REA Elektronik GmbH

REA PI Library 3.6

REA PI Library

REA JET HR, HR pro, CL

www.rea.de

High-Tech für die Praxis

REA Elektronik GmbH
Teichwiesenstrasse 1
D-64367 MÜHLTAL

Tel.: +49 6154-638-0
Fax: +49 6154-638-195
E-Mail: reainfo@rea.de

REA JET

REA-Elektronik GmbH

Teichwiesenstraße 1

64367 Mühltal

Tel.:+49 6154 638 0

Document Version 3.6.6

Version	Author	Date	Annotation
1.0.2	VH	2009-11-26	1.0 release
1.60	MKU	2009-12-14	1.6 release
1.70.1	MKU	2010-05-20	1.7 release
1.80.0	MKU	2010-11-18	1.8 release
2.1.0.0	MKU	2010-12-10	2.1 release
3.22.2	KD	2013-01-28	3.2 release
3.3.5.1	KD	2014-11-20	3.3 release
3.4.01	KD	2015-11-23	Command GetDeviceInfo aktualisiert: neue Eigenschaften Artikelnummer und FPGA-Version
3.5.0.0	KD MMa	2015-07-13 2015-06-25	Review; new Function Get-/SetNTPConfig
3.6.0.0	KD	2015-12-08	Funktionalität Kommando TRIGGERGENERICEVENT und Event GENERICEVENT
3.6.0.1	KD	2015-12-22	Minor Bugfixes in passage control of word
3.6.0.2	KD	2016-01-18	Minor Bugfixes in passage Event GENERICEVENT, version update to 3.6
3.6.0.4	KD	2016-04-28	Getting and setting of product sensor and digital I/O levels
3.6.1.2	KD	2016-05-04	Installation, description provided software

Table of Contents

1 Introduction

The library implements the REA-PI (XML) communication protocol.

The REA-PI Library supports Linux 2.6 and Windows XP, Windows7 and Windows8 operating systems.

The library API is implemented as a C language API and can be used from many programming languages.

For C# / .NET a wrapper ("reapisharp.dll") of the REA-PI Library is available. The usage of the API is mostly identical to the C language API, disregarding the language syntax.

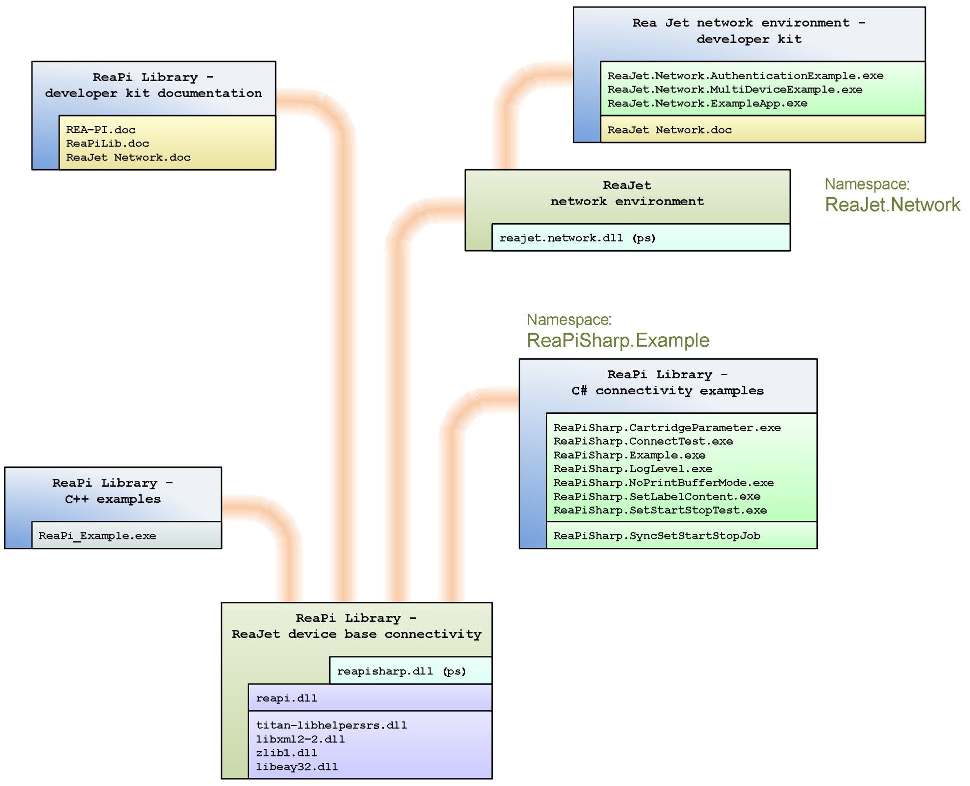
Commands can be sent in synchronous as well as in asynchronous mode. Currently, only asynchronous mode is implemented.

2 Installation

For using the REA-PI Library at least the **ReaJet device base connectivity** components have to be installed, i.e., the main library reapi.dll and all its dependencies.

ℹ	If you'd like to execute an application in your development environment, make sure that all dependencies may be resolved and loaded! It is recommended to copy reapi.dll, reapisharp.dll and all necessary libraries in your debug or release directory (listed under **`ReaJet device base connectivity`** in the following diagram).

For this purpose, C++ developers need the include headers and lib file to be linked:


Rea Jet network environment - developer kit
ReaJet.Network.AuthenticationExample.exe
ReaJet.Network.MultiDeviceExample.exe
ReaJet.Network.ExampleApp.exe
ReaJet Network.doc
ReaPi Library - developer kit documentation
REA-PI.doc
ReaPiLib.doc
ReaJet Network.doc
ReaJet network environment
reajet.network.dll (ps)
Namespace: ReaJet.Network
Namespace: ReaPiSharp.Example
ReaPi Library - C# connectivity examples
ReaPiSharp.CartridgeParameter.exe
ReaPiSharp.ConnectTest.exe
ReaPiSharp.Example.exe
ReaPiSharp.LogLevel.exe
ReaPiSharp.NoPrintBufferMode.exe
ReaPiSharp.SetLabelContent.exe
ReaPiSharp.SetStartStopTest.exe
ReaPiSharp.SyncSetStartStopJob
ReaPi Library - C++ examples
ReaPi_Example.exe
ReaPi Library - ReaJet device base connectivity
reapisharp.dll (ps)
reapi.dll
titan-libhelpersrs.dll
libxml2-2.dll
zlib1.dll
libeay32.dll

Additionally the **C++ connectivity examples** will be installed containing a small example program which demonstrates the use of the reapi.dll introductorily.

Developers using .NET C# or VB.NET only will reference the C# wrapper reapisharp.dll in their project in order to use the reapi.dll. They do not need any header or libraries.

Also for C# there are some small **C# connectivity examples** from above, which uses the reapisharp.dll and in consequence the reapi.dll.

Developers using .NET C# or VB.NET can use reajet.network.dll for using objects which are implementing all functionality from reapisharp.dll in a comfortable way.

Tbd installer result: folder description

```
Win7
  C:\Program Files (x86)\
    REA-JET
      ReaPi - Developer kit
        <VersionKey> eg. ‚3.6'
          bin
          doc
          licence

  C:\Users\kdiehl\AppData\Roaming
    REA-JET
      ReaPi - Developer kit
        <VersionKey> eg. ‚3.6'
          ??
          ??

  C:\Users\username\Documents
    REA-JET
      REA JET Label Creator
        data
          images
            blue_man_wink.jpg
            ...
        rea-jet
          bitmaps
          device
          fonts
            singleline
          installations
          jobs
          labels
          service
          vectorgraphics
      ReaPi - Developer kit
        ImageFilterBin.Sample
        ReaJet.Net.SampleApp
        ReaPiLib
          bin
          bindebug
          include
          lib
          exmples
            reapi_example
            ReaPiSharp_Example
```

Tbd client distribution, licence, disclaimer

Feel free to distribute and ship all libraries installed in document folder and used by your software. The appropriate files are located in reapilib/bin or -/bin64 or rea-jet.network/bin or -/bin64.

Note that, when an application using reapi.dll will be distributed; all libraries which are listed in the previous diagram have to be shipped as well with your software.

3 Basic API concepts

3.1 Content of Chapter

3.2 Introduction

The library itself is implemented in C++, whereas the API is implemented in C.

On Windows operating systems the __declspec(dllexport) __stdcall calling convention is used, which in principle allows other programming languages such as Visual Basic, C#, Delphi to use the library, too.

API-functions only use basic data types – integers, floats and strings. Complex data types such as structures and arrays (required for some commands), are created, accessed and destroyed by API-functions. For complex data-structures the creator-functions return a handle for a particular data object. This handle should also be used for accessing and destroying the corresponding data object.

All strings used in API-functions as input variables should be encoded as UTF-8 (7-bit ASCII-strings conform this requirement automatically). All strings returned by the API are encoded as UTF-8.

3.3 Communication Scenarios

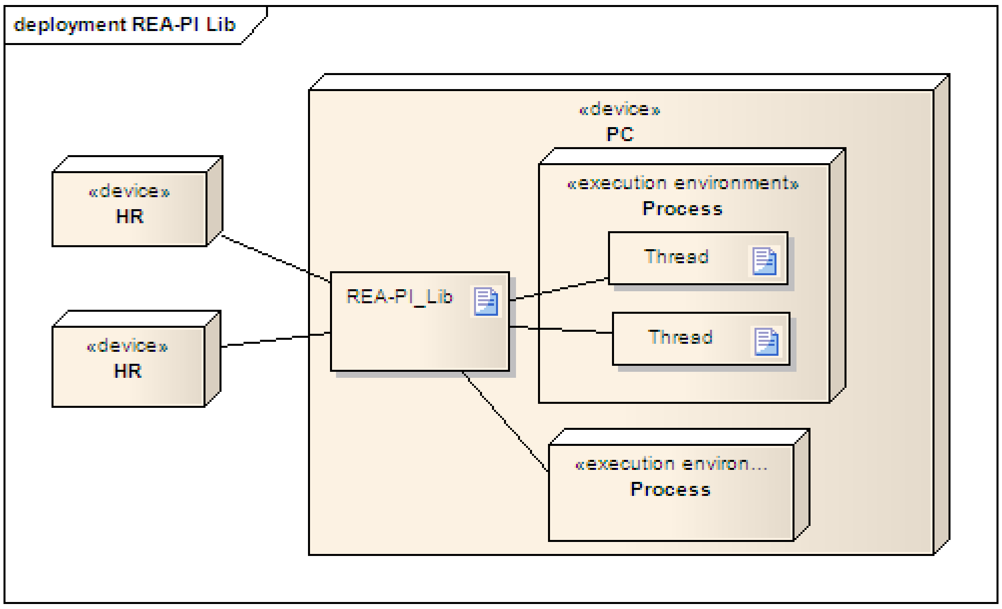


Figure 1: Communication scenarios using REA-PI Communication Library

As the Figure 1 suggests an external host (e.g., PC) can communicate with REA JET devices, e.g., CL, FL, HR (*pro*). Several processes on the same host may use the REA-PI Library for communication simultaneously. Several threads within one process are allowed. Only one thread can execute one command at the same time, all other threads will be blocked on accessing the library API.

3.4 API definition

The C language API functions are defined in the file **reapi.h**. The data types are defined in reapi_types.h. The error codes are defined in reapi_errors.h. The protocol is implemented in reapi.dll for Windows and libreapi.so for Linux respectively. The client code should be linked against reapi.lib.

More information about the used reapi.dll may be retrieved at runtime by informational library functions described later in this document.

3.5 Basic API types

Following types are commonly used in REA-PI functions:

- **`TConnectionId`** - signed integer type used to address an established connection. 0 means a pending connection, positive value should be interpreted as a connection handle, a negative number should be interpreted as error-code which corresponds to values in the type `TErrorCode`.
- **`TConStatus`** - signed integer type used in connection callback for state of connection

- **`TCmdId, TEventId`** - signed integer type used as unique identifier for REAPI Commands or Events.
- **`TErrorCode`** - signed integer type returned by API containing error-code. Zero means successful command execution. Negative value should be interpreted as error-code defined in file reapi-errors.h
- **`TResponseHandle`** - signed integer type to address device response-elements in asynchronous transfer mode. In synchronous transfer mode the `TResponseHandle` should be interpreted as `TErrorCode`.
  Valid response handles in asynchronous transfer mode are positive values including zero. Negative values should be interpreted as error-code like `TErrorCode` defined in file reapi-errors.h.
  In asynchronous transfer mode TResponseHandles are invalid outside of the corresponding callback routine. Although they can be saved for later use.
- **`TJobId, TEncoderId, TSensorId`** - signed integer type used as identifier for a job, an encoder or a product sensor on the HR device.
  Note: All versions of device firmware up to current version supports only one job, encoder and product sensor.
- **`TLabelContentHandle, TLabelObjectHandle`** - signed integer type to address a library internal container for label contents or label object, which are used to submit label property changes to the device.
- **`TLabelPropertyType`** - signed integer type to distinguish label property changes.

3.6 Basic library functions

3.6.1 *Querying Library Information*

Syntax:

```
REAPI_DLL const char* REAPI_LibInfo();
```

Returns:

`const char*`	Returns information like version, build-date, copyright etc. in a human readable form.

Syntax:

```
REAPI_DLL const char* REAPI_GetRevision();
```

Returns:

`const char*`	Return the library version/revision string of the locally used reapi.dll in LINUX format. For example: “3.30.52765”.

Remark:

The reapi.dll revision and the protocol version used for the communication must not be confused! E.g., if the revision of the reapi.dll is ‘3.30.52765’, a protocol version ‘3.3’ or smaller will be supported depending on the device communicated with.

Where the first digit is the major version, the following two digits concatenate minor and revision number. The next five digits represent the build number w.r.t. an internal version control system.

At last after a tilde '~' it's possible to get special version description, for example '~beta', '~alpha' or '~debug'.

3.7 Using Asynchronous Mode

Asynchronous mode means that a call to a function of the REA-PI Library immediately returns after starting a separate thread of execution. Hence the result of the function call does not denote the result of the corresponding action but rather a handle, which shall be used to retrieve the result of the action when it was executed. This is to be done within callback functions.

Callback functions should be implemented in client-code and registered in reapi DLL in order to get events as well as responses to asynchronous calls. All callback functions are called from a separate context, i.e., execution thread. Be careful with sharing data between the two contexts!

Three types of callback functions are defined: Connection, Event, and Response.

3.7.1 Connection callback

Syntax:

```
typedef void (REAPI* C_connectionCallbackPtr)(
   TConnectionId connection,
   TConStatus status,
   TErrorCode error,
   void* context );
```

`C_connectionCallbackPtr` – callback function pointer for notification of asynchronous connection and disconnection requests. Connection callback functions should be implemented by client. If asynchronous mode is desired, the pointers to these implemented functions should be handed over to REA-PI in API-function `REAPI_RegisterConnectionCallback()`.

Parameters:

`connection`	To be interpreted as connection id if this value is positive.
`status`	the connection status identifier showing whether the connection is • established with `CON_CONNECTED`. This state occurs always with no error. `positive handle, CON_CONNECTED, OK, void*` on successful connection request. • rejected with `CON_CONNECTIONERROR` • closed with `CON_DISCONNECTED` `positive handle, CON_DISCONNECTED, OK, void*` on graceful disconnection request.
`error`	The error code showing the occurrence of a problem. Occurs with states • `CON_CONNECTIONERROR` if a connection could not be established, for example when the device could not be found by IP address: Example: `invalidhandle, CON_CONNECTIONERROR, ERRORCDE_CANNOT_CONNECT_TO_DEVICE, void*` • `CON_DISCONNECTED` when an established connection breaks down because of an error, for example the network went down.

Example: `positive handle, CON_DISCONNECTED, ERRORCODE_TRANSMISSION_ERROR, void*`

`context`	Context pointer usually required for dispatching. The value is handed over previously by client to REA-PI and returned by REA-PI unchanged. It is not used in the library itself. As all callback-functions are called from another thread, the context pointer can be used to synchronize values back into the calling context. It might even simply hold a `this()` pointer, if used in an object oriented programming language.

```
void REAPI onConnectionChanged( TConnectionId connection,
                                TConStatus status,
                                TErrorCode error,
                                void* context )
{
   if (status == CON_CONNECTED)
   {
      printf( "Connection <%d> established\n", connection );

      REAPI_RegisterResponseCallback( connection,
                                      &onResponse, 0 );
      REAPI_RegisterEventCallback( connection,
                                   &onEvent, 0 );

      REAPI_SubscribeJobSet( connection, 1 );
      REAPI_SubscribeIOConfiguratonSet( connection, 1 );
   }
   else if (status == CON_DISCONNECTED)
   {
      if (error == ERRORCODE_OK)
      {
         // graceful disconnect, probably triggered
         //   by self-called REAPI_Disconnect( connection );
         return;
      }

      // sudden disconnect reasoned by network down,
      //    did someone stumble over the network cable?
   }
   else if (status == CON_CONNECTIONERROR)
   {
      // the connection could not be established
      //   maybe IP address wasn't correct?
   }
}
```

Example 1: Connection callback implementation

3.7.2 Register a connection callback

Syntax:

```
typedef void (REAPI* C_connectionCallbackPtr)(
   TConnectionId connection,
   TConStatus status,
   TErrorCode error,
   void* context );

REAPI_DLL TErrorCode
   REAPI_RegisterConnectionCallback(
               C_connectionCallbackPtr callback,
               void* context );
```

Parameters:

`callback`	Address of callback function which should be called on connection specific events from the connection
`context`	context pointer that will be given to callback function to reference needed context members

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	Successfully registered.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

Remark:

This function does not cause any data to be sent to or received from the device.

The **context** pointer is handed over to the REA-PI and returned unchanged by REA-PI in the event callback. It is not used in the library itself. As all callback-functions are called from another thread, the context pointer can be used to synchronize values back into the calling context. It might even simply hold a `this()` pointer, if used in an object oriented programming language.

3.7.3 Response callback

Syntax:

```
typedef void (REAPI* C_responseCallbackPtr)(
   TResponseHandle response,
   TConnectionId connection,
   TCmdId command,
   TErrorCode error,
   void* context );
```

`C_responseCallbackPtr` - callback function pointer for notification of asynchronous command responses. Response callback functions should be implemented by client. If asynchronous mode is desired, the pointers to these implemented functions should be handed over to REA-PI by calling the `RegisterResponseCallback()` function.

Parameters:

`response`	A unique `TResponseHandle` that can be used to access response message values. The response handle is valid only inside the callback function and should only be used there! For saving a response handle later use outside of callback function see 3.9.1.
`connection`	The connection id of the appropriate device connection.
`command`	The Command ID to signal the message type. The corresponding command definitions can be found in the **reapi.h** file.
`error`	Obsolete. This error isn't used anymore, because • the call of an command to device failed with returning an error code, see later in this document • The error state about the request you can retrieve by response access functions, see also chapter 3.9.2.
`context`	Context pointer usually required for dispatching. The value is handed over previously by client to REAPI and returned by REAPI unchanged. It is not used in the library itself. As all callback-functions are called from another thread, the context pointer can be used to synchronize values back into the calling context. It might even simply hold a `this()` pointer, if used in an object oriented programming language.

Note: It's important to know what a call to a REA-PI command directed to device does! Basically there are three kinds of responses w.r.t. a REA-PI command:

- Simple execution of a function: Get the result of the call by retrieving the error code from the response handle. An example for such a function is REAPI_SetDateTime().
- Retrieve information from device: Get the result of the call by retrieving the error code and the returned information from the response handle. An example for such a function is REAPI_GetDeviceInfo().
- Complex command execution: A process is initiated, whose end is signalled by an event. By retrieving the error code from the response handle, the host will be informed whether the attempt to start this process was successfully or not.

The result of the complex command execution itself is delivered by an event.
An example for such a function is REAPI_SetJob().

```
void REAPI onResponse( TResponseHandle response,
                       TConnectionId connection,
                       TCmdId command,
                       TErrorCode,
                       void* context ) {
   TErrorCode err;
   // get the result of the function response
   int32_t domain = REAPI_GetErrorDomain( response, &err );
   int32_t error =  REAPI_GetErrorCode( response, &err );
   const char* msg = REAPI_GetErrorMessage( response, &err );

   switch (command) {
       case CMD_SETJOB: {
         if (error == CODE_JOB_NOT_EXISTS) {
            // order to set the job faild because file of the job doesn't exist
         }
         else if (error != CODE_OK) {
            // advise setting the job faild because of -> see domain / error / msg
         }

         // order to set the job was successful, but is not currently set. Therefore
         //   you have to wait for the event EVENT_JOBSET!
      }
      break;

      case CMD_SETDATETIME: {
         if (error != CODE_OK) {
            // setting datetime failed because of -> see domain / error / msg
         }

         // the new time was set on device successfully
      }
      break;

      case CMD_GETDEVICEINFO: {
         if (error != CODE_OK) {
            // getting device information failed because of -> see domain / error / msg
         }

         // for the device information call use response handle with getter functions
         const char* version =
            REAPI_GetFirmware( response, &err );
         const char* serial =
            REAPI_GetSerialNumber( response, &err );
         const char* devicetype =
            REAPI_GetDeviceType( response, &err );
      }
      break;

      default: {
         if (error != CODE_OK) {
            // other command faild because of -> see domain / error / msg
         }
      }
      break;
   }
}
```

Example 3: Response callback implementation

3.7.4 Register a response callback

Syntax:

```
typedef void (REAPI* C_responseCallbackPtr)
    (TResponseHandle, TConnectionId, TCmdId,
     TErrorCode, void*);
```

```
REAPI_DLL TErrorCode
   REAPI_RegisterResponseCallback(
               TConnectionId connection,
               C_responseCallbackPtr callback,
               void* context );
```

Parameters:

`connection`	Connection id returned by REAPI_connect or the corresponding connection callback
`callback`	Address of callback function which should be called on responses from the connection
`context`	context pointer that will be given to callback function to reference needed context members

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	Successfully registered.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

Remark:

This function does not cause any data to be sent to or received from the device.

The **context** pointer is handed over to the REAPI and returned unchanged by REAPI in the event callback. It is not used in the library itself. As all callback-functions are called from another thread, the context pointer can be used to synchronize values back into the calling context. It might even simply hold a `this()` pointer, if used in an object oriented programming language.

3.7.5 *Event callback*

Syntax:

```
typedef void (REAPI* C_eventCallbackPtr)(
   TResponseHandle response,
   TConnectionId connection,
   TEventId event,
   void* context );
```

`C_eventCallbackPtr` - callback function pointer for event notification from REAPI. Should be implemented by client and registered in REAPI using `RegisterEventCallback` function.

Parameters:

`response`	A unique `TResponseHandle` that can be used to access event message values. The response handle is only valid and should only be used inside the callback function! For saving a response handle later use outside of callback function see 3.9.1.
`connection`	The connection id of the returning device connection.
`event`	The EventID to signal the event number. The corresponding event definitions can be found in the **reapi.h** file.
`context`	Context pointer usually required for dispatching. The value is handed over previously by client to REAPI and returned by REAPI unchanged. It is not used in the library itself. As all callback-functions are called from another thread, the context pointer can be used to synchronize values back into the calling context. It might even simply hold a `this()` pointer, if used in an object oriented programming language.

```
void REAPI myEventCallback( TResponseHandle response,
                            TConnectionId connection,
                            TEventId event_id,
                            void* context) {
   printf( "Event from connection %d\n", connection );
   switch( event_id ) {
      case EVENT_JOBSET:     /*do something*/ break;
      case EVENT_JOBSTARTED: /*do something*/ break;
      case EVENT_JOBSTOPPED: /*do something*/ break;
      case EVENT_PRINTSTART: /*do something*/ break;
      case EVENT_PRINTEND:   /*do something*/ break;
      default: break;
   }
}
```

Example 2: Event callback implementation

3.7.6 *Register an event callback*

Syntax:

```
typedef void (REAPI* C_eventCallbackPtr)
   (TResponseHandle, TConnectionId, TEventId, void*);

REAPI_DLL TErrorCode
   REAPI_RegisterEventCallback(
               TConnectionId connection,
               C_eventCallbackPtr callback,
               void* context );
```

Parameters:

`connection`	Connection id returned by REAPI_connect or the corresponding connection callback
`callback`	Address of callback function which should be called on events from the connection
`context`	context pointer that will be given to callback function to reference needed context members

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	Successfully registered.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

Remark:

This function does not cause any data to be sent to or received from the device.

The **context** pointer is handed over to the REAPI and returned unchanged by REAPI in the event callback. It is not used in the library itself. As all callback-functions are called from another thread, the context pointer can be used to synchronize values back into the calling context. It might even simply hold a `this()` pointer, if used in an object oriented programming language.

3.8 Using Synchronous Mode

3.8.1 Synchonous calls

A synchonous call means that the API function will be suspended until the desired action has been executed on the device and the corresponding response has been received and processed. Therefore, no response callbacks will be used, whereas for events, a different mechanism will be used.

Again, the return code of the API function indicates whether submitting the command has failed. In this case, the return code is a negative number being interpreted as `TErrorCode`, e.g., `ERRORCODE_INVALID_CONNECTION` (there was no connection to the device) or `ERRORCODE_TIMEOUT` (the device hasn't answered within a given timeout).

If submission (network transfer to the device) of the command was successful, the function returns with a positive code denoting a `TResponseHandle`. Of course, execution of the corresponding action could also have been failed on the device.

Thus, it is crucial to check the error code which has been received from the device as well. This must not be confused with the aforementioned error code of the function call. The following code snippet stresses this difference:

```
TResponsehandle response = REAPI_GetDeviceInfo( connection );
if ((TErrorCode)response < ERRORCODE_OK) {
   // order not sent to device because internal REAPI Lib TErrorCode
}

// else valid response handle, function was ececuted on device

TErrorCode err;
// get the result of the function response
int32_t domain = REAPI_GetErrorDomain( response, &err );
int32_t error =  REAPI_GetErrorCode( response, &err );
const char* msg = REAPI_GetErrorMessage( response, &err );
if (error >= CODE_OK) {
   // function failed on device because of because error, errorcode
   // later in this document
{

// successful, for the device information call use response handle
//    with getter functions
const char* version =
   REAPI_GetFirmware( response, &err );
const char* serial =
   REAPI_GetSerialNumber( response, &err );
const char* devicetype =
   REAPI_GetDeviceType( response, &err );
```

3.9 Using response handles

3.9.1 *Keeping alive and releasing response handles*

There are two reasons, to keep alive response handles for later use.

- In synchronous mode calling a function will result in a response handle. Use this response handle to retrieve the error code and the returned information. An Example for such a function is REAPI_GetDeviceInfo()
- In asynchronous mode the response handle is valid within the scope of callback functions and will be deleted after leaving them. Normally response access functions retrieve data in the callback and have to be saved in variables. To keep the response handle alive, use REAPI_ResponseHandleAddRef() inside and REAPI_ResponseHandleRelease() later outside of the callback functions.

Both functions strictly have to be used pairwise. Omitting to release a response handle will result in memory leaks!

Syntax:

```
REAPI_DLL TErrorCode REAPI_ResponseHandleAddRef (
               TResponseHandle response );

REAPI_DLL TErrorCode REAPI_ResponseHandleRelease(
               TResponseHandle response );
```

Parameters:

`response`	Handle of the response returned by event callback function.

Return:

`ERRORCODE_INVALID_RESPONSE_HANDLE`	response handle is not valid
`ERRORCODE_OK`	response handle successfully saved or released.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

3.9.2 Get Error Values from Response / Event

Each response and event does have some common attributes which can be retrieved from response handle by the following access functions.

REAPI_GetErrorDomain
REAPI_GetErrorCode

The error domains and codes sent from the device are described in chapter 0 , page 203.

```
REAPI_DLL int32_t REAPI_GetErrorDomain(
               TResponseHandle response,
               TErrorCode*     error );
```

```
REAPI_DLL int32_t REAPI_GetErrorCode(
               TResponseHandle response,
               TErrorCode*     error );
```

```
REAPI_DLL const char* REAPI_GetErrorMessage(
               TResponseHandle response,
               TErrorCode*     error );
```

REAPI_GetWarningCode

The warning codes sent from the device are described in chapter 0 , page 203.

Warning codes represent additional information when no error is indicated, but something unusual happened, which might lead to an undesired behaviour.

E.g., SetJob() may be used to assign a label, which is too high to be printed on the selected group. This will result in the warning CODE_LABEL_EXCEEDS_PRINT.

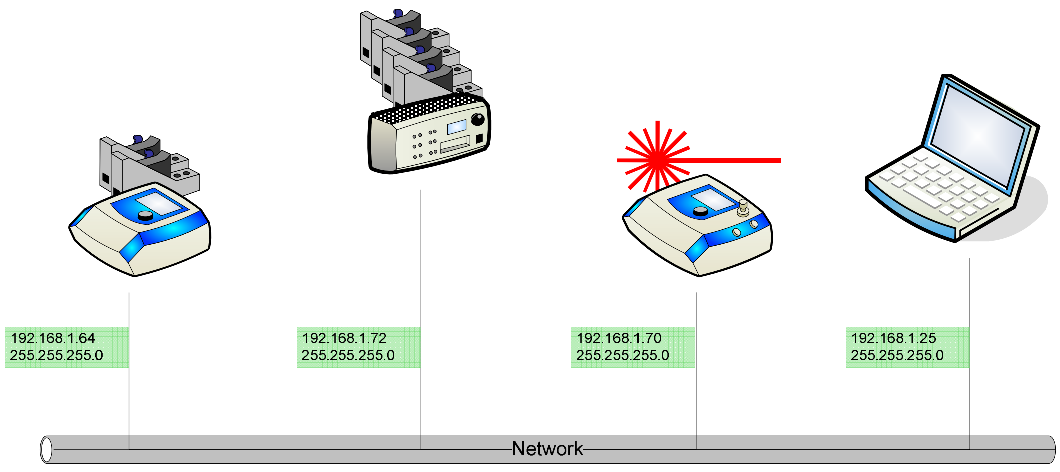
```
REAPI_DLL int32_t REAPI_GetWarningCode(
               TResponseHandle response,
               TErrorCode*     error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

4 Connection management

4.1 Content of Chapter

4.2 Getting connected



4.2.1 *Asynchronous connect to device*

Use only in asynchronous mode. Therefore you have to implement a connection callback and register it with REAPI_RegisterConnectionCallback.

For synchronous mode use `ConnectWait()` instead.

Syntax:

```
REAPI_DLL TConnectionId REAPI_Connect(
               const char* connectionString );
REAPI_DLL TConnectionId REAPI_ConnectContext(
               const char* connectionString,
               void* context );
```

Parameters:

`connectionstring`	Device connection string containing device address and connection type. The format is: <Connection Type>://<Address>[:<port>], where

<Connection Type> can currently be only **TCP** or **SSL**

<Address> is connection specific address. For TCP/SSL - an IP-address or a host name

<port> optional parameter, currently not used, defaults to 22171(TCP) or 22172(SSL)

`context`	A generic pointer. The referenced context will be used for parameter in connection callback.

Return:

The return value `ERRORCODE_OK` means successful. Other values signals an error ans must be interpreted as `TErrorCode`.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

4.2.2 *Disconnect from device*

Disconnects from the device.

You have to implement a connection callback and register it with REAPI_RegisterConnectionCallback.

Syntax:

```
REAPI_DLL TErrorCode REAPI_Disconnect(
                TConnectionId connection );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback

Return:

The return value `ERRORCODE_OK` means successful. Other values signals an error ans must be interpreted as `TErrorCode`.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

4.3 Information about connection

4.3.1 *Get protocol version*

Get the actual used protocol version between device and the actual connection. It will be the highest possible protocol version for communication between the used reapi.dll and the firmware version on the device.

The protocol version is automatically established after the handshake between device and the actual connection during REAPI_Connect().

Examples:
reapi.dll V3.22 and device FW 3.21 use protocol version 3.2.
reapi.dll V1.84 and device FW 3.21 use protocol version 1.8.
reapi.dll V3.22 and device FW 3.02 use protocol version 3.0.

Syntax:

```
REAPI_DLL const char* REAPI_ProtocolVersion(
               TConnectionId connection );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback

5 Job management commands

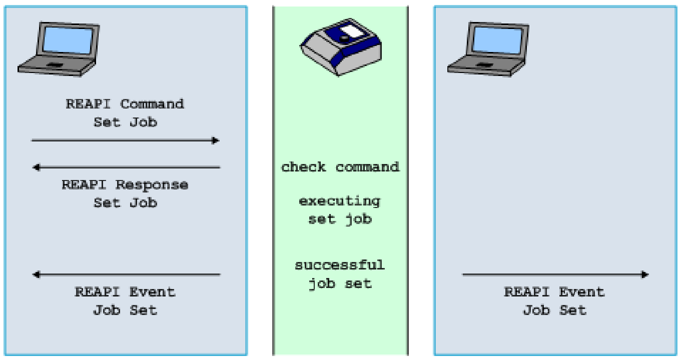
5.1 Content of Chapter

5.2 Assign jobs

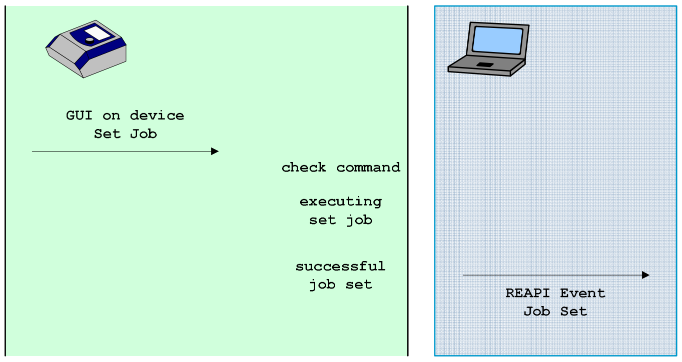
Assigning a job denotes the procedure, which is started by issuing the command for setting a job, i.e., process the label to be printed w.r.t. the given settings. Subsequently, the print environment is configured such that printing can be activated. Any previously assigned job will automatically be removed when a new job is set. Note, that a job can be removed manually, too, by the command for clearing a job.

The event, when a (new) job is set, is called JobSet event and contains the name (filename) of this job as a parameter. Note, that this kind of event will also occur when a job was cleared. In that case an empty filename will be returned. No job will be assigned then.

In order to always receive the most recent data, it is strongly recommended to subscribe the Job-Set Event immediately after a connection was established. The subscription of the event can either be terminated by calling the unsubscribe function or will be terminated automatically when the network connection is closed.



When setting or clearing a job, the response from the device only indicates whether the command has been accepted. The response MUST NOT be used to determine which job is currently assigned. Since a print job could also be assigned by another application or the GUI, only the Job-Set Event gives relevant information about the correct state of the job assignment.



5.2.1 *Command Set job*

Request the device to load the specified job file and prepare settings and label(s) for printing.

This command is available with a Firmware-Version ≥ 1.80.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SetJob(
               TConnectionId connection,
               TJobId job,
               const char* filename );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
`filename`	Filename of job definition file.

	The jobfile defined in filename has to exist in the “job” folder on the device share

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to

	identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

5.2.2 *Clear Job Command*

The command requests the device to unload / clear the job specified by the job identifier.

This command is available with a Firmware-Version ≥ 1.80.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_ClearJob(
                TConnectionId connection,
                TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

5.3 Event JobSet

The device sends this event when a new job was assigned by device or any client including own call.

This event requires a subscription and has event parameters. First time the event will occur immediately after subscription to inform which set is set.

This event is available since firmware version ≥ 1.80.

5.3.1 Subscription

The subscription is effective while a connection. It is recommended to subscribe to this event immediately after a connection to a device is established.

The subscription of the event can either be terminated by calling the unsubscribe function or will be terminated automatically when the network connection is closed.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SubscribeJobSet(
              TConnectionId connection,
              TJobId job );
```

```
REAPI_DLL TResponseHandle REAPI_UnsubscribeJobSet(
              TConnectionId connection,
              TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

5.3.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from within the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the currently set job or the job which is now empty, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetJobFilename

Get the Filename of the job which is now currently set or an empty string if no job is set.

```
REAPI_DLL int32_t REAPI_GetJobFilename (
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetJobFixedInstallation

Returns if the currently set job is a fixed installation or not.

```
REAPI_DLL bool REAPI_GetJobFixedInstallation(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

5.4 Starting and stopping jobs

5.4.1 Commands

The print environment is activated by issuing the command to start a job. Subsequently, the system becomes ready to print. The print heads and the laser tube, respectively, will be turned on according to the settings. Furthermore, the activities of the product sensor and the encoder can be observed. Calling the command to stop a job causes the print environment to be deactivated.

When starting or stopping a job a job, the response from the device only indicates whether the command has been accepted. The response MUST NOT be used to determine the current status of an activated job. Since a print job could also be started / stopped by another application or the GUI, only the corresponding events give relevant information about the correct job status.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_StartJob(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle REAPI_StopJob(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

5.5 Event JobStarted and JobStopped

The device raises these events when a job was started and stopped, respectively, by any client. This means, an application may receive a JobStopped event when it calls StopJob itself, as well as when the device stopped the job in case of an error condition.

Both events work within the context of the job. Therefore, these events should be subscribed immediately after a job has been set. The recommended way is to call the subscription when the JobSet event was received to ensure reading the most recent status. The subscription of the events can be cancelled by either calling the corresponding unsubscribe functions or is terminated automatically when the job is removed or replaced by another job.

Subscribing to the events when a job was started or stopped, respectively, allows a client to be informed about the current state of the job activation. With help of the parameter of these events you can determine which job was started or stopped.

These events are available since firmware version ≥ 1.80.

5.5.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or unloaded.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SubscribeJobStarted(
        TConnectionId   connection,
        TJobId job );
```

```
REAPI_DLL TResponseHandle REAPI_UnsubscribeJobStarted(
        TConnectionId connection,
        TJobId job );
```

```
REAPI_DLL TResponseHandle REAPI_SubscribeJobStopped(
        TConnectionId   connection,
        TJobId job );
```

```
REAPI_DLL TResponseHandle REAPI_UnsubscribeJobStopped(
        TConnectionId connection,
        TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

5.5.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the job which was startet or stopped, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

5.6 Purging a printhead

5.6.1 Purge head command

The command requests the device to purge the specified printheads with corresponding parameters.

In order to purge a job must be loaded before. Only heads which are included in this job are able to be purged. It is only available in stopped print mode and not during active printing for that job.

Since Firmware-Version ≥ 3.2 the device will send a corresponding event when the process to purge printheads is finished.

This command is available with a Firmware-Version ≥ 1.80.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_StartPurgeHead(
               TConnectionId connection,
               int32_t head,
               int32_t drops,
               int32_t frequency );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`head`	The heads to be purged are specified by id, which 1 is first head, 2 second head and so on. For more comfortable usage it is possible to specify only one printhead with a special identifier 0. In this case all available printheads in job 0 are purged with this parameter.

Pay attention: it is not possible to mix both versions with exact specified printheads and the wildcard identifier -1!

`drops`	Number of drops, allowed value from 500 - 6000 drops.
`frequency`	Frequency per nozzle in Hz, allowed value 1000-6000Hz

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

5.6.2 *Event PurgeCompleted*

The device always raises this event whenever the purge process is finished. However this event needs no subscription.

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the job which was startet or stopped, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
                TResponseHandle response,
                TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6 Print management commands

6.1 Content of Chapter

6.2 Print control events and functions

The functionality described in this section mainly concerns events which occur during the printing process, either as notifications for successful operation or as error indications and warnings respectively.

The former comprise events when a print was triggered, was actually started or completed or when certain print phases are passed through:

- PrintTrigger: A print was triggered.
- PrintStart: A print / print sequence was started.
- PrintEnd: A print / print sequence was completed.
- ImageStart: An individual print within a whole sequence was started.
- ImageEnd: An individual print within a whole sequence was completed.
- PrintAborted: A print sequence was interrupted.

The following events belong to the latter group:

- MissingContent: Upon trigger, no valid print buffer was available.
- PrintRejected: No print was performed as there was no valid print buffer available.
- PrintSpeedError: Product speed too high w.r.t. maxmimum nozzle frequency and horizontal resolution.
- PrintStatus: Miscellaneous errors / warnings.

All these events are automatically unsubscribed when a new job is assigned or the current job is cleared.

6.3 Event PrintTrigger

The device raises this event when the trigger for a print has occurred. Normally it is caused by product sensor or digital input or any combination of these.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 1.80.

6.3.1 *Subscription*

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintTrigger(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintTrigger(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.3.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
                TResponseHandle response,
                TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.4 Event PrintStart

The device raises this event on the beginning of a printing process. Here, the term “printing process” relates to the first print head of the first label / print head group in chronological order and denotes a sequence of one or more consecutive prints, depending on the print mode (cf. chapter 6.5 Event ImageStart, page 45):

- Using print mode “normal”, there is only one individual print per trigger. Hence every (individual) print raises one (separate) PrintStart event.
- Operated in print mode “cyclic”, one PrintStart event is raised for the configured sequence.
- The same goes for print mode “continous”, no matter whether a limitation for the number of (individual) prints is set.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 1.80.

6.4.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintStart(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintStart(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.

`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.4.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
              TResponseHandle response,
              TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.5 Event ImageStart

The device raises this event on the beginning of the printing process for every individual print (cf. chapter 6.4 Event PrintStart, page 42):

- Using print mode “normal”, there is only one individual print per trigger. Hence ImageStart coincides with the PrintStart event.
- Operated in print mode “cyclic”, one ImageStart event is raised for every individual print of the configured sequence.
- The same goes for print mode “continous”, no matter whether a limitation for the number of (individual) prints is set.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.40.

6.5.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribeImageStart(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeImageStart(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to

	identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.5.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
              TResponseHandle response,
              TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.6 Event ImageEnd

The device raises this event on the end of the printing process for for every individual print (cf. chapter 6.7 Event PrintEnd, page 51):

- ➢ Using print mode “normal”, there is only one individual print per trigger. Hence ImageEnd coincides with the PrintEnd event.
- ➢ Operated in print mode “cyclic”, one ImageEnd event is raised for every individual print of the configured sequence.
- ➢ The same goes for print mode “continous”, no matter whether a limitation for the number of (individual) prints is set.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.40.

6.6.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribeImageEnd(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeImageEnd(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to

	identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.6.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
                TResponseHandle response,
                TErrorCode* error );
```

REAPI_GetPrintDuration

Gets the time duration from the beginning to the end of the printing process.

```
REAPI_DLL double REAPI_GetPrintDuration(
                TResponseHandle response,
                TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.7 Event PrintEnd

The device raises this event on the beginning of a printing process. Here, the term “printing process” relates to the first print head of the first label / print head group in chronological order and denotes a sequence of one or more consecutive prints, depending on the print mode (cf. chapter 6.6 Event ImageEnd, page 48):

- Using print mode “normal”, there is only one individual print per trigger. Hence every (individual) print raises one (separate) PrintEnd event.
- Operated in print mode “cyclic”, one PrintEnd event is raised for the whole sequence configured.
- The same goes for print mode “continous”, no matter whether a limitation for the number of (individual) prints is set.

This event requires a subscription and has event parameters.

6.7.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintEnd(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintEnd(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to

	identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.7.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetPrintDuration

Gets the time duration from the beginning to the end of the printing process for the currently assigned label.

```
REAPI_DLL double REAPI_GetPrintDuration(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.8 Event PrintAborted

The device raises this event if the print of a label was aborted. Normally this event occurred in continuous printing mode sequence. It is the equivalent of the PrintEnd event for this special case.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 1.80.

6.8.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintAborted(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintAborted(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.8.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Get the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
              TResponseHandle response,
              TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.9 Event PrintSpeedError

The device raises this event when the product speed is too high, e.g., w.r.t. maxmimum nozzle frequency and horizontal resolution.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 1.80.

6.9.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintSpeedError(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintSpeedError(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.9.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetJobErrorDomain
REAPI_GetJobErrorCode
REAPI_GetJobErrorMessage

Get the error information why this speed error ocurrs. For more information about errorcodes see chapter 0 , page 203.

```
REAPI_DLL int32_t REAPI_GetJobErrorDomain(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL int32_t REAPI_GetJobErrorCode(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetJobErrorMessage(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.10 Event MissingContent

The device raises this event when a print is triggered but no print buffer is available. This event signals the beginning of a period, in which a new printbuffer may become available and would eventually be printed. This period would end either with the event PrintRejected if no print buffer was available at all or with the event PrintStart if it was possible to create a print buffer created in sufficient time.

Particularly, the event MissingContent would occurs send in single print mode if variable data for the upcoming print was not refreshed and a possible tolerance (configurable, cf. chapter 6.13 Print Rejection and "No Print Buffer Mode"

, page 65) has passed.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.30.

6.10.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribeMissingContent(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeMissingContent(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to

identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.10.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetJobErrorDomain
REAPI_GetJobErrorCode
REAPI_GetJobErrorMessage

Get the error information why this content is missing

```
REAPI_DLL int32_t REAPI_GetJobErrorDomain(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL int32_t REAPI_GetJobErrorCode(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetJobErrorMessage(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetPrintTriggerID
REAPI_GetNoBufferMode
REAPI_GetNoBufferOffset
REAPI_GetPrintRejectMode

Get information about the behavior when no printbuffer is available. For more information about these parameters see the function SetNoPrintBufferMode().

```
REAPI_DLL TSensorId REAPI_GetPrintTriggerID(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL int32_t REAPI_GetNoBufferMode(
               TResponseHandle response,
               TErrorCode* error );

REAPI_DLL double REAPI_GetNoBufferOffset(
               TResponseHandle response,
               TErrorCode* error );

REAPI_DLL int32_t REAPI_GetPrintRejectMode(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.11 Event PrintRejected

The device raises this event when the actual printing on a product is rejected because no print buffer was available. This event signals the end of a time span, in which a new print buffer would be accepted and printed. This time span starts with an event MissingContent. See also event MissingContent.

Particularly, the event PrintRejected would occurs send in single print mode if variable data for the upcoming print was not refreshed and a possible tolerance (configurable, cf. chapter 6.13 Print Rejection and “No Print Buffer Mode”

, page 65) has passed.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 1.80.

ℹ	The meaning and the behavior of this event has been modified significantly since version 3.3 in comparison to older versions! See also the chapter 6.13 Print Rejection and “No Print Buffer Mode” , page 65 and the Event MissingContent, page 58.

6.11.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintRejected(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintRejected(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid

`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.11.2 *Event parameters*

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetJobErrorDomain
REAPI_GetJobErrorCode
REAPI_GetJobErrorMessage

Get the error information why this print is rejected

```
REAPI_DLL int32_t REAPI_GetJobErrorDomain(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL int32_t REAPI_GetJobErrorCode(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetJobErrorMessage(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetPrintTriggerID
REAPI_GetNoBufferMode
REAPI_GetNoBufferOffset
REAPI_GetPrintRejectMode

Get information about the behavior when no printbuffer is available. For more information about these parameters see the function SetNoPrintBufferMode().

```
REAPI_DLL TSensorId REAPI_GetPrintTriggerID(
               TResponseHandle response,
               TErrorCode* error );
REAPI_DLL int32_t REAPI_GetNoBufferMode(
               TResponseHandle response,
               TErrorCode* error );
REAPI_DLL double REAPI_GetNoBufferOffset(
               TResponseHandle response,
               TErrorCode* error );
REAPI_DLL int32_t REAPI_GetPrintRejectMode(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

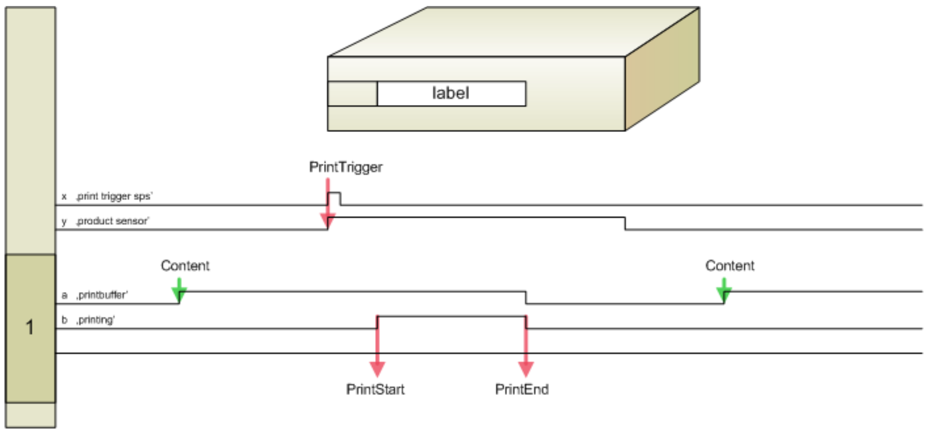
6.12 EventPrintStatus

The device raises this event whenever an unexpected error or warning or information on current printing occurs. However this event needs no subscription and can occur as long as a connection to the device is established. Hence the functions REAPI_GetErrorDomain() and REAPI_GetErrorCode() directly apply.

This event is available with a Firmware-Version ≥ 1.80.

6.13 Print Rejection and “No Print Buffer Mode”

The following diagram depicts - in a simplified way - the sequence of some of the actions and events when executing a print. Usually, the content to be printed has been set early enough before the print is triggered. Hence the print is executed with just this content, while the events PrintStart and PrintEnd signal the beginning and end of the printing process respectively:



Since this represents the regular flow, the settings for the “No Print Buffer Mode” are of no relevance.

However, in case the content to be printed could not be prepared in time, the subsequent behaviour depends on the settings for NoBufferMode. In detail, there are the following possibilities:

0	Reject
1	Wait Infinite
2	Wait Delay
3	Wait Delay (incl. SO)
4	Wait Delay (incl. SO & LO)

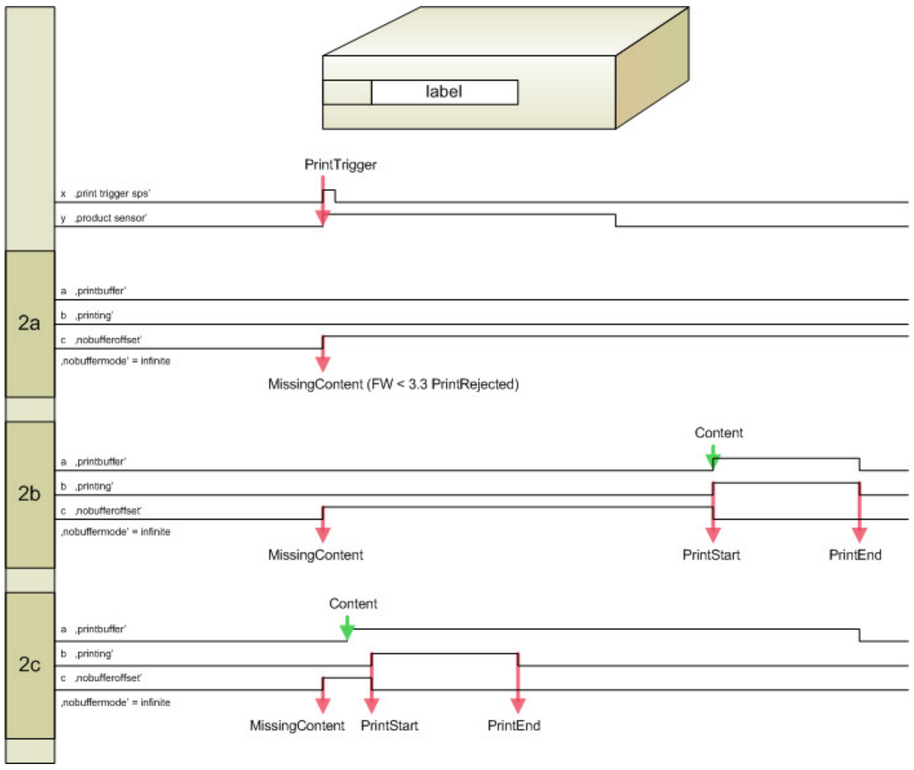
6.13.1 NoBufferMode “Wait Infinite”

If NoBufferMode is set to “Wait Infinite”, the device will directly issue an event MissingContent upon the print trigger. Neither the parameter NoBufferOffset nor PrintRejectMode has an effect in this case.

As soon as the the content to be printed has been prepared, the print will be executed, i.e., if necessary, the device will wait indefinitely long for a content.

Therefore, a displacement may occur for the print when the content finally has arrived. The the events PrintStart and PrintEnd signal the beginning and end of this - possibly delayed - printing process respectively.

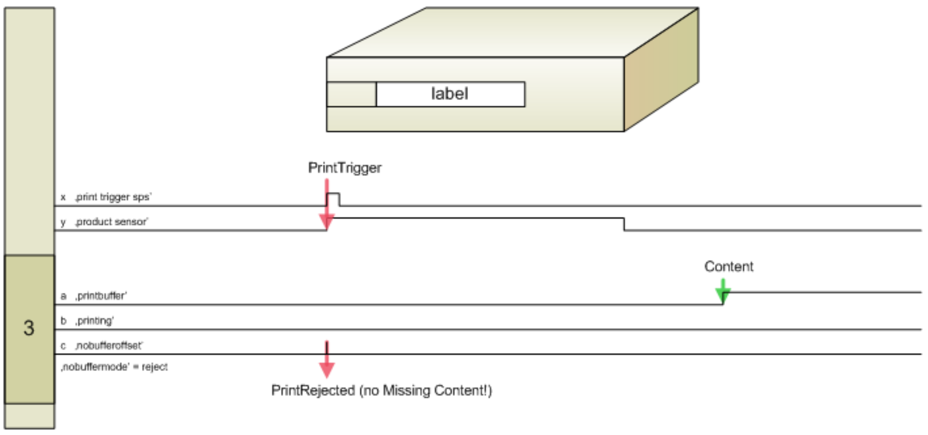
The setting described here corresponds to the behaviour of firmware version 3.25 and before with the exception that an event PrintRejected will be raised instead of an event MissingContent as described above.



6.13.2 NoBufferMode "Reject"

If NoBufferMode is set to "Wait Infinite", the device will directly issue an event PrintRejected upon the print trigger. The parameter NoBufferOffset is not applica-ble, but PrintRejectMode with one of the values:

0	Stop Printing
1	Wait for Confirmation
2	Ignore and Wait

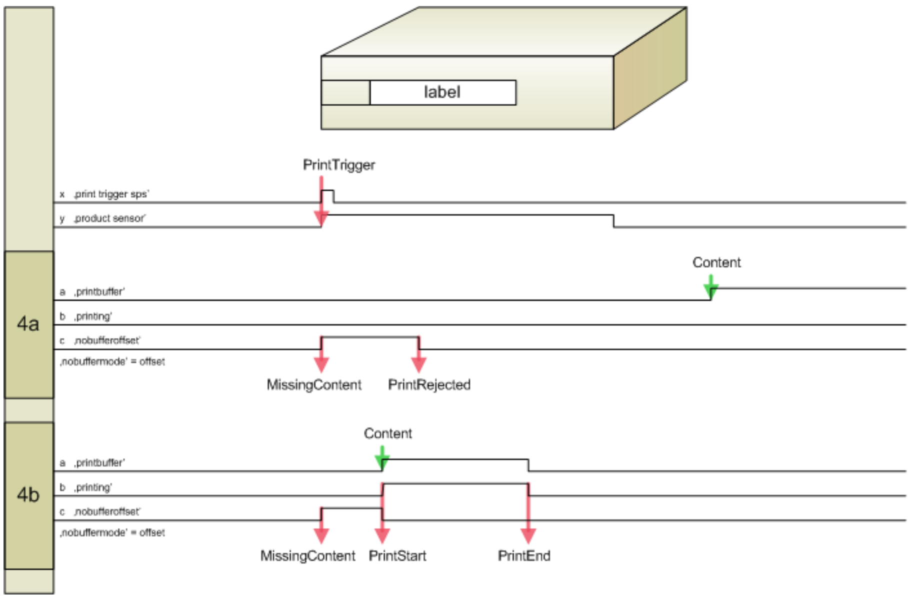


Since the content to be printed was not prepared in time, the print will not be executed. Afterwards

- the job will be stopped, when PrintRejectMode is "Stop Printing".
- any print trigger will be discarded until the status is explicitely confirmed (cf. ConfirmPrintReject), when PrintRejectMode is "Wait for Confirmation".
- the condition is ignored and the next print trigger will result in a regular printing process, when PrintRejectMode is "Wait and Ignore".

6.13.3 NoBufferMode „Wait Delay"

The variants "Wait Delay","Wait Delay (incl. SO)" and "Wait Delay (incl. SO & LO)", reflect some sort of combination of NoBufferMode "Wait Infinite" and NoBufferMode "Reject" respectively:



Upon print trigger an event MissingContent will be issued immediately. The parameter NoBufferOffset specifies a tolerance (in mm) which may pass before an event PrintRejected will occur.

- If the content to be printed can be prepared within the given tolerance, the print will be executed. However, a displacement will occur for the print, the events PrintStart and PrintEnd signal the beginning and the and of this print respectively.

  If the content to be printed was not prepared within the given delay, the trigger will be discarded, i.e., no print will be executed, and an event PrintRejected will be issued. The further process is identical to NoBufferMode “Reject”, i.e., depends on the setting for PrintRejectMode:

  - the job will be stopped, when PrintRejectMode is “Stop Printing”.
  - any print trigger will be discarded until the status is explicitely confirmed (cf. ConfirmPrintReject), when PrintRejectMode is “Wait for Confirmation”.
  - the condition is ignored and the next print trigger will result in a regular printing process, when PrintRejectMode is “Wait and Ignore”.

.

6.13.4 Get no-print-buffer mode

This command gets the configuration and parameters for the behaviour of the printer if a printtrigger instructs a print, but no printbuffer is available.

This command is available with a Firmware-Version ≥ 3.30.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_GetNoPrintBufferMode(
                TConnectionId connection,
                TSensorId printtriggerid );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`printtriggerid`	Identifier of the related print trigger

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.13.4.1 Command return

With the following functions the command return can be retrieved when the functions are called from the command callback by the corresponding commandID.

Use the response handle of the command callback to get the corresponding command return. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.3 Response callback, page 15.

REAPI_GetPrintTriggerID
REAPI_GetNoBufferMode
REAPI_GetNoBufferOffset
REAPI_GetPrintRejectMode

Get information about the behavior when no printbuffer is available. For more information about these parameters see the function SetNoPrintBufferMode().

```
REAPI_DLL TSensorId REAPI_GetPrintTriggerID(
                TResponseHandle response,
                TErrorCode* error );
```

```
REAPI_DLL int32_t REAPI_GetNoBufferMode(
               TResponseHandle response,
               TErrorCode* error );
REAPI_DLL double REAPI_GetNoBufferOffset(
               TResponseHandle response,
               TErrorCode* error );
REAPI_DLL int32_t REAPI_GetPrintRejectMode(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

6.13.5 *Set no-print-buffer mode*

This command sets the configuration and parameters for the behaviour of the printer if a printtrigger instructs a print, but no printbuffer is available.

This command is available with a Firmware-Version ≥ 3.30.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SetNoPrintBufferMode(
               TConnectionId connection,
               TSensorId printtriggerid,
               int nobuffermode,
               double nobufferoffset,
               int printrejectmode );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`printtriggerid`	Identifier of the related print trigger
`nobuffermode`	Mode of the behaviour if no print buffer is available on print. Allowed values are: Reject = 0, Infinite = 1, Offset = 2
`nobufferoffset`	Distance in millimeter before the print is finally rejected.
`printrejectmode`	Mode of the behaviour if the print is finally rejected. Allowed values are: Stop = 0, NeedConfirm = 1, Ignore = 2

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

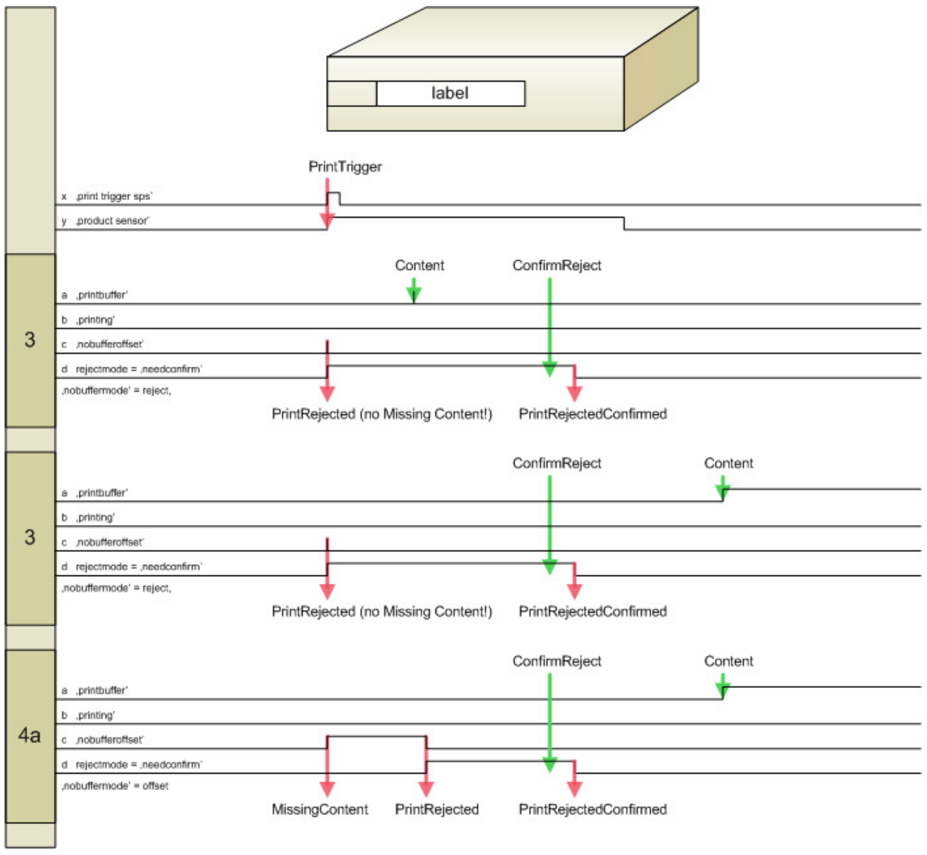
6.14 Confirm PrintRejection

When PrintRejectMode is set to “Wait for Confirmation” (cf. chapter 6.13 Print Rejection and “No Print Buffer Mode”, page 65), then

- ➢ no new content will be accepted
- ➢ all of the following print triggers will be discarded

until the status is explicitely confirmed by the ConfirmPrintReject command. Any content that is currently under preparation by the print system will be discarded as well. Only after the above command a new content is allowed to be set.

Additionally, an event PrintRejectedConfirmed may be subscribed to get informed about the new status, particularly by clients, which do not send ConfirmPrintReject themselves.



6.14.1 ConfirmPrintReject

This command confirms the occurrence of the PrintRejected event and thus terminates the dismission of further printis (see also chapter 6.13 Print Rejection and "No Print Buffer Mode", page 65).

This command is available with a Firmware-Version ≥ 3.30.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_ConfirmPrintReject(
              TConnectionId connection,
              TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.14.2 *Event PrintRejectedConfirmed*

The device raises this event when a print rejected was confirmed.

For further information see the command ConfirmPrintReject, page 73 and chapter 6.13 Print Rejection and “No Print Buffer Mode”

, page 65.

This event requires a subscription and has event parameters.

This event is available with a Firmware-Version ≥ 3.30.

6.14.2.1 Subscription

The subscription is only effective while a job has been loaded. It is recommended to subscribe to this event when the event JobSet has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribePrintRejectConfirmed(
               TConnectionId connection,
               TJobId job );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribePrintRejectConfirmed(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

6.14.2.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7 Printing - Label manipulation

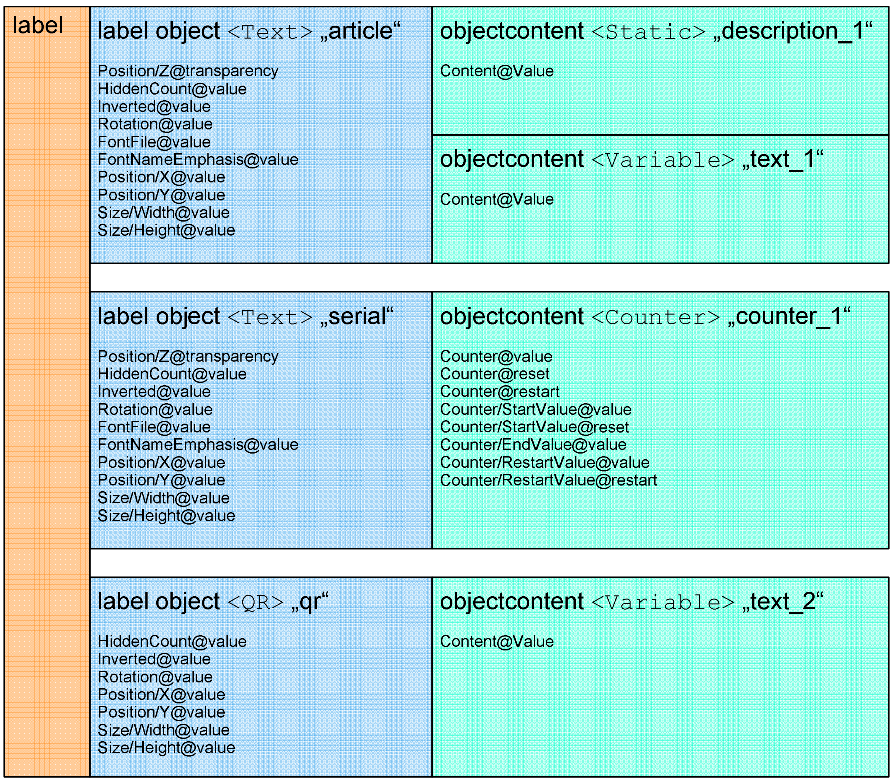
7.1 Content of Chapter

7.2 What are properties and variable contents

This chapter describes how to query or change label contents and label properties when a job is assigned or prints are executed.

To demonstrate label manipulating, we assume a job with one printhead in a group named ‘Front’ which is printing an example label. The following example label contains three label objects ‘article’, ‘serial’ and ‘qr’, each of them with one or more contents and many label object properties, depending on the types of the label objects:

The following diagram shows the structure of the label, label objects and label contents:

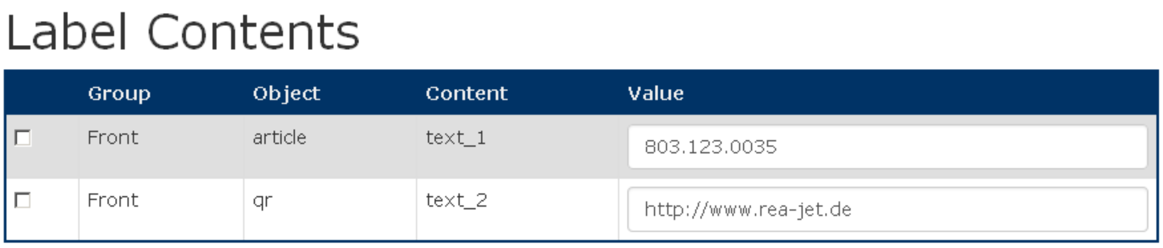


label	label object	objectcontent
label	label object `<Text>` „article“ Position/Z@transparency HiddenCount@value Inverted@value Rotation@value FontFile@value FontNameEmphasis@value Position/X@value Position/Y@value Size/Width@value Size/Height@value	objectcontent `<Static>` „description_1“ Content@Value objectcontent `<Variable>` „text_1“ Content@Value
	label object `<Text>` „serial“ Position/Z@transparency HiddenCount@value Inverted@value Rotation@value FontFile@value FontNameEmphasis@value Position/X@value Position/Y@value Size/Width@value Size/Height@value	objectcontent `<Counter>` „counter_1“ Counter@value Counter@reset Counter@restart Counter/StartValue@value Counter/StartValue@reset Counter/EndValue@value Counter/RestartValue@value Counter/RestartValue@restart
	label object `<QR>` „qr“ HiddenCount@value Inverted@value Rotation@value Position/X@value Position/Y@value Size/Width@value Size/Height@value	objectcontent `<Variable>` „text_2“ Content@Value

For more information about the structure of a label and properties of label objects and label contents see the documentation REA JET XML Labelformat.doc. Generally there are two ways to manipulate labels during printing:

- A command to set only variable label contents of type <Variable> and <Choice> in a performant way. Since Firmware 3.50 also contents of type <ContentBuffer> are supported.

In the example label assumed with a job in group 'Front' there are two variable contents that can be set with the command to set variable contents. This can also be done using the WebGUI:


Label Contents

	Group	Object	Content	Value
☐	Front	article	text_1	803.123.0035
☐	Front	qr	text_2	http://www.rea-jet.de

➢ A command to set label object properties and content properties. This command supports also setting of variable label contents as the previous command for supporting an atomic call to set all contents and properties together.
Setting properties and variable contents with this command is less performant than setting only variable label contents with the command mentioned above.
In the example label assuming loaded with a job in group 'Front' there are many label object properties and variable contents that can be set with the command to set object properties and variable contents.

The current REA-PI Version supports following label object properties:

`Type@value`

String with literal name of the label object analog to the names used in the label, eg “Text”, “Counter”, “GS1DataMatrix”. This property is read only!

For more information about label objects see the documentation REA JET XML Labelformat.doc.

Note: Supported by devices since FW 3.40

`Position/X@value`

numeric float with X position in millimeter, eg “12”, “40.123”

Note: Supported by devices since FW 3.02

`Position/Y@value`

numeric float with Y position in millimeter, eg “12”, “40.123”

Note: Supported by devices since FW 3.02

`Position/Z@transparency`

boolean text, sets the transparency of an label object.

‘true’	transparent, underlying labelobjects are visible.
‘false’	opaque, not transparent, underlying labelobjects are covered.

Note: Supported by HR printing devices since FW 3.20, not supported by laser marking devices; there they are always transparent

Note: On HR printing devices supported by all types of label object, except barcodes; they are never transparent

`Size/Width@value`

numeric float with X dimension size in millimeter, eg “12”, “40.123”

Note: Supported by devices since FW 3.20

Note: The displayed content to be printed or marked will change its appearance. So e.g. texts could be truncated, barcodes will change its module size or graphics will be stretched!

`Size/Height@value`

numeric float with Y dimension size in millimeter, eg “12”, “40.123”

Note: Supported by devices since FW 3.20

Note: The displayed content to be printed or marked will change its appearance. So e.g. texts could be truncated, barcodes will change its module size or graphics will be stretched!

`Rotation@value`

Rotates the label object from base position in degrees.

numeric float with angle in degrees are allowed, eg “0”, “90”, “23”, “113.45”.

Note: Supported by devices since FW 1.84

Note: On HR printing devices also numeric float with angle in degrees are allowed to set, but only 90 degree can be printed, values are rounded to 0, 90, 180, 270 degrees.

`Inverted@value`

inverts black/white pixels in region of label object

‘true’	inverted
‘false’	not inverted

Note: Supported by HR printing devices since FW 1.84, not supported by Laser marking systems.

`HiddenCount@value`

instructs how often a label object should not be printed.

0: not hidden

n: hide the label object for the next n prints

-1: hide the label object infinite times

Note: Supported by devices since FW 1.72

`Font/NameEmphasis@value`

have to be valid strings for installed fonts with font name and font emphasis separated by slash, e.g. “FreeSans/Medium”

Note: Supported by devices since FW 3.20.

Note: Supported by types of label objects which displays text, as there are Text, Counter, Date or Time and barcodes which support humanreadable text.

`Font/File@value`

instructs how often a label object should not be printed.

0: not hidden

n: hide the label object for the next n prints

-1: hide the label object infinite times

Note: Supported by devices since FW 3.20

Note: Supported by types of label objects which displays text, as there are Text, Counter, Date or Time and barcodes which support humanreadable text.

The current REA-PI Version supports following label object content properties:

`Type@value`

String with literal name of the label object content analog to the names used in the label, eg "Variable", "Static", "Counter", "Time", etc. This property is read only!

For more information about label object contents see the documentation REA JET XML Labelformat.doc.

Note: Supported by devices since FW 3.40

`Content@value`

sets the text value of a content, possible for static text, variable text, variable key values of choice and static barcode identifiers.
For variable text and key values of choice this function works identical to SetLabelContent(). Static texts and barcode identifiers are still not variable, but setting is available with this function!

Note: Supported by devices since FW 3.20

Note: For variable text and key values of choice this function works identical to SetLabelContent(). Static texts and barcode identifiers are still not variable, but setting is available with this function!

`Counter@value`

numeric integer, sets the current counter value, e.g "101"

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter@reset`

no argument, will be ignored, resets the counter to start value in label

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter@restart`

no argument, will be ignored, restarts the counter from restart value in label

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter/StartValue@value`

numeric integer, sets a new start value, e.g "0"

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter/StartValue@reset`

numeric integer, sets a new start value and resets the counter, e.g "0"

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter/EndValue@value`

numeric integer, sets a new end value, e.g "1000"

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter/RestartValue@value`

numeric integer, sets a new restart value, e.g "-15"

Note: Supported by devices since FW 3.20 with label object of type Counter.

`Counter/RestartValue@restart`

numeric integer, sets the actual restart value and restart the counter, e.g "-15"

Note: Supported by devices since FW 3.20 with label object of type Counter.

`BufferSize@value`

reserved for future use.

Note: Supported by devices since FW 3.50 with label object of type ContentBuffer.

`BufferPrinted@value`

numeric integer, gets a snapshot of number of printed entries from content buffer since last time when a job was set. This property is read only!

Note: Supported by devices since FW 3.50 with label object of type ContentBuffer.

`BufferExpired@value`

numeric integer, gets a snapshot of number of expired entries from content buffer since last time when a job was set. This property is read only!

Note: Supported by devices since FW 3.50 with label object of type ContentBuffer.

7.3 Containers to transport contents and properties

In order to gather the values to be set for label contents and properties containers are used. So they can be sent over the network in one step.

A label content container is an object which contains a list of label contents, a label object container is a list of label object properties. When a command sends the values in a container over the network, the information is treated atomically. This means all contents and properties are set simultaneously.

The container of desired properties and / or contents to be sent to the device is created, managed and deleted using auxiliary library functions. Example calls:

7.3.1 *Example for setting*

```
// no dataset -> all entries
// explicite specified
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "1", "EAN8", "Text", "" );
```

7.3.2 *Example for getting*

```
// no dataset -> all entries
// explicite specified
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "1", "EAN8", "Text", "" );

// not trackable dataset -> no result entries
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "", "", "", "" );

// all Variables of all groups/labels
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, ".*", ".*", ".*", ""  );

// all Variables of the groups '1' to '2'
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "[1-2]", ".*", ".*", "" );

// all Variables of the groups '1'
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "1", ".*", ".*", "" );

// all Variables of a explicite labelobject by name of all
groups/labels
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, ".*", "FamilyName", ".*", "" );

// beliebige contents eines expliziten labelobjectes eines Labels /
einer Gruppe
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "1", "FamilyName", ".*", "" );

// alle variablen Inhalte, die auf 'Name' enden
```

```
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, ".*", ".*Name", ".*", "" );

// alle variablen Inhalte die 'Value' heißen von zwei expliziten
Labelobjekten (sollte genau zwei Treffer landen)
err = ReaPi.PrepareLabelContent( _handleLabelContent,
                   1, "1", "CIPCity|FamilyName", "Value", "" );
```

7.4 Using containers to transport contents

7.4.1 Create the container for label contents

Creates a new label content container object in the REA-PI library and returns the handle of this object. The handle should be used in the following API-functions for addressing this container.

Syntax:

```
REAPI_DLL TLabelContentHandle REAPI_CreateLabelContent();
```

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	Creating container for label contents failed.
`TLabelContentHandle`	Handle of the new created container for label contents.

7.4.2 Reset the container for label contents

Resets an existing label content container object. After reset all data of this label object is lost.

Syntax:

```
REAPI_DLL TErrorCode REAPI_ResetLabelContent(
                TLabelContentHandle contents );
```

Parameters:

`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent().

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	Label content container object reset successfully.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.4.3 Remove the container for label contents

Removes an existing label content container object. After this call the handle becomes invalid.

Syntax:

```
REAPI_DLL TErrorCode REAPI_RemoveLabelContent(
                TLabelContentHandle contents );
```

Parameters:

`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent().

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	Label content container object removed successfully.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.4.4 Add a label content to container

Adds a new value to a label content container or replaces the value if a referenced entry already exists.

See the description oof the function SetLabelContent() for how the search queries have to be built.

Syntax:

```
REAPI_DLL TErrorCode REAPI_PrepareLabelContent(
                TLabelContentHandle contents,
                TJobId jobId,
                char* group,
                char* object,
                char* content,
                char* value );
```

Parameters:

`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent().
`jobId`	Job identifier, 1 for first job Note: All versions of device firmware up to current version supports only one job
`group`	Group-name as specified in installation settings
`object`	Object-name as specified in label definition
`content`	Content-name as specified in label definition
`value`	Value to be set

Return:

`ERRORCODE_INVALID_PARAMETER`	At least one of the parameter was not accepted, null-pointer for strings are not possible.
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	Label content container object content added/set successfully.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.4.5 *Create an container for label content values*

Creates a new label content values container object in the REA-PI library and returns the handle of this object. This type of container corresponds to the <Content-Buffer> content type. The handle should be used in the following API-functions for addressing this container.

This command is available with a Firmware-Version ≥ 3.50.

Syntax:

```
REAPI_DLL TLabelContentValuesHandle
       REAPI_CreateLabelContentValues();
```

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	Creating container for label content values failed.
`TLabelContentHandle`	Handle of the new created container for label content values.

7.4.6 *Add a value to the container of label content values*

Add a new value to an existing label content values container.

This command is available with a Firmware-Version ≥ 3.50.

Syntax:

```
REAPI_DLL TErrorCode REAPI_AddLabelContentValues(
               TLabelContentValuesHandle contentvalues,
               char* value );
```

Parameters:

`contentvalues`	Handle of the label content values container previously created with CreateLabelContentValues().
`value`	content value to be added and set

Return:

ERRORCODE_INVALID_LABEL_HANDLE	No label content values container exists with provided handle
ERRORCODE_OK	Add label content value to container successful.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.4.7 *Remove the container for label content values*

Remove an existing label content values container object. After this call the handle becomes invalid.

This command is available with a Firmware-Version ≥ 3.50.

Syntax:

```
REAPI_DLL TErrorCode REAPI_RemoveLabelContentValues(
              TLabelContentValuesHandle contentvalues );
```

Parameters:

contentvalues	Handle of the label content values container previously created with CreateLabelContentValues().

Return:

ERRORCODE_INVALID_LABEL_HANDLE	No label content values container exists with provided handle
ERRORCODE_OK	Label content values container successful removed.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.4.8 *Add label content values to container*

Add label content values to a label content container or replaces values if a referenced entry already exists.

See the description of the function SetLabelContent() for how the search queries have to be built.

This command is available with a Firmware-Version ≥ 3.50.

Syntax:

```
REAPI_DLL TErrorCode REAPI_PrepareLabelContentValues(
              TLabelContentHandle contents,
              TJobId jobId,
              char* group,
              char* object,
              char* content,
              TLabelContentValuesHandle valuelist );
```

Parameters:

`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent ().
`jobId`	Job identifier, 1 for first job Note: All versions of device firmware up to current version supports only one job
`group`	Group-name as specified in installation settings
`object`	Object-name as specified in label definition
`content`	Content-name as specified in label definition
valuelist	Handle of the label content values container previously created with CreateLabelContentValues()

Return:

`ERRORCODE_INVALID_PARAMETER`	At least one of the parameter was not accepted, null-pointer for strings are not possible.
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content values container exists with provided handle
`ERRORCODE_OK`	Label content values container content successful added/set.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.5 Use container for transport properties

7.5.1 Create the container for label objects

Create a new label object container in the REA-PI library and returns the handle of this object. The handle should be used in the following API-functions for addressing this label object.

Syntax:

```
REAPI_DLL TLabelObjectHandle REAPI_CreateLabelObject();
```

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	Creating container for label objects failed.
`TLabelContentHandle`	Handle of the new created container for label objects.

7.5.2 Reset the container for label objects

Reset an existing label object container. After reset all data of this label object container is lost.

Syntax:

```
REAPI_DLL TErrorCode REAPI_ResetLabelObject(
              TLabelObjectHandle objectproperties );
```

Parameters:

`objectproperties`

Handle of the label object container previously created with CreateLabelObject().

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	No label object container exists with provided handle.
`ERRORCODE_OK`	Label object container successful reset.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.5.3 Remove the container for label objects

Removes an existing label object container. After this call the handle becomes invalid.

Syntax:

```
REAPI_DLL TErrorCode REAPI_RemoveLabelObject(
              TLabelObjectHandle objectproperties );
```

Parameters:

`objectproperties`	Handle of the label object container previously created with CreateLabelObject().

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	No label object container exists with provided handle.
`ERRORCODE_OK`	Label object container successful removed.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.5.4 *Add a label object to container*

Adds a new entry to label object container or replaces the value if a referenced entry already exists.

Syntax:

```
REAPI_DLL TErrorCode REAPI_PrepareLabelObject(
              TLabelObjectHandle objectproperties,
              TJobId jobId,
              char* group,
              char* object,
              char* content,
              char* propertypath,
              char* value );
```

Parameters:

`objectproperties`	Handle of the label object container previously created with CreateLabelObject().
`jobId`	Job identifier, 1 for first job Note: All versions of device firmware up to current version supports only one job
`group`	Name of group as specified in installation settings
`object`	Name of label object as specified in label definition
`content`	Name of content as specified in label definition
`propertypath`	Complete name of property to set
`value`	Value to be set

Return:

`ERRORCODE_INVALID_LABEL_HANDLE`	No label object container exists with provided handle.
`ERRORCODE_OK`	Label object container successful reset.

For further information for error codes `TErrorCode` see chapter 3.5 Basic API types, page 9).

7.6 Dynamically change label contents of a loaded job

7.6.1 *Query label contents*

This command retrieves a current snapshot of variable contents for types <Variable> and <Choice>. Since version 3.50 of the firmware variable contents of type <ContentBuffer> are supported, too.

It is possible to restrict the query to certain objects / contents by specifying job, group, object and contens respectively. If the handle for label content is empty, i.e., no label content object was added with `PrepareLabelContent`, all variable contents of all labels in the job are retrieved.

It is possible to create one or more handles for label contents with `PrepareLabelContent` to retrieve especially these specified contents.

This command is available with a Firmware-Version ≥ 3.20.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_GetLabelContent(
               TConnectionId connection,
               TLabelContentHandle contents );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent(). Used as Mask to select label contents to query.

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.6.1.1 Command return

With the following functions the return value of the command can be retrieved when the functions are called from the command callback by the corresponding commandID.

Use the response handle of the command callback to get the corresponding command return. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.3 Response callback, page 15.

REAPI_GetLabelPropertyNumber

Get the number of data records with label properties. The returned number should be used to iterate the data records with following functions for retrieve the label properties.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyNumber(
               TResponseHandle    response,
               TErrorCode* error );
```

REAPI_GetLabelPropertyType

Get the type of the label property. The only possibility in this context is `TYPE_LABELCONTENT`.

```
REAPI_DLL TLabelPropertyType REAPI_GetLabelPropertyType(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyJobID

Get the job identifier the label property.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyJobID(
               TResponseHandle    response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyGroupName

Get the name of the group of the label property.

```
REAPI_DLL const char* REAPI_GetLabelPropertyGroupName(
               TResponseHandle     response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyLabelObject

Get the name of the label object of the label property.
This function is available since version 3.41.

```
REAPI_DLL const char* REAPI_GetLabelPropertyLabelObject(
               TResponseHandle    response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyLabelObjectType

Get the type of the label object of the label property.
This function is available since version 3.41.

```
REAPI_DLL const char* REAPI_GetLabelPropertyLabelObjectType(
               TResponseHandle    response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyObjectContent

Get the name of the content of the label property. This function only works with label property types `TYPE_LABELCONTENT` and `TYPE_LABELCONTENTPROPERTY`.

```
REAPI_DLL const char* REAPI_GetLabelPropertyObjectContent(
               TResponseHandle    response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyObjectContentType

Get the type of the content of the label property. This function only works with label property types `TYPE_LABELCONTENT` and `TYPE_LABELCONTENTPROPERTY`.
This function is available since version 3.41.

```
REAPI_DLL const char* REAPI_GetLabelPropertyObjectContentType(
               TResponseHandle    response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyValue

Get the value of the requested property. This function works for variable object contents of the types <Variable> and <Choice> which contains only a singe value. It is recommended to check the type of the object content with the function REAPI_GetLabelPropertyObjectContentType().

```
REAPI_DLL const char* REAPI_GetLabelPropertyValue(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyValueNumber

Get number of the values of the requested object content. This function works for variable object contents of the types <Variable> and <Choice> as well as for object contents of the types <ContentBuffer> which can contain multiple values.
This function is available since version 3.41.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyValueNumber(
                TResponseHandle response,
                int32_t labelpropertyId,
                TErrorCode* error );
```

REAPI_GetLabelPropertyValueValue

Get the value by index of the requested object content. This function works for variable object contents of the types <Variable> and <Choice> as well as for object contents of the types <ContentBuffer> which can contain multiple values.
This function is available since version 3.41.

```
REAPI_DLL const char* REAPI_GetLabelPropertyValueValue(
                TResponseHandle response,
                int32_t labelpropertyId,
                int32_t valueId,
                TErrorCode* error );
```

This command retrieves a current snapshot of variable contents for types <Variable> and <Choice>. Since version 3.50 of the firmware variable contents of type <ContentBuffer> are supported, too.

`response`	Handle of the response returned by response callback function.
`labelpropertyId`	Label property number within range [1 .. LabelPropertyNumber]
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.6.2 *Set label contents*

This command sets and replaces one or more variable label contents of current loaded label(s) on device.

Apart from a content of the types <Variable> und <Choice> also a content of type <ContentBuffer> may be given, a firmware of version 3.50 and above, supporting the type <ContentBuffer>, supposed.

Fundamental to this command is, that the container provided will be treated atomically, i.e., either all of the values from one call to SetLabelContent() will be assigned or none at all.

In contrast to this, consecutive calls to SetLabelContent will be processed separately. Hence the modified values could be distributed over several prints in that case.

This command is available with a firmware version ≥ 1.80.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SetLabelContent(
               TConnectionId connection,
               TLabelContentHandle contents );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent() containing the label contents to set (see chapter 7.4 Using containers to transport contents, page 85).

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.6.3 *Add label contents*

This command adds one or more variable label contents to a content of type <ContentBuffer>. Like the function SetLabelContent, also this function treats the given references atomically, i.e., either adds all of them to the specified content buffers or none.

In contrast to this, consecutive calls of AddLabelContent result in separate assignments and thus may be spread over several prints.

This command first appeared in firmware 3.38 but is generally available with a firmware version ≥ 3.50.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_AddLabelContent(
               TConnectionId connection,
               TLabelContentHandle contents );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent() containing the label contents to set (see chapter 7.4 Using containers to transport contents, page 85).

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.7 Manipulate label object properties of a loaded job

7.7.1 *Get label object properties*

This command retrieves a snapshot of a specified query of variable contents of types ‘Variable’ and ‘Choice’ and the specified label object properties of all used labels in an assigned job.

If the handle for label content container is empty, no label content object is added with `PrepareLabelContent`, all variable contents of all labels in the job are retrieved.

It is possible to create one or more handles for label contents with `PrepareLabelContent` to retrieve especially these specified contents.

This command is available with a Firmware-Version ≥ 3.20.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_GetLabelObject(
               TConnectionId connection,
               TLabelContentHandle contents,
               TLabelObjectHandle objectproperties );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent(). Used as mask to select label contents to query.
`objectproperties`	Handle of the label object property container previously created with REAPI_CreateLabelObject(). Used as mask to select label object properties to query.

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container or label object property container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.7.1.1 Command return

With the following functions the command return can be retrieved when the functions are called from the command callback by the corresponding commandID.

Use the response handle of the command callback to get the corresponding command return. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.3 Response callback, page 15.

Overview for the label property types with its corresponding functions:

	CONTENT	LABEL PROP-ERTY	OBJECT-PROP-ERTY	CONTENT PROP-ERTY
REAPI_GetLabelPropertyNumber()	X	X	X	X
REAPI_GetLabelPropertyType()	X	X	X	X
REAPI_GetLabelPropertyJobID()	X	X	X	X
REAPI_GetLabelPropertyGroupName()	X	X	X	X
REAPI_GetLabelPropertyLabelObject()	X		X	X
REAPI_GetLabelPropertyObjectContent()	X			X
REAPI_GetLabelPropertyProperty()		X	X	X
REAPI_GetLabelPropertyValue()	X	X	X	X

REAPI_GetLabelPropertyNumber

Get the number of data records with label properties. The number should be used to iterate the data records with following functions for retrieve the label properties.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyNumber(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetLabelPropertyType

Get the type of the label property. Possible types are `TYPE_LABELCONTENT`, `TYPE_LABELOBJECTPROPERTY` and `TYPE_LABELCONTENTPROPERTY`. The possible usage of the following functions is depending on this type.

```
REAPI_DLL TLabelPropertyType REAPI_GetLabelPropertyType(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyJobID

Get the job identifier the label property.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyJobID(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyGroupName

Get the name of the group of the label property.

```
REAPI_DLL const char* REAPI_GetLabelPropertyGroupName(
               TResponseHandle  response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyLabelObject

Get the name of the label object of the label property.

```
REAPI_DLL const char* REAPI_GetLabelPropertyLabelObject(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyObjectContent

Get the name of the content of the label property. This function only works with label property types `TYPE_LABELCONTENT` and `TYPE_LABELCONTENTPROPERTY`.

```
REAPI_DLL const char* REAPI_GetLabelPropertyObjectContent(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyProperty

Get the name of the requested changed property. The function works with label property types `TYPE_LABELOBJECTPROPERTY` and `TYPE_LABELCONTENTPROPERTY` only.

```
REAPI_DLL const char* REAPI_GetLabelPropertyProperty(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyValue

Get the value of the requested changed property.

```
REAPI_DLL const char* REAPI_GetLabelPropertyValue(
               TResponseHandle  response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

`response`	Handle of the response returned by response callback function.
`labelpropertyId`	Label property number within range [1 .. LabelPropertyNumber]
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.

`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid.
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.7.2 *Set Label object properties*

This command changes one or more label object properties, content properties and variable label contents of current loaded label(s) on device.

All entries of one command to set all object properties and label contents are treated atomic altogether. This means for one specific printing operation all modifications in the label are done simultaneously or no modification will be done at all.

In contrast to this, consecutive commands to set object properties and label contents are not atomic; modifications will be done successively and thus may be spread across several prints.

Previously with `PrepareLabelObject and PrepareLabelContent` written content is actually send to the device.

This command is available with a Firmware-Version ≥ 1.80.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SetLabelObject(
               TConnectionId connection,
               TLabelContentHandle contents,
               TLabelObjectHandle objectproperties );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`contents`	Handle of the label content container previously created with REAPI_CreateLabelContent() containing the label contents to set
`objectproperties`	Handle of the label content container previously created with REAPI_CreateLabelContent() containing the label properties to set

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container or label object property container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.8 Label manipulation control events

7.8.1 *Event ReadyForNextContent*

The device sends this event when it is safe to send new data for a variable, i.e., the print buffer for the current print has been locked and thus sending new data may only change what is printed subsequently. Furthermore, note that, in single print mode, this event also indicates that new data has to be sent, as otherwise MissingContent and / or PrintRejected could occur upon the next trigger.

In contrast to the event ReadyForNextContent, which applies to the content types <Variable> and <Choice>, the event BufferUnderrun only occurs for contents of type <ContentBuffer> and therefore behaves differently.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.40.

7.8.1.1 Subscription

The subscription is only effective while a job and a label has been loaded. It is recommended to subscribe to this event when the JobSet event has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
              REAPI_SubscribeReadyForNextContent(
                      TConnectionId connection,
                      TJobId job,
                      const char* group );
REAPI_DLL TResponseHandle
              REAPI_UnsubscribeReadyForNextContent(
                      TConnectionId connection,
                      TJobId job,
                      const char* group );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
`group`	Group name of the print head group that will be observed

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid

`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.8.1.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from within the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId
REAPI_GetGroupname

Get the identifier of the job and the group in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetGroupname (
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.8.2 *Event InvalidContent*

The device sends this event if a label contents or label properties could not be rendered by the device. In this case no print image (buffer) will be generated and subsequently printing is not possible.

This event may indicate that a content which has been set by the function SetLabelContent or SetLabelObject is invalid.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 1.80.

7.8.2.1 Subscription

The subscription is only effective while a job and a label has been loaded. It is recommended to subscribe to this event when the event for a new job was set has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SubscribeInvalidContent(
              TConnectionId connection,
              TJobId job,
              const char* group );
REAPI_DLL TResponseHandle REAPI_UnsubscribeInvalidContent(
              TConnectionId connection,
              TJobId job,
              const char* group );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
`group`	Group name of the print head group that will be observed

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.8.2.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId
REAPI_GetGroupname

Get the identifier of the job and the group in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetGroupname (
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetJobErrorDomain
REAPI_GetJobErrorCode
REAPI_GetJobErrorMessage

Get the error information why this content is invalid

```
REAPI_DLL int32_t REAPI_GetJobErrorDomain(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL int32_t REAPI_GetJobErrorCode(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetJobErrorMessage(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command

`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.8.3 *Event Label Property Changed*

The device sends this event if one or more label contents or label properties has been changed.

These changes are caused not only by commands to set label contents or label properties, instead they may also be caused by dynamic label objects like date/time <Time>, counter <Counter>, etc. For more information about dynamic label objects see the documentation REA JET XML Labelformat.doc.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 2.00.

7.8.3.1 Subscription

The subscription is only effective while a job and a label has been loaded. It is recommended to subscribe to this event when the event for a new job was set has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle REAPI
   REAPI_SubscribeLabelPropertyChanged(
               TConnectionId connection,
               TJobId job,
               const char* group );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeLabelPropertyChanged(
               TConnectionId connection,
               TJobId job,
               const char* group );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
`group`	Group name of the print head group that will be observed

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.

`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.8.3.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetLabelPropertyNumber

Get the number of data records with label properties. The number should be used to iterate the data records with following functions for retrieve the label properties.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyNumber(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetLabelPropertyType

Get the type of the label property. Possible types are `TYPE_LABELCONTENT`, `TYPE_LABELOBJECTPROPERTY` and `TYPE_LABELCONTENTPROPERTY`. The possible usage of the following functions is depending on this type.

```
REAPI_DLL TLabelPropertyType REAPI_GetLabelPropertyType(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyJobID

Get the job identifier of the label property.

```
REAPI_DLL int32_t REAPI_GetLabelPropertyJobID(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyGroupName

Get the name of the group of the label property.

```
REAPI_DLL const char* REAPI_GetLabelPropertyGroupName(
               TResponseHandle  response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyLabelObject

Get the name of the label object of the label property.

```
REAPI_DLL const char* REAPI_GetLabelPropertyLabelObject(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyObjectContent

Get the name of the content of the label property. This function only works with label property types `TYPE_LABELCONTENT` and `TYPE_LABELCONTENTPROPERTY`.

```
REAPI_DLL const char* REAPI_GetLabelPropertyObjectContent(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyProperty

Get the name of the requested changed property. This function works with label property types `TYPE_LABELOBJECTPROPERTY` and `TYPE_LABELCONTENTPROPERTY` only.

```
REAPI_DLL const char* REAPI_GetLabelPropertyProperty(
               TResponseHandle response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

REAPI_GetLabelPropertyValue

Get the value of the requested property which was changed.

```
REAPI_DLL const char* REAPI_GetLabelPropertyValue(
               TResponseHandle  response,
               int32_t labelpropertyId,
               TErrorCode* error );
```

`response`	Handle of the response returned by response callback function.
`labelpropertyId`	Label property number within range [1 .. LabelPropertyNumber]
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid.

`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.8.4 *Event LabelEvent*

The device sends this event when a label event has been raised due to a change of a digital input or output and label properties are affected.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.20.

7.8.4.1 Subscription

The subscription is only effective while a job and a label has been loaded. It is recommended to subscribe to this event when the event for a new job was set has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
    REAPI REAPI_SubscribeLabelEvent(
               TConnectionId connection,
               TJobId job,
               const char* labelevents );
```

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeLabelEvent(
               TConnectionId connection,
               TJobId job,
               const char* labelevents );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
labelevents	The required argument specifies the requested label events by identifier. The format is one label event identifier or more label event identifiers separated by semicolon or a range of label event identifiers with hyphen.

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.8.4.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId

Gets the identifier of the job in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetLabelEvents

Get a string with a label event identifier when a label event has been raised due to a change of a digital input or output.
Remark that each event sends only one identifier, regardless how many label events are sucscribed.

```
REAPI_DLL const char* REAPI_GetLabelEvents(
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.9 Dynamically change Label Contents using a Queue

This section deals with the management of the “content buffer” functionality, i.e., a FIFO which holds values to be assigned (from print to print) to the corresponding entities in the label. There are commands for the configuration of the FIFO as well as events for the notification about the current buffer status:

- SetContentBufferControl(): Set parameters for the buffer. These include
  - BufferUnderrun - threshold when event BufferUnderrun should be issued
  - BufferFull - threshold when event BufferFull should be issued
  - Expiration - new entries should expire after given delay
  - ExpireDelay - delay for new entries to expire
- SetContentBufferControl():
  - MaxBufferSize - currently not implemented (always 0)
  - BufferPrinted - number of entries which have been taken from the buffer for printing
  - BufferExpired - number of entries which have been taken from the buffer because of expiry
- BufferUnderrun: This event indicates that the number of entries left in the buffer is below the configured threshold (parameter “BufferUnderrun” see above).
- BufferFull: This event indicates that the number of entries left in the buffer is above the configured threshold (parameter “BufferFull” see above).

7.9.1 Buffered Data Transfer in a Serialisation Application

When doing serialisation, it is crucial to prevent the printing of duplicate fields. For that purpose, REA JET CL, FL, HR (*pro*) and related systems all provide the so-called "single print mode", which invalidates every variable content after each print. Therefore a host application has to (re-) fill variable content, so that it becomes prepared in time on the print system for the subsequent print.

For REA JET CL, FL, HR (*pro*) and related systems there are generally three ways to accomplish this task:

- Unbuffered[1]: If the content type is "Variable", its content will have to be set for each print using the SetLabelContent() function. Since the current print image (bitmap) will not be fixed as soon as the print is started, the content for the subsequent print may and should be set upon receipt of the corresponding PrintStart / ReadyForNextContent event. This process is "real-time" in the sense that, whenever there is a significant delay in the com-

[1] REA JET CL, FL, HR (*pro*) and related systems firmware 1.84 and above

munication between device and host, it might happen that the print image cannot be prepared in time for the subsequent print.

The consequence of a missing print image can be configured by the SetNoPrintBufferMode()[2] function.

- FIFO[3]: Using "ContentBuffer" as a content type, new content may also be fed every time a print is executed. Though, in contrast to a "Variable", the "ContentBuffer" may hold several entries which are to be used subsequently. The event the host listens on, remains the same, although, the function AddLabelContent() instead of SetLabelContent() is used to insert new elements at the end of the queue. Despite of the fact, that the queue (FIFO) has firstly to be initialized with the desired number of elements by the function SetLabelContent(), the workflow is the same as before.
- Queue (Advanced)[4]: However, there is an advanced queue mechanism available with "ContentBuffer":

  During initialisation, the queue will be filled to a certain degree. Additionally, the event BufferUnderrun may be configured to raise when the queue's fill level falls below the threshold defined. Upon receipt of this BufferUnderrun event, the host then sends several entries at a time (using AddLabelContent), so that the desired fill level of the queue is restored.

  Additionally, it is possible to define an expiration for queue elements, i.e., a time after which an individual element gets removed from the queue automatically, cf. SetContentBufferControl() function.

  An example for the advanced queue mechanism would look like:

  1. Configure BufferUnderrun to raise when less than 51 elements remain in the queue
  2. Initialise queue with 150 elements
  3. Execute 100 prints
  4. Receive a BufferUnderrun event
  5. Re-Fill 100 elements using AddLabelContent()

Because of the the advanced queue mechanism, the content type "ContentBuffer" is designated with "Queue" in the Label Creator. The Choice of these content types will be found in the "Contents" tab of the properties dialog for a label object.

[2] Here, "buffer" denotes the storage location of the aforementioned print image - it does not relate to a buffer in the communication between host and device.

[3] REA JET CL, FL, HR (*pro*) and related systems firmware 3.50 and above, as well as 3.38 and 3.39, but not 3.4x versions

[4] REA JET CL, FL, HR (*pro*) and related systems firmware 3.50 and above

7.9.2 Retrieving information about content buffer control

This command requests the device to report the current buffer configuration, cf. the parameters above.

This command is available with a firmware version ≥ 3.50.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SetContentBufferControl (
                TConnectionId connection,
                int32_t nMaxBufferSize,
                int32_t nBufferUnderrun,
                int32_t nBufferFull,
                bool_t fExpiration,
                int32_t nExpireDelay );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`nMaxBufferSize`	maximal numberof content entries
`nBufferUnderrun`	low water mark of content entries
`nBufferFull`	high water mark of content entries
`fExpiration`	flag for content entries to expire or not
`nExpireDelay`	delay time in seconds after a content entry goes expired

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.9.3 Set content buffer control parameters

This command sets the buffer configuration, cf. the parameters above.

This command is available with a firmware version ≥ 3.50.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SetContentBufferControl (
               TConnectionId connection,
               int32_t nMaxBufferSize,
               int32_t nBufferUnderrun,
               int32_t nBufferFull,
               bool_t fExpiration,
               int32_t nExpireDelay );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`nMaxBufferSize`	maximal numberof content entries
`nBufferUnderrun`	low water mark of content entries
`nBufferFull`	high water mark of content entries
`fExpiration`	flag for content entries to expire or not
`nExpireDelay`	delay time in seconds after a content entry goes expired

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_INVALID_LABEL_HANDLE`	No label content container exists with provided handle
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.9.4 *Event BufferUnderrun*

The device sends this event when the FIFO is going to run out of data, i.e., the number of entries in the buffer is below the threshold, which was configured by the “BufferUnderrun” parameter.

In contrast to the event ReadyForNextContent, which applies to the content types <Variable> and <Choice>, the event BufferUnderrun only occurs for contents of type <ContentBuffer> and therefore behaves differently.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.50.

7.9.4.1 Subscription

The subscription is only effective while a job and a label has been loaded. It is recommended to subscribe to this event when the JobSet event has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
        REAPI_SubscribeBufferUnderrun(
                TConnectionId connection,
                TJobId job,
                const char* group );
```

```
REAPI_DLL TResponseHandle
        REAPI_UnsubscribeBufferUnderrun(
                TConnectionId connection,
                TJobId job,
                const char* group );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
`group`	Group name of the print head group that will be observed

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to

identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.9.4.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId
REAPI_GetGroupname

Gets the identifier of the job and the group in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetGroupname (
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

7.9.5 *Event BufferFull*

The device sends this event when the number of entries in the buffer is above the threshold, which was configured by the “BufferFull” parameter.

The event only relates to contents of type <ContentBuffer> when the fill level of the corresponding queue discourages to send new data.

This event requires a subscription and has event parameters.

This command is available with a Firmware-Version ≥ 3.50.

7.9.5.1 Subscription

The subscription is only effective while a job and a label has been loaded. It is recommended to subscribe to this event when the JobSet event has been received.

Unsubscribe the event if it is no longer needed. The event is automatically unsubscribed, when the job is replaced by another job or is simply unloaded.

Syntax:

```
REAPI_DLL TResponseHandle
              REAPI_SubscribeBufferFull(
                      TConnectionId connection,
                      TJobId job,
                      const char* group );
```

```
REAPI_DLL TResponseHandle
              REAPI_UnsubscribeBufferFull(
                      TConnectionId connection,
                      TJobId job,
                      const char* group );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`job`	Job identifier, 1 for first job *Note: All versions of device firmware up to current version support only one job*
`group`	Group name of the print head group that will be observed

Return:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`ERRORCODE_OK`	In synchronous mode only: the subscription was successful.
`TResponseHandle`	In asynchronous mode only: the response handle. This handle should be used to identify this command in response callback and get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

7.9.5.2 Event parameters

With the following functions the event parameters can be retrieved when the functions are called from the event callback by the corresponding eventID.

Use the response handle of the event callback to get the corresponding event parameters. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.5 Event callback, page 18.

REAPI_GetJobId
REAPI_GetGroupname

Gets the identifier of the job and the group in which the event occurred, 1 for the first job.
Note: All versions of device firmware up to current version support only one job.

```
REAPI_DLL int32_t REAPI_GetJobId(
               TResponseHandle response,
               TErrorCode* error );
```

```
REAPI_DLL const char* REAPI_GetGroupname (
               TResponseHandle response,
               TErrorCode* error );
```

`response`	Handle of the response returned by event callback function.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

8 Device settings HR Family

8.1 Content of Chapter

8.2 Event CartridgeChanged

The device sends this event whenever the state of a cartridge changes, there exists no subscription or unsubscription. Examples are the insertion or removal of a cartridge or when the ink falls below the error or warning level.

With the following functions the event parameters could be retrieved.

Usually, the following functions are called from an event callback when eventID is JobStopped. Use the response handle to get the corresponding event parameter.

The response handle is only valid and should only be used inside the callback function! For retrieving an event see chapter 3.7.5 Event callback, page 18.

8.3 Manual query cartridge states

This command retrieves a snapshot of information and states of all cartridges. Some information is available only if the cartridge is involved in the loaded job.

This command is available with a Firmware-Version ≥ 1.80.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetCartridges(
               TConnectionId connection,
               TErrorCode* error );
```

Parameters:

connection Connection id returned by REAPI_Connect() or the corresponding connection callback **error** obsolete, do not use this parameter

Returns:

TResponseHandle	ResponseHandle in synchronous mode to retrieve response elements, ERRORCODE_OK in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

All getter functions below should be called in the response callback context of this call.

Remark, that the quality of information is a snapshot and depends on the state of the engine. So for example you can not get information for number of printable labels without any label set or you will get a wrong number of printable labels for a cartridge which is not included in the actual job.

8.4 Functions for retrieving cartridge states

8.4.1 Get number of Cartridges

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetCartridgeNumber(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Number of Cartridges present in HR-device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.2 Get PrintHead ID of Cartridge

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetCartridgePrintHeadId(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command

is not valid

Returns:

int32_t	Identifier of print head in which the cartridge is inserted. This parameter could also be called cartridge identifier as these identifiers are identical.First printhead /cartridge is 1, second is 2 and so on.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.3 Get name of Cartridge

Syntax:

```
REAPI_DLL const char*
   REAPI_GetCartridgeName(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Name of the cartridge, more precisely the name of the ink in this cartridge.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.4 Get ink identifier of Cartridge

Syntax:

```
REAPI_DLL const char*
   REAPI_GetCartridgeInkID(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

response Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **cartridgeId** Cartridge index within range [1 .. CartridgeNumber] **error** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Identifier of the ink. It is derived from article number and can be interpreted as unique type identifier.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.5 Get maximum fill level of Cartridge

Syntax:

```
REAPI_DLL double
   REAPI_GetCartridgeMaxLevel(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

`Parameters:`

response Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **cartridgeId** Cartridge index within range [1 .. CartridgeNumber] **error** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that

command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

double	Maximum fill level / size of the cartridge

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.6 Get current fill level of Cartridge in ml

Syntax:

```
REAPI_DLL double
   REAPI_GetCartridgeGaugeMl(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

double	Current fill level of the cartridge in milliliter.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.7 Get current fill level of Cartridge in percent

Syntax:

```
REAPI_DLL double
   REAPI_GetCartridgeGaugePercent(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

double	current fill level/gauge of selected cartridge in per-cent

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.8 Get remaining number of printable labels

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetCartridgeLabelsPrintable(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Remaining number of labels printable with current label configuration and ink level of cartridge. -1 if not involved in a job.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges` call.

8.4.9 Get ink usage per printed label

Syntax:

```
REAPI_DLL double
   REAPI_GetCartridgeInkUsagePerPrint(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

double	Returns ink usage per print for specified cartridge in picoliter as double value. -1 if not involved in a job.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.10 Get information about bulk ink storage

Syntax:

```
REAPI_DLL bool_t
   REAPI_GetCartridgeIsBulk(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	Indicates whether a bulk cartridge is used. Returns true for Bulk or false otherwise for specified cartridge.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.11 Get current cartridge temperature

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetCartridgeTemperature(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Returns the current temperature of the specified cartridge in degree Centigrade or -1, if the temperature is not available. Note, that no working cartridge will have a temperature of -1°!

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.12 Get number of defect nozzles

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetCartridgeDefectNozzles(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Returns the number of defect nozzles or -1, if no measurement is available.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.13 Get State of Cartridge

Syntax:

```
REAPI_DLL const char*
   REAPI_GetCartridgeState(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Returns current state of specified cartridge.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.14 *Get information about spitting status of cartridge*

Syntax:

```
REAPI_DLL bool_t
   REAPI_GetCartridgeIsSpitting(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridges() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	Returns true when spitting is active or false if spitting is not active.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.15 Get information about pulse warming status of cartridge

Syntax:

```
REAPI_DLL bool_t
   REAPI_GetCartridgeIsPulseWarming(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	Returns true when pulse warming is active or false if pulse warming is not active

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.16 Get Warnings of Cartridge

Syntax:

```
REAPI_DLL const char*
   REAPI_GetCartridgeWarning(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.

ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid
ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command
ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

const char*	Returns warnings of specified cartridge. e.g. “ink” if cartridge gauge is lower than the defined ink warning value

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.4.17 Get Errors of Cartridge

Syntax:

```
REAPI_DLL const char*
   REAPI_GetCartridgeError(
               TResponseHandle response,
               int32_t cartridgeIndex,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridges() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Returns errors of specified cartridge. e.g. “ink” if cartridge gauge is lower than the defined ink error value

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridges()` call.

8.5 Query cartridge parameter information

Query cartridge parameter information from a device. One query resolves all cartridge parameter information for all possible cartridges of a device. In the response callback function, the user has to read the specific information for every cartridge with the following getter functions.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetCartridgeParameter(
               TConnectionId connection );
```

Parameters:

`connection`	Connection id returned by REAPI_Connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

All getter functions below should be called in the response callback context of this call.

8.6 Functions for retrieving cartridge parameters

8.6.1 *Get number of Cartridges*

Syntax:

```
REAPI_DLL int32_t REAPI_GetInkErrorCartridgeNumber(
              TResponseHandle response,
              TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridgeParameter() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Maximum number of cartridges which could be pre-sent in device. This number depends on the device, e.g., two cartridges are possible for the device "REA JET HR 2K".

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridgeParameters()` call.

8.6.2 *Get ink warning level*

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetWarningLevel(
              TResponseHandle response,
              int32_t cartridgeId,
              TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridgeParameter() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid

ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

int32_t	Returns ink warning level in milliliter. The device will announce an ink warning, if the fill level of a cartridge falls below this parameter. This warning is independent of the ink warning behavior described below.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridgeParameters()` call.

8.6.3 Get ink error level

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetErrorLevel(
                TResponseHandle response,
                int32_t cartridgeId,
                TErrorCode* error );
```

Parameters:

response Handle of the response object previously returned by REAPI_GetCartridgeParameter() or its callback function **cartridgeId** Cartridge index within range [1 .. CartridgeNumber] **error** TErrorCode* used to return ERRORCODE_OK regarding a successful call. ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

int32_t	Returns ink error level in milliliter. The device will announce an ink error if the fill level of a cartridge falls below this level. The indication of an ink error is independent of the ink error behavior described below.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridgeParameters()` call.

8.6.4 Get ink error behavior

Syntax:

```
REAPI_DLL const char*
   REAPI_GetErrorEvent(
               TResponseHandle response,
               int32_t cartridgeId,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetCartridgeParameter() or its callback function
`cartridgeId`
Cartridge index within range [1 .. CartridgeNumber]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Returns ink error behaviour as string. In case a loaded and started print job should be stopped when the cartridge fill level falls below the ink error level, the string is ‘StopJob’. Otherwise (empty string or ‘None’) there is no special behaviour in addition to announcing an error.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridgeParameters()` call.

8.6.5 Get fixed voltage setting

Syntax:

```
REAPI_DLL bool_t
   REAPI_GetFixedVoltageSetting(
               TResponseHandle response,
               int32_t cartridgeId,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetCartridgeParameter() or its callback function **`cartridgeId`** Cartridge index within range [1 .. CartridgeNumber] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	Returns true if fixed voltage is used.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetCartridgeParameters()` call.

8.7 Changing cartridge parameters

Syntax:

```
TResponseHandle REAPI
   REAPI_SetCartridgeParameter(
               TConnectionId connection,
               int32_t warninglevel,
               const char* warningevent,
               int32_t errorlevel,
               const char* errorevent,
               bool_t usefixedvoltage );
```

Parameters:

`connection`
Connection id returned by REAPI_connect or the corresponding connection call-back
`warninglevel`
Sets level of ink in milliliter as integer to announce an ink warning. The value must be larger than the the corresponding parameter for ink error value!
`errorlevel`
Sets level of ink in milliliter as integer to announce an ink error. The value must be smaller than the corresponding parameter for ink warning!
`errorevent`
For this parameter you can set 'StopJob' to immediately stop printing when ink is empty at given level. To reset event handler not to do any action on ink error, send 'None' or an empty string.
`usefixedvoltage`
For parameter VoltageMode two parameters are possible: 'FixedVoltage' to use default voltage or 'ChipVoltage' to use optimal voltage from chip data.

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - parameter set successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Change cartridge parameters of the device.

This supplementary API function causes data to be sent to and received from the device.

9 Device settings CL Family

9.1 Content of Chapter

9.2 Event LaserUnitInfo

The device sends this event whenever the state of the laser unit changes.

With the following functions the event parameter could be retrieved.

Normally the following functions are called from event callback when eventID is JobStopped. Use the response handle to get the corresponding event parameter.

The response handle is only valid and should only be used inside the callback function! For retrieving an event see chapter 3.7.5 Event callback, page 18.

9.3 Query laser unit information

There is no separate command to query the laser unit status information. Currently, it is implemented only as an event instead. However, like other status events, it will be received containing the current status at time of subscription.

9.4 Functions for retrieving laser unit infos

9.4.1 Subscribe to the Laser Unit Info Event

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SubscribeLaserUnitInfo(
               TConnectionId connection );
```

Parameters:

`connection` Connection id returned by REAPI_Connect() or the corresponding connection call-back

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - LASERUNITINFO subscription successful.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Subscribe for the LASERUNITINFO event. The device sends this event if the status of one of the laser units changes. E.g. the shutter is opened or closed

This API function causes data to be sent to and received from the device.

The following functions can be used to access the event data received with a LASERUNITINFO event. The functions should only be called in the event callback context of a LASERUNITINFO event.

`REAPI_GetNumberOfLaserUnits()`

`REAPI_GetLaserUnitId()`

`REAPI_IsExtInterlockOpen()`

`REAPI_IsLaserShutterOpen()`

`REAPI_IsOverTempError()` (replaces since version 3.2 the old Function `REAPI_IsOverTempWarning()`)

9.4.1.1 Unsubscribe the Laser Unit Info Event

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeLaserUnitInfo(
               TConnectionId connection );
```

Parameters:

`connection` Connection id returned by REAPI_Connect() or the corresponding connection call-back

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - LASERUNITINFO unsubscription successful.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Cancel a subscription for the LASERUNITINFO event. The device sends this event if the status of one of the laser units changes.

This API function causes data to be sent to and received from the device.

9.4.1.2 Get number of Laser Units

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetNumberOfLaserUnits(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

response
Handle of the event object of the LASERUNITINFO event
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Number of Laser Units present in CL-Device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of a LASERUNIT-INFO event.

9.4.1.3 Get Laser Unit Identifier of a Laser Unit

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetLaserUnitId(
               TResponseHandle response,
               int32_t laserunitID,
               TErrorCode* error );
```

Parameters:

response
Handle of the event object of the LASERUNITINFO event
laserunitID
Laser unit index within range [1 .. NumberOfLaserUnits]
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	ID of laser unit

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of a LASERUNIT-INFO event.

9.4.1.4 Get information about the external interlock status

Syntax:

```
REAPI_DLL bool_t
   REAPI_IsExtInterlockOpen(
               TResponseHandle response,
               int32_t laserunitID,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the event object of the LASERUNITINFO event
`laserunitID`
Laser unit index within range [1 .. NumberOfLaserUnits]
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	Returns 1 for an open external interlock. That means marking can not be started until the interlock is closed

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of a LASERUNIT-INFO event.

9.4.1.5 Get information about the shutter status

Syntax:

```
REAPI_DLL bool_t
   REAPI_IsLaserShutterOpen(
               TResponseHandle response,
               int32_t laserunitID,
               TErrorCode* error );
```

Parameters:

response Handle of the event object of the LASERUNITINFO event **laserunitID** Laser unit index within range [1 .. NumberOfLaserUnits] **error** TErrorCode* used to return ERRORCODE_OK regarding a successful call. ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

bool_t	Returns 1 for an open shutter. That means the laser optical path is ready for marking.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of a LASERUNIT-INFO event.

9.4.1.6 Get information about the pilot laser

Syntax:

```
REAPI_DLL bool_t
   REAPI_IsPilotLaserActive(
               TResponseHandle response,
               int32_t laserunitID,
               TErrorCode* error );
```

Parameters:

response Handle of the event object of the LASERUNITINFO event **laserunitID** Laser unit index within range [1 .. NumberOfLaserUnits] **error** TErrorCode* used to return ERRORCODE_OK regarding a successful call. ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

bool_t	Returns 1 for an active pilot laser.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of a LASERUNIT-INFO event.

9.4.1.7 Get information about the laser tube temperature error state

Syntax:

```
REAPI_DLL bool_t
   REAPI_IsOverTempError(
               TResponseHandle response,
               int32_t laserunitID,
               TErrorCode* error );
```

Parameters:

`response` Handle of the event object of the LASERUNITINFO event **`laserunitID`** Laser unit index within range [1 .. NumberOfLaserUnits] **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	Returns 1 if the laser tube is in over temperature error state and can not be used until it is cooled down.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of a LASERUNIT-INFO event.

This function replaces since Version 3.2 the older and now obsolete function `REAPI_IsOverTempWarning()`.

9.4.2 Set laser device properties

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SetDeviceProperty(
               TConnectionId connection,
               const TDevicePropertyID propertyid,
               const TDevicePropertyIDValue valueid );
```

Parameters:

`connection`
Connection id returned by REAPI_connect or the corresponding connection call-back
`propertyid`
Identifier of the property to set. See remarks for more information.
`valueid`
Value of the property to set. See remarks for more information.

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - time zone set successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

In actual version the function supports only a few device properties specified by identifier and values with special meaning.

```
TDevicePropertyID   DEVICE_PROPERTY_LU_SHUTTER 1
TDevicePropertyIDValue
   DEVICE_PROPERTY_LU_VALUE_SHUTTER_OPEN (1)
   DEVICE_PROPERTY_LU_VALUE_SHUTTER_CLOSED (2)

TDevicePropertyID   DEVICE_PROPERTY_LU_PILOTLASER 2
TDevicePropertyIDValue
   DEVICE_PROPERTY_LU_VALUE_SHUTTER_PILOTLASER_ON (3)
   DEVICE_PROPERTY_LU_VALUE_SHUTTER_PILOTLASER_OFF (4)
```

This supplementary API function causes data to be sent to and received from the device.

10 Device I/O Configuration

10.1 Content of Chapter

10.2 Getting the current I/O configuration of the device

The current I/O configuration setting can also be manually queried from a device. One query resolves all I/O configuration specific information for one device. In the response callback function, the user has to read the specific information with the following getter functions.

The same information can be obtained automatically, whenever there is some change on the I/O configuration. In this case the corresponding event has to be subscribed, preferably when the connection is established.

10.2.1 Manual query of information

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetIOConfiguration(
               TConnectionId connection );
```

Parameters:

connection Connection identifier returned by REAPI_connect() or the corresponding connection callback

Returns:

TResponseHandle	ResponseHandle in synchronous mode to retrieve response elements, ERRORCODE_OK in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

The getter function REAPI_GetIOConfigurationFilename() below should be called in the response callback context of this call.

10.2.2 SubscribeIOConfigurationSet

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SubscribeIOConfigurationSet(
        TConnectionId connection,
        TJobId job
);
```

Parameters:

`connection`
Connection identifier returned by REAPI_connect() or the corresponding connection callback
`job`
Job identifier, 1 for first job
Note: All versions of device firmware up to current version supports only one job

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - IOCONFIGURATIONSET subscription successful.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Subscribe for the IOCONFIGURATIONSET event. The device sends this event if the I/O configuration of the specified job is changed by any client.

This API function causes data to be sent to and received from the device.

The getter function REAPI_GetIOConfigurationFilename() below should be called in the response callback context of this call.

10.2.3 UnsubscribeIOConfigurationSet

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_UnsubscribeIOConfigurationSet(
               TConnectionId connection,
               TJobId job );
```

Parameters:

`connection` Connection identifier returned by REAPI_connect() or the corresponding connection callback **`job`** Job identifier, 1 for first job *Note: All versions of device firmware up to current version supports only one job*

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - IOCONFIGURATIONSET unsubscription successful.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Cancel a subscription for the IOCONFIGURATIONSET event. The device sends this event if the I/O configuration of the specified job is changed by any client.

This API function causes data to be sent to and received from the device.

10.2.3.1 Get current I/O configuration filename

Syntax:

```
REAPI_DLL const char*
   REAPI_GetIOConfigurationFilename(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

response Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED **error** TErrorCode* used to return ERRORCODE_OK regarding a successful call. ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to REAPI_GetIOConfiguration() or the event callback of the event IOCON-FIGURATIONSET.

10.2.4 Setting I/O configuration of the device

Syntax:

```
TResponseHandle REAPI
   REAPI_SetIOConfiguration(
               TConnectionId connection,
               TJobId job,
               const char* filename );
```

Parameters:

`connection`
Connection identifier returned by REAPI_connect() or the corresponding connection callback
`job`
Job identifier, 1 for first job
Note: All versions of device firmware up to current version supports only one job
`filename`
name of the file that will be used as I/O configuration setting

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - job started successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Change current I/O configuration settings of the device.

This API function causes data to be sent to and received from the device.

10.3 Getting of product sensor levels

The current I/O configuration setting can also be manually queried from a device. One query resolves all I/O configuration specific information for one device. In the response callback function, the user has to read the specific information with the following getter functions.

The same information can be obtained automatically, whenever there is some change on the I/O configuration. In this case the corresponding event has to be subscribed, preferably when the connection is established.

10.3.1 Manual query of information of product sensor inputs

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetProductSensorLevel(
               TConnectionId connection );
```

Parameters:

connection Connection identifier returned by REAPI_connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

The getter function REAPI_GetNumberOfProductSensors() and REAPI_GetProductSensorLevelValue() below should be called in the response callback context of this call.

10.3.1.1 Get number of product sensors

Syntax:

```
REAPI_DLL const char*
   REAPI_GetNumberOfProductSensors(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetIOConfiguration()` or the event callback of the event IOCON-FIGURATIONSET.

10.3.1.2 Get current level of product sensor

Syntax:

```
REAPI_DLL const char*
   REAPI_GetProductSensorLevelValue(
                TResponseHandle response,
                TErrorCode* error );
```

Parameters:

response Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED **error** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetIOConfiguration()` or the event callback of the event IOCON-FIGURATIONSET.

10.4 Getting digital I/O input levels

The current I/O configuration setting can also be manually queried from a device. One query resolves all I/O configuration specific information for one device. In the response callback function, the user has to read the specific information with the following getter functions.

The same information can be obtained automatically, whenever there is some change on the I/O configuration. In this case the corresponding event has to be subscribed, preferably when the connection is established.

10.4.1 Manual query of information of digital inputs

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetIOInputLevel(
               TConnectionId connection );
```

Parameters:

`connection` Connection identifier returned by REAPI_connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

The getter function REAPI_GetNumberOfInputs() and REAPI_GetIOInputLevelValue() below should be called in the response callback context of this call.

10.4.1.1 Get number of digital I/O inputs

Syntax:

```
REAPI_DLL const char*
   REAPI_GetNumberOfInputs(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED
`error` `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetIOConfiguration()` or the event callback of the event IOCON-FIGURATIONSET.

10.4.1.2 Get current level of digital I/O input

Syntax:

```
REAPI_DLL const char*
   REAPI_GetIOInputLevelValue(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetIOConfiguration()` or the event callback of the event IOCON-FIGURATIONSET.

10.5 Getting and setting of digital I/O output levels

The current I/O configuration setting can also be manually queried from a device. One query resolves all I/O configuration specific information for one device. In the response callback function, the user has to read the specific information with the following getter functions.

The same information can be obtained automatically, whenever there is some change on the I/O configuration. In this case the corresponding event has to be subscribed, preferably when the connection is established.

10.5.1 Manual query of information of digital outputs

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetIOOutputLevel(
               TConnectionId connection );
```

Parameters:

`connection` Connection identifier returned by REAPI_connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

The getter function REAPI_GetNumberOfOutputs() and REAPI_GetIOOutputLevelValue() below should be called in the response callback context of this call.

10.5.1.1 Get number of digital I/O outputs

Syntax:

```
REAPI_DLL const char*
   REAPI_GetNumberOfOutputs(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetIOConfiguration()` or the event callback of the event IOCON-FIGURATIONSET.

10.5.1.2 Get current level of digital I/O output

Syntax:

```
REAPI_DLL const char*
   REAPI_GetIOOutputLevelValue(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetIOConfiguration() or its callback function or the event callback of IOCONFIGURATIONCHANGED **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetIOConfiguration()` or the event callback of the event IOCON-FIGURATIONSET.

10.5.2 *Setting of digital outputs*

Syntax:

```
TResponseHandle REAPI
   REAPI_SetIOOutputLevel(
               TConnectionId connection,
               uint32_t outMask,
               uint32_t outLevels );
```

Parameters:

`connection` Connection identifier returned by REAPI_connect() or the corresponding connection callback **`outMask`** Job identifier, 1 for first job *Note: All versions of device firmware up to current version supports only one job* **`outLevels`** name of the file that will be used as I/O configuration setting

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - job started successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Change current I/O configuration settings of the device.

This API function causes data to be sent to and received from the device.

11 Device settings Common

11.1 Content of Chapter

11.2 Getting information about the device

Query device information from a device. One query resolves device information for one device. In the response callback function, the user has to read the specific information with the following getter functions.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetDeviceInfo(
               TConnectionId connection );
```

Parameters:

`connection`	Connection id returned by REAPI_Connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

All getter functions below should be called in the response callback context of this call.

The following getter functions can be used to access the response data received with this function. The functions should only be called in the response callback context.

`REAPI_GetFirmware()`

`REAPI_GetSerialNumber()`

`REAPI_GetArticleNumber()`

`REAPI_GetFPGAVersion()`

`REAPI_GetDeviceType()`

11.3 Getting production data

Query product information from a device. One query resolves production data like print counter, operation hour, etc. In the response callback function, the user has to read the specific information with the following getter functions.

This command is available with a Firmware-Version ≥ 3.50.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetProductionData (
               TConnectionId connection );
```

Parameters:

`connection`	Connection id returned by REAPI_Connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This API function causes data to be sent to and received from the device.

All getter functions below should be called in the response callback context of this call.

The following getter functions can be used to access the response data received with this function. The functions should only be called in the response callback context.

`REAPI_GetSerialNumber()`

`REAPI_GetArticleNumber()`

`REAPI_GetSytemPrintCounter()`

11.3.1 *Getter functions for information*

11.3.1.1 Get firmware version

Syntax:

```
REAPI_DLL const char*
   REAPI_GetFirmware(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetDeviceInfo() or its callback function
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Firmware version description

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetDeviceInfo()`.

11.3.1.2 Get serial number

Syntax:

```
REAPI_DLL const char*
   REAPI_GetSerialNumber(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetDeviceInfo() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	serial number of the device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetDeviceInfo() and REAPI_GetProductionData().`

11.3.1.3 Get article number

Syntax:

```
REAPI_DLL const char*
   REAPI_GetArticleNumber(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetDeviceInfo() or its callback function
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	article number of the device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetDeviceInfo() and REAPI_GetProductionData().`

11.3.1.4 Get FPGA version

Syntax:

```
REAPI_DLL const char*
   REAPI_GetFPGAVersion(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetDeviceInfo() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Returns the FPGA version string

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetDeviceInfo()`.

11.3.1.5 Get device type identifier

Syntax:

```
REAPI_DLL const char*
   REAPI_GetDeviceType(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetDeviceInfo() or its callback function
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	device type identification string

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetDeviceInfo()`.

11.3.1.6 Get number of prints since begin operation

Syntax:

```
REAPI_DLL const
   uint64_t REAPI_GetSystemPrintCounter (
               TResponseHandle  response,
               int32_t printheadId,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetDeviceInfo() or its callback function
`printheadId`
Identifier of the printhead within range [1..4]. Depending on type of device, not all identifier printheads are available. In this case none print is returned.
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

uint64	number of prints for the specified printhead since begin of operation.

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a call to `REAPI_GetProductionData()`.

11.4 Query time zone information

Query currently set time zone information from a device. One query resolves all time zone information for one device. In the response callback function, the user has to read the specific information with the following getter functions. The time zone information is based on the tzdata project. Valid settings can be found on http://en.wikipedia.org/wiki/List_of_tz_database_time_zones

11.4.1 Getting information about the time zone

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetTimezone(
               TConnectionId connection );
```

Parameters:

`connection`	Connection id returned by REAPI_connect() or the corresponding connection callback

Returns:

`TResponseHandle`	Response handle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

The following getter functions can be used to access the response data received with this function. The functions should only be called in the response callback context.

`REAPI_GetDateTimeRegion()`

`REAPI_GetDateTimeZone()`

`REAPI_GetDateTimeDSTChangeover()`

11.4.1.1 Get time zone region

Syntax:

```
REAPI_DLL const char*
   REAPI_GetDateTimeRegion(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetTimezone() or its callback function
error
TErrorCode* used to return ERRORCODE_OK regarding a successful call.
ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid
ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command
ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

const char*	Current time zone region set

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a REAPI_GetTimezone() call.

11.4.1.2 Get time zone city / country

Syntax:

```
REAPI_DLL const char*
   REAPI_GetDateTimeZone(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetTimezone() or its callback function
error
TErrorCode* used to return ERRORCODE_OK regarding a successful call.
ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid
ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command
ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

const char*	current time zone city

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetTimezone()` call.

11.4.1.3 Get time zone daylight saving time status

Syntax:

```
REAPI_DLL bool_t
   REAPI_GetDateTimeDSTChangeover(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetTimezone() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool_t	True if daylight saving time is used, else false

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetTimezone()` call.

11.4.2 Changing the time zone setting of the device

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SetTimezone(
               TConnectionId connection,
               const char* region,
               const char* timezone,
               bool_t dst );
```

Parameters:

`connection`
Connection id returned by REAPI_connect or the corresponding connection call-back
`region`
Time zone region
`timezone`
Time zone city / country
`dst`
Activate automatic dst (daylight saving time) detection

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - time zone set successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Change current time zone settings in device.

This supplementary API function causes data to be sent to and received from the device.

11.5 Query date and time setting

Query clock and date information from a device. One query resolves all clock information for one device. In the response callback function, the user has to read the specific information with the following getter functions.

11.5.1 Get information about the date and time setting of the device

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetDateTime(
               TConnectionId connection );
```

Parameters:

connection Connection identifier returned by REAPI_Connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

The following getter functions can be used to access the response data received with this function. The functions should only be called in the response callback context.

```
REAPI_GetDateTimeYear()
REAPI_GetDateTimeMonth()
REAPI_GetDateTimeDay()
REAPI_GetDateTimeHour()
REAPI_GetDateTimeMinute()
REAPI_GetDateTimeSecond()
```

11.5.1.1 Get date year setting

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetDateTimeYear(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetDateTime() or its callback function
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Current year setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetDateTime()` call.

11.5.1.2 Get date month setting

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetDateTimeMonth(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetDateTime() or its callback function
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Current month setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetDateTime()` call.

11.5.1.3 Get date day of month setting

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetDateTimeDay(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetDateTime() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Current day of month setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetDateTime()` call.

11.5.1.4 Get clock hour setting

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetDateTimeHour(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetDateTime() or its callback function
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Current hour setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetDateTime()` call.

11.5.1.5 Get clock minute setting

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetDateTimeMinute(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetDateTime() or its callback function
error
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Current minute setting

Remark:

This API function causes data to be sent to and received from the device.

The function should only be called in the response callback context of a `REAPI_GetDateTime()` call.

11.5.1.6 Get clock second setting

Syntax:

```
REAPI_DLL int32_t
   REAPI_GetDateTimeSecond(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetDateTime() or its callback function
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

int32_t	Current second setting

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetDateTime()` call.

11.5.2 *Changing date and time of device*

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_SetDateTime(
               TConnectionId connection,
               int32_t year,
               int32_t month,
               int32_t day,
               int32_t hour,
               int32_t minute,
               int32_t second );
```

Parameters:

`connection`
Connection identifier returned by REAPI_connect() or the corresponding connection callback
`year`
year to be set
`month`
month to be set
`day`
day of month to be set
`hour`
hour of the day
`minute`
minute of the hour
`second`
second to be set

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - date set successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Changes current time settings of the internal clock in the device.

This supplementary API function causes data to be sent to and received from the device.

11.6 Network and NTP setting

Gets or sets information about network configuration and NTP server of a device.

With these functions it is possible to get or set network configuration like TCP address, gateway, etc, or information about NTP server configuration, like TCP address of NTP server, etc.

With the corresponding getter functions, the user has to read the specific information from response.

11.6.1 Getting information about the network configuration of the device

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetNetworkConfig(
               TConnectionId connection );
```

Parameters:

`connection` Connection id returned by REAPI_Connect() or the corresponding connection callback

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to retrieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This supplementary API function causes data to be sent to and received from the device.

All getter functions below should be called in the response callback context of this call.

11.6.1.1 Get IP address

Syntax:

```
REAPI_DLL const char*
   REAPI_GetIPAddress(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetNetworkConfig() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current IP Address of the device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetNetworkConfig()` call.

11.6.1.2 Get subnet mask

Syntax:

```
REAPI_DLL const char*
   REAPI_GetSubnetmask(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetNetworkConfig() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current subnet mask of the device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a REAPI_GetNetworkConfig() call.

11.6.1.3 Get gateway IP address

Syntax:

```
REAPI_DLL const char*
   REAPI_GetGateway(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object previously returned by REAPI_GetNetworkConfig() or its callback function **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	Current gateway IP address setting of the device

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetNetworkConfig()` call.

11.6.1.4 Get DHCP configuration

Syntax:

```
REAPI_DLL bool
   REAPI_IsDhcpActive(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object previously returned by REAPI_GetNetworkConfig() or its callback function
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.

`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

bool	Returns true if the device is configured to use dhcp to set its network address

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetNetworkConfig()` call.

11.6.2 *Changing network configuration of the device*

Syntax:

```
TResponseHandle REAPI
   REAPI_SetNetworkConfig(
               TConnectionId connection,
               const char* ipaddress,
               const char* netmask,
               const char* gateway,
               bool_t useDHCP );
```

Parameters:

`connection`
Connection id returned by REAPI_Connect() or the corresponding connection call-back
`ipaddress`
String containing the new IP address in format:
“xxx.xxx.xxx.xxx” e.g. “192.168.0.2”
`netmask`
String containing the new subnet mask in format:
“xxx.xxx.xxx.xxx” e.g. “255.255.255.0”
`ipaddress`
String containing the new gateway IP address in format:
“xxx.xxx.xxx.xxx” e.g. “192.168.0.1”
`useDHCP`
Set to true to force the device the acquire its network settings from a DHCP server

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - job started successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Changes current network configuration settings of the device.

This supplementary API function causes data to be sent to and received from the device.

This function call causes a disconnect from the device. The client application needs to reconnect to the new device address afterwards.

11.6.3 Getting information about the NTP configuration of the device

This command is available with a Firmware-Version ≥ 3.40.

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_GetNTPConfig(
               TConnectionId connection );
```

Parameters:

`connection` Connection id returned by REAPI_Connect() or the corresponding connection call-back

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to re-trieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This API function causes data to be sent to and received from the device.

All getter functions below should be called in the response callback context of this call.

11.6.3.1 Get IP address of NTP server

Syntax:

```
REAPI_DLL const char*
   REAPI_GetNTPServerAddress(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetNTP() or its call-back function
error
TErrorCode* used to return ERRORCODE_OK regarding a successful call.
ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid
ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command
ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

const char*	Current IP address used as NTP server

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetNTPConfig()` call.

11.6.3.2 Is NTP server fixed by DHCP

Syntax:

```
REAPI_DLL const bool_t
   REAPIIsFixedNTPServer(
               TResponseHandle  response,
               TErrorCode* error );
```

Parameters:

response
Handle of the response object previously returned by REAPI_GetNTPConfig() or its callback function
error
TErrorCode* used to return ERRORCODE_OK regarding a successful call.
ERRORCODE_INVALID_RESPONSE_HANDLE - response handle was not valid
ERRORCODE_COMMAND_ERROR - the response handle can not be used for that command
ERRORCODE_INVALID_DEVICE_RESPONSE - the returned value for that command is not valid

Returns:

bool_t	Returns wether the NTP server is fixed set by DHCP

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the response callback context of a `REAPI_GetNTPConfig()` call.

11.6.4 *Changing NTP server configuration of the device*

Syntax:

```
TResponseHandle REAPI
   REAPI_SetNTPConfig(
               TConnectionId connection,
               const char* ntpserver );
```

Parameters:

connection Connection id returned by REAPI_Connect() or the corresponding connection call-back **ntpserver** String containing the IP address of the NTP server to use in format: "xxx.xxx.xxx.xxx" e.g. "192.168.0.2"

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid
`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - job started successfully.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Changes current NTP server configuration of the device.

This API function causes data to be sent to and received from the device.

This function call causes a disconnect from the device. The client application needs to reconnect to the new device address afterwards.

12 Generic event protocol

12.1 Content of Chapter

12.2 Introduction

Das Generic event protocol dient zur einfachen Kommunikation zwischen zwei Clients. Das bedeutet, die Drucksteuerung des Kennzeichnungssystems leitet die entsprechenden Nachrichten nur weiter, die Kommunikation von Client zu Client ist für das Drucksystem also völlig transparent.

Für diese Kommunikation gibt es zwei Funktionalitäten:

- Senden einer Nachricht mit TriggerGenericEvent
- Empfangen einer Nachricht als Event Generic Event.

Beide Clients vereinbaren für diese Kommunikaton gemeinsame eindeutige Identifier aus, die jeweils ein Kommando darstellen. Zum Beispiel:

```
Client1:
SubscribeGenericEvent( connection, "cmdDoSomething" );
   Client 2:
   SubscribeGenericEvent( connection, "cmdSomethingDone" );


Client1:
TriggerGenericEvent( connection, "cmdDoSomething", "switch on coffee
machine");

   Client 2:
   -> Event GenericEvent mit Parametern "cmdDoSomething", "switch on coffee
machine"
   TriggerGenericEvent( connection, "cmdSomethingDone", "coffee machine is
working");

Client1:
-> Event GenericEvent mit Parametern "cmdSomethingDone", "coffee machine is
working"
```

Der Identifier sollte eindeutig sein. Es wird dringend empfohlen, nur ASCII-Zeichen zu verwenden. Ein Semikolon ';' wird als Trennzeichen bei mehreren Identifier verwendet und darf nicht im Identifier enthalten sein!

Im Parameter rawData können beliebige Daten als Zeichenkette übergeben werden. Dies können literale Texte, Xml-Daten, JSON oder auch binäre Daten als Base64 sein.

12.3 Sending generic events

Syntax:

```
REAPI_DLL TResponseHandle
   REAPI_TriggerGenericEvent(
               TConnectionId connection,
               const char* identifier,
               const char* rawdata );
```

Parameters:

`connection` Connection id returned by REAPI_Connect() or the corresponding connection call-back
`identifier` Specifies common used identifier as eg. Command. This parameter must not con-tain a semicolon ';'!
`rawdata` Arbitary used data as string. Can be literal text as well as Xml-Data, Json or base64 coded binary Data

Returns:

`TResponseHandle`	ResponseHandle in synchronous mode to re-trieve response elements, `ERRORCODE_OK` in asynchronous mode

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

This API function causes data to be sent to and received from the device.

All getter functions below should be called in the response callback context of this call.

12.4 Receiving generic events

12.4.1 Subscribe and unsubscribe

Syntax:

```
REAPI_DLL TResponseHandle REAPI_SubscribeGenericEvent (
        TConnectionId connection,
        const char* identifier );
```

```
REAPI_DLL TResponseHandle REAPI_UnsubscribeGenericEvent (
        TConnectionId connection,
        const char* identifier );
```

Parameters:

`connection`
Connection identifier returned by REAPI_connect() or the corresponding connec-tion callback
`identifier`
Specifies common used identifier as eg. Command. This parameter must not con-tain a semicolon ';'!

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection id is not valid

`ERRORCODE_OK`	In asynchronous mode - asynchronous call processed successfully, result will be returned by callback. In synchronous mode - IOCONFIGURA-TIONSET subscription successful.

For further information for response handler `TResponseHandler` see chapter 3.5 Basic API types, page 9).

Remark:

Subscribe for the GENERICEVENT event. The device sends this event if the I/O configuration of the specified job is changed by any client.

This API function causes data to be sent to and received from the device.

12.4.2 Get event data

12.4.2.1 Get identifier

Syntax:

```
REAPI_DLL const char*
   REAPI_GetIdentifier(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response` Handle of the response object given in the event callback of GENERICEVENT **`error`** `TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	identifier of the generic event

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of the event GENERICEVENT.

12.4.2.2 Get raw data

Syntax:

```
REAPI_DLL const char*
   REAPI_GetRawData(
               TResponseHandle response,
               TErrorCode* error );
```

Parameters:

`response`
Handle of the response object given in the event callback of GENERICEVENT
`error`
`TErrorCode*` used to return `ERRORCODE_OK` regarding a successful call.
`ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid
`ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command
`ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid

Returns:

const char*	raw data of the generic event as string

Remark:

This function does not cause any data to be sent to or received from the device.

The function should only be called in the event callback context of the event GENERICEVENT.

13 File Transfer Commands

13.1 Content of Chapter

13.2 Get list of files or directories from device

Query files or directories over REAPI protocol. This function uses the directory names in the device share ‘rea-jet’.

The command requests a list of directory elements from in the given path. In the response callback function, the user has to read the specific information with appropriated getter functions.

A simple example to get the job file from device could be:

```
ReaPi.ResponseHandle response = ReaPi.ListDirectory( _connectionID,
                                                     "jobs" );
if ((int)response >= 0)
  _pendingQueryJobFiles = (int)response;
```

```
int err = 0;

int count = REAPI_ GetNumberOfDirectoryElements( response, out err );
for (int n = 1; n <= count; n++)
{
   string name = REAPI_GetNameOfDirectoryElement( response, n, out err );
   bool isfile = REAPI_IsDirectoryElementAFile( response, n, out err );

   //_listDirectoryElements.Add( name, isfile );
}
```

13.2.1 *Getting list of files from device*

This command requests a list of directory elements from device.

Syntax:

```
REAPI_DLL TResponseHandle REAPI_ListDirectory(
               TConnectionId connection,
               const char* path );
```

Parameters:

`connection`	Connection identifier returned by REAPI_Connect() or the corresponding connection callback
`path`	destination path from where to query files and/or directories

Returns:

`ERRORCODE_INVALID_CONNECTION`	Connection identifier is not valid
`TResponseHandle`	the response handle In asynchronous mode only: This handle should be used to identify this command in response callback and use the getter functions to get the result of this function later in response callback. In synchronous mode: the call was was successful and use directly the getter functions to get the result of this function.

For further information for response handle `TResponseHandle` see chapter 3.5 Basic API types, page 9).

13.2.1.1 Command return

With the following getter functions the command return can be retrieved.

Use the response handle of the response callback to get the corresponding command return. The response handle is only valid and should only be used inside the callback function!

For retrieving an event see chapter 3.7.3 Response callback, page 15.

REAPI_GetDirectoryName

Get the name of the directory on device from which the directory elements were generated. This is a repetition of the second parameter of the function call.

```
REAPI_DLL const char* REAPI_GetDirectoryName(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetNumberOfDirectoryElements

Get the number of directory elements generated. Use the number for the following getter functions as index in parameter elementid.

```
REAPI_DLL int32_t REAPI_GetNumberOfDirectoryElements(
               TResponseHandle response,
               TErrorCode* error );
```

REAPI_GetNameOfDirectoryElement
REAPI_IsDirectoryElementAFile

Get information about a single directory element from elementid. Here they are the name of the element and whether the element is a file or not.

```
REAPI_DLL const char* REAPI_GetNameOfDirectoryElement(
               TResponseHandle response,
               int elementid,
               TErrorCode* error );
```

```
REAPI_DLL bool REAPI_IsDirectoryElementAFile(
               TResponseHandle response,
               int elementid,
               TErrorCode* error );
```

`response`	Handle of the response previously returned by REAPI_ListDirectory() by function call or event callback function.
`elementid`	The elementid is an index of the list starting with 1 for the first and number of elements for the last entry.
`error`	`TErrorCode*` used to return `ERRORCODE_OK` regarding a success-ful call. `ERRORCODE_INVALID_RESPONSE_HANDLE` - response handle was not valid. `ERRORCODE_COMMAND_ERROR` - the response handle can not be used for that command `ERRORCODE_INVALID_DEVICE_RESPONSE` - the returned value for that command is not valid.

14 Appendix A: Error Codes

14.1 Error codes from functions of the REA-PI Library

Error code (Domain/Code)	**Code**	**Description**
ERRORCODE_OK	0	no error
ERRORCODE_RUNTIME	-1	runtime error
ERRORCODE_INVALID_CONNECTION	-2	function was called with a not valid connection
ERRORCODE_TIMEOUT	-3	function call was timed out
ERRORCODE_INVALID_LABEL_HANDLE	-4	function was called with a not valid label handle
ERRORCODE_UNKNOWN	-5	unspecified error occurred
ERRORCODE_COMMAND_ERROR	-6	command error
ERRORCODE_INVALID_RESPONSE_HANDLE	-7	function was called with a not valid response handle
ERRORCODE_INVALID_DEVICE_RESPONSE	-8	device send a not valid response
ERRORCODE_CANNOT_CONNECT_TO_DEVICE	-9	connection failed, possibly un-reachable device or invalid connect parameters
ERRORCODE_TRANSMISSION_ERROR	-10	network error, transmission of data failed
ERRORCODE_SSL_HANDSHAKE_ERROR	-11	ssl handshake failed, possible invalid parameter for ssl
ERRORCODE_INVALID_PARAMETER	-12	function was called with invalid parameter, possible null pointer for strings are not allowed
ERRORCODE_NOT_IMPLEMENTED	-10000	invalid or unknown function was called, possible called function is not supported or not implemented

14.2 Errorcodes from print control

Error code (Domain/Code)	**Domain / Code**	**Description**
CODE_OK	ANY/0	no error
CODE_FAILED	ANY/2	failed for unknown reason
CODE_UNDEFINED	ANY/3	unknown error
CODE_INVALID_PARAMS	ANY/4	invalid parameters
CODE_FATAL	ANY/5	fatal error, device will restart
CODE_MEMORY	ANY/6	a memory exception occurred
DOMAIN_GENERIC		
CODE_GENERIC	99/99	Generic error occurred
DOMAIN_SYSTEM_MISC		
CODE_FILE_NOT_FOUND	500/10	file not found
CODE_FILE_OPEN_FAILED	500/11	cannot open file
CODE_SCRIPT_EXECUTION_FAILED	500/20	execution of script or sub-process failed
CODE_SCRIPT_EXECUTION_DENIED	500/21	execution of script or sub-process denied
CODE_SCRIPT_FIRMWAREUPDATE_FAILED	500/30	execute script for firmware update failed
CODE_SCRIPT_FIRMWAREUPDATE_DENIED	500/31	execute script for firmware update denied
CODE_SCRIPT_FIRMWAREUPDATE_WHILEPRINTING	500/32	execute script for firmware update while printing
CODE_SCRIPT_FIRMWAREUPDATE_IN_PROGRESS	500/33	process of firmware update is in progress
CODE_SCRIPT_FACTORYDEFAULT_FAILED	500/35	execute script for factory default failed
CODE_SCRIPT_FACTORYDEFAULT_DENIED	500/36	execute script for factory default denied
CODE_SCRIPT_FACTORYDEFAULT_WHILEPRINTING	500/37	execute script for factory default while printing
CODE_INVALID_NETWORKCONFIG	500/40	invalid network configuration to set
CODE_FONTUPDATE_FAILURE	500/80	Update fonts failed
CODE_FREETYPE_FAILED	500/81	Resource freetype not available or failed
CODE_INVALID_FONT	500/82	Font file is not a valid freetype usable font
CODE_XML_INVALID_DIOVERSION	500/90	Xml-IOConfiguration version not supported
DOMAIN_ENGINE_LABEL		
CODE_INVALID_TAG	101/10	Invalid label tag
CODE_MISSING_RESOURCE	101/20	Label used resource missing
CODE_MISSING_FONT	101/30	Label used font missing
CODE_INVALID_LABEL_OBJECT_WIDTH	101/31	Invalid width of label object
CODE_INVALID_LABEL_OBJECT_HEIGHT	101/32	Invalid height of label object
CODE_LABELOBJECT_NOTFOUND	101/40	Label object not found / available in label

CODE_OBJECTCONTENT_NOTFOUND	101/41	Object content not found / available in label object
CODE_PERMISSION_DENIED	101/90	access to label or label tag not granted
CODE_CONTENT_UNCHANGED	101/96	object content set without any changes
CODE_REQUEST_STOPPRINT	101/99	object content set produces a stop condition/request for printing
DOMAIN_ENGINE_PRINTCONTROL		
CODE_JOB_STILL_RUNNING	102/10	Job is still running
CODE_JOB_NOT_RUNNING	102/11	Job is not running
CODE_NO_JOB_ASSIGNED	102/12	There is no active assigned job
CODE_NO_GROUP_ASSIGNED	102/13	Job contains no group
CODE_JOB_NOT_EXISTS	102/14	Job does not exist
CODE_NO_LABEL_ASSIGNED	102/15	Group contains no label
CODE_GROUP_NOT_ASSIGNED	102/16	Tried action on not assigned group
CODE_INVALID_INKINFO	102/17	Invalid ink info
CODE_HEAD_NOT_READY	102/18	Printhead not ready
CODE_SHAFTENCODER_NOT_SET	102/19	Shaft encoder not set
CODE_ENTITY_ACTIVE	102/21	A entity of the printing system is active and cannot be (re-)parameterized
CODE_RENDER_FAILED	102/22	Rendering failed, no label-image from renderer available
CODE_JOB_PURGING	102/23	Purging already running
CODE_PURGING_HEAD_NOT_FOUND	102/24	Please assign a job which includes the printhead to be purged
CODE_PURGING_DUPLICATE_HEAD_FOUND	102/25	Can not purge with double existing printheads/cartridges
CODE_PURGING_PURGE_WITHOUT_HEADS	102/26	Can not purge without any printhead / cartridge
CODE_PURGING_JOB_IS_RUNNING	102/27	Can not purge with a printhead included in a running job
CODE_PURGING_CARTRIDGE_NOT_CONNECTED	102/28	Please insert a cartridge or lock the inserted cartridge
CODE_GROUP_NOT_EXISTS	102/29	Group does not exist
CODE_GROUP_NOT_ACTIVE	102/30	Group must not be active to perform
CODE_GROUP_ACTIVE	102/31	Group have to be active to perform
CODE_GROUP_NO_IMAGEBUFFER	102/32	No buffer to store the image data for group
CODE_GROUP_NO_PRINTHEAD	102/33	Group have to contain at least one printhead
CODE_GROUP_HARDWARE_NOT_SUPPORTED	102/34	Desired hardware not supported
CODE_GROUP_DUPLICATE_PRINTHEAD	102/35	Group must not contain printheads with identical IDs
CODE_GROUP_PURGING	102/36	Purging in progress. Job can not be started!
CODE_EXTERNAL_STOP_CONDITION	102/37	Group/job can not be started within external stopped condtion
CODE_PRINTIMAGE_PRINTCOUNT_EXCEEDED	102/38	No buffer data to print reasoned by printcount

		exceeded
CODE_JOB_DUPLICATE_GROUP	102/40	Job must not contain groups with not unique name
CODE_SPITTING_INK_TYPE	102/50	Type of ink for spitting doesn't match with settings
CODE_INK_INVALID_ERROR_WARNING_LEVEL	102/51	levels for ink warnings and ink error doesn't match together
CODE_SPEEDSENSORPULSELOST	102/70	Print speed too fast, max print pulses exceeded
CODE_SPEEDSENSORTOOFAST	102/71	Print speed too fast, speed sensor pulses exceeded for resolution
CODE_SPEEDSENSORSLICESLOST	102/72	Print speed too fast, speed sensor pulses for generating slices exceeded
CODE_GROUP_LASERPEN_NOT_FOUND	102/80	Needed set of laser pen parameter in group not found
CODE_GROUP_PARAMETER_ERROR	102/81	Erroneous parameters
CODE_CHIPREQUIRED	102/90	Required cartridge chip not present
DOMAIN_ENGINE_PARSER		
CODE_PARSE_EXCEPTION	103/10	Common error in parsing xml
CODE_XML_SYNTAX	103/11	Error in xml syntax
CODE_XML_INVALID_ROOT	103/12	xml-document doesn't contain valid root node
CODE_XML_INVALID_JOBFILE	103/14	file of xml-document for job or installation is invalid (or not found)
CODE_XML_INVALID_JOB	103/15	xml-document invalid or missing job node
CODE_XML_INVALID_JOBVERSION	103/16	Job version is not supported
CODE_XML_INVALID_INSTALLATIONVERSION	103/17	Installation version is not supported
CODE_XML_LABELFILE_NOT_FOUND	103/19	Label could not be found
CODE_XML_INVALID_LABEL	103/20	xml-document invalid or missing label node
CODE_XML_INVALID_LABELVERSION	103/21	xml-document Label version not supported
CODE_LOGIC_EXCEPTION	103/40	Logical error in xml-tag
CODE_TAG_LABEL	103/41	Error in xml-tag label
CODE_TAG_LAYOUT	103/42	Error in xml-tag layout
CODE_TAG_OBJECT	103/44	Error in xml-tag object
CODE_TAG_RENDERER	103/45	Error in xml-tag renderer
CODE_RENDERER_NOTSUPPORTED	103/46	Xml-tag renderer type (of label) not supported
CODE_TAG_CONTENT	103/47	Error in xml-tag content (of label object)
CODE_CONTENT_NOTSUPPORTED	103/48	Xml-tag content type (of label object) not supported
CODE_RENDERER_NEEDSCONTENT	103/49	Xml-tag renderer type needs content
CODE_TAG_FORMAT	103/51	Xml-tag format doesn't match specification
CODE_TAG_UNRESOLVED_REFERENCE	103/54	Referenced content can not be found
CODE_TAG_INSTALLATION	103/60	Error in xml-tag installation
CODE_FEATURE_EXCEPTION	103/70	Unknown feature
DOMAIN_ENGINE_RENDERER		
CODE_CANNOT_RENDER	104/10	Rendering print image failed

CODE_RENDERTHREAD_NOT_STARTED	104/11	Cannot create or start render thread
CODE_PURGETHREAD_NOT_STARTED	104/12	Cannot create or start purge thread
CODE_INVALID_IMAGEFACTORY	104/29	Not valid renderer reasoned by invalid image factory
CODE_NO_RENDERER_CREATE	104/30	Cannot create renderer
CODE_NO_TEXTRENDERER_CREATE	104/31	Cannot create renderer for text
CODE_NO_IMAGERENDERER_CREATE	104/35	Cannot create renderer for images
CODE_NO_BARCODERENDERER_CREATE	104/36	Cannot create renderer for barcodes
CODE_NO_GRAPHICRENDERER_CREATE	104/40	Cannot create renderer for graphical objects
CODE_INITIALIZE_FREETYPE	104/50	Cannot initialize freetype library
CODE_INITIALIZE_TBARCODE	104/60	Cannot initialize tbarcode library
CODE_CREATE_BARCODE_FAILED	104/62	Cannot render barcode object
CODE_MISSING_BITMAP_FILE	104/79	Bitmap file could not be found
CODE_INVALID_BITMAP_FORMAT	104/80	Rendering failed due to invalid format of source image
CODE_INVALID_BITMAP_SIZE	104/81	Rendering failed due to invalid size of source image
CODE_PRINTCOUNT_EXPIRED	104/83	Cannot render image because of expired print count
CODE_INVALID_RENDEREVENT	104/84	Invalid count event trigger
CODE_INVALID_SHIFTCODE	104/90	Render failed on invalid shiftcode
CODE_INSUFFICIENT_CODELISTENTRIES	104/110	Code list contains insufficient entries
CODE_INVALID_INCREMENT_ZERO	104/120	Invalid increment, must not be 0, zero
CODE_INVALID_INCREMENT_DIRECTION	104/121	Invalid increment, violate start/end direction
CODE_INVALID_STARTEND_EXCEEDED	104/122	Invalid counter have to be in range of start/end condition
CODE_INVALID_STARTEND_CONDITION	104/123	Invalid start/end condition, must not be equal
DOMAIN_FPGA_MISC		
CODE_LABEL_TOO_LATE	200/10	Label data too late for FPGA
CODE_SHAFTENCODER_TOO_FAST	200/11	Pulses from shaft encoder too fast
CODE_SHAFTENCODER_TOO_UNSTEADY	200/12	Unsteady pulses from shaft encoder
CODE_INVALID_SHAFTENCODER_PARAM	200/20	Invalid shaft encoder parameter
CODE_NO_SUCH_HARDWARE	200/30	Hardware subsystem missing
CODE_NUMBER_LAYERS_EXCEEDED	200/40	Number of layers exceeded / not supported
CODE_INVALID_CONFIGURATION	200/50	Invalid FPGA configuration
CODE_COPY_FPGA_IMAGE	200/60	Failed to copy image to FPGA
CODE_OPERATION_AND_2EDGE	200/70	Logical AND-operation with edges not allowed
CODE_OPERATION_OR_2EDGE	200/71	Logical AND-operation with falling edges not allowed
DOMAIN_HEAD		
CODE_NOTCONNECTED	300/11	Not all printheads, which are referenced from the job,

		are connected
CODE_VERRIEGELUNG_OPEN	300/12	Cartridge is not locked
CODE_PENDRIVER_INIT_FAILED	300/13	Pendriver init failed
CODE_NO_CARTRIDGE	300/14	Printhead has no cartridge
CODE_INK_EMPTY	300/15	Ink empty
CODE_TEMPERATURE_HIGH	300/16	Printhead temperature is too high
DOMAIN_CHIPCARD		
CODE_NO_CHIP	310/10	Cartridge with no chip
CODE_READWRITE_FAILED	310/11	Read write failed
DOMAIN_COMMUNICATION		
CODE_SUBSCRIPTION_FAILED	400/10	Event subscription failed
CODE_XML_INVALID_REAPIVERSION	400/20	Xml-document REA-PI version not supported
CODE_VERSION_ALREADY_SELECTED	400/21	Version already selected
CODE_REAPICOMMAND_INVALID	400/30	REA-PI xml-command is invalid
CODE_REAPICOMMAND_INCOMPLETE	400/31	REA-PI xml-command is missing parameters
DOMAIN_FILESYSTEM		
CODE_ACCESS_DENIED	600/10	Access to file or directory denied
CODE_FILE_LOCKED	600/11	File is locked
CODE_INVALID_PATH	600/12	Invalid path
CODE_INVALID_FILENAME	600/13	Invalid filename
CODE_FILE_ALREADY_EXISTS	600/14	File already exists
CODE_COULDNOTGETFILE	600/15	Could not get file
CODE_COULDNOTWRITEFILE	600/16	Could not create or write file
CODE_TRANSMISSION_FAILED	600/20	Transmission failed
CODE_MD5SUM_ERROR	600/21	Error in MD5 check sum
DOMAIN_LASER_MISC		
CODE_INVALID_CORRECTION_FILE	700/10	Invalid correction file
CODE_NUMBER_LABELOBJECTS_EXCEEDED	700/20	Number of LabelObjects exceeded / not supported
CODE_LABELOBJECT_DOES_NOT_EXIST	700/21	Label object does not exist
CODE_INVALID_LABELOBJECT_LIST	700/22	Invalid list of label ojects
CODE_OVERTEMPERATURE_FAULT	700/30	Overtemperature fault
CODE_SECURITY_LOCK_OPEN	700/31	Security lock open
CODE_LASER_TUBE_NOT_READY	700/32	Laser tube not ready
CODE_SAFETY_CIRCUIT_FAILURE	700/33	Safety circuit failure
CODE_SHUTTER_CONTROL_NOT_POSSIBLE	700/34	Shutter control not possible

Additional to error codes in some cases one or more warnings can occure:

DOMAIN_WARNINGS		
CODE_LABEL_EXCEEDS_PRINT	800/201	Label exceeds the printable height