rx ANT2, tx ANT3
Answer:  EE B5 00 B5 00
         ok
```

Monitoring of Stack Max state

Using get_stack_status query the control program can poll Stack Max to send status. This status include also current status of front panel LEDs (led_shadow and mix_shadow). This information is sufficient to control program to simulate Stack Max front panel. Here are some examples.

```
Query:   EE D6 00 D6 00
         get stack status
Answer:  EE B6 08 00 00 01 01 00 04 00 01 C5 00
         LEDs on: red 1
Query:   EE D6 00 D6 00
         get stack status
Answer:  EE B6 08 00 00 02 02 04 10 00 02 D8 00
         LEDs on: red 2
Query:   EE D6 00 D6 00
         get stack status
Answer:  EE B6 08 80 00 04 04 00 40 01 04 8B 01
         LEDs on: red 3, red T/R
```

This method has one drawback. Items led_shadow and mix_shadow indicates current status of LEDs. In the case, when some LED is flashing, corresponding bit follows its state. It depends on the time of query what is reported. This can be problem when remote control is realized through the net with possible long delays. To overcome this problem the control program can evaluate first five status items to get the state of LEDs. Because interpretation of these items depends on configuration we will reduce our example only for the case the Stack Max is configured to device “4 ANTENNA SWITCH”. Here is the list of interesting items with its possible values.

```
status_aux:           0x00=split off, 0x80=split on
status_rx:            0x01=ANT1, 0x02=ANT2, 0x04=ANT3, 0x08=ANT4
status_tx:            0x01=ANT1, 0x02=ANT2, 0x04=ANT3, 0x08=ANT4
status_flags & 0x01:  not applied because PTT is active (LEDs flashing)
status_flags & 0x04:  current status of PTT
```

Use the method below to get LEDs state from them:

```
byte antennas;
bool flashing = false;
if ((status_aux & 0x80) && (status_flags & 0x04)) // split on & PTT active
  antennas = status_tx;
else                      // split off or PTT inactive
  antennas = status_rx;
switch (antennas) {
  case 0x01:              // antenna 1
    led_shadow = 0x04;    // red 1
    break;
  case 0x02:              // antenna 2
    led_shadow = 0x10;    // red 2
    break;
  case 0x04:              // antenna 3
    led_shadow = 0x40;    // red 3
    break;
  case 0x08:              // antenna 4
    led_shadow = 0x02;    // red 4
    break;
}
if (status_flags & 0x01) // status not applied
  flashing = true;        // antenna buttons LEDs are flashing
if (status_aux & 0x80)    // split on
  mix_shadow = 0x01;      // red T/R
else
  mix_shadow = 0x00;
```

Same examples of possible answers:

```
 Answer:  EE B6 08 00 00 01 01 00 04 00 01 C5 00
          split: off; selection: ANT1; PTT:RX;   LEDs: red 1 on

 Answer:  EE B6 08 00 00 02 02 01 00 00 01 C4 00
          split: off; selection: ANT2; PTT:RX;   LEDs: red 2 flashing
          Selection will be applied after PTT raise and drop. This is result of protection
          when Stack Max prevent output to change since start up until first PTT pulse is
          received.

 Answer:  EE B6 08 00 00 02 02 00 10 00 02 D4 00
          split: off; selection: ANT2; PTT:RX;   LEDs: red 2 on

 Answer:  EE B6 08 00 00 02 02 04 10 00 02 D8 00
          split: off; selection: ANT2; PTT:TX;   LEDs: red 2 on

 Answer:  EE B6 08 00 00 04 04 05 00 00 02 CD 00
          split: off; selection: ANT3; PTT:TX;   LEDs: red 3 flashing
          Selection will be applied after PTT drop.

 Answer:  EE B6 08 00 00 04 04 00 40 00 04 0A 01
          split: off; selection: ANT3; PTT:RX;   LEDs: red 3 on
          Selection is already applied.

 Answer:  EE B6 08 80 00 04 04 00 40 01 04 8B 01
          split: on; selection: rx ANT3, tx ANT3; PTT:RX;   LEDs: red 3 on, T/R on

 Answer:  EE B6 08 80 00 01 04 00 04 01 01 49 01
          split: on; selection: rx ANT1, tx ANT3; PTT:RX;   LEDs: red 1 on, T/R on

 Answer:  EE B6 08 80 00 01 04 04 40 01 04 8C 01
          split: on; selection: rx ANT1, tx ANT3; PTT:TX;   LEDs: red 3 on, T/R on

 Answer:  EE B6 08 80 00 01 08 05 02 01 04 53 01
          split: on; selection: rx ANT1, tx ANT4; PTT:TX;   LEDs: red 4 flashing, T/R on
          tx selection will be applied after next PTT raise

 Answer:  EE B6 08 80 00 01 08 00 04 01 01 4D 01
          split: on; selection: rx ANT1, tx ANT4; PTT:RX;   LEDs: red 1 on, T/R on

 Answer:  EE B6 08 80 00 01 08 04 02 01 08 56 01
          split: on; selection: rx ANT1, tx ANT4; PTT:TX;   LEDs: red 4 on, T/R on
          tx selection is already applied

 Answer:  EE B6 08 80 00 01 08 00 04 01 01 4D 01
          split: on; selection: rx ANT1, tx ANT4; PTT:RX;   LEDs: red 3 on, T/R on
```

Note: Keep in mind that every 0xEE inside the packet is doubled. It means that sender sends single 0xEE only at the start of packet and every next 0xEE from inside the packet is sended two times. On the other side when receiver receives single 0xEE it interprets it as start of packet and if receives two consecutive 0xEE place only one to buffer. Here are some examples.

```
 Query:   EE D6 00 D6 00
 Answer:  EE B6 08 00 00 01 02 25 00 00 08 EE EE 00

 Query:   EE D5 05 0E 00 00 02 04 EE EE 00
 Answer:  EE B5 00 B5 00
```

Version information

The device flash memory contains the version information record. This record is set by the manufacturer and it is permanent. It describes device hardware and the bootloader version.

byte cbl_ver_minor
Minor version of bootloader, most significant bit is beta flag.

byte cbl_ver_major
Major version of bootloader.

byte product_type
Type of device.
Possible values:

0	not specified
1	micro Band Decoder
2	micro Stack Max
other	reserved for the future

byte hardware_version
Version of electronic hardware.

byte mechanical_version
Version of mechanical arrangement.

word serial_number
Serial number.

byte reserved
Reserve for the future. It is always 0xFF.

The application firmware contains its own version information too. It describes currently loaded application, so it will change after the firmware upgrade.

byte appl_product_type
Type of device, what this application is dedicated to.

byte appl_min_hardware_version
Requirement for minimal version of electronic hardware.

byte appl_min_mechanical_version
Requirement for minimal version of mechanical arrangement.

byte appl_ver_minor
Minor version of application, most significant bit is beta flag.

byte appl_ver_major
Major version of application.

To get the version information from the device, see the section *How to get the version information*.

Note: MicroHAM Device Configurator checks, if the device type and the version (`product_type`, `hardware_version`, `mechanical_version`) satisfy the firmware requirements (`appl_product_type`, `appl_min_hardware_version`, `appl_min_mechanical_version`) before firmware is upgraded. If not, upgrade is not allowed.
Also the application itself checks, if the `appl_product_type` is equal to the `product_type` immediately after power up. If not, it starts bootloader. So it is ensured that only the firmware compatible with the device will be started.

Firmware file

MicroHAM firmware file contains the encoded firmware for some of microHAM devices. Its name form is usually "DDD_release_XX_XX.cbl", where DDD is a short device name and XX_XX is a version of the firmware.
The firmware file consists of the blocks of several types. Generally the block has the following format.

General firmware file data block

```
byte   BlockType
byte   BlockLength
byte   BlockContent[BlockLength]
```

Version 2.0 of the firmware file format has defined four block types.

***Flash data block**, downloaded to device code memory.*

```
byte   0x01
byte   0x86
byte   FlashDataBlockContent[0x86]
```

***EEPROM data block**, downloaded to device EEPROM memory.*

```
byte   0x02
byte   Length
byte   EepromDataBlockContent[Length]
```

***Version specification block**, used by configurator to check the compatibility.*

```
byte   0x03
byte   0x05
byte   appl_product_type
byte   appl_min_hardware_version
byte   appl_min_mechanical_version
byte   appl_ver_minor
byte   appl_ver_major
```

***Comment block**, ignored by configurator.*

```
byte   0x20
byte   CommentLength     ; usually 0x20
byte   Comment[CommentLength]
```

To upgrade device firmware, see the section *How to upgrade the firmware*.

Note: Because EEPROM data block and the version specification block was not defined in the previous version 1.0 of firmware file format, current version 2.0 is fully supported by microHAM Device Configurator since v1.2 and by the bootloader since v2.0.
MicroHAM Device Configurator, older than v1.2, ignores EEPROM data block and treats the version specification block as an error. Therefore it cannot download newer firmware files that contain the version specification block.
The bootloader v1.0 doesn't support the download to EEPROM. Therefore the configurator doesn't try to download EEPROM data blocks to the device with the bootloader v1.0.

Configuration

The configuration of any microHAM device is stored in its internal EEPROM memory. Size of this memory is 2 kbytes and its address range is from 0x0000 to 0x07FF. It is possible to access any part of this memory from computer through *application protocol*. Configuration parameters are stored on the lowest addresses. It depends on the device type and the firmware version how much memory is occupied. Some devices also store their working data at the highest addresses of EEPROM. While configuration data are taken into account at the application start up, working data can be accessed by the application at any time.

To modify the device configuration, whole or its part, see the section *How to configure the device*.

Band Decoder configuration

Band Decoder configuration parameters are presented by category. To find their location look to *EEPROM memory map* at the end of this chapter. Some boolean parameters (does not related each other) are grouped to the complex byte parameters as `cfg_flags`, `cfg_flags_2` and `uconf_flags`.

Band data source

Parameters of this category determine what source of band data should the Band Decoder use. It is possible to get band data from the four-bit parallel input on ACC connector or from some of the CAT interfaces.

`bit use_yaesu_4bit_band_data`

If this flag is set, the Band Decoder gets band data from the four-bit parallel input on ACC connector. Independently it works as level converter between selected rig interface and the serial interface of the computer (PC), but the serial communication on this line is ignored. Except the `rig_interface`, all other "CAT related" parameters from this category are ignored.
If this flag is clear, the Band Decoder decodes band data from the serial data received from selected CAT interface of the rig. Details of this CAT communication are described by other parameters of this category.

`byte rig_interface`

This parameter determine what CAT interface is used to receive data from the rig. The Band Decoder has three rig interfaces.
Possible values:

0	no interface is used
1	C-IV (resp. FIF232) interface
2	IF-232 interface
3	RS-232 interface
other	defaults to 0

`bit respect_cts`

If this flag is set, the Band Decoder respects CTS signal on the selected interface. This is necessary when the rig uses RTS/CTS handshake. It is possible on IF-232 and RS-232 interfaces only. If CI-V interface is selected, this flag is ignored.
If this flag is clear, signal CTS is ignored (no handshake is used).

`byte cat_baud_rate`

This parameter determines the baud rate on CAT interface. It must correspond to baud rate settings on the rig.
Possible values:

0xA0	1200 bps
0xD0	2400 bps
0xE8	4800 bps
0xF4	9600 bps
0xFA	19200 bps
0xFD	38400 bps
0xFE	57600 bps
0xFF	115200 bps
other	defaults to 9600 bps

byte cat_protocol

This parameter determines what CAT protocol is used. It must correspond to the rig type. Possible values:

0x00	no protocol, no data transmitted, received data ignored	
0x01	Kenwood protocol	
0x02	Icom general protocol	
0x03	Icom IC-735 protocol	
0x04	Yaesu FT-100 compatible protocol	
0x05	Yaesu FT-1000MP compatible protocol	
0x06	Yaesu FT-8x7 compatible protocol	
0x07	Yaesu FT-900 compatible protocol	
0x08	Yaesu FT-920 compatible protocol	
0x09	Yaesu FT-990 compatible protocol	
0x0A	Yaesu FT-1000D compatible protocol	
0x0B	Yaesu FT-757 compatible protocol	
0x0C	Yaesu FT-847 compatible protocol	
0x0D	Yaesu FT-890 compatible protocol	
0x0E	Yaesu FT-767 compatible protocol	*not implemented yet*
0x20	TenTec ORION protocol	
0x21	TenTec JUPITER protocol	
0x22	TenTec ARGONAUT protocol	
0x22	TenTec PEGASUS protocol	*not implemented yet*
0x30	Barret 900 protocol	
0x31	JRC JST-145/245 protocol	*not implemented yet*
0x40	Yaesu FTdx9000/FT-2000 compatible protocol	
other	defaults to 0	

byte icom_address

This parameter specifies the rig address in the Icom protocol. It is used only if the `cat_protocol` is set to some Icom protocol, general or IC-735. It can be automatically adjusted, if its auto detection is enabled (see `autodetect_icom_address`).

bit autodetect_icom_address

This flag, if it is set, allows adjusting the icom_address, from "Send frequency data (command 0)" packet received from the Icom rig. Parameter icom_address in EEPROM is not changed, but the Band Decoder works with auto detected value. This feature is undesirable when there are two or more rigs on CI-V bus.
Note: The firmware supports this parameter since version 2.3. Previous versions make auto detection always and this feature cannot be disabled.

bit enable_cat_substitution_by_yaesu_4bit

If this flag is set, the Band Decoder is allowed to use alternative band data from four-bit parallel input on ACC connector when they are valid. It means that band data from CAT are taken into account only if there are invalid data an ACC (0 or higher than 10). It allows to connecting a manual BCD controller to the Band Decoder via ACC, and combining the manual switching with the automatic (CAT) switching without reconfiguring the Band Decoder.
Note: The firmware supports this parameter since version 2.0.

CW and PTT

Parameters of this category determine what source of CW and PTT should the Band Decoder use and what conditions must be satisfied to pass these signals to the CW and PTT outputs.

`bit enable_cw_ptt_from_lpt`
`bit enable_cw_ptt_from_com`

If the flag `enable_cw_ptt_from_lpt` is set, CW signal is passed from LPT pin 17 to the CW output and PTT signal from LPT pin 16 to the PTT output.
If the flag `enable_cw_ptt_from_lpt` is clear and the `enable_cw_ptt_from_com` is set, CW signal is passed from DTR pin of the serial port PC to the CW output and PTT signal from RTS pin of serial port PC to the PTT output.
If both flags are clear, CW and PTT outputs are still inactive.

`bit disable_cw_ptt_out_of_bands`

If this flag is set, CW and PTT signals are passed to their outputs only if the current frequency is within one of specified bands and they are blocked if the frequency is out of any band. If the Band Decoder gets band data from the four-bit parallel input, CW and PTT signals are never blocked because in this case the Band Decoder does not know the frequency and so it cannot detect if the frequency is out of band.
If this flag is clear, CW and PTT signals are no blocked dependently on frequency.

`bit disable_cw_ptt_when_pc2rig_faults`

If this flag is set, CW and PTT outputs stays inactive since the Band Decoder is powered up to the first rig answer to computer is received and decoded.
If this flag is clear, CW and PTT signals are no blocked.

Band plan

Parameters of this category define boundaries of bands. Currently the Band Decoder supports 11 bands indexed from 1 to 11 and named 160m, 80m, 40m, 30m, 20m, 17m, 15m, 12m, 10m, 6m and 60m. User can change the default band plan to any one with satisfying the conditions that new bands are not overlapped and they preserve the original band order with increasing frequency: 1, 2, 11, 3, 4, 5, 6, 7, 8, 9, 10.
Note: The default band plan used by MicroHAM Device Configurator you will find in source code. See the macros `DEFAULT_BAND_BOUNDARIES_VALUES` and `DEFAULT_SPLIT_FREQUENCY_VALUES` in the file *uBD_prot.h*.
Note: The firmware supports 60m band since version 2.1. This band is signaled on the front panel by two LEDs 80m and 40m.

`dword band_boundaries[band].low`

This parameter specifies the lowest frequency of band in Hz.

`dword band_boundaries[band].high`

This parameter specifies the highest frequency of band in Hz.

`dword split_frequency[band]`

This parameter specifies the frequency in Hz that splits band to two subbands. It is possible to generate different output vector on BAND DATA OUTPUTS for each subband. See the parameter *`split_mask`*.

Outputs

Parameters of this category define output vectors associated to bands or subbands respectively. These parameters also describes what bands are split and how.
Each band has associated two output vectors. If band is not split, only one of them, `lo_sub`, is used. If band is split, both vectors are used.
There are two possibilities how to split band. One is frequency split. In this case `lo_sub` vector is used only if the frequency is not higher than *`split_frequency`* and if the frequency is higher, `hi_sub` vector is used. The second possibility is split by external switch. In this case `lo_sub` vector is used if external switch is off and `hi_sub` vector is used if external switch is on.

`word split_mask`

This is bitwise parameter. Each band has associated one bit. Low significant bit is associated to the band 1 (160m), etc.
If bit corresponding to some band is set, this band is split. Type of split is defined by parameter `external_switch_split_mask`.

`bit allow_external_switch`

This flag changes function of the Band Decoder SET input.
If this flag is set, SET input is considered as an external switch input and the band split by external switch is allowed.
If this flag is clear, SET input has its original function. It allows display the Band Decoder state and some configuration flags on the front panel and it allows manual band switching when CAT band data are not available.

`word external_switch_split_mask`

This is bitwise parameter like a `split_mask`.
If bit corresponding to some band is set, this band is split by external switch.
But if corresponding bit in `split_mask` is cleared, this bit is ignored and band is not split.
Also if flag `allow_external_switch` is cleared whole this parameter is ignored and split by external switch is not allowed.

`word out_vectors[band][sub_band]`

Output vector associated to the subband of band.
Index sub_band = 0 (previously lo_sub) correspond to lower subband or to external switch turned off respectively.
Index sub_band = 1 (previously hi_sub) correspond to higher subband or to external switch turned on respectively.
If four positional switch is connected to ACC (`external_switch_on_acc` is set) than four vectors are associated to each band. Values of sub_band from 0 to 3 correspond to positions on that switch.

Special features

Parameters of this category determine what special feature will be used and specify details of this feature.

`byte multi_out_mode`

This parameter determines what special feature will be active.
Possible values:

0	no special function
1	split info output
2	IC-PW1 control
3	IC-2KL/IC-4KL control
5	the same band protection on multi stations
6	generation of pulse after band/output is changed
other	defaults to 0

`byte icompw1_baud_rate`

This parameter determines baud rate of the communication with IC-PW1 amplifier on multifunctional pin when IC-PW1 control is enabled by parameter `multi_out_mode`.
Possible values:

0xA0	1200 bps
0xD0	2400 bps
0xE8	4800 bps
0xF4	9600 bps
other	defaults to 9600 bps

`byte icompw1_address`

When Band Decoder communicates with IC-PW1 amplifier, it simulates the Icom rig with address specified by this parameter. IC-PW1 control must be enabled by parameter `multi_out_mode`.

`byte protection_group_size`

This parameter is used when the same band protection on multi stations is enabled by parameter `multi_out_mode`. It specifies how much of stations (band decoders) is interconnected. Band Decoder allows switching only if detects the specified number of band decoders on the bus. It is possible to set the autodetection and the number of stations will be set dynamically. But this is not so safe because it is not possible to detect the bus interruption. Total number of stations is limited to 6. It means that maximally 5 Band Decoders can be connected to our Band Decoder.
Possible values:

0	auto
1	2 stations
2	3 stations
3	4 stations
4	5 stations
5	6 stations
other	unusable

`byte change_pulse_length`

This parameter determines the length of generated pulse when generation of pulse after band/output change is enabled by parameter `multi_out_mode`.

`byte change_pulse_mode`

This is complex parameter used when generation of pulse after band/output change is enabled by parameter `multi_out_mode`. Its bitwise form is PEEEDDDD, see the text below.
Most significant bit determine pulse polarity P. If it is set, pulse is positive.
Other bits determine the size of frequency intervals used to generate pulses also when the frequency pass through the boundaries of these intervals. The interval is specified in the form M*10^E.

The mantissa M is specified in reverse form M=10/D, where the divisor D is stored in four lowest significant bits of this parameter. If the divisor D is zero, pulses are generated only if band or output vector is changed but not at frequency changes within a band.
Possible values:

DDDD	*M*
1010	1.00
1000	1.25
0110	1.67
0101	2.00
0100	2.50
0011	3.33
0010	5.00
0000	pulses are not generated at frequency changes
other	useless

Exponent E is stored in tree bits of this parameter.
Possible values:

EEE	*10^E*
000	1 Hz
001	10 Hz
010	100 Hz
011	1 kHz
100	10 kHz
101	100 kHz
110	1 MHz
111	10 MHz

Configurator related parameters

The Band Decoder ignores parameters of this category. They are used only by microHAM Device Configurator that stores here some additional data necessary to correctly display the configuration read from the device.
The third party software should store the recommended values to these parameters.

`byte rig_type`

This parameter contains ID of "Rig (band data source)" combo box list item. This combo box allows setting all band data source related parameters (interface, baud rate, etc.) in one step by selecting the rig type.
The list of rigs supported by MicroHAM Device Configurator with default values of the related parameters you will find in source code. See the array `TConstants_uBD::RIG_TYPE_LIST[]` in the file *uBD_const.cpp*.
The third party software should store the zero here that corresponds to "custom settings".

`byte antenna_switch_type`

This parameter contains index to "Antenna switch" combo box list. This combo box allows setting predefined output vectors and display some antenna switch related informations such as cable colors, names of ports and terminal pins in one step by selecting the antenna switch.
The list of antenna switches supported by MicroHAM Device Configurator with default output vectors you will find in source code. See the array `TConstants_uBD::ANTENNA_SWITCH_LIST[]` in the file *uBD_const.cpp*.
The third party software should store the zero here that correspond to "custom settings".

`byte full_rig_interface`

This parameter contains the index to "Rig interface" combo box list. It integrates two parameters `rig_interface` and `respect_cts`.
The third party software should store the value according to table.
Possible values:

0	no interface
1	C-IV interface
2	IF-232 interface
3	IF-232 interface with RTS/CTS handshake
4	RS-232 interface
5	RS-232 interface with RTS/CTS handshake
6	FIF232 interface

`bit use_default_band_boundaries`

This parameter determine the state of "Use default band boundaries" checkbox.
The default band plan used by MicroHAM Device Configurator you will find in source code. See the macros `DEFAULT_BAND_BOUNDARIES_VALUES` and `DEFAULT_SPLIT_FREQUENCY_VALUES` in the file *uBD_prot.h*.
The third party software should clear this flag that correspond to unchecked state.

`bit only_one_out_active`

This parameter determine the state of "One out active only" checkbox.
The third party software should clear this flag that correspond to unchecked state.

Band Decoder EEPROM memory map

***EEPROM memory map**, configuration*

```
0000: byte  cat_baud_rate
0001: byte  cat_protocol
0002: byte  icom_address
0003: byte  rig_interface
0004: byte  cfg_flags
            bit0: use_yaesu_4bit_band_data
            bit1: respect_cts
            bit2: enable_cw_ptt_from_com
            bit3: enable_cw_ptt_from_lpt
            bit4: disable_cw_ptt_out_of_bands
            bit5: disable_cw_ptt_when_pc2rig_faults
            bit6: enable_cat_substitution_by_yaesu_4bit
            bit7: allow_external_switch
0005: byte  reserved
0006: word  split_mask, bitx: (0<=x<=10) band_x+1_split
0008: dword band_boundaries[1].low
000C: dword band_boundaries[1].high
0010: dword band_boundaries[2].low
0014: dword band_boundaries[2].high
0018: dword band_boundaries[3].low
001C: dword band_boundaries[3].high
0020: dword band_boundaries[4].low
0024: dword band_boundaries[4].high
0028: dword band_boundaries[5].low
002C: dword band_boundaries[5].high
0030: dword band_boundaries[6].low
0034: dword band_boundaries[6].high
0038: dword band_boundaries[7].low
003C: dword band_boundaries[7].high
0040: dword band_boundaries[8].low
0044: dword band_boundaries[8].high
0048: dword band_boundaries[9].low
004C: dword band_boundaries[9].high
0050: dword band_boundaries[10].low
0054: dword band_boundaries[10].high
0058: dword split_frequency[1]
005C: dword split_frequency[2]
0060: dword split_frequency[3]
0064: dword split_frequency[4]
0068: dword split_frequency[5]
006C: dword split_frequency[6]
0070: dword split_frequency[7]
0074: dword split_frequency[8]
0078: dword split_frequency[9]
007C: dword split_frequency[10]
0080: word  out_vectors[1][0]  // lo_sub
0082: word  out_vectors[1][1]  // hi_sub
0084: word  out_vectors[2][0]  // lo_sub
0086: word  out_vectors[2][1]  // hi_sub
0088: word  out_vectors[3][0]  // lo_sub
008A: word  out_vectors[3][1]  // hi_sub
008C: word  out_vectors[4][0]  // lo_sub
008E: word  out_vectors[4][1]  // hi_sub
0090: word  out_vectors[5][0]  // lo_sub
0092: word  out_vectors[5][1]  // hi_sub
0094: word  out_vectors[6][0]  // lo_sub
```

```
0096: word  out_vectors[6][1]  // hi_sub
0098: word  out_vectors[7][0]  // lo_sub
009A: word  out_vectors[7][1]  // hi_sub
009C: word  out_vectors[8][0]  // lo_sub
009E: word  out_vectors[8][1]  // hi_sub
00A0: word  out_vectors[9][0]  // lo_sub
00A2: word  out_vectors[9][1]  // hi_sub
00A4: word  out_vectors[10][0]  // lo_sub
00A6: word  out_vectors[10][1]  // hi_sub
00A8: byte  rig_type
00A9: byte  antenna_switch_type
00AA: byte  uconf_flags
            bit0: use_default_band_boundaries
            bit1: only_one_out_active
00AB: byte  full_rig_interface
00AC: byte  icompw1_baud_rate
00AD: byte  icompw1_address
00AE: word  external_switch_split_mask
            bitx: (0<=x<=10) band_x+1_split_by_external_switch
00B0: byte  multi_out_mode
00B1: byte  change_pulse_length
00B2: byte  change_pulse_mode
            bit0-6: change_pulse_interval = (10/D)*10^E
              bit0-3: divisor D
              bit4-6: exponent E
            bit7: change_pulse_polarity
00B3: byte  cfg_flags_2
            bit0: autodetect_icom_address
            bit1: force_auto_answers_mode
            bit2: accept_frequency_from_controller
            bit3: external_switch_on_acc
00B4: dword band_boundaries[11].low
00B8: dword band_boundaries[11].high
00BC: dword split_frequency[11]
00C0: word  out_vectors[11][0]  // lo_sub
00C2: word  out_vectors[11][1]  // hi_sub
00C4: byte  protection_group_size
00C5: byte  hot_switch_protection_time
00C6: word  out_vectors[1][2]
00C8: word  out_vectors[1][3]
00CA: word  out_vectors[2][2]
00CC: word  out_vectors[2][3]
00CE: word  out_vectors[3][2]
00D0: word  out_vectors[3][3]
00D2: word  out_vectors[4][2]
00D4: word  out_vectors[4][3]
00D6: word  out_vectors[5][2]
00D8: word  out_vectors[5][3]
00DA: word  out_vectors[6][2]
00DC: word  out_vectors[6][3]
00DE: word  out_vectors[7][2]
00E0: word  out_vectors[7][3]
00E2: word  out_vectors[8][2]
00E4: word  out_vectors[8][3]
00E6: word  out_vectors[9][2]
00E8: word  out_vectors[9][3]
00EA: word  out_vectors[10][2]
00EC: word  out_vectors[10][3]
00EE: word  out_vectors[11][2]
00F0: word  out_vectors[11][3]
00F2:
```

Stack Max configuration

Stack Max configuration parameters are presented by category. To find their location look to *EEPROM memory map* at the end of this chapter. Some boolean parameters (does not related each other) are grouped to the complex byte parameter `cfg_flags`.

Switch operation

Parameters of this category describe stack max operation.

byte stack_type

This parameter determines what device is connected to the controller. It may not be real stack switch. List contains also vertical arrays or simple antenna switch.
Possible values:

0x00	no device
0x01	micro STACK SWITCH
0x02	WX0B STACK MASTER
0x03	WX0B STACK MATCH
0x04	WX0B STACK MATCH used in N2NU arrangement to utilize BOP
0x05	WX0B FOUR SQUARE
0x06	WX0B TRIANGLE VERTICAL ARRAY
0x07	WX0B DOUBLE VERTICAL ARRAY
0x08	Comtek Hybrid Phasing Coupler ACB-4
0x09	micro STACK SWITCH QRO 3 ANT
0x0A	micro STACK SWITCH QRO 2 ANT
0x0B	N4TZ STACK DESIGN
0x0C	OM2KW STACK
0x0D	Comtek Stack Yagi System SYS-3
0x0E	4 ANTENNA SWITCH
0x0F	Comtek Stack Yagi System STACK-2
0x10	Comtek Phased Vertical System PVS-2
0x11	Comtek Antenna Switch System RCAS-8
other	reserved for future devices, defaults to 0

byte enabled_antennas

Four least significant bits of this parameter defines mask of enabled antennas.
Ignored for vertical arrays.

byte inhibit_time

Hot switch protection time in milliseconds. This time is duration of inhibit signal or delay of PTT output signal respectively, it depends on how `ptt_out_instead_of_inh` is set.

bit ptt_out_instead_of_inh

INH output can be used in two modes.
If this parameter is cleared there is generated inhibit signal after leading edge of PTT input signal. Duration of inhibit signal is defined by parameter `inhibit_time`.
If this parameter is set there is generated PTT output signal. Its leading edge is delayed after PTT input signal by `inhibit_time`.

bit ptt_acc_enabled

This parameter enables accepting of PTT input signal. If it is cleared signal is ignored.
Independently it is still possible to receive PTT signal from computer via serial protocol.

bit inh_acc_enabled

This parameters enables generating of inhibit (or delayed PTT resp.) signal on INH output.
Independently it is still possible to send this signal to computer via serial protocol.

bit load_memo1_at_power_up

If this is set initial status after power up is retrieved from the first memory `status[0]`.

byte enabled_aux

`micro STACK SWITCH:`
Four least significant bits of this parameter defines mask of antennas that can be connected to subradio AUX. Four most significant bits of this parameter defines mask of antennas that "may not" be connected to main radio feed.
`WX0B STACKs:`
Least significant bit of this parameter enables using of function AUX.
Ignored for other devices.

byte bop_list_length

Length of bop_list. Its maximal value is 4. Higher value defaults to 4.

byte bop_list[index]

This array contains list of allowed BOP combinations. These combinations are restricted by device design. Currently only these values are possible:

```
micro STACK SWITCH: 0x23, 0x26
WX0B STACK MATCH: 0x00
WX0B STACK MATCH (N2NU BOP mod.): 0x33
```

Button control

Parameters of this category describe operation mode of buttons.

`bit toggle_mode`

If set, toggle mode instead of exclusive mode is used as base button mode.

`bit memory_mode_enabled`

If set, memory button mode is enabled.

`bit tr_split_enabled`

This enables split function. It allows using of different selections for RX and TX.

`bit base_mode_enabled`

If set, base button mode is enabled.

`bit allow_memory_modification`

If set, it is possible to write current selection to some memory from the front panel.

`bit memory_mode_at_power_up`

If set, memory button mode instead of base mode is activated at power up. Memory mode must be enabled, else it is ignored.

Output to micro INFO Panel

Parameters of this category describe what is displayed on connected micro INFO Panel.

`bit display_tx_rx_simultan`

If set, RX and TX selections are displayed simultaneously on micro INFO Panel.
This applies only for base (exclusive/toggle) button mode.

`bit display_tx_rx_in_two_lines`

If set, RX and TX selections are displayed in two lines instead of one. If `display_tx_rx_simultan` is cleared it is ignored.
This applies only for base (exclusive/toggle) button mode.

`bit generate_mem_description`

If set, there is display generated description of current selection (like in base mode) instead of strings from configuration.
This applies only for memory button mode.

`char base_button_label[5][4]`
`char mem_button_label[5][4]`

These button labels are displayed in bottom line of micro INFO Panel. One set is for base mode and second for memory mode.

`char mem_description[24][4]`

These strings are displayed in memory mode in top line of micro INFO Panel as description of currently active memory. Parameter `generate_mem_description` must be cleared.

`char call_sign[12]`
`char switch_description[24]`

These strings are displayed for a while at power up.

Memories

Parameters of this category can be read and modified at process time. It is content of four memories used in memory mode. These memories are indexed from 0 to 3.

`byte status[mem_index].aux`

Stored value of status_aux.

`byte status[mem_index].bop_index`

Stored value of status_bop_index.

`byte status[mem_index].rx`

Stored value of status_rx.

`byte status[mem_index].tx`

Stored value of status_tx.

Stack Max EEPROM memory map

In addition to configuration parameters Stack Max stores to EEPROM some *working data*.

***EEPROM memory map**, configuration*

```
0000: byte  stack_type
0001: byte  enabled_antennas
0002: word  inhibit_time
0004: byte  cfg_flags
            bit0: toggle_mode
            bit1: memory_mode_enabled
            bit2: tr_split_enabled
            bit3: base_mode_enabled
            bit4: allow_memory_modification
            bit5: ptt_out_instead_of_inh
            bit6: ptt_acc_enabled
            bit7: inh_acc_enabled
0005: byte  cfg_flags_2
            bit0: display_tx_rx_simultan
            bit1: display_tx_rx_in_two_lines
            bit2: generate_mem_description
            bit3: memory_mode_at_power_up
            bit4: load_mem1_at_power_up
0006: byte  enabled_aux
0007: byte  bop_list_length
0008: byte  bop_list[4]
000C: char  base_button_label[5][4]
0020: char  mem_button_label[5][4]
0034: char  mem_description[24][4]
0094: char  call_sign[12]
00A0: t_status    status[4]
00B0: char  switch_description[24]
00C8:
```

```
typedef struct {
  byte aux;
  byte bop_index;
  byte rx;
  byte tx;
} t_status;
```