USB/CAN Interface

Insolated CAN Adaptor

Rev. 1.0
Documentation Rev. 4

Reusch Elektronik

www.reusch-elektronik.de

http://products.reworld.eu/uci.htm

File: uci_10_Manual
Created: 2020-04-05
Changed: 2020-05-01

Table of Contents

Please note:

This document refers to revision 1.0 of the USB/CAN Interface module. If you are using a device with another revision number, please refer to the corresponding documentation!

Reusch Elektronik
Dipl.-Ing. (FH) Rainer Reusch

Blumenstr. 13
D-88097 Eriskirch
Germany

Phone: +49-7541-81484
Fax: +49-7541-81483
E-Mail: info@reusch-elektronik.de
Homepage: www.reusch-elektronik.de

1. Introduction

USB/CAN Interface is an insolated and easy to use CAN adaptor in industrial quality for a low price. The software is designed for *Microsoft Windows*®. The driver interface is based on the open source LibUSB-Win32 driver set. A Windows application is available as well as C++ sources of the interface to design own applications with *Embarcadero RADStudio XE*®.

2. Specifications

2.1 USB

Power Supply	USB powered (4.75 to 5.25V DC)
Supply Current	typ. 51mA, max. 100mA
USB Connection	USB 2.0, full speed (12MBit/s)

2.2 CAN

Connector	Sub-D 9 pin (male)
Bus Termination	none or 120Ω (controlled by software)
Insolation Voltage	max. 100V
Supported CAN speeds	10, 20, 50, 100, 125, 250, 500, 1000kBit/s
CAN Filters	up to 14 (controlled by software)

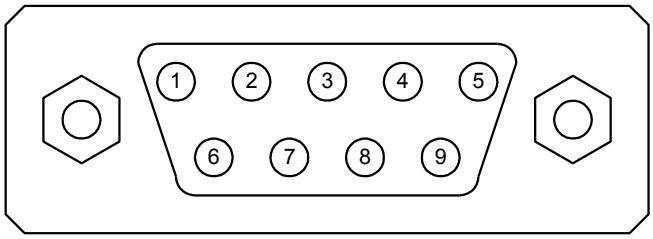
2.3 Mechanical

Dimensions	64 x 95 x 33mm³
Weight	110g

The USB and CAN connector is ESD protected.

3. Technical Information

3.1 CAN Connector



9 pin Sub-D connector, male

Pin Assignment

Pin	Description
2	CAN Low
3	CAN Ground
6	CAN Ground
7	CAN High

The other pins of the SUB-D connector are not connected. All the signals on these pins are electrically insolated against the USB connector.

Please note:
The shielding of the SUB-D connector is **not insolated**! It is connected with the casing and the shield of the USB connector.

3.2 Activity LED

The LED is turning on, when the device is recognized by the operating system. It is flashing bright at data activity.

3.3 Device Properties

The device offers a set of eight standardized CAN bus transfer rates from 10kBit/s up to 1MBits/s. After a power cycle the lowest speed is set (10kBit/s). The transmission behavior can be controlled by software. The automatic retry of failed message transmissions can be activated (power on default) or disabled. Also a loop back mode can be activated for test purposes. That means, the logical output and input is connected. The device is receiving its own transmitted messages.

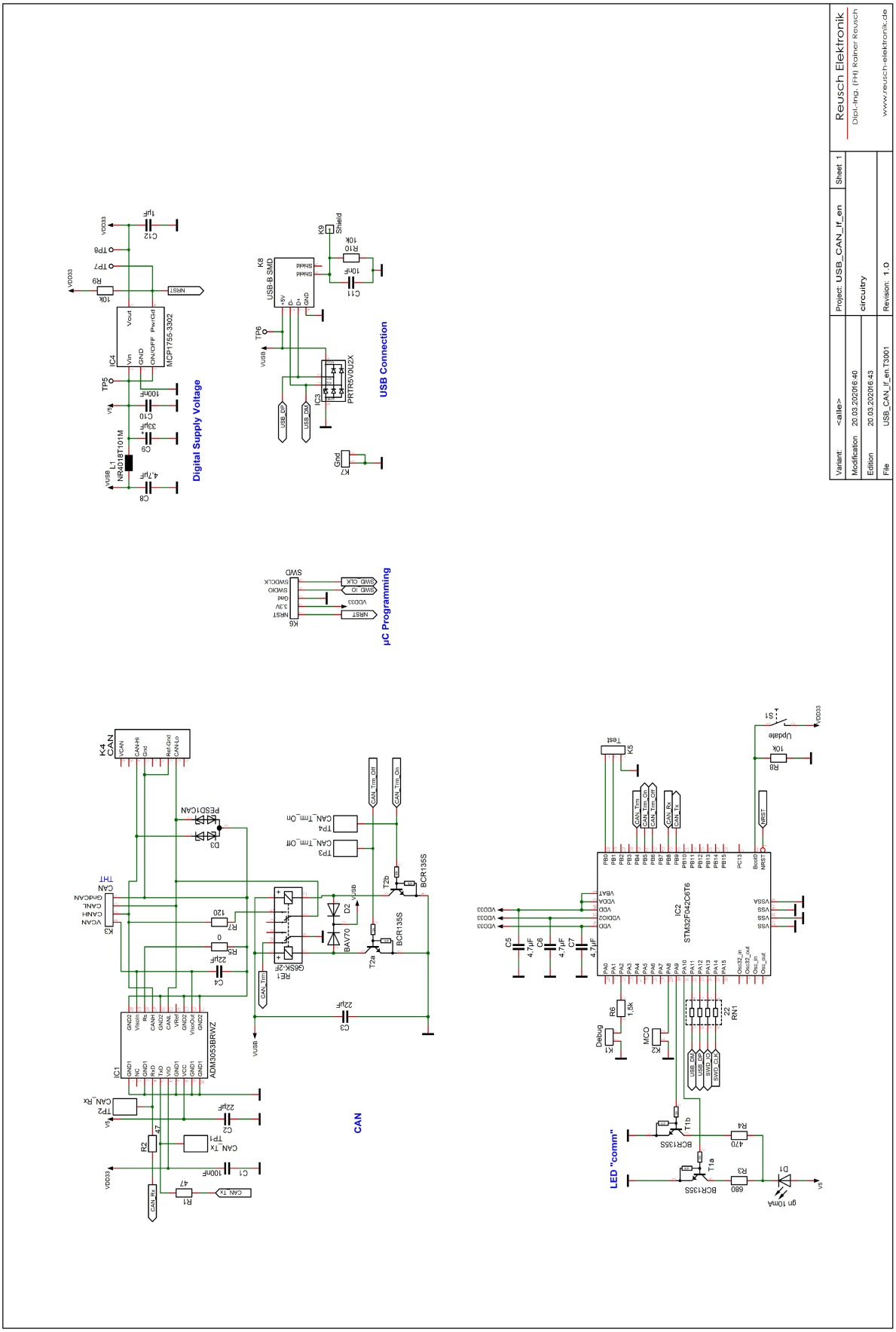
The bus termination of the device can be turned on and off by software. This setting will be kept in the powerless state (usage of a bistable relay).

The firmware of the device supports a FIFO buffer for up to eight CAN messages to transmit. As long as this buffer is full, the device will not accept new messages by the USB communication.

To receive messages, a FIFO buffer for up to 64 messages is realized. Messages will be lost, if this buffer becomes full and no messages will be retrieved by USB communication.

The microcontroller of the device allows the specification of up to 14 message filters. After power on all messages will be received (none will be filtered out). Filter bank 0 is initialized for this behavior.

3.3 Schematic



4. Programming Interface

4.1 Intruduction

Sources for *Embarcadero RAD Studio XE* C++Builder are available. The following files has to be included into your project:

usb.h	Based on the original “usb.h”, published with the LibUSB-Win32 sources. Contains a small modification.
libusbdyn.h libusbdyn.cpp	Based on the original files, which are published with the LibUSB-Win32 sources. The driver DLL will be linked dynamically.
usbdevice_libdyn.h usbdevice_libdyn.cpp	Contains basic routines and the declaration of a basic class for USB devices of this type.
uci_if.h uci_if.cpp	Contains the class, which represents the USB/CAN Interface device.
uci_common.h	Common declarations of the USB/CAN Interface device firmware and the interface class, declared in “uci_if.h”.

All these files has to become a part of your project. Two files has to be included by the include directive:

```
#include “uci_common.h”
#include “uci_if.h”
```

In the first step, the function *USB_Initialize()* can be called explicit (if not, it will be called automatically). This function returns *true*, if the LibUSB-Win32 drivers are installed. An explicit call and evaluation of the return value offers you the opportunity, to flash an error message and stop further proceeding, because the device can't be accessed, if the result is *false*.

In the next step an instance of the class TUCI has to be generated:

```
TUCI *uci = new TUCI();
```

This class offers the function

```
int __fastcall ScanForDevices(TStrings *DeviceList);
```

Parameter is an instanced class of *TStrings* (a list of Strings). A call of this function fills the list with serial numbers of all found USB/CAN Interface devices. The function returns the number of found devices. An entry of the list can be used, to open a specific device. To open a device, call the function

```
uci->Open(“ASerialnumber”)
```

The parameter is a serial number of the device list, you got by calling of *ScanForDevices*. *Open* also can be called with an empty string. In this case, the first found device will be opened and a call of *ScanForDevices* is not necessary. The call of *Open* returns *true*, if the connection to the device is established. Hence, an access to the device is possible, by calling the other functions of the *TUCI* class. More details will be shown in the reference part of this documentation.

At the end (to cut the connection to the device), the *TUCI* function *Close()* should be called.

4.2 Class declaration

The interface class TUCI (file *uci_if.h*) is defined as follows:

```
class TUCI : public TUSBDevice
{
private:
protected:
public:
      __fastcall TUCI(void);
      __fastcall ~TUCI(void);
      virtual bool __fastcall Open(AnsiString SerialNo);
      virtual void __fastcall Close(void);
      bool __fastcall Status(TStatus *Status);
      bool __fastcall ClearStatus(bool Rx, bool Tx, bool ErrorCounters);
      bool __fastcall DeviceInfo(TDeviceInfo *di);
      bool __fastcall GetDeviceConfig(TDeviceConfig *dc);
      bool __fastcall SetDeviceConfig(TDeviceConfig *dc);
      bool __fastcall CAN_Rx(TCANData *cd);
      bool __fastcall CAN_Tx(TCANData *cd);
      bool __fastcall CAN_SetFilter(TCANFilter *cf);
      bool __fastcall CAN_SetFilterAll(void);
};
```

This class is derived from *TUSBDevice*, which is defined in the file *usbdevice_libdyn.h*. The methods and parameter declarations are described in the following chapter.

4.3 Reference

This reference refers to firmware version V1.xx.

`bool USB_Initialize(void)`

Declared in *usbdevice_libdyn.h*. Initialize the USB interface. The function returns the result *false*, if the LibUSB-Win32 drivers are not installed and therefor the interface library can't be used!

`__fastcall TUCI(void)`

The constructor of the USB/CAN Interface class.

`__fastcall ~TUCI(void)`

The destructor of the class.

`__property bool Opened`

Defined in the inherited class *TUSBDevice* and read only. Contains *true*, if the device communication is opened (successful call of *Open*).

`__property AnsiString SerialNumber`

Defined in the inherited class *TUSBDevice* and read only. Contains the serial number of the opened device. This property is helpful to get the serial number, if *Open()* is called with an empty string as serial number.

int __fastcall ScanForDevices(TStrings *DeviceList)

Defined in the inherited class *TUSBDevice*. Scan for USB/CAN Interface devices. *DeviceList* has to be instanced. The list will be cleared and filled will the serial numbers of all found devices. Return value is the number of found devices (can be zero).

A call of this function causes a call of *USB_Initialize()*, if not explicit done before.

bool __fastcall DeviceRemoved(void)

Defined in the inherited class *TUSBDevice*. Check, if the device is removed (unplugged). The function returns *true*, if the opened device is no longer present. In this case, the logical connection will be closed automatically.

Usually this method is be called within the handle of the Windows message WM_DEVICECHANGE. This message occurs, if a device is connected or removed. This method helps you to detect, if the assigned device of the *TUCI* class instance is the cause for the message.

bool __fastcall Open(AnsiString SerialNo)

Open the USB communication with a device. The parameter *SerialNo* specifies the device to open by its serial number. The serial numbers of all connected devices can be retrieved by calling *ScanForDevices()*. If the *Open()* function is called with an empty string as parameter, the first found device will be opened. The function returns *true*, if a connection to the specified device is established.

Please note: A successful open call is necessary for further operations with the device!

void __fastcall Close(void)

Close the connection of an opened device. Hence no more device operations (except *Open*) will work. This function will be called implicit, when the *TUCI* class will be destroyed.

bool __fastcall Status(TStatus *Status)

Request the device status. The method returns *true*, if succeeded. In this case the structure *Status* is filled with information.

Declaration of *TStatus* (defined in *uci_common.h*):

```
typedef struct {
  TCmd cmd;             // used for USB communication (contains USB_STATUS)
  uint16_t Reserved;    // not used
  uint32_t Status;      // Status Bits (see STATUS_ constants in uci_common.h)
  uint32_t RxLost;      // number of lost CAN messages at receive
  uint32_t TxLost;      // number of lost CAN messages at transmission
  uint32_t TxError;     // number of not transmitted CAN messages (tx error)
} TStatus;
```

For details about the field *Status* refer to the source of *uci_common.h*.

bool __fastcall ClearStatus(bool Rx, bool Tx, bool ErrorCounters)

Reset status information and buffers.

Rx: true = clear receiving FIFO buffer (erase all received and not retrieved messages)
Tx: true = clear transmission FIFO buffer (erase all messages for transmission)
ErrorCounters: true = reset the status error counters (*RxLost*, *TxLost* and *TxError*)

`bool __fastcall DeviceInfo(TDeviceInfo *di)`

Request the device information. If the function call returns *true*, the structure *di* will be filled with information.

Declaration of *TDeviceInfo* (defined in *uci_common.h*):

```
typedef struct {
  TCmd cmd;                 // used for USB communication (contains USB_INFO)
  uint8_t  Reserved1;       // not used
  uint8_t  Revision;        // board revision (0 for Rev. 1.0)
  uint16_t Version;         // firmware version
  uint16_t Reserved2;       // not used
  xchar SerialNo[16];       // serial number
  uint32_t Reserved3[10];   // reserved for future use
} TDeviceInfo;
```

`bool __fastcall GetDeviceConfig(TDeviceConfig *dc)`

Request the device configuration. If *true* is returned, the structure *dc* is filled with the information.

Declaration of *TDeviceInfo* (defined in *uci_common.h*):

```
typedef struct {
  TCmd cmd;             // contains USB_CONFIG_R or USB_CONFIG_W
  uint8_t Speed;        // CAN bus speed (SPEED_ constant)
  uint8_t Termination;  // CAN bus termination (TERMINATION_ constant)
  uint8_t Retransmit;   // 1 = repeat transmission at error (default)
  uint8_t LoopBackMode; // 1 = loop back mode (for debug purposes)
  uint8_t Reserved1;    // not used
  uint8_t Reserved2;    // not used
} TDeviceConfig;
```

The *Speed* variable contains one of the following constants to specify the CAN bus transfer rate (bus speed):

SPEED_10K, *SPEED_20K*, SPEED_50K, SPEED_100K, SPEED_125K, SPEED_250K, SPEED_500K or SPEED_1M.

The *Termination* variable contains one of the following constants, to specify, if the CAN bus termination is turned off or on:

TERMINATION_OFF (no termination by the device)
TERMINATION_ON (bus terminated by the device)

Please note: The termination setting will be kept, if the device is unpowered (e.g. unplugged from USB).

After a power cycle the device is set to a bus speed of 10kBit/s, retransmission is activated and loop back mode is deactivated. The bus termination will be in its last state.

bool __fastcall SetDeviceConfig(TDeviceConfig *dc)

Set a device configuration. The structure *dc* has to be initialized as required (except the field *cmd*). The function returns *true*, if succeeded.

Example:

```
TDeviceConfig dc;
dc.Speed = SPEED_125K;
dc.Termination = TERMINATION_ON;
dc.Retransmit = 1;  // activate retransmission
dc.LoopBackMode = 0;  // no loop back mode
if (!uci->SetDeviceConfig(&dc)) ShowMessage("error");
```

bool __fastcall CAN_Rx(TCANData *cd)

Retrieve a received CAN message. The function returns *true*, if a message was retrieved. In this case, the structure is filled with the data.

Declaration of *TCANData* (defined in *uci_common.h*):

```
typedef struct {
  TCmd cmd;                 // contains USB_CAN_TX or USB_CAN_RX
  uint8_t  IdType;          // Std. or ext. message (CAN_ID_ constant)
  uint8_t  RemoteRequest;   // data or remote request frame (CAN_RTR_ constant)
  uint8_t  Reserved;        // not used
  uint8_t  DataLength;      // number of data bytes (0..8)
  uint32_t Identifier;      // CAN identifier
  uint8_t  Data[8];         // data
} TCANData;
```

Example:

```
TCANData cd;
if (CAN_Rx(&cd)) {
  if (cd.Id==CAN_ID_STD) ShowMessage("Standard Message");
  else ShowMessage("Extended Message");
  for (int i=0; i<cd.DataLength; i++)
    ShowMessage(IntToStr(cd.Data[i]));
}
```

For a received data frame, *RemoteRequest* contains *CAN_RTR_DATA* (0). If *RemoteRequest* contains *CAN_RTR_REMOTE* (2), the received frame is a remote request. *DataLength* contains the requested amount of data and the contents of *Data* is not relevant.

bool __fastcall CAN_Tx(TCANData *cd)

Transmit a CAN message. The structure *cd* has to be initialized with the data to transmit (except *cmd*). The function returns *true*, if succeeded. If it returns *false*, there is no connection to the device or the FIFO buffer in the device is full. In this case, simply repeat the transmission call later.

Example:

```
TCANData cd;
cd.IdType = CAN_ID_EXT;  // an extended message
cd.RemoteRequest = CAN_RTR_DATA;  // a data frame
```

```
cd.Identifier = 0x1234;
cd.Data]0] = 0x0F;
cd.Data]1] = 0x1E;
cd.Data]2] = 0x2D;
cd.DataLength = 3;
if (!CAN_Tx(&cd)) ShowMessage("transmission failed");
```

To transmit a remote request frame, set the *RemoteRequest* field to *CAN_RTR_REMOTE* and specify the requested data amount in the *DataLength* field. Data doesn't have to be initialized.

bool __fastcall CAN_SetFilter(TCANFilter *cf)

Set a CAN filter. The structure of *cf* has to be initialized with the required filter parameters. With this function call one of 14 available filters is configured. The function returns *true*, if succeeded.

Declaration of *TCANFilter* (defined in *uci_common.h*):

```
typedef struct {
  TCmd cmd;                 // contains USB_CAN_FILTER
  uint16_t Reserved1;
  uint8_t IdType;           // filter for Std. or Ext. messaged (CAN_ID_...)
  uint8_t Bank;             // filter bank number (0..13)
  uint8_t Active;           // 0 = filter off, 1 = filter active
  uint8_t Mode;             // filter modus (CAN_FILTERMODE_ constant)
  union {
    struct {                // parameters for "mask" mode
      uint32_t ID;          // message ID template
      uint32_t Mask;        // mask (relevant bits of ID)
    };
    struct {                // parameter for "list" mode
      uint32_t ID1;         // 1st ID (standard or extended)
      uint32_t ID2;         // 2nd ID (standard or extended)
    };
  };
} TCANFilter;
```

The field *IdType* specifies, if the *ID* fields are containing a standard or an extended message identifier. *Bank* specifies the filter bank number. A number from 0 to 13 is allowed. *Active* turns on (1) or off (0) the filter bank.

There are two ways, to specify a filter. In the "list" mode (*CAN_FILTERMODE_IDLIST*) two CAN message IDs will be specified. With this method only messages will pass, which are exactly matching with one of the two IDs.

The "mask" mode (*CAN_FILTERMODE_IDMASK*) allows to specify a block of message IDs, which will pass the filter. **Example:** If the ID field is set to 0x0A50 and the mask field is set to 0x0A5E, the messages with the ID 0x0A50 and 0x0A51 will pass the filter. If the mask 0x0A5C is used, the messages with the ID 0x0A50 up to 0x0A53 will pass the filter.

bool __fastcall CAN_SetFilterAll(void)

Set the filter bank 0 for passing all CAN messages. Therefor settings of the other filter banks will have no effect. Hence all messages will be received!

Please note: After a power cycle of the device all filter banks are deactivated, except filter bank 0. This bank is set to receive all messages (same as calling the method *CAN_SetFilterAll()*).

5. Concluding Remarks

5.1 Downloads

You will find product information, this manual, an application software for *Microsoft Windows*® and source files for download on the product homepage. Please visit the following URL:
http://products.reworld.eu/uci.htm

5.2 Links

Visit the following links for more information about other products, used software modules and tools.

Reusch Elektronik homepage for electronic products:
http://products.reworld.eu

LibUSB-Win32, USB drivers for *Microsoft Windows*®:
http://libusb-win32.sourceforge.net/

Zadig – USB driver installation made easy
https://zadig.akeo.ie/

5.3 Conformity statement and Disclaimer

This electronic device is designed under best known engineering guidelines. It confirms the appropriate design rules. No warranty or liability is given for adherence, assured properties, or damages which might be caused by the usage of this hardware or the accessory software.

Note: This equipment has been tested and found to comply with the limits for a Class B digital device. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy, and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation.

5.4 Technical Support

If You have any questions, technical problems or need additional information, please contact us by e-mail (preferred) or FAX.

E-Mail: support@reusch-elektronik.de

FAX: +49-7541-81483 (turned off outside the business time)

We are able to handle inquiries in english and german language.

Don't hesitate to contact us, if You have demand for custom specific solutions!