Protocol Specification

RBP w/ HRT version 2.1.1

MenloSystems GmbH

Authors: A. Thaller <a.thaller@menlosystems.com>,
H. Schmitz <h.schmitz@menlosystems.com>

Issue 03 – 27-Apr-2021

Introduction

This document defines the specification for the protocol used for the serial communication with a Menlo Systems GmbH device. The first part of the document describes the general concepts of message transmission over a serial bus as is common to a variety of Menlo Systems GmbH products.

Scope

This document describes the register based protocol (RBP) with hierarchical register tree (HRT), ver. 2.1.1. The concept of HRT introspection is introduced as a general method of querying device specific features. For a detailed description of the HRT layout exposed by a specific device, refer to the device specific HRT documentation.

General Concept

The design of the protocol includes features that allow its application in flexible environments:

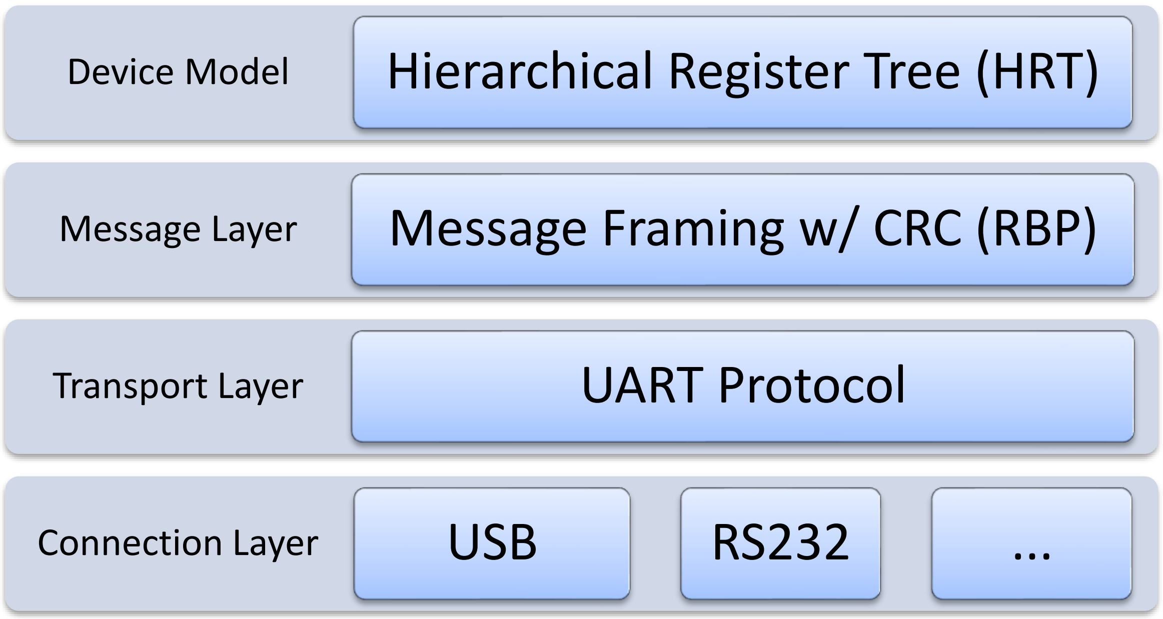
- Message Framing: Messages are transferred within a message frame with unambiguous start and stop markers for easy detection of a complete message. The message inside the frame uses a suitable encoding to guarantee the unambiguity of the message frame.
- Addressing: The message header contains a source and destination ID to identify the two end-points of the transmission. While not necessary in ordinary two-peer environments, this allows implementation of multi-peer systems (e.g. RS232 with message relaying, or RS485, or mixed systems)
- Message Classes: Each message starts with an ID describing the purpose of the message, e.g. data write commands, data read requests, operations, datagram, etc.
- Error detection: With each message, a 16bit checksum is transferred to allow for detection of transmission errors. Whether false messages can be discarded or must trigger a suitable reply to the sender depends on the respective device requirements.

Document Version History

Issue	Date	Author	Annotations
01	2019-06-28	AT	First issue for HRT-2.1.1, new document naming scheme
02	2020-10-26	AT	Add descriptions of SYNCRO specific register types 0xae, 0xe0, 0xe1 and 0xe4 (p 10)
03	2021-03-31	AT	Clarify CRC related information

Protocol definition

Protocol layer structure



The device functions and parameters are represented by registers organized in a tree-like structure. Read and write operations on these registers involve messages passed between the device and the remote entity. The integrity of each message frame is ascertained by a CRC16-XMODEM checksum.

Message Structure

A message consists of the following elements:

MESSAGE:	[SOT][BODY][EOT]
BODY:	[HEADER][COMMAND][DATA][CRC16]
HEADER:	[DESTINATION][SOURCE]
DESTINATION:	[byte]
SOURCE:	[byte]
COMMAND:	[byte]
DATA:	[byte] | [byte][DATA]
CRC16:	[MSB][LSB]
LSB:	[byte]
MSB:	[byte]
SOT:	[byte]
EOT:	[byte]

Note: Mnemonics used in this document are not related to similar or identically sounding mnemonics commonly known from other character tables (ASCII) or protocol specifications

Note: Multi-byte data like *word* (unsigned 16 bit) or *long int* (signed 32bit) are always transmitted LSB first (little-endian). The only exception to this is the CRC16 code that is transmitted MSB first (big-endian).

The complete message body including the checksum is subject to character escaping, i.e. replacing characters with special meaning (e.g. SOT, EOT) by a sequence of escape marker (SOE) and disambiguated character (*character value* BITWISE-OR *64*).

Destination and source IDs must be unique throughout the system and may be configurable in the individual devices. In general, a message sent from A to B can only result in a response from B to A. However, a specific device may react to certain messages sent to another destination ID than its own. One special destination address (FF hex) is reserved for broadcasting messages to all devices. In general, using the broadcast address in a peer-to-peer-only setup is good practice allowing for a simplified driver architecture.

Message escaping/unescaping

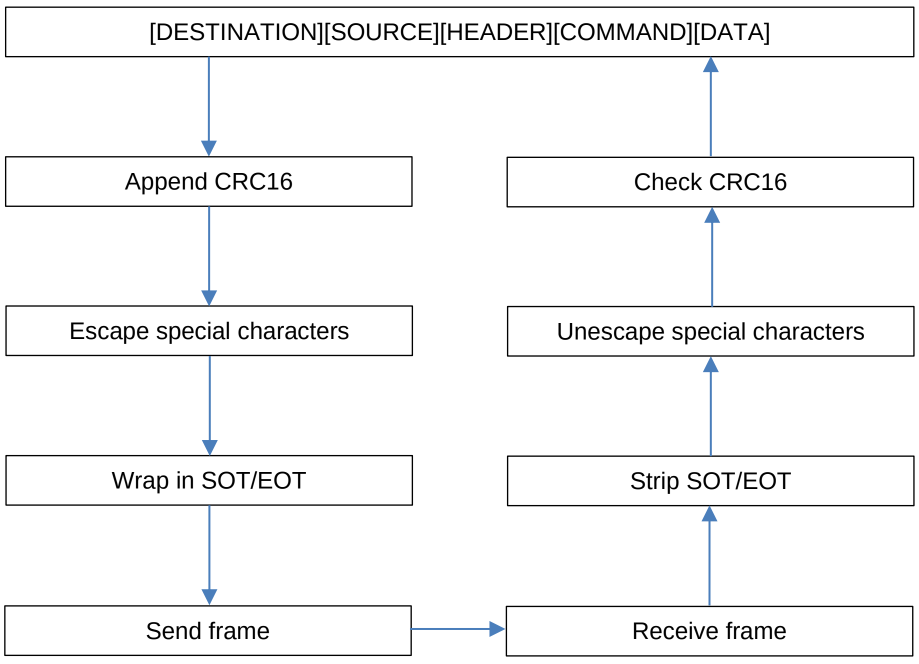
Any one character in the message body (including header and checksum) that matches any of the characters in the following table must be preceded with a SOE and the character itself must be masked by adding the value 64 (Note that this excludes any character value above 255-64=191 from escaping)

Mnemonic	Dec	Hex	Char	Description
EOT	10	0A	'\n'	End of transmission
SOT	13	0D	'\r'	Start of transmission
X-ON	17	11	Ctrl+S	Used in software flow control as "transmission on" signal
X-OFF	19	13	Ctrl+Q	Used in software flow control as "transmission off" signal
ECC	64	40		
SOE	94	5E		Start of Escape Sequence

Note: Some devices do not implement software flow-control and thus do not care about X-ON/X-OFF character escaping. In that case, software flow-control cannot be used safely. Refer to the device specific documentation for details. If not documented, the default is to not use software flow control.

When receiving a transmission, all characters between SOT and EOT need to be scanned for occurrences of SOE. The SOE itself is to be discarded and the following character must be unmasked by subtracting the value 64. The result need not be a character from the above table (though it usually is).

Message frame life cycle


[DESTINATION][SOURCE][HEADER][COMMAND][DATA]
Append CRC16
Escape special characters
Wrap in SOT/EOT
Send frame
Receive frame
Strip SOT/EOT
Unescape special characters
Check CRC16

Error protection

Before message escaping, the CRC-16/XMODEM checksum of a message is calculated and appended to the message. After receiving and unescaping, this transmitted code (calculated by the sender) can be checked against the code calculated by the receiver to check for transmission problems. Calculating the checksum of all received unescaped data including the original checksum should result in 0x0000 if the communication was without errors.

In general, the checksum code `crc` is calculated by setting `crc` to zero and updating the value with every character in the message as shown in the following code examples. Please note that these examples mainly serve for demonstration purposes. Wherever possible, the use of test-proven libraries shall be considered for production use:

- C-code optimized for µControllers, uses TXraw() to add single bytes to the serial transmit buffer:

```
word crc16_add_1(byte datagram, word crc)
{
  crc = (byte)(crc >> 8) | (crc << 8);
  crc ^= datagram;
  crc ^= (byte)(crc & 0xff) >> 4;
  crc ^= crc << 12;
  crc ^= (crc & 0xff) << 5;
  return crc;
}

static unsigned int tx1(byte datagram, unsigned int crc)
{
  crc = crc16_add_1(datagram,crc);
  if(datagram == SOT ||
     datagram == EOT ||
     datagram == SOE ||
     datagram == XON ||
     datagram == XOFF)
  {
    datagram |= ECC;
    TXraw(SOE);
  }
  TXraw(datagram);
  return crc;
}

void protocol_send(byte dest, byte src, word size, byte cmd, byte *datagram)
{
  unsigned int crc_value;
  byte tt;

  TXraw(channel, SOT);                             // Start a new telegram
  crc_value = tx1(channel, dest, 0);               // Transmit destination address
  crc_value = tx1(channel, src, crc_value);        // Transmit source address
  crc_value = tx1(channel, cmd, crc_value);        // Transmit command
  for(tt=0; tt<size; tt++)
  {
    crc_value = tx1(channel, *datagram,crc_value); // Transmit data
    datagram++;
  }
  TXnocrc(channel, (byte)((crc_value>>8) & 0xFF)); // MSB of CRC
  TXnocrc(channel, (byte)(crc_value & 0xFF));      // LSB of CRC
  TXraw(channel, EOT); // End telegram
}
```

  In this example, the types “byte” and “word” denote unsigned 8-bit and 16-bit, respectively. Modifications to the example code may be required for specific architectures so that correct checksum codes are produced.

- Python code example for calculation the checksum from a list of 8-bit values:

```
    def checksum(v):
        """ Calculate CRC-16/XMODEM checksum.

        Appending the checksum to the data MSB first and calculating the checksum again
        will result to 0.

        :param list[int] v: Data
        :return: 16-bit checksum (CRC-16/XMODEM)
        :rtype: int
        """
        crc = 0
        for i in v:
            crc = crc << 8 | crc >> 8
            crc ^= i
            crc ^= (crc & 0xff) >> 4
            crc ^= crc << 12
            crc ^= (crc & 0xff) << 5
            crc &= 0xffff  # make sure to only keep the lower 16 bits
        crc = int(crc)
        return crc
```

- Example patterns for validating custom implementations:

Byte-Array	Checksum
0x00	0x0000
0x01	0x1021
0x01, 0x10, 0x21	0x0000
0x42, 0x11, 0x04, 0x0F, 0x06	0x94C0
0x42, 0x11, 0x04, 0x0F, 0x06, 0x94, 0xC0	0x0000

more patterns can be generated using tools for CRC-16/XMODEM calculation, e.g. at https://crccalc.com/ (no affiliation).

Commands

The command identifier is used to denote the general purpose of the message; each data may be followed by command-specific data:

ID	Command	Remarks
0	Nack	Sent in response to a read or write command if requested operation failed Data: [Read|Write][REGISTER][ERRCODE]
1	CRC error	Sent in response to a command with mismatched CRC (no additional data)
3	Ack	Sent in response to a write command if requested operation was successful
4	Read	Requests data readout from specific device registers
5	Write	Change data or operational state in the device
8	Datagram	Contains data sent in response to a read request
9	Echo	Requests the destination device to send back the supplied data. Any kind of data up to 100 bytes may be sent.
10	Reply	Sent in response to an Echo command. Contains the message size, message header and all data from the echo command

Note: The Echo/Reply commands can be used to test the communication channel or synchronize the protocol after an error occurred.

Error Codes

If an error occurs, a Nack reply will be sent in response to a Read command (instead of a Datagram) or a Write command (instead of an Ack). In any case, the Nack message contains the first element of the register address specified in the Read/Write command and an optional error code:

Error	Code	Remarks
NOERROR	0x0000	Not sent in any reply, command Ack is used instead
BUFFER_OVERFLOW	0x0001	Severe internal error, should not occur in normal operation
NOT_WRITABLE	0x0002	A write operation was requested for a RO register
ARGSIZE_LOW	0x0003	Too few arguments were supplied to the Write command
ARGSIZE_HIGH	0x0004	Too many arguments were supplied to the Write command
INTERNAL_ERROR	0x0005	An error in the internal data structures was detected; should not occur in normal operation
AUTHORIZATION_REQUIRED PROTERR_NOT_READABLE PROTERR_WRONG_ARGUMENT	0x0006 0x0007 0x0008	The specified register can only be written to in service mode The specified register is write-only At least one of the given arguments is wrong, i.e. out of range

It is important to note that a CRC16 mismatch will not result in any response from the device at all. In this case, waiting for a reply from the device (Ack, Nack or Datagram) will lead to a timeout.

Typical device specific specifications

Typically Menlo devices are equipped with a standard RS232 jack (D-SUB-9). The serial link can be established with the following settings:

115200 baud, 8N1, no handshake (message integrity check is performed based on CRC16 codes)

Alternatively (but not simultaneously), the device can be connected to the host PC via the USB connector. The USB interface comprises a USB-to-serial converter configured to the same settings as the RS232 port.

If not specified otherwise, no software flow control is used.

Device registers

Device registers are used to read data from the device and write data to the device.

Contents of device registers can be requested using the read command and the register ID as command data (with HID: Host ID, DID: Device ID):

[HID][DID][04][ADDRESS bytes][CRC msb][CRC lsb]

This command will trigger a datagram message to be sent back from the device:
[DID][HID][08][ADDRESS 1st byte][DATA][..][CRC msb][CRC lsb]

Example Communication

The following example shows how to read out the *Device Date Register* present in some devices (please note that not all devices do implement this register, refer to the device register tree specification for relevant registers):
Address: `0x0f 0x06`
Destination: `0x42`
Source: `0x11` (escaped `0x5e 0x51`)

Sent (`0x04`) message: `0d 42 5e 51 04` **`0f 06`** `94 c0 0a`
Received (`0x08`) message (datagram): `0d 5e 51 42 08` **`0f`** `0e 0c 09 77 cc 0a`

⇨	DATE.day:	14 (`0x0e`)
⇨	DATE.month:	12 (`0x0c`)
⇨	DATE.year:	09 (`0x09`)

Please note byte **`0f`** in the received answer. It indicates the top-level node of the requested register path. Before verifying the checksum in these examples, the message requires unescaping, i.e. replacing the sequence `0x5e 0x51` with `0x11`.

Minimal device register tree

The following table shows a list of the minimal device register tree. Each register is addressed by a path comprising 1..n – bytes (column “hex”). The allowed operations (read/write) are shown in the “R/W” column. The “struct” ID denotes the underlying data structure (hex IDs are described in detail in section “Definition of Register Types”).

Certain entries represent nodes in the register tree that have subnodes but cannot be read/written in their own right. In the table, the register “DEV” (marked in blue) serves as an example.

Register	hex	R/W	TypeID	Description
DEV	0x0f	--	0x02	Node for sub-registers below
Addr	0x0f 0x01	rw	0x07	Address
Type	0x0f 0x02	r-	0x08	device type
Serial	0x0f 0x03	rw	0x09	serial number
saveset	0x0f 0x04	-w	0x50	save settings for system including all subsystems
Tstamp	0x0f 0x08	r-	0x0d	Timestamp
Uptime	0x0f 0x11	r-	0x0d	Timestamp
ID	0x0f 0x0a	r-	0x0f	String
Ver_HW	0x0f 0x0b	rw	0x10	Version
Ver_FW	0x0f 0x0c	r-	0x10	Version
RegVers	0xfd	r-	0x03	version register tree engine
Subregs	0xfe	r-	0x04	subregisters (childs)
RegDef	0xff	r-	0x05	register type

Note: RO registers can be read from at any time. If the specified register does not exist, the reply message will be NACK with the first register node address as argument.

Note: Writing to a RW register will result in changing the content of the respective register or trigger a state change of the related subdevice. Each write command is responded to by an ACK or NACK message that can be used solely for detection of protocol errors. To verify if the write command succeeded to change the register content or invoke a state change, the respective register needs to be monitored.

Note: The node DEV may be located at register “0x0f” as shown above, but this may vary depending on the device. For the register tree of a specific device, please refer to the device’s register tree specification.

Definition of Register Types

The following table contains definitions for all known register types.

TypeID	Mnemonic	Structure	Description
0x01	NONE	-	None (feature not yet implemented)
0x02	NODE	-	node
0x03	REGVERS	U8	version register engine
0x04	SUBREGS	NxU8	List of subregs (N: number of subregisters)
0x05	REGDEF	RGIF	register info
0x07	ADDRESS	U8	address
0x08	TYPE	U16	device type
0x09	SER	SERS	serial number
0x0a	REMOTE_SERVICEMODE	U8	state service mode
0x0b	RTCTIME	DATE	RTC date
0x0c	RTCDATE	TIME	RTC time
0x0d	TSTAMP	TSTAMP	timestamp
0x0f	DEVID	Cstring	character string
0x10	VERS	VERS	version
0x11	LOGENTRY	LOGENTRY	Entry of error/log history
0x20	EDIP_BMP	-	eDIP bitmap
0x21	EDIP_FW	-	eDIP firmware
0x30	REMOTE_EDIP	U8	state menu/eDIP
0x31	REMOTE_AMP	U8	state
0x32	REMOTE_SEED	U8	state seed
0x32	REMOTE_SPI	-	SPI tunnel
0x50	SAVESETTINGS	U8	save
0x51	U16_mA	U16	Current (mA)
0x52	CALIB_DAC	DACS	calibration D/A-converter
0x53	CALIB_ADC	ADCS	calibration A/D-converter
0x54	S32_mV	S32	Voltage in mV
0x55	U08_enum	U8	Enumeration
0x56	U08_bool	U8	Binary value
0x57	S32_ms	S32	Time in ms
0x58	S32	S32	Discrete Stepping
0x59	S32_mC	S32	Temperature in m°C
0x5a	S32_uC	S32	Temperature in µ°C
0x5b	U32_Hz	U32	Frequency in Hz
0x5c	U32_mHz	U32	Frequency in mHz
0x5d	U32_kHz	U32	Frequency in kHz
0x5e	U32_mW	U32	Power in mW
0x5f	S32_uV	S32	Voltage in µV
0x60	U32_us	U32	Time in us
0x61	S32_mA	S32	Current in mA
0x62	S32_mW_K	S32	Heat flow rate in mW/K
0x63	S32_mJ_K	S32	Heat capacity in mJ/ K
0x64	U16_pm	U16	Value in permille
0x65	U8_pc	U8	Value in percent
0x66	U16_pmy	U16	Value in permyriad
0x67	S32_uA	S32	Current in µA
0x68	U16_mV	U16	Voltage in mV
0x69	cmd_val	U8_U16	Command (Byte) and Value (16 Bit)
0x6a	U32_cW	U32	Power in cW
0x6b	U32_uW	U32	Power in µW
0x70	AmpConf	U8	Bit-mask denoting diodes in diode-group
0x80	U16_Hz	U16	Frequency in Hz
0x81	U16	U16	Generic value
0x82	U16_ms	U16	Time in ms
0x83	8xU8	8xU8	8 generic Bytes
0x90	SP64	2xS32	Signed double value given by a+b/1000000
0x91	UP64	2xU32	Unsigned double value given by a+b/1000000
0xab	PUMPDIODELOG	PUMPDIODELOG	
0xac	SYNCRO_TECLOGALL	SYNCRO_TECLOGALL	
0xad	SYNCRO_TECLOG	SYNCRO_TECLOG	
0xae	SYNCRO_TRACKLOG	SYNCRO_TRACKLOG	
0xaf	SYNCRO_TRACKTECLOG	SYNCRO_TRACKTECLOG	
0xe0	MLD_AC_WEIGHT	4xS32	
0xe1	MLD_AC_LEVELS	8xS32	
0xe2	MLD_ML_LEVELS	MLD_ML_LEVELS	
0xe3	FXMHIST	FXMHIST	
0xe4	LOGHIST	LOGHIST	
0xf?	SANDBOX	Arbitrary	0xf0…0xff: Reserved, do not use

Definition of Basic Data Types

Data type	Structure		Description
U8	1 byte	Byte, [0..2^8-1]	
U16	2 byte	Word, [0..2^8-1], LSB first	
U32	4 byte	Long, [0.. 2^{16}-1], LSB first	
S32	4 byte	Lint, [-2^{15}.. 2^{15}-1], LSB first, two's complement	
Cstring	X byte	Zero terminated string of characters	

Definition of Data Structures

Data type	Structure		Description
RGIF	(2+x) byte U8 Cstring U8	 register type label RW	Register type data structure register type (see table), e.g. `0x60` register label, e.g. "“`DEV_ADDR`”. Read/write direction
TCDS	22 byte U16 U16 U16 S32 S32 S32 S32	 controller_state warnings tset divider iact tact tact_dev	TC state and measurement monitor data structure state of the controller state machine bitfield containing warnings (see below) devide by 1'000 to get temperature controller preset value in °C determines number of measurements in running average (if > 1) devide by 1'000*divider to get value in Ampere devide by 1'000*divider to get temperature reading in °C devide by 10'000*divider to get deviation from tset in K
DACS	20 byte S32 S32 S32 S32 U16 U16	 raw_0 phys_0 raw_1 phys_1 range_min range_max	Digital-Analog-Converter adjustment data structure adjustemt point 1, DAC raw value adjustemt point 1, physical value adjustemt point 2, DAC raw value adjustemt point 2, physical value physical value valid minimum physical value valid maximum
ADCS	16 byte S32 S32 S32 S32	 raw_0 phys_0 raw_1 phys_1	Analog-Digital-Converter adjustment data structure adjustemt point 1, DAC raw value adjustemt point 1, physical value adjustemt point 2, DAC raw value adjustemt point 2, physical value
TCSPS	6 byte U16 U16 U16	 controller_state warnings tset	TC state and preset data structure state of the controller state machine bitfield containing warnings (see below) devide by 1'000 to get PID temperature preset value °C
DATE	3 byte U8 U8 U8	 day month year	Date data structure day of month [1-31] month [1-12] year [2000-2099]
TIME	3 byte U8 U8 U8	 hour min sec	Time data structure hour [0-23] minute [0-59] second [0-59]
TSTAMP	6 byte S32 S16	 sec msec	Time stamp data structure (time since… [epoche or time difference]) seconds [-2^{31}.. 2^{31}-1], milliseconds [-999…999]
CNFS	4 byte U16 U16	 device_options imax	Temperature controller unit configuration data structure device option bitfield maximum current for TEC/heater element in mA
VERS	4 byte U8 U8 U8 U8	 build patchlevel minor major	Version information data structure
SERS	4 byte U8 U8 U16	 year month serial	Serial number data structure
OPS	4 byte U32	 Operation flags	Device configuration Operation flags (see below)
LOGENTRY	16 byte TSTAMP U8 U8 S32 S32	 context event val ref	Entry of log history timestamp context of event event code, context specific data value involved in case, reference value that was exceeded
PUMPDIODELOG	16 byte S32 S32 S32 S32 S32	 Iset Tset Iact Tact Pact	LDMV monitoring structure mA m°C µA m°C a.u.

Data type	Structure		Description
SYNCRO_TECLOGALL			
SYNCRO_TECLOG			
SYNCRO_TRACKLOG	16 byte		Tracker monitoring structure
	S32	LB.out	µV
	S32	Track1.pos	(position)
	S32	LB.in	µV
	S32	Track2.pos	(position)
SYNCRO_TRACKTECLOG			
MLD_AC_WEIGHT	16 byte		ML-value calculation-weights structure
	S32	w1	
	S32	w2	
	S32	w3	
	S32	w4	
MLD_AC_LEVELS	32 byte		Modelock detector readings (may be different for specific devices)
	S32	ML	Combined modelock-level
	S32	f0	Power level in the fundamental frequency (laser repetition rate)
	S32	fn	Power level in one of the harmonic frequencies
	S32	noise	Noise floor below the fundamental frequency
	S32	DC	Average power level
	S32	TPA	Two-photon-detector level (if applicable)
	S32	Pout	Power at laser head output (if applicable)
	S32	CW	CW-spike detector value
MLD_ML_LEVELS			
FXMHIST			
LOGHIST	16 byte		Event log buffer structure (for interpretation, contact Menlo Systems)
	U32	seconds	System time of log entry (seconds + milliseconds)
	U16	milliseconds	
	U8	remain	Remaining entries in buffer (>0: continue reading)
	U8	level	
	U8	context	
	U8	subcontext1	
	U8	subcontext2	
	U16	msg-ID	
	U16	lineno	
	S32	value	
	S32	ref	

HRT Introspection

All RBP protocol versions with HRT support introspection by providing special registers 0xfe and 0xff. These two registers can be used to readout the complete HRT structure from a device, including the register clear text name, read/write operation permissions and type-id of the underlying data. While all other registers may be relocated to different paths and their name may be changed from one firmware revision to the next, the introspection registers will always be kept at the same location. A third register located at 0xfd allows querying the RBP version. A full-featured driver should always use the RBP version number to choose between implementations.

Reading the register **0xfe** will return the main nodes of the HRT, to read the sub registers of a node read **0xfe** followed by a node address.

Examples (please note that the shown registers may not necessarily be present in all devices or may have a different function):
Reading register 0xfe:
Send: `0d 42 5e 51 04 fe 35 b2 0a`
Received message: `0d 5e 51 42 08 fe 01 02 05 06 6a 6b 6c ac fc fd fe ff 22 31 0a`

⇨ `Subregs: 0x01 0x02 0x05 0x06 0x6a 0x6b 0x6c 0xac 0xfc 0xfd 0xfe 0xff`

Reading register 0xfe 0x05:
Send: `0d 42 5e 51 04 fe 05 84 53 0a`
Received: `0d 5e 51 42 08 fe 01 02 03 05 06 10 11 12 51 5a 0a`

⇨ `Subregs:0x01 0x02 0x03 0x05 0x06 0x10 0x11 0x12`

Reading the register 0xff followed by a specific register address will return the type as a byte and the associated label of that register as a zero-terminated string. The allowed operations are returned in a 1-byte bitmask following the string.

Starting with HRT-2.1.1, the permission byte will carry extended permission flags after a *protocol upgrade* is performed by writing 3 bytes [0, 0, 1] to register **0xfd**:

Bit	Direction
0-1	1: R/W 2: Read-only 3: Write-only
2	SU: register requires service mode for writing
3	OP: register reflects an operational state rather than a device configuration parameter
4	TC: changing the register contents may affect the register tree composition

Examples:
Reading register 0xff 0x05:
Send: `0d 42 5e 51 04 ff 05 b7 62 0a`
Received: `0d 5e 51 42 08 ff 02 4d 4f 54 4f 52 30 00 00 29 73 0a`

⇨ `RegDef.type: 0x02`
⇨ `RegDef.label: MOTOR0`
⇨ `RegDef.perm: --`

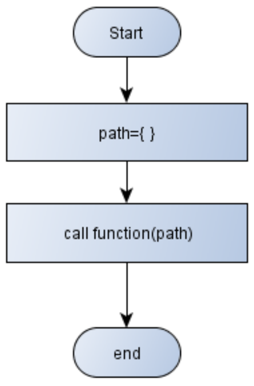
Reading register 0xff 0x05 0x01:

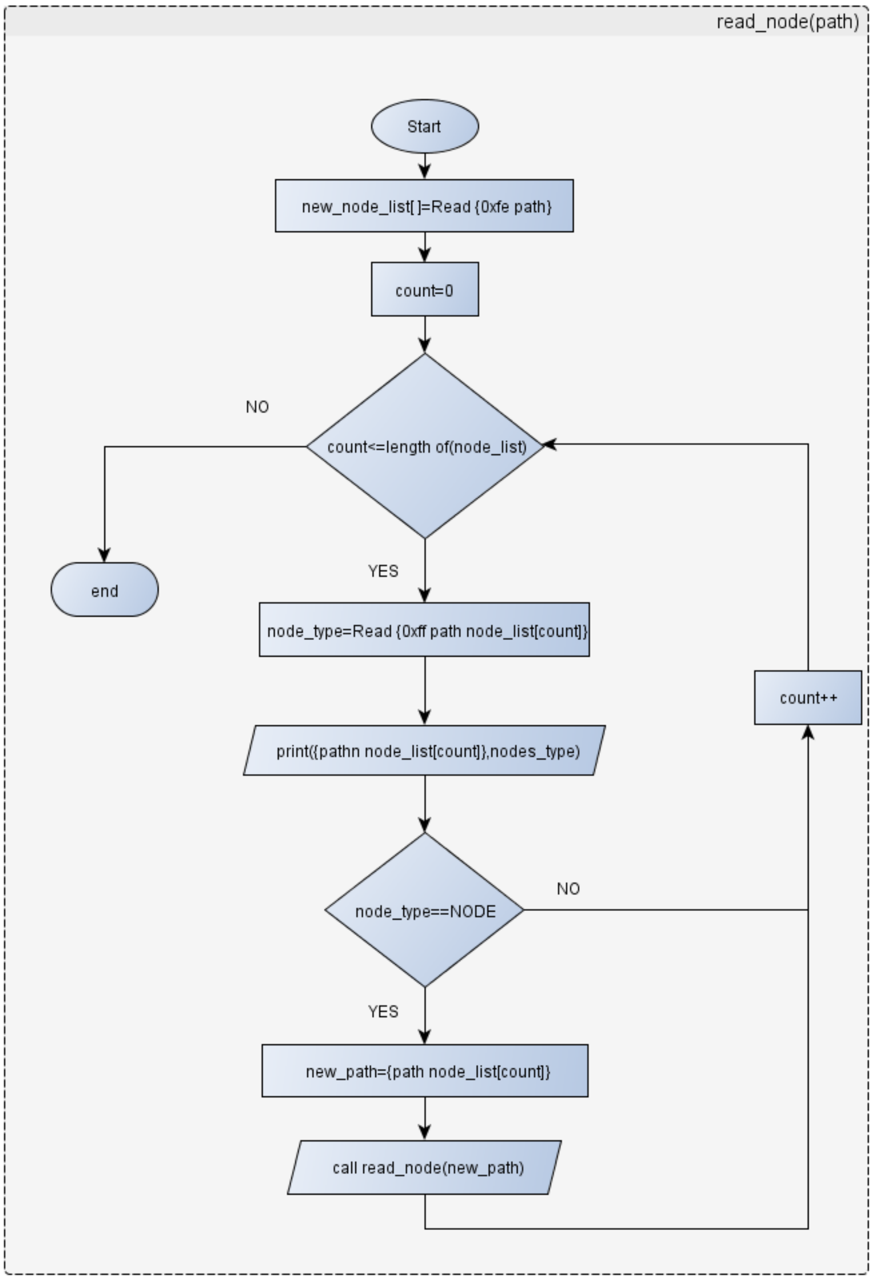
Send: `0d 42 5e 51 04 ff 05 01 a5 1d 0a`
Received: `0d 5e 51 42 08 ff 58 50 4f 53 49 54 49 4f 4e 00 01 0f 2b 0a`

⇨ `RegDef.type: 0x58`
⇨ `RegDef.label: POSITION`
⇨ `RegDef.perm: RW`

Note: If you try to query information on a non-existing register using 0xfe or 0xff, the device returns nothing.

The flow chart given below describes how to extract a complete register tree using registers 0xfe and 0xff.




read_node(path)
Start
new_node_list[ ]=Read {0xfe path}
count=0
count<=length of(node_list)
NO
end
YES
node_type=Read {0xff path node_list[count]}
print({pathn node_list[count]},nodes_type)
node_type==NODE
NO
count++
YES
new_path={path node_list[count]}
call read_node(new_path)

Device Recognition

Register	hex	R/W	Structure	Description
DEV_ADDR	0x0f 0x01	RW	1xU8	device address, factory default is set to hex 0x42
DEV_TYPE	0x0f 0x02	RO	1xU16	device type
DEV_ID	0x0f 0x0a	RW	Cstring	device ID
DEV_HW	0x0f 0x0b	RW	VERS	device HW version
DEV_FW	0x0f 0x0c	RO	VERS	device FW version
DEV_SERIAL	0x0f 0x03	RW	SERS	device serial number
DEV_CONF	0x0f 0x31	RO	1xOPS	device configuration

Reading from register DEV_ID returns a zero-terminated string of characters in the form
“Manufacturer,Device Name,Serial number,Firmware version (Compilation Date)”
In case of a SYNCRO unit this would read something like:

```
“Menlo Systems GmbH,SYNCRO,LE0011209,1.0.0 (Jun 1 2010)”
```

Additionally, the device type can be read from register DEV_TYPE (see appendix A).

Appendix A: Protocol commands C-header file

```
#ifndef PROTOCOL_HEADER
#define PROTOCOL_HEADER

#define MENLO_DEVICE_TYPE_SMA1000    0x0100
#define MENLO_DEVICE_TYPE_THETA      0x0200
#define MENLO_DEVICE_TYPE_AC1550     0x0300
#define MENLO_DEVICE_TYPE_ORANGE_1   0x0400
#define MENLO_DEVICE_TYPE_ORANGE_A   0x0500
#define MENLO_DEVICE_TYPE_FIBERLINK  0x0600
#define MENLO_DEVICE_TYPE_LFC_REC    0x0700
#define MENLO_DEVICE_TYPE_SYNCRO     0x0800

#define PROTO_CMD_NACK     0x00
#define PROTO_CMD_CRCERR   0x01
#define PROTO_CMD_ACK      0x03
#define PROTO_CMD_READ     0x04
#define PROTO_CMD_WRITE    0x05
#define PROTO_CMD_DATAGRAM 0x08
#define PROTO_CMD_ECHO     0x09
#define PROTO_CMD_REPLY    0x10

#define PROTERR_NOERROR                0x00
#define PROTERR_BUFFER_OVERFLOW        0x01
#define PROTERR_NOT_WRITABLE           0x02
#define PROTERR_ARGSIZE_LOW            0x03
#define PROTERR_ARGSIZE_HIGH           0x04
#define PROTERR_INTERNAL_ERROR         0x05
#define PROTERR_AUTHORIZATION_REQUIRED 0x06


#endif
```

Appendix B: Register type definition C-header file

```
enum
{
  REGDEF_NONE                         = 0x01,
  REGDEF_NODE                         = 0x02,
  REGDEF_REGVERS                      = 0x03,
  REGDEF_SUBREGS                      = 0x04,
  REGDEF_REGDEF                       = 0x05,
  REGDEF_ADDRESS                      = 0x07,
  REGDEF_TYPE                         = 0x08,
  REGDEF_SER                          = 0x09,
  REGDEF_REMOTE_SERVICEMODE           = 0x0A,
  REGDEF_RTCTIME                      = 0x0B,
  REGDEF_RTCDATE                      = 0x0C,
  REGDEF_TSTAMP                       = 0x0D,
//REGDEF_TDIFF                        = 0x0E,
  REGDEF_DEVID                        = 0x0F,
  REGDEF_VERS                         = 0x10,
  REGDEF_LOGENTRY                     = 0x11,
//REGDEF_LOG                          = 0x11,

  REGDEF_EDIP_BMP                     = 0x20,
  REGDEF_EDIP_FW                      = 0x21,

  REGDEF_REMOTE_EDIP                  = 0x30,
  REGDEF_REMOTE_AMP                   = 0x31,
  REGDEF_REMOTE_SEED                  = 0x32,
  REGDEF_REMOTE_SPI                   = 0x33,

  REGDEF_SAVESETTINGS                 = 0x50,
  REGDEF_U16_mA                       = 0x51,
  REGDEF_CALIB_DAC                    = 0x52,
  REGDEF_CALIB_ADC                    = 0x53,
  REGDEF_S32_mV                       = 0x54,
  REGDEF_U08_enum                     = 0x55,
  REGDEF_U08_bool                     = 0x56,
  REGDEF_S32_ms                       = 0x57,
  REGDEF_S32                          = 0x58,
  REGDEF_S32_mC                       = 0x59,
  REGDEF_S32_uC                       = 0x5a,
  REGDEF_U32_Hz                       = 0x5b,
  REGDEF_U32_mHz                      = 0x5c,
  REGDEF_U32_kHz                      = 0x5d,
  REGDEF_U32_mW                       = 0x5e,
  REGDEF_S32_uV                       = 0x5f,
  REGDEF_U32_us                       = 0x60,
  REGDEF_S32_mA                       = 0x61,
  REGDEF_S32_mW_K                     = 0x62,
  REGDEF_S32_mJ_K                     = 0x63,

  REGDEF_U16_Hz                       = 0x80,
  REGDEF_U16                          = 0x81,
  REGDEF_U16_ms                       = 0x82

  // type IDs above and including E0 are reserved!!
};
```