CAN via UDP

Configuration, Usage, and Protocol Specification

V 1.0

by Wolfgang Büscher, MKT Systemtechnik

Changes to the specification may be possible at any time, and may take place without any explicit notification. Please contact the developer for an update, if you consider to implement this protocol in your own application.

Sorry, there is no german translation of this document, and never will. It was written *by* a developer *for* developers.

Original file location: WoBu5:\C:\DRIVERS1\DOKU\art85140_CAN_via_UDP.odt Rev. Date: 2011-06-08

Contents

1. Revision History

Versionsnummer	Datum (ISO8601)	Autor	Bemerkungen, Änderungsgrund
V 0.1	2011-04-26	W. Büscher	Created this document as a 'placeholder'
V 0.2	2011-05-10	W. Büscher	First functional implementation of 'CAN-via-UDP' in an MKT-View II .

2. Preface

This document specifies a simple, UDP-based protocol used in various embedded devices by MKT Systemtechnik to transmit CAN bus messages over an Ethernet connection.

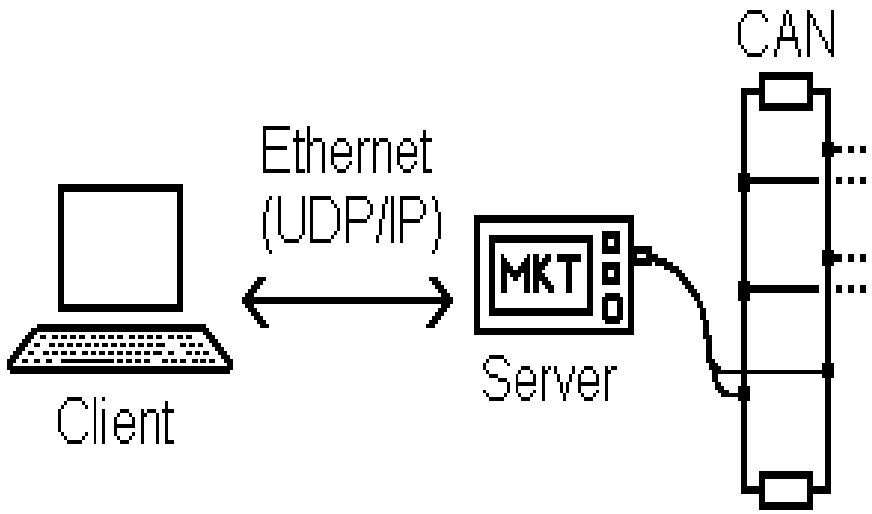
The primary goals at the time of development were:

- keep it simple, but not necessarily stupid
- keep it expandable
- make it easy to implement
- make it portable (don't use any fancy dot-something-stuff)

The CAN-via-UDP protocol uses a simple client / server model.

The Server has a 'physical' CAN bus interface.

The Client typically only has an Ethernet port, but uses the remote server's CAN interface.

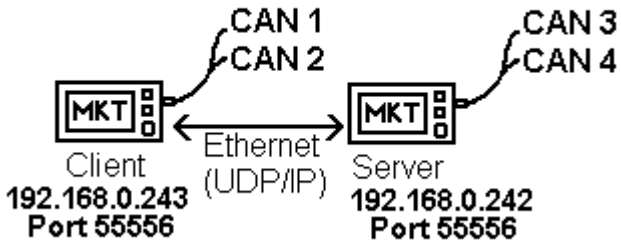


Example:
Server at **192.168.000.242:55556** (IP-address and UDP port number)
Client at **192.168.000.042:**55557

The client's own port number doesn't really matter, only the server's port number must be fixed - see suggestions for the default settings in the chapter 'IP Usage'.

3. Using CAN-via-UDP to have *four* CAN ports on the MKT-View (II)

At least the MKT-View II has an integrated CAN-via-UDP client and server, which allows two of these devices being used in the following configuration:



MKT-View II with four 'logical' CAN interfaces, two of them
(CAN 3 and CAN 4) provided by a second MKT-View (server).

In this case, the two devices (client and server) were connected with a simple (straight through) RJ-45 Ethernet cable, without any other network equipment in between.

Note: This is possible because the Ethernet 'PHY' controller used in the MKT-View II can automatically reverse the 'TD' and 'RD' pairs if a straight-through cable is used. At the time of this writing, it was unknown if all (future) devices would offer the same flexibility. Depending on the PHY controller type, it may be necessary to use a cross-over cable instead.

3.1 Configuration of the MKT-View (II) for two additional CAN ports

The following screenshots show the network configuration of the client (left) and the server (right). In this example, both devices were MKT-View II .

```
NETWORK SETUP (24) NAV

MAC:00-50-C2-8E-70-39

IPaddr:192.168.000.243
Subnet:255.255.255.000
Gatew.:192.168.000.001
Local TCP Server Ports
  HTTP:80 TN:23 FTP:21
CAN via UDP : CLIENT
  RemoteIP:192.168.000.242
  Ports: R=55556  L=55556

F3: Save & Exit [↑] [↓]
 CAN-via-UDP mode
```

```
NETWORK SETUP (24) NAV

MAC:00-50-C2-8E-70-01

IPaddr:192.168.000.242
Subnet:255.255.255.000
Gatew.:192.168.000.001
Local TCP Server Ports
  HTTP:80 TN:23 FTP:21
CAN via UDP : SERVER
  RemoteIP:192.168.000.243
  Ports: R=55556  L=55556

F3: Save & Exit [↑] [↓]
 CAN-via-UDP mode
```

More details about the Network Setup (menu) in the MKT-View, and similar displays, can be found in document #85115 (System Menu and Setup).

A test application for the MKT-View 2 (and possibly any later device) is in ../programs/arm7_special/4buses.cvt . It transmits on CAN1, and receives signals on CAN2, CAN3, and CAN4. The latter two ports use the remote CAN-via-UDP client.

3.2 Performance of the MKT-View (II) with two additional CAN ports

The performance (max message rate, latency, ..) of the two additional CAN ports depends on the local network (switches or hubs, ..), and on the CPU load inside the MKT-View caused by the display task, and other tasks performed by the device at the same time.

At the time of this writing (2011-06), no measurements had been made.

Some notes about CAN-via-UDP in the MKT-View II :

- The two additional CAN ports ("CAN3" and "CAN4") have a larger latency than the local ports ("CAN1" and "CAN2"). If your application implements its own communication protocol, for example using the built-in script language, you should use CAN1 and CAN2 for bidirectional, acknowledged communications if the round trip delay between sending the 'request' and receiving the 'response' via CAN is critical. Reason: The round trip delay caused by the local network (Ethernet and IP protocol) will be added to the CAN bus delay, which may be in the order of a few milliseconds.
  Use CAN3 and CAN4 to collect 'display data', where an additional delay of a few milliseconds doesn't matter.
- The CAN logger built inside the MKT-View (II) does NOT support four CAN ports yet. It only works with the two 'local' CAN ports ("CAN1" and "CAN2") .
- The CAN snooper built inside the MKT-View (II) does support the additional CAN ports.
- To reduce the latency for CAN3 + CAN4 in the MKT-View (II), do ***not*** use the web-based remote control feature, because the load caused by the TCP/IP traffic (mainly the transmission of compressed framebuffer images from the HTTP server to the web browser) increases the latency of UDP transmission and reception. Reason: Both TCP/IP and UDP/IP run in the same worker task (using LwIP), on a comparably slow 72 MHz CPU.

4. Using CAN-via-UDP to upload files and configurations

Certain external devices (like the planned Flexray interface for MKT-Views) may require a very complex configuration, which -for example- is created by some kind of 'Flexray Configuration Tool'. Such a tool could place the entire configuration for the Flexray box in a file on a memory card. That memory card could be placed in the MKT-View (because the Flexray box may be inaccessable for the user). The Flexray box could then read the file, using the trivial file transfer server implemented in the MKT View Firmware. No details about such a Flexray box were known at the time of this writing (2011-06-08).

In addition, the UPT (and "CANdb Terminal") programming tool by MKT contains a simple file transfer utility, which can be used to access files on a remote CAN-via-UDP server. Details about that are in the manual for the UPT programming tools, document #85110 (available after installation of the tool in the "Doku" folder, or online at the MKT website).

5. IP Usage

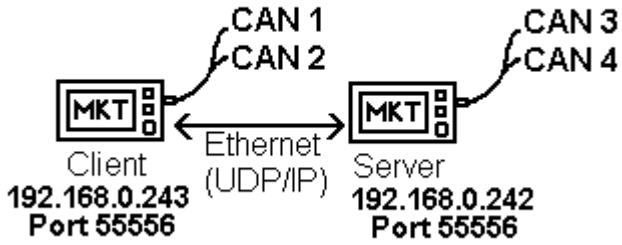
The CAN-via-UDP protocol uses, as the name says, not TCP/IP but UDP/IP.
UDP is much easier to implement on a microcontroller with limited resources than TCP; and when used on a local network (not "via the internet") UDP is sufficiently reliable for this application (remember: CAN also doesn' guarantee that a message is successfully delivered to the intended recipient; the same applies to UDP).

There are no port numbers 'registered' for this protocol, so we can only specify anything here, but only ***suggest*** the following ***default settings*** (it's up to you to use or ignore them; the CAN-via-UDP client may use any available port for receiving the response from the server):

CAN-via-UDP SERVER port number: 55556 (=local UDP port number used by the server, to 'listen' for incoming datagrams)

CAN-via-UDP CLIENT port number: 55557 (=local UDP port number used by the client, if both client and server shall run on the same machine).

In most cases, client and server will run on different machines, so they can use the same UDP port number (locally) as in the following example. The client (MKT-View on the left side) has two own ('physical') CAN interfaces "CAN1" and "CAN2", and uses two remote CAN interfaces on a remote server ("CAN3" and "CAN4" on the right side.



CAN-via-UDP example using two MKT-Views (2) .
Note the different IP addresses ! Details in chapter 3. .

The first device by MKT which supported CAN-via-UDP (client ***or*** server) was a display called 'MKT-View II'. Like many other embedded devices with light-weight internet protocol stack, this device uses a fixed IP address to keep things simple. The default IP was originally 192.168.0.100, but it was later changed to 192.168.0.242 to reduce the chance of conflicts with dynamically allocated IPs, and other network devices (like routers, switches, etc).

5.1 Using CAN-via-UDP on a PC

You can use this, for example, if the PC doesn't have a CAN interface installed. In this case, you can use any of MKT's devices (with a CAN-via-UDP-***server*** built inside) as an interface between your PC and the CAN bus. The PC will be the CAN-via-UDP-***client*** then.

Use your PC's 'ipconfig' and 'ping' command to check the IP settings. The network setup in the remote server may have to be adjusted, to allow using it in your local network. Example:

>**ipconfig**

```
Windows-IP-Konfiguration

Ethernetadapter LAN-Verbindung:

        Verbindungsspezifisches DNS-Suffix:
        IP-Adresse. . . . . . . . . . . . : 192.168.0.24
        Subnetzmaske. . . . . . . . . . . : 255.255.255.0
        Standardgateway . . . . . . . . . : 192.168.0.254
```

Check if the embedded device's IP address, and subnet mask, is compatible with the results displayed by ipconfig. The subnet mask ("Subnetzmaske") *should* be identical, and all bits in the IP address which are not zero in the subnet mask *must* be identical. At least one of the bits *which are zero in the subnet mask* must be *different* in the IP addresses (this is usually only the 4th byte, in 99.9 % of all cases - thus the common subnet mask "255.255.255.0").

5.2 Excursion: Check if the device's fixed IP address is still available

Let's try if the suggested 'default' IP address 192.168.0.242 is not occupied yet (***before*** connecting the new device to the LAN):

>**ping 192.168.0.242**

```
Ping wird ausgeführt für 192.168.0.242 mit 32 Bytes Daten:

Zeitüberschreitung der Anforderung.  (timeout on an english PC)
```

This confirms that, *before* connecting the embedded device, no other device has occupied the default IP address of the CAN-via-UDP server, so we can safely connect the embedded device (e.g. "MKT-View II" operating as the server) to the LAN.
For the next step, connect the embedded device to the LAN, then try the ping command again :

>**ping 192.168.0.242**

```
Ping wird ausgeführt für 192.168.0.242 mit 32 Bytes Daten:

Antwort von 192.168.0.242: Bytes=32 Zeit<1ms TTL=255
Antwort von 192.168.0.242: Bytes=32 Zeit<1ms TTL=255
Antwort von 192.168.0.242: Bytes=32 Zeit<1ms TTL=255
Antwort von 192.168.0.242: Bytes=32 Zeit<1ms TTL=255

Ping-Statistik für 192.168.0.242:
 Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0 (0% Verlust),
```

```
Ca. Zeitangaben in Millisek.:
 Minimum = 0ms, Maximum = 0ms, Mittelwert = 0ms
```

Typical 'ping' turnaround times should be less than a millisecond (as measured above - the "ping" command from windows cannot measure the short delay properly). Otherwise, check your network equipment - there may be a faulty switch/hub/etc- because with a latency of more than a a millisecond, CAN-via-UDP will suffer from a performance penalty. But even if a UDP frame took one millisecond for 'delivery', the protocol can transfer more than 1000 CAN messages per second because the protocol will stuff more than one CAN message into a single UDP datagram (~ one Ethernet frame).

5.3 Trivial File Transfer Protocol (embedded in the CAN-via-UDP stack)

<TBD>

The CAN-via-UDP protocol handler contains a very simple ("trivial") file transfer protocol, which can be used to upload (send, write) files from a client into a server, and to download (receive, read) files from a remote server. The remote server is typically a programmable device (like the MKT-View II), a standalone CAN / Ethernet gateway, or (future plan) a Flexray / Ethernet gateway which must be fed with some kind of configuration file (which is outside the scope of this document).

Similar to (but not compatible with) the Trivial File Transfer Protocol (TFTP) described in RFC1350, the file transfer is always initiated by the client.

The 'opcodes' (as RFC1350 likes to call them) are defined in the header file CanViaUDP.H . A few more 'opcodes' were added, so at the moment, the following opcodes are used *in the payload of a CAN-via-UDP frame, if the 'struct type' is* *CanUDP_STRUCT_TYPE_TFTP* . This is NOT the same as the payload of a UDP datagram (see chapter 7) ! From CanViaUDP.h :

```
#define CanUDP_TFTP_OPCODE_NONE   0 /* dummy / test                          */
#define CanUDP_TFTP_OPCODE_RRQ    1 /* "read request"  similar to RFC1350    */
#define CanUDP_TFTP_OPCODE_WRQ    2 /* "write request" similar to RFC1350    */
#define CanUDP_TFTP_OPCODE_DATA   3 /* "data" (512 byte netto) as in RFC1350 */
#define CanUDP_TFTP_OPCODE_ACK    4 /* "acknowledgement" as in RFC1350       */
#define CanUDP_TFTP_OPCODE_ERROR  5 /* "error"           as in RFC1350       */
#define CanUDP_TFTP_OPCODE_LIST   6 /* request to "list" the directory       */
#define CanUDP_TFTP_OPCODE_L_RESP 7 /* response, one line of the dir-listing */
#define CanUDP_TFTP_OPCODE_FIN    8 /* to finish/abort a listing by the client*/
```

Please ask the software development engineer (Mr. Büscher) for an up-to-date copy of that file, if you need to implement your own client or server using the file transfer protocol compatible with MKT's devices.

Note: To aid development, sent and received UDP datagrams can be displayed in the UPT programming tool during file transfer. In the file transfer window, select '*Options*' .. ***Enable Debugging*** before you start to upload or download a file. The debug messages are dumped in the programming tool's main window, on the tabsheet ***Errors and Messages*** . The hexadecimal UDP payload is truncated after the 16th byte. For more detailed analysis, use WireShark instead (see one of the next chapters).
Symbols in the message dump like 'RRQ', 'WRQ', 'DATA', 'ACK', 'ERROR' have the same meaning as for the 'Trivial File Transfer Protocol', described in RFC 1350. Despite that, the UDP payload is not compatible with RFC 1350, because certain features (like the exchange of the file size and file date+time) were missing in RFC 1350.

5.4 Remote Directory Listing

As an extension to the TFTP protocol (see previous chapter), the CAN-via-UDP protocol handler (implemented in CanViaUDP.C) also supports listing a remote directory.
The client initiates the transfer (using the proprietary 'LIST' command, opcode 6).
The server responds with the first directory list item ('DIR_ENTRY', opcode 7), containing one line of the directory listing.
The client acknowledges this (TFTP opcode 4), causing the server to send more directory entries. After the last directory entry, the server sends a 'FIN' (opcode 8) to indicate that the transmission of the directory listing is finished.
The structure of a directory entry is defined as 'TCanUDP_TFTP_DirEntry' in CanViaUDP.h.
Using the 'debug display' in the UPT programming tool (see previous chapter), the UDP message dump looks like this:

```
CvUDP_Connect: 'My' IP address is 192.168.0.24, 'my' port 55556 .
Created transfer THREAD .
CvUDP Server reports t=475501875, f=40000 Hz .
CvUDP: TX 192.168.000.243:55556 05 00 5c 00 06 00 00 00 00 00 00 00 2a 2e 2a 00..  TFTP:LIST "*.*"
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 10 00..  TFTP:DIR_ENTRY "data_flash"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 10 00..  TFTP:DIR_ENTRY "font_flash"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 10 00..  TFTP:DIR_ENTRY "audio_flash"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 10 00..  TFTP:DIR_ENTRY "memory_card"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 10 00..  TFTP:DIR_ENTRY "ramdisk"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 40 00..  TFTP:DIR_ENTRY "serial1"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 7c 00 07 00 00 00 00 00 00 00 00 40 00..  TFTP:DIR_ENTRY "serial2"
CvUDP: TX 192.168.000.243:55556 05 00 04 00 04 00 00 00                         TFTP:ACK[#0]
CvUDP: RX 192.168.000.243:55556 05 00 04 00 08 00 ff ff                         TFTP:FIN

        [remote IP addr] [port] [-header -] [--- payload / TFTP opcode, .. ---]    [ decoded info ]
                                |  |  |___|
                                |  |  |
                                |  |  |_ struct size (16 bit)
                                |  |
                                |  |_ Reserve (see TCanUDP_StructHeader )
                                |
                                |_ struct type
```

6. Data Types and Structures

All structures which are used as 'payload' in UDP frames are defined (as "C" structs) in the file CanViaUDP.h .

That file is available on request by the developer (Mr. Wolfgang Büscher).

< TBD >

6.1 Structure Types

MKT's 'struct type code' is in the first byte *in the payload* of any UDP datagram (actually, it's the first byte of any 'struct header', and there may be more than one struct header in a datagram).

As most other defines which are relevant for the implementation of the CAN-via-UDP protocol, the type codes are defined in CanViaUDP.h :

```
// Identifiers for the different 'structures' which may be packed
//    into an UDP datagram.  Used in T_CanUDP_StructHeader.bStructType .
#define CanUDP_STRUCT_TYPE_NOTHING     0  // "nothing" or a "dummy frame"
#define CanUDP_STRUCT_TYPE_HEADER      1  // -> TCanUDP_StructHeader
#define CanUDP_STRUCT_TYPE_STRING      2  // a TEXT STRING, command or response
#define CanUDP_STRUCT_TYPE_COMMAND     3  // -> TCanUDP_CommandParams plus args
#define CanUDP_STRUCT_TYPE_CAN_MESSAGE 4  // -> TCanUDP_CAN_Message
#define CanUDP_STRUCT_TYPE_TFTP        5  // -> TFTP (trivial file transfer)
#define CanUDP_STRUCT_TYPE_UPT_TRANSFER 6 // transfer a UPT DISPLAY PROGRAM
```

For most of these 'structure type codes', there is a "C" data type ("typdef struct") defined in CanViaUDP.H, with a matching name, for example:

```
typedef struct tCanUDP_StructHeader
{ // four-byte prefix before any structure mapped into a UDP datagram (frame):
  BYTE bStructType; // contains CanUDP_STRUCT_TYPE_CAN_MESSAGE, etc etc
  BYTE bReserve;    // for DWORD-alignment, and as a future reserve
  WORD wStructSize; // Number of bytes AFTER the T_CanUDP_StructHeader.
} TCanUDP_StructHeader;


typedef struct tCanUDP_CAN_Message // format of a CAN MESSAGE in a UDP frame
{ // Use with ..StructHeader.bStructType = CanUDP_STRUCT_TYPE_CAN_MESSAGE [...]
  DWORD dwCANid;  /* CAN-ID (11 Bit or 29-Bit, Bit 29 is a FLAG then)     */
  DWORD dwFlags;  /* bits 3..0  = DLC (Data Length Code)   [...]          */
  DWORD dwTimestamp;  /* Zeitstempel (mit hardwarespezifischer Frequenz)  */
  DWORD dw2Data[2];   /* Datenfeld (für CAN: maximal 8 Datenbytes)        */
} TCanUDP_CAN_Message;
```

7. Sample UDP traffic (analysed with Wireshark)

The following EXAMPLE shows the first CAN-via-UDP datagram received on the SERVER side, with 'Data' = UDP *payload* (*netto*, without IP headers), copied from a Wireshark dump.

```
Data: 03 00 1400  01 00 00 04  C0A800F2  04D90000  C0A80018  04D90000
      |_________|  |_____________________________________________________|
      | header     | Command, CAN-message, or similar (here: Command)
      |            |  |  |  |  |         |         |         |
      |            |  |  |  |  |         |         |         |_ dwArg[3]
      |            |  |  |  |  |         |         |___________ dwArg[2]
      |            |  |  |  |  |         |_____________________ dwArg[1]
      |            |  |  |  |  |_______________________________ dwArg[0]
      |            |  |  |  |
      |            |  |  |  |
      |            |  |  |  |
      |            |  |  |  |___ bNumArgs: 4 arguments (command specific)
      |            |  |  |______ bBusIndex : 0x00
      |            |  |_________ bCmdFlags : 0x00
      |            |____________ bCommand  : 0x01 = CanUDP_CMD_Connect
      |
      |____
      |TCanUDP_StructHeader: always four bytes long.
      |  BYTE bStructType : 0x03   = CanUDP_STRUCT_TYPE_COMMAND
      |  BYTE bReserve    : 0
      |  WORD wStructSize : 0x0014 = 20 byte
      |____          = Size of the "whatever-it-is" on the right side.
```

There may be more than one such entry in a single UDP datagram.
For example, see CanUDP_SendMultipleCANMessagesFromPhysicalRxFifo() : This function uses a common header to pack up to 10 (?) CAN bus messages into a single UDP frame:

```
  struct // structure to send multiple CAN messages in a single UDP frame:
   { T_CanUDP_StructHeader hdr;
     #define L_MAX_TX_CAN_MSG_PER_DATAGRAM 10
     TCanUDP_CAN_Message  msgs[L_MAX_TX_CAN_MSG_PER_DATAGRAM];
   } tx_block;

 ...
     n := Number of CAN messages to send in a single UDP datagram;
 ...
     // Fill out a combination of HEADER and a CAN-MESSAGE :
     tx_block.hdr.bStructType = CanUDP_STRUCT_TYPE_CAN_MESSAGE;
     tx_block.hdr.bReserve    = 0;
     tx_block.hdr.wStructSize = n * sizeof(TCanUDP_CAN_Message);
     for( i=0; i<n; ++i )
      { tx_block.msgs[i] = CanPhysicalRxFifo[j];
        j = (j+1) & (CanUDP_PHYSICAL_CAN_RX_BUFFER_LENGTH-1);
      }
...
```

<T.B.D.>