Remote control programming examples for R&S®PR100/R&S®EM100

Radiomonitoring & Radiolocation
Application Brochure | 01.00

Remote control programming examples
for R&S®PR100/R&S®EM100

ROHDE&SCHWARZ

Contents

This application brochure gives you an overview on how to include Rohde & Schwarz receivers in your own system to create an automatic measurement and monitoring process.

Products from Rohde & Schwarz

- R&S®PR100 portable receiver
- R&S®EM100 digital compact receiver

Introduction 3

Monitoring solution 4

Application 5

Specifications in brief 5

Ordering information 5

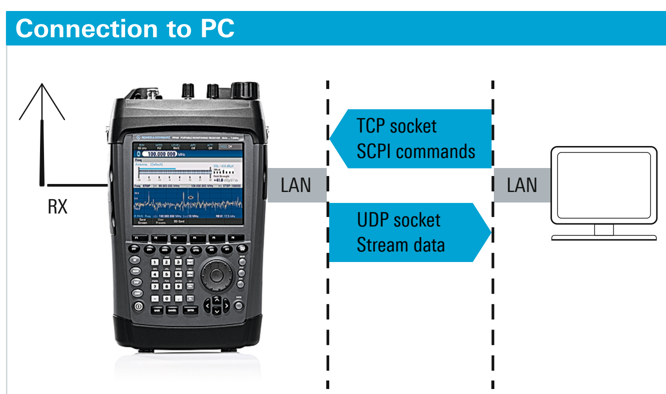
Appendix: Programming examples 6

Introduction

Your task

You want to connect and communicate with the receiver over the PC. You may already have your own system including devices which are working with different software solutions and you want to integrate the R&S®PR100/R&S®EM100 receivers into your network/process/system. You want to stream data directly from the receiver to the PC or get recorded files from the SD card. It might also be necessary to know how the recorded or streamed data is structured and which information it contains or how to transform it into the required format. You want to create your own automatic monitoring program and switch automatically between operating modes if your own conditions are true.

Connection to PC



Monitoring solution

Radiomonitoring solutions are needed whenever an unknown signal environment must be examined to gain more knowledge about the signals on air. Finding interfering signals to avoid dropped calls or any inconvenience for the user of wireless devices is another typical application for radiomonitoring solutions.

The complete functionality provided by the R&S®PR100/ R&S®EM100 receivers can also be used via remote control.

This allows you to integate the R&S®PR100 into your own system or even to program your own application for automatic measurements and monitoring tasks with the receiver.

To make it easier to understand which steps have to be done, a programming example has been created. It is written in the programming language C so you can follow every command in chronological order. To keep it comprehensible, the code is well commented. So it is easy to follow which are the basic steps for connecting the receiver and the PC over a LAN network and what is necessary to build up communications between them.

The template gives a quick overview on how to start the basic operating modes by remote control. There is at least one example of each basic application such as storing realtime spectrum data, panorama scan data, memory scan data, frequency scan data, audio data and audio live stream, complex baseband data (I/Q data) and direction finding data, as well as examples where different modes are combined.

It is possible to store the data in the SD card and transfer the file to the PC or to stream it directly into the PC. Both cases are also explained in the programming example. The code also contains the header structures for every mode. This makes it easy to get useful information out of the incoming data. In some cases it is necessary to restructure the data and put it into another format in the programming example (e.g. audio stream data is converted into a .WAV format).

Application

For applications such as air traffic control (ATC) it can be useful to combine different operating modes (e.g. memory scan with direction finding functionality). In this case the programming example shows how you can monitor up to 1024 predefined channels which use different demodulation modes and bandwidths. If there is a signal over squelch, the receiver will also store the direction data of the signal.

If you want to know which signals exist in an unknown area and where they come from, you can combine the panorama scan with the direction finding mode. The panorama scan itself gives you a quick overview of the spectrum occupancy. If any signal appears over the squelch level which you set before, the scan will give you the direction information too. It is important to know that the signal has to be there for some time, because the program first scans the entire chosen spectrum and then switches to direction finding mode.

In some situations it might save manpower when leaving the receiver alone during the monitoring process. The internal SD card combined with remote control gives you the possibility to disconnect the receiver from the PC after starting the monitoring program. It will store all the data to the SD card. When you reconnect the receiver to the PC, you can stop the recording and download the file, so no one has to be next to the receiver while recording a certain scenario.

Specifications in brief

Specifications in brief		
R&S®PR100	frequency range, base unit	9 kHz to 7.5 GHz
R&S®EM100	frequency range, base unit	9 kHz to 3.5 GHz
	with optional frequency extension	9 kHz to 7.5 GHz
Realtime bandwidth	for spectral display and waterfall	1 kHz to 10 MHz
Demodulation bandwidth	for demodulation, level measurement and I/Q streaming	150 Hz to 500 kHz
Demodulation modes	all demodulation bandwidths	AM, FM, φM, PULSE, I/Q
	IF bandwidths ≤ 9 kHz	LSB, USB, CW
	IF bandwidths ≥ 1 kHz	ISB
Panorama scan	for wideband spectral display and waterfall	up to 2 GHz/s
Frequency scan		up to 150 channels/s
Memory scan		1024 programmable memory locations
	speed	up to 120 channels/s

Ordering information

Designation	Type	Order No.
Portable Monitoring Receiver	R&S®PR100	4079.9011.02
Digital Compact Receiver	R&S®EM100	4070.4800.02

Appendix: Programming examples

Contents

1 Streaming into the PC 7
1.1 Connecting the PC to the receiver 7
1.2 Basic header of the UDP packet 11
1.3 IFPan streaming (realtime spectrum) 12
1.3.1 Settings 12
1.3.2 Optional header 12
1.3.3 Streaming 13
1.4 PScan 14
1.4.1 Settings 14
1.4.2 Optional header 15
1.4.3 Streaming 16
1.5 Audio streaming into .WAV file 18
1.5.1 Settings 18
1.5.2 Optional header 19
1.5.3 Streaming 20
1.6 Audio live streaming 23
1.6.1 Settings 23
1.6.2 Optional header 24
1.6.3 Streaming 24
1.7 I/Q data streaming 27
1.7.1 Settings 27
1.7.2 Optional header 28
1.7.3 Streaming 28
1.8 Frequency scan streaming into .WAV file 30
1.8.1 Settings 30
1.8.2 Optional header 31
1.8.3 Streaming 32
1.9 Level data 35
1.9.1 Settings 35
1.9.2 Optional header 36
1.9.3 Streaming 36
1.10 MScan 38
1.10.1 Settings 38
1.10.2 Optional header 39
1.10.3 Streaming 39
1.11 Main 40

2 Store into SD card 41
2.1 Connect 41
2.2 Settings 44
2.2.1 IFPAN 44
2.2.2 PScan 45
2.2.3 Audio stream 46
2.2.4 I/Q stream 47
2.2.5 FScan 48
2.2.6 Store level 49
2.2.7 MScan 50
2.3 Store to SD card 51
2.4 Copy from receiver to PC 52
2.5 Main 53

3 Memory scan and direction finding 54
3.1 Connecting to the receiver 54
3.1.1 Headers 54
3.1.2 Windows sockets 58
3.1.3 TCP socket 59
3.1.4 UDP socket 61
3.1.5 Connect 63
3.1.6 Sending SCPI commands 65
3.2 Settings 66
3.3 Data stream 69
3.4 Main 73

4 Panorama scan and direction finding 74
4.1 Connecting to the receiver 74
4.1.1 Headers 74
4.1.2 Windows sockets 77
4.1.3 TCP socket 78
4.1.4 UDP socket 80
4.1.5 Connect 82
4.1.6 Sending SCPI commands 83
4.2 Settings 84
4.3 Data stream 86
4.4 Main 91

1 Streaming into the PC

1.1 Connecting the PC to the receiver

```
/*****************************************************************/
/* Connecting to the Receiver                                    */
/*****************************************************************/
/***************************Header-Files*************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctime> //just necessary for Audio live streaming
#include <winsock.h>
/* winsock.h has to be included to connect with the receiver,
additionally the library wsock32.lib has to be included, for
Visual Studio:
Project -> test Properties -> Configuration Properties ->
Linker -> Input, then type in wsock32.lib to the other libraries
(other development environments may need a different workflow)*/

/***************************Functions****************************/
/***************************Network-Connection-Test***************/
int checkWinSock()
{
    /* type WSADATA is defined in WinSock.h */
       WSADATA wsaData;
       /* Testing if Network-Connection is ok */
       if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)
              /* Returning 0 if Connection is ok */
       {
              /* Tell the user that we couldn't find a usable */
              /* WinSock DLL by returning 1                    */
              return 1;
       }
       /* Confirm that the WinSock DLL supports 2.0.*/
       /* Note that if the DLL supports versions greater  */
       /* than 2.0 in addition to 2.0, it will still return */
       /* 2.0 in wVersion since that is the version we    */
       /* requested.                         */
       if (LOBYTE(wsaData.wVersion) != 2 ||
              HIBYTE(wsaData.wVersion) != 0 )
       {      /* Tell the user that we couldn't find a usable */
              /* WinSock DLL by returning 2                    */
              return 2;
       }
       /* The WinSock DLL is acceptable. Proceed. */
       return 0;
}
/*****************************************************************/
```

```c
/**************************Create TCP Socket*********************/
int createTCPSocket(SOCKET *TCPsock, unsigned long *pulRemoteAddress,
                                  unsigned short *pusPort)
{
       /* Socket Initialization */
       int iReturn = 0;
       /* Test the Network Connection (see Network-Connection-Test)*/
       iReturn = checkWinSock();

       /* Create a receiver control TCP socket */
       /* The function socket(address family, socket type, protocol) is defined
       in WinSock.h
       It builds up a socket for the connection between the receiver and the PC*/
       *TCPsock = socket(AF_INET, SOCK_STREAM, 0);
       /* Testing if there is an error */
       if(*TCPsock == SOCKET_ERROR)
       {
              iReturn = 3;
       }
       /* Set the socket options */
       int sopt = 1;
       /* The function setsockopt(socket, level, optname, optval, optlen)
       is defined in WinSock.h
       It sets the current value for a socket option associated with a socket of
       any type, in any state */
       if ( setsockopt( *TCPsock, IPPROTO_TCP, TCP_NODELAY,
              (char *)&sopt, sizeof( sopt ) ) == SOCKET_ERROR )
       {
              iReturn = 4;
       }
       /* Create socket address structure for INET address family */
       /* The struct sockaddr_in(family, type, address, port)
       is defined in WinSock.h
       This struct defines the structure of the socket, it consists of
       the address family, the address of the server you are connecting to
       and the port you want to connect */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
    /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
    /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port    = htons(*pusPort);
    /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = *pulRemoteAddress;


       /* Connect socket to receiver device */
       /* The function connect(socket, socket_addr, size of struct socket_addr)
       is defined in WinSock.h
       It is connecting the receiver and the PC over the socket */
       if (connect(*TCPsock, (struct sockaddr *)&addrDevice,
                 sizeof(addrDevice)) != 0)
       {
              iReturn = 5;
       }
       return iReturn;
}
/********************************************************************/
```

```
/***************************Create UDP Socket**********************/
int createUDPSocket(SOCKET *sock, unsigned long *pulRemoteAddress,
                                   unsigned short *pusPort)
{
       /* Socket Initialization*/
       int iRet = 0;
       /* Test the Network Connection (see Network-Connection-Test)*/
       iRet = checkWinSock();
       if (iRet != 0) return iRet;
       /* Create a receiver control TCP socket */
       /* The function socket(address family, socket type, protocol) is defined
       in WinSock.h
       It builds up a socket for the connection between the receiver and the PC*/
       *sock = socket(AF_INET, SOCK_DGRAM, 0);
       /*Testing if there is an error*/
       if (*sock == SOCKET_ERROR)
       {
              iRet = 3;
       }
       /* fill a struct which is defined in WinSock.h with information
       for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
    /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
    /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port   = htons(*pusPort);
    /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = *pulRemoteAddress;
       /* The function bind(socket, pointer to the address structure, size of the struct)
    is defined in WinSock.h
       It associates a local address with a socket */
       if ( bind(*sock, (struct sockaddr *)&addrDevice, sizeof(addrDevice)) == SOCKET_ERROR)
       {
              printf("Couldn't bind TRACe socket - errno = %d\n", WSAGetLastError());
              closesocket(*sock), *sock = INVALID_SOCKET;
              iRet = 4;
       }
       return iRet;
}
/******************************************************************/
```

```
/*************************Connect******************************/
void connect(SOCKET *pSock, SOCKET *uSock)
{
       /* IP-address of the receiver (for the TCP socket) */
       const char *pcDeviceAddress = "172.25.9.88";
       unsigned long ulRemoteAddress = inet_addr(pcDeviceAddress);
       unsigned short usSCPIPort = 5555;
       // Port number for the receiver is always 5555

       /* IP-address of the PC (for the UDP streaming) */
       const char *pcPCAddress = "172.25.10.27";
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       unsigned short usUDPPort = 9000;
       // Port for UDP Mass Data Output you should be sure that its not
    already used
       printf("Connecting to PR100...\n");
       /* Creating a TCP socket */
       int iRet = createTCPSocket(pSock, &ulRemoteAddress, &usSCPIPort);
       if (iRet != 0)
       {      /* Error returning:*/
              switch(iRet)
              {
              case 1:
                     printf("Error retrieving windows socket dll.\n");
              case 2:
                     printf("Error retrieving correct winsock version 2.0.\n");
              case 3:
                     printf("Error creating TCP socket.\n");
              case 4:
                     printf("setsockopt (TCP_NODELAY) failed - errno %d\n",
                            WSAGetLastError());
              case 5:
                     printf("Error connecting to %s [SCPI port = %d]\n",
                  pcDeviceAddress, usSCPIPort);
              }
       }
       else
       {
              printf("TCP Socket created!\n");
       }
       /* Creating an UDP Socket */
       int iUDP = createUDPSocket(uSock, &ulPCAddress, &usUDPPort);
       if (iUDP != 0)
       {      /* Error returning: */
              switch(iUDP)
              {
              case 1:
                     printf("Error retrieving windows socket dll.\n");
              case 2:
                     printf("Error retrieving correct winsock version 2.0.\n");
              case 3:
                     printf("Error creating UDP socket.\n");
              case 4:
                     printf("\n");
              }
       }
       else
       {
              printf("UDP Socket created!\n");
       }
}
/**************************************************************/
```

```
/**************************send string**************************/
int sendSCPI(int sd, char *pBuffer)
{
       unsigned int nLen;
       /* The function send(socket, buffer, size of buffer, flags)
       is defined in WinSock.h
       It sends bytes to the receiver over the socket and returns the
       number of bytes sent */
       nLen = send(sd, pBuffer, strlen(pBuffer), 0);
       /* Testing if all bytes are sent */
       if (nLen != strlen(pBuffer))
       {
              printf("Error writing to socket. Len = %d\n", nLen);
       }
       return nLen;
}
/*******************************************************************/
```

1.2 Basic header of the UDP packet

Every UDP packet has an EB200 header and an attribute header. If the flag Optional Header is activated, then every packet also includes an optional header. The structure of the optional header depends on the activated tag.

```
/**************************Basic Header*************************/
typedef struct Eb200Header
{
       unsigned long MagicNumber;
       unsigned short      VersionMinor;
       unsigned short      VersionMajor;
       unsigned short      SeqNumber;
       unsigned short      Reserved;
    unsigned long    DataSize;
} EB200_HEADER_TYPE;
#define EB200_HEADER_SIZE  (sizeof( EB200_HEADER_TYPE ))

typedef struct UdpDatagramAttribute
{
       unsigned short      Tag;
       unsigned short      Length;
       unsigned short      NumItems;
       unsigned char ChannelNumber;
       unsigned char OptHeaderLength;
       unsigned long SelectorFlags;
       unsigned char OptHeader[22];
} UDP_DATAGRAM_ATTRIBUTE_TYPE;
#define ATTRIBUTE_HEADER_SIZE  (sizeof( UDP_DATAGRAM_ATTRIBUTE_TYPE ))

/*******************************************************************/
```

1.3 IFPan streaming (realtime spectrum)

1.3.1 Settings

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
	/* Get the basic information about the receiver with the
	SCPI command *idn? */
	sendSCPI(TCPsock, "*idn?\n");
	char pRxBuf[256];
	int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
	if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
	puts( pRxBuf );
	/* sending SCPI commands for the needed settings */
	/* set the frequency mode to fixed */
	sendSCPI(TCPsock, "frequency:mode fixed\n");
	/* set the frequency to 98.5 MHz */
	sendSCPI(TCPsock, "frequency 98.5 MHz\n");
	/* set the IF spectrum to 10 MHz */
	sendSCPI(TCPsock, "sense:freq:span 10 MHz\n");
	/* set the IF spectrum mode to average */
	sendSCPI(TCPsock, "calculate:ifpan:average:type scalar\n");
	/* set the measurement time to 50 ms */
	sendSCPI(TCPsock, "measure:time 50 ms\n");
	/* set the measurement mode to periodic */
	sendSCPI(TCPsock, "measure:mode periodic\n");
	/* switch attenuation off*/
	sendSCPI(TCPsock, "input:attenuation:state off\n");
	/* switch the automatic frequency control off*/
	sendSCPI(TCPsock, "sense:frequency:afc off\n");
	/* set the needed tags and flags for IFPan streaming */
	sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, ifpan\n");
	sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'volt:ac',
    'swap', 'opt'\n");
}
/****************************************************************/
```

1.3.2 Optional header

The optional header for the IFPan streaming is only necessary if the information inside the header is needed or if the headers must be parsed because only the raw data is needed.
The structure of the optional header is shown below.

```
/**************************Optional Header***********************/
typedef struct OptHeaderIFPan
{
	unsigned long		Freq_low;
	unsigned long		FSpan;
	short			AvgTime;
	short			AvgType;
	unsigned long		MeasureTime;
	unsigned long		Freq_high;
	signed long		DemodFreqChannel;
	unsigned long		DemodFreq_low;
	unsigned long		DemodFreq_high;
	DWORDLONG		OutputTimestamp; /* nanoseconds since Jan 1st, 1970,
					   without leap seconds */
```

```
} OPT_HEADER_IFPAN_TYPE;
#define OPT_HEADER_IFPAN_SIZE (sizeof(OPT_HEADER_IFPAN_TYPE))
/*******************************************************************/
```

1.3.3 Streaming

Otherwise it is possible to store the whole UDP package (without parsing any headers) in the file, like in this example:

```
/***************************streaming*****************************/
void streaming(SOCKET UDPsock)
{
       /* create new File */
       FILE * stream;
       /* open File */
       if((stream = fopen("testfile_IFPan.rtr","w+b"))==NULL)
       {
              printf("\nCould not open File");
              exit(1);
       }

       /* create buffer for receiving */
       char buffer[0x80000];
       /* IP address of the PC */
       const char *pcPCAddress = "172.25.10.27";
       /* bring the IP address in the right format for WinSock.h functions */
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       /* port for UDP Mass Data Output */
       unsigned short usUDPPort = 9000;

       /* fill a struct with information for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
       /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
       /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port   = htons(usUDPPort);
       /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = ulPCAddress;
       /* size of the socket address struct */
       int remlen = sizeof(sockaddr_in);
       /* counter variable for the amount of packages */
       int packages = 0;
       /* variable for the returning value of the fwrite function */
       int stor = 0;
       /* receiving loop in this example for receiving 500 UDP packages */
       while(packages < 500)
       {
              /* The function recvfrom(socket, buffer, size of buffer, flags,
              socket address struct, size of socket address) is defined in WinSock.h
              It receives the UDP packet from the receiver and returns the
        number of received bytes */
              int reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
                     (struct sockaddr *)&addrDevice, &remlen);
              if(reclen==SOCKET_ERROR)
              {
                     printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
              }
```

```
            /* compare the length of the received data with the datasize
         in the EB200 Header */
            EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
            int datasize = ntohl(pEb200Header->DataSize);
            if(reclen != datasize)
            {
                   printf("Error: received bytes: %d != data size %d\n",
                   reclen, datasize);
                   exit(1);
            }
            /* write the buffer content into the File */
            if(fwrite(buffer, reclen, 1, stream) != 1)
                   printf("Error: received less bytes than expected!");
            /* count up for the next package */
            packages++;
       }
       /* close the File */
       fclose(stream);
}
/*******************************************************************/
```

1.4 PScan

1.4.1 Settings

```
/****************************settings********************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to PScan */
       sendSCPI(TCPsock, "frequency:mode pscan\n");
       /* set the start frequency to 880 MHz */
       sendSCPI(TCPsock, "frequency:pscan:start 880 MHz\n");
       /* set the stop frequency to 960 MHz*/
       sendSCPI(TCPsock, "frequency:pscan:stop 960 MHz\n");
       /* set the number of Scans to infinite */
       sendSCPI(TCPsock, "pscan:count infinity\n");
       /* set the step size of the PScan to 50 kHz */
       sendSCPI(TCPsock, "sense:pscan:step 50 kHz\n");
       /* set the measurement time to 1 ms */
       sendSCPI(TCPsock, "measure:time 1 ms\n");
       /* set the measurement mode to periodic */
       sendSCPI(TCPsock, "measure:mode periodic\n");
       /* switch attenuation off*/
    sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* Set the needed tags and flags for the PScan streaming */
       sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, pscan\n");
       sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'freq:low:rx',
       'freq:high:rx', 'volt:ac', 'swap', 'opt'\n");
}
/*******************************************************************/
```

1.4.2 Optional header

```
/*************************Optional Header***********************/
typedef struct OptHeaderPScan
{
	unsigned long		StartFreq_low;
	unsigned long		StopFreq_low;
	unsigned long		StepFreq;
	unsigned long		StartFreq_high;
	unsigned long		StopFreq_high;
	char			reserved[4];
	DWORDLONG		OutputTimestamp; /* nanoseconds since Jan 1st, 1970,
						  without leap seconds */
} OPT_HEADER_PSCAN_TYPE;
/**************************************************************/
```

1.4.3 Streaming

Otherwise it is possible to store the whole UDP package (without parsing any headers) in the file, like in this example:

```
/*************************streaming****************************/
void streaming(SOCKET UDPsock, SOCKET TCPsock)
{
	/* create new File */
	FILE * stream;
	/* open File */
	if((stream = fopen("testfile_PScan.rtr","w+b"))==NULL)
	{
		printf("\nCould not open File");
		exit(1);
	}

	/* create buffer for receiving */
	char buffer[0x80000];
	/* IP address of the PC */
	const char *pcPCAddress = "172.25.10.27";
	/* bring the IP address in the right format for WinSock.h functions */
	unsigned long ulPCAddress = inet_addr(pcPCAddress);
	/* port for UDP Mass Data Output */
	unsigned short usUDPPort = 9000;

	/* fill a struct which is defined in WinSock.h with information
	for the socket address format */
	sockaddr_in addrDevice;
	/* reserve memory space for the address struct */
	memset(&addrDevice, 0, sizeof(addrDevice));
	/* setting for TCP/UDP address family */
	addrDevice.sin_family = AF_INET;
	/* put the information for the UDP Mass Data Output port into the struct */
	addrDevice.sin_port    = htons(usUDPPort);
	/* put the information for socket address into the struct */
	addrDevice.sin_addr.s_addr = ulPCAddress;
	/* size of the socket address struct */
	int remlen = sizeof(sockaddr_in);
	/* counter variable for the amount of packages */
	int packages = 0;
	/* start the PScan */
	sendSCPI(TCPsock, "init\n");
	Sleep(2000); // otherwise the program is too fast

	/* receiving loop in this example for receiving 500 UDP packages */
	while(packages < 500)
	{
		/* The function recvfrom(socket, buffer, size of buffer, flags,
		socket address struct, size of socket address) is defined in WinSock.h
		It receives the UDP packet from the receiver and returns the number
	 of received bytes */
		int reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
			(struct sockaddr *)&addrDevice, &remlen);
		if(reclen==SOCKET_ERROR)
		{
			printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
		}
```

```
            /* compare the length of the received data with the datasize in the
        EB200 Header */
            EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
            int datasize = ntohl(pEb200Header->DataSize);
            if(reclen != datasize)
            {
                printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
                exit(1);
            }
            /* write the buffer content into the File */
            if(fwrite(buffer, reclen, 1, stream) != 1)
                printf("Error: received less bytes than expected!");
            /* count up for the next package */
            packages++;
        }
        /* stop the Scan */
        sendSCPI(TCPsock, "abort\n");
        fclose(stream);
}
/*******************************************************************/
```

1.5 Audio streaming into .WAV file

1.5.1 Settings

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to fixed */
       sendSCPI(TCPsock, "frequency:mode fixed\n");
       /* set the frequency to 95 MHz */
       sendSCPI(TCPsock, "frequency 95 MHz\n");
       /* set the bandwidth to 120 kHz */
       sendSCPI(TCPsock, "bandwidth 120 kHz\n");
       /* set the demodulation mode to fm */
       sendSCPI(TCPsock, "demodulation fm\n");
       /* set the measurement time to default */
       sendSCPI(TCPsock, "measure:time default\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
    /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch the automatic frequency control on */
       sendSCPI(TCPsock, "sense:frequency:afc on\n");
       /* set gcontrol to automatically generated */
       sendSCPI(TCPsock, "sense:gcontrol:mode agc\n");
       /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
       /* choose the audio mode, see table 6-8 in the manual */
       sendSCPI(TCPsock, "system:audio:remote:mode 2\n");
       /* set the needed tags and flags for audio streaming */
       sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, audio\n");
       sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'opt'\n");
}
/****************************************************************/
```

1.5.2 Optional header

To stream UDP packages with audio data in a .WAV format file, the optional header must be parsed and the .WAV format header has to be implemented. The structure of the optional header and the .WAV headers is shown below.

```
/***************************Optional Header************************/
typedef struct OptHeaderAudio
{
    unsigned short      AudioMode;
    unsigned short      FrameLength;
    unsigned long       Freq_low;
    unsigned long       Bandwidth;
    unsigned short      Demodulation;
    char                DemodulationString[8];
    unsigned long       Freq_high;
    char                reserved[6];
    DWORDLONG           OutputTimestamp; /* nanoseconds since Jan 1st, 1970, without leap sec-
onds */
} OPT_HEADER_AUDIO_TYPE;
/*******************************************************************/
/**************************WAV Header****************************/
/* structures for the WAV Header format */
struct struRIFFHeader
{
       char     cRIFFID[4];     // RIFF file id ('RIFF')
       unsigned unLength;       // Length in bytes after RIFF header
       char     cFormatID[4];   // Format id ('WAVE' for .WAV files)
};

struct struChunkHeader
{
       char     cChunkID[4];
       unsigned unChunkSize;    // Chunk length in bytes excluding header
};

struct struFormatChunkBody
{
       char                    ChunkID[4];     // 'fmt '
       unsigned                fmtLength;
       short            sFormatTag;
       unsigned short   usChannels;
       unsigned                unSamplesPerSec;
       unsigned                unAvgBytesPerSec;
       short            sBlockAlign;
       short            sBitsPerSample;
  // Note: there may be additional fields here, depending upon wFormatTag
};
#define WAV_HEADER ((sizeof(struRIFFHeader)+sizeof(struChunkHeader)+sizeof(struFormatChunkBody))-8)
/*******************************************************************/
```

1.5.3 Streaming

In this example the headers of the UDP packages are parsed and the raw data is saved first. Then a .WAV header is added to the entire received raw data.

```
/*************************streaming****************************/
void streaming(SOCKET usock)
{
	/* create new File */
	FILE * stream;
	/* open File */
	if((stream = fopen("testfile_Audio.wav","w+b"))==NULL)
	{
		printf("\nCould not open File");
		exit(1);
	}
	/* create buffer for receiving */
	char buffer[0x80000];
	/* IP address of the PC */
	const char *pcPCAddress = "172.25.10.27";
	/* bring the IP address in the right format for WinSock.h functions */
	unsigned long ulPCAddress = inet_addr(pcPCAddress);
	/* port for UDP Mass Data Output */
	unsigned short usUDPPort = 9000;
	/* fill a struct which is defined in WinSock.h with information
	for the socket address format */
	sockaddr_in addrDevice;
	/* reserve memory space for the address struct */
	memset(&addrDevice, 0, sizeof(addrDevice));
	/* setting for TCP/UDP address family */
	addrDevice.sin_family = AF_INET;
	/* put the information for the UDP Mass Data Output port into the struct */
	addrDevice.sin_port   = htons(usUDPPort);
	/* put the information for socket address into the struct */
	addrDevice.sin_addr.s_addr = ulPCAddress;
	/* size of the socket address struct */
	int remlen = sizeof(sockaddr_in);

	/* write a WAV Header "dummy" to the beginning of the file */
	fseek(stream, 0, SEEK_SET);
	/* RIFF Header */
	struRIFFHeader ctRIFFHeader;
		strncpy(ctRIFFHeader.cFormatID, "WAVE",4);
		strncpy(ctRIFFHeader.cRIFFID,"RIFF",4);
		ctRIFFHeader.unLength             = WAV_HEADER;
	/* write the RIFF Header into the file */
	if (fwrite(&ctRIFFHeader, sizeof(struRIFFHeader), 1, stream) != 1)
		printf("Error: received less bytes than expected!");
	/* the settings of the Format Header depend on the chosen audio mode */
	/* Format Header */
	struFormatChunkBody ctFmtChunkBody;
		strncpy(ctFmtChunkBody.ChunkID, "fmt ",4);
		ctFmtChunkBody.fmtLength          =      16;
		ctFmtChunkBody.sFormatTag       = 0x0001;
		ctFmtChunkBody.usChannels       = 1; // 1-Kanal
		ctFmtChunkBody.unSamplesPerSec  = 32000;
		ctFmtChunkBody.sBlockAlign      = 2;
		ctFmtChunkBody.unAvgBytesPerSec = 64000;
		ctFmtChunkBody.sBitsPerSample   = 16;
	/* write the Format Header into the file */
	if (fwrite(&ctFmtChunkBody, sizeof(struFormatChunkBody), 1, stream) != 1)
		printf("Error: received less bytes than expected!");
```

```
/* Chunk Header */
struChunkHeader ctChunkHeader;
strncpy(ctChunkHeader.cChunkID, "data",4);
ctChunkHeader.unChunkSize = 0;
/* write the Chunk Header into the file */
if (fwrite(&ctChunkHeader, sizeof(struChunkHeader), 1, stream) != 1)
       printf("Error: received less bytes than expected!");

/* variables */
/* counter variable for the amount of packages */
int packages = 0;
/* rawAudio counts the received raw Audio Bytes per package */
int rawAudio = 0;
/*rawAudioSum counts the totally received raw Audio Bytes */
int rawAudioSum = 0;
/* receiving loop in this example for receiving 500 UDP packages */
while(packages < 500)
{
       /* The function recvfrom(socket, buffer, size of buffer, flags,
       socket address struct, size of socket address) is defined in WinSock.h
       It receives the UDP packet from the receiver and returns the number
 of received bytes */
       int reclen = recvfrom(usock, buffer,sizeof(buffer),0,
              (struct sockaddr *)&addrDevice, &remlen);
       if(reclen==SOCKET_ERROR)
       {
              printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
       }
       /* compare the length of the received data with the datasize in
 the EB200 Header */
       EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
       int datasize = ntohl(pEb200Header->DataSize);
       if(reclen != datasize)
       {
              printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
       }
       /* getting the size of the Optional Header */
       UDP_DATAGRAM_ATTRIBUTE_TYPE *attrHeader =
 (UDP_DATAGRAM_ATTRIBUTE_TYPE*)(buffer + EB200_HEADER_SIZE);
       /* always read out the length of the Optional Header from the Attribute
 Header because in case something is changing in the Opt. Header it fits
 automatically */
 /* calculate the size ot the Headers */
       int headersize = EB200_HEADER_SIZE + ATTRIBUTE_HEADER_SIZE
                 + (attrHeader->OptHeaderLength);
       /* pointer to the beginning of the raw Audio Data */
       char *audio = buffer + headersize;
       /* calculate the length of the raw Audio Data */
       rawAudio = reclen - headersize;
       /* calculate the sum of the received raw Audio Data */
       rawAudioSum = rawAudioSum + rawAudio;
       /* store the raw Audio Data in the File */
       if(fwrite(audio, rawAudio, 1, stream)!=1)
              printf("Error: received less bytes than expected!");
       /* increase the packages */
       packages ++;
}
/* fill the WAV Header with the information about the Audio Data */
fseek(stream, 0, SEEK_SET);
/* RIFF Header */
strncpy(ctRIFFHeader.cFormatID, "WAVE",4);
strncpy(ctRIFFHeader.cRIFFID,"RIFF",4);
```

```
        ctRIFFHeader.unLength = WAV_HEADER + rawAudioSum;
        /* write the RIFF Header into the file */
        if (fwrite(&ctRIFFHeader, sizeof(struRIFFHeader), 1, stream) != 1)
                printf("Error: received less bytes than expected!");
        /* the settings of the Format Header depend on the chosen audio mode */
        /* Format Header */
        strncpy(ctFmtChunkBody.ChunkID, "fmt ",4);
        ctFmtChunkBody.fmtLength           =      16;
        ctFmtChunkBody.sFormatTag       = 0x0001;
        ctFmtChunkBody.usChannels       = 1;
        ctFmtChunkBody.unSamplesPerSec  = 32000;
        ctFmtChunkBody.sBlockAlign      = 2;
        ctFmtChunkBody.unAvgBytesPerSec = 64000;
        ctFmtChunkBody.sBitsPerSample   = 16;
        /* write the Format Header into the file */
        if (fwrite(&ctFmtChunkBody, sizeof(struFormatChunkBody), 1, stream) != 1)
                printf("Error: received less bytes than expected!");
        /* Chunk Header */
        strncpy(ctChunkHeader.cChunkID, "data",4);
        ctChunkHeader.unChunkSize = rawAudioSum;
        /* write the Chunk Header into the file */
        if (fwrite(&ctChunkHeader, sizeof(struChunkHeader), 1, stream) != 1)
                printf("Error: received less bytes than expected!");
        /* close the File */
        fclose(stream);
}
/*******************************************************************/
```

1.6 Audio live streaming

1.6.1 Settings

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
	/* Get the basic information about the receiver with the
	SCPI command *idn? */
	sendSCPI(TCPsock, "*idn?\n");
	char pRxBuf[256];
	int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
	if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
	puts( pRxBuf );
	/* sending SCPI commands for the needed settings */
	/* set the frequency mode to fixed */
	sendSCPI(TCPsock, "frequency:mode fixed\n");
	/* set the frequency to 95 MHz */
	sendSCPI(TCPsock, "frequency 95 MHz\n");
	/* set the bandwidth to 120 kHz */
	sendSCPI(TCPsock, "bandwidth 120 kHz\n");
	/* set the demodulation mode to fm */
	sendSCPI(TCPsock, "demodulation fm\n");
	/* set the measurement time to default */
	sendSCPI(TCPsock, "measure:time default\n");
	/* set the measurement mode to continuous */
	sendSCPI(TCPsock, "measure:mode continuous\n");
	/* switch attenuation off*/
	sendSCPI(TCPsock, "input:attenuation:state off\n");
	/* switch the automatic frequency control on */
	sendSCPI(TCPsock, "sense:frequency:afc on\n");
	/* set gcontrol to automatically generated */
	sendSCPI(TCPsock, "sense:gcontrol:mode agc\n");
	/* switch tone off */
	sendSCPI(TCPsock, "output:tone off\n");
	/* choose the audio mode, see table 6-8 in the manual */
	sendSCPI(TCPsock, "system:audio:remote:mode 2\n");
	/* set the needed tags and flags for audio streaming */
	sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, audio\n");
	sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'opt'\n");
}
/******************************************************************/
```

1.6.2 Optional header

```
/*************************Optional Header***********************/
typedef struct OptHeaderAudio
{
    unsigned short AudioMode;
    unsigned short FrameLength;
    unsigned long Freq_low;
    unsigned long Bandwidth;
    unsigned short Demodulation;
    char DemodulationString[8];
    unsigned long Freq_high;
    char reserved[6];
    DWORDLONG OutputTimestamp; /* nanoseconds since Jan 1st, 1970, without
    leap seconds */
} OPT_HEADER_AUDIO_TYPE;
/**************************************************************/
```

1.6.3 Streaming

```
/*************************streaming****************************/
void streaming(SOCKET usock)
{
       /* create buffer for receiving */
       char buffer[0x80000];
       /* IP address of the PC */
       const char *pcPCAddress = "172.25.10.27";
       /* bring the IP address in the right format for WinSock.h functions */
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       /* port for UDP Mass Data Output */
       unsigned short usUDPPort = 9000;
       /* fill a struct which is defined in WinSock.h with information
       for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
       /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
       /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port   = htons(usUDPPort);
       /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = ulPCAddress;
       /* size of the socket address struct */
       int remlen = sizeof(sockaddr_in);

       /* declare a variable sound of the structure WAVEFORMATEX which is
       defined in the MMSystem.h header file.
       The settings for the structure depend on the chosen audio mode */
       WAVEFORMATEX  sound;
              sound.wFormatTag = WAVE_FORMAT_PCM; // PCM (pulse-code modulated) data
                                                 in integer format
              sound.nChannels = 1;
              sound.nSamplesPerSec = 32000;
              sound.nAvgBytesPerSec = 64000;
              sound.nBlockAlign = 2;
              sound.wBitsPerSample = 16;
              sound.cbSize = 0;
       /* declare a handle to the waveform-audio output device */
       HWAVEOUT test;
```

```
/* declare a variable audiodata of the structure WAVEHDR which is
defined in the MMSystem.h header file. */
WAVEHDR audiodata[1000];
/* The function waveOutOpen opens the given waveform-audio output
device for playback, it is defined in MMSystem.h */
waveOutOpen(&test, WAVE_MAPPER, &sound, 0, 0, CALLBACK_NULL);

/* number of UDP packages */
int packages = 0;
/* number of raw audio data */
int rawAudio = 0;
/* sum of the audio data */
int rawAudioSum = 0;
/* number of received data */
int datasize = 0;
int headersize = 0;
/* receiving depending on UDP packages */
while(packages < 1000)
{
       /* The function recvfrom(socket, buffer, size of buffer, flags,
       socket address struct, size of socket address) is defined in WinSock.h
       It receives the UDP packet from the receiver and returns the number of
 received bytes */
       int reclen = recvfrom(usock, buffer,sizeof(buffer),0,
              (struct sockaddr *)&addrDevice, &remlen);
       if(reclen==SOCKET_ERROR)
       {
              printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
       }
       /* compare the length of the received data with the datasize in the
 EB200 Header */
       EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
       datasize = ntohl(pEb200Header->DataSize);
       if(reclen != datasize)
       {
              printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
       }

       /* getting the size of the Optional Header */
       UDP_DATAGRAM_ATTRIBUTE_TYPE *attrHeader =
 (UDP_DATAGRAM_ATTRIBUTE_TYPE*)(buffer + EB200_HEADER_SIZE);
       /* always read out the length of the Optional Header from the Attribute
 Header because in case something is changing in the Opt. Header it fits
 automatically */
 /* calculate the size ot the Headers */
       int headersize = EB200_HEADER_SIZE + ATTRIBUTE_HEADER_SIZE
                  + (attrHeader->OptHeaderLength);
       /* parsing the headers to get the raw audio data */
       char *audio = buffer + headersize;
       /* number of raw audio data */
       rawAudio = reclen - headersize;
       /* fill the WAVEHDR struct audiodata with information */
       audiodata[packages].lpData = (LPSTR)malloc(rawAudio);
       memcpy(audiodata[packages].lpData, audio, rawAudio);
       audiodata[packages].dwBufferLength = rawAudio;         /* length of data buffer */
       audiodata[packages].dwFlags = 0;
       audiodata[packages].dwUser = 0;
       /* The function waveOutPrepareHeader prepares a waveform-audio data block
       for playback, it is defined in MMSystem.h */
       waveOutPrepareHeader(test, &audiodata[packages], sizeof(audiodata[packages]));
```

```
		/* The function waveOutWrite sends a data block to the given waveform-audio
		output device, it is defined in MMSystem.h */
		waveOutWrite(test, &audiodata[packages], sizeof(audiodata[packages]));
		/* increase the number of received packages */
		packages ++;
	}
	/* counter variable for the amount of unprepared headers */
	int unprep = 0;
	/* unprepare the wave out headers */
	while(unprep < packages)
	{
		/* wait if the data is not replayed yet */
		do{
		}while(!(audiodata[unprep].dwFlags & WHDR_DONE));
		/* The function waveOutUnprepareHeader cleans up the preparation performed
		by the waveOutPrepareHeader function. This function must be called after the
		device driver is finished with a data block. It is defined in MMSystem.h */
		waveOutUnprepareHeader(test, &audiodata[unprep], sizeof(audiodata[unprep]));
		unprep++;
	}
	/* The waveOutClose function closes the given waveform-audio output device.
	It is defined in MMSystem.h */
	waveOutClose(test);
}
/*******************************************************************/
```

1.7 I/Q data streaming

1.7.1 Settings

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to fixed */
       sendSCPI(TCPsock, "frequency:mode fixed\n");
       /* set the frequency to 300 MHz */
       sendSCPI(TCPsock, "frequency 390 MHz\n");
       /* set the bandwidth to 500 kHz */
       sendSCPI(TCPsock, "bandwidth 500 kHz\n");
       /* set the demodulation mode to iq */
       sendSCPI(TCPsock, "demodulation iq\n");
       /* set the measurement time to default */
       sendSCPI(TCPsock, "measure:time def\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch the automatic frequency control off */
       sendSCPI(TCPsock, "sense:frequency:afc off\n");
       /* switch the automatic gain control off */
       sendSCPI(TCPsock, "gcontrol:mode mgc\n");
       sendSCPI(TCPsock, "gcontrol 0\n");
       /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
       /* set the needed tags and flags for IQ data streaming */
       sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, if\n");
       sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'volt:ac',
            'swap', 'opt'\n");
}
/******************************************************************/
```

1.7.2 Optional header

The optional header for the PScan streaming is only necessary if the information inside the header is needed or if the headers must be parsed because only the raw data is needed. The structure of the optional header is shown below.

```
/**************************Optional Header************************/
typedef struct OptHeaderIF
{
       unsigned short        IFMode;
       unsigned short        FrameLength;
       unsigned long         SamplerRate;
       unsigned long         Freq_low;
       unsigned long         Bandwidth;
       unsigned short        Demodulation;
       short                 RxAttenuation;
       unsigned short        Flags;
       short                 kFactor;
       char                  DemodulationString[8];
       DWORDLONG             SampleCount;
       unsigned long         Freq_high;
       char                  reserved2[4];
       DWORDLONG             StartTimestamp; /* nanoseconds since Jan 1st, 1970,
                                                without leap seconds */
} OPT_HEADER_IF_TYPE;
/*****************************************************************/
```

1.7.3 Streaming

Otherwise it is possible to store the whole UDP package (without parsing any headers) in the file, like in this example:

```
/**************************streaming*****************************/
void streaming(SOCKET UDPsock)
{
       /* create new File */
       FILE * stream;
       /* open File */
       if((stream = fopen("testfile_IQdata.riq","w+b"))==NULL)
       {
              printf("\nCould not open File");
              exit(1);
       }
       /* create buffer for receiving */
       char buffer[0x80000];
       /* IP address of the PC */
       const char *pcPCAddress = "172.25.10.27";
       /* bring the IP address in the right format for WinSock.h functions */
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       /* port for UDP Mass Data Output */
       unsigned short usUDPPort = 9000;
       /* fill a struct which is defined in WinSock.h with information
       for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
       /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
       /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port    = htons(usUDPPort);
```

```
        /* put the information for socket address into the struct */
        addrDevice.sin_addr.s_addr = ulPCAddress;
        /* size of the socket address struct */
        int remlen = sizeof(sockaddr_in);
        /* counter variable for the amount of packages */
        int packages = 0;
        /* receiving loop in this example for receiving 500 UDP packages */
        while(packages < 500)
        {
               /* The function recvfrom(socket, buffer, size of buffer, flags,
               socket address struct, size of socket address) is defined in WinSock.h
               It receives the UDP packet from the receiver and returns the number of
         received bytes */
               int reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
                      (struct sockaddr *)&addrDevice, &remlen);
               if(reclen==SOCKET_ERROR)
               {
                      printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
               }
               /* compare the length of the received data with the datasize in the
         EB200 Header */
               EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
               int datasize = ntohl(pEb200Header->DataSize);
               if(reclen != datasize)
               {
                      printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
                      exit(1);
               }
               /* write the buffer content into the File */
               if(fwrite(buffer, reclen, 1, stream)!=1)
                      printf("Error: received less bytes than expected!");
               /* count up for the next package */
               packages++;
        }
        /* close the File */
        fclose(stream);
}
/*******************************************************************/
```

1.8 Frequency scan streaming into .WAV file

1.8.1 Settings

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
    /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to sweep */
       sendSCPI(TCPsock, "frequency:mode sweep\n");
       /* number of sweeps is set to 10 */
       sendSCPI(TCPsock, "sweep:count inf\n");
       /* set the start frequency to 118 MHz */
       sendSCPI(TCPsock, "frequency:start 118 MHz\n");
       /* set the stop frequency to 138 MHz */
       sendSCPI(TCPsock, "frequency:stop 138 MHz\n");
       /* set the frequency stepwidth to 25 kHz*/
       sendSCPI(TCPsock, "sweep:step 25 kHz\n");
       /* switch on sweep control */
       sendSCPI(TCPsock, "sweep:control on\n");
       /* set dwel time to 3 s */
       sendSCPI(TCPsock, "sweep:dwel 3 s\n");
       /* set hold time to 1 s */
       sendSCPI(TCPsock, "sweep:hold:time 1 s\n");
       /* switch on squelch */
       sendSCPI(TCPsock, "output:squelch on\n");
       /* set squelch threshold to 10 dbuV */
       sendSCPI(TCPsock, "output:squelch:threshold 10 dbuV\n");
       /* set the bandwidth to 9 kHz */
       sendSCPI(TCPsock, "bandwidth 9 kHz\n");
       /* set the demodulation mode to am */
       sendSCPI(TCPsock, "demodulation am\n");
       /* set the measurement time to default */
       sendSCPI(TCPsock, "measure:time default\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch automatic frequency control off */
       sendSCPI(TCPsock, "sense:frequency:afc off\n");
       /* set gcontrol to automatically generated */
       sendSCPI(TCPsock, "sense:gcontrol:mode agc\n");
       /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
       /* choose the audio mode, see table 6-8 in the manual */
       sendSCPI(TCPsock, "system:audio:remote:mode 2\n");
       /* set the needed tags and flags for audio and fscan streaming */
       sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, fscan, audio\n");
    sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'freq:low:rx', 'freq:high:rx',
    'volt:ac', 'opt', 'swap'\n");
}
/*******************************************************************/
```

1.8.2 Optional header

```
/*************************Optional Header***********************/
typedef struct OptHeaderFScan
{
	short		CycleCount;
	short		HoldTime;
	short		DwellTime;
	short		DirectionUp;
	short		StopSignal;
	unsigned long	StartFreq_low;
	unsigned long	StopFreq_low;
	unsigned long	StepFreq;
	unsigned long	StartFreq_high;
	unsigned long	StopFreq_high;
	char		reserved[2];
	DWORDLONG	OutputTimestamp; /* nanoseconds since Jan 1st, 1970 */
	short			level;
	unsigned long	 FreqLow;
	unsigned long	 FreqHigh;
} OPT_HEADER_FSCAN_TYPE;

typedef struct OptHeaderAudio
{
	unsigned short	AudioMode;
	unsigned short	FrameLength;
	unsigned long	Freq_low;
	unsigned long	Bandwidth;
	unsigned short	Demodulation;
	char		DemodulationString[8];
	unsigned long	Freq_high;
	char		reserved[6];
	DWORDLONG	OutputTimestamp; /* nanoseconds since Jan 1st, 1970 */
} OPT_HEADER_AUDIO_TYPE;

/***************************************************************/
/*************************WAV Header***************************/
/* structures for the WAV Header format */
struct struRIFFHeader
{
	char	cRIFFID[4];	// RIFF file id ('RIFF')
	unsigned unLength;	// Length in bytes after RIFF header
	char	cFormatID[4];	// Format id ('WAVE' for .WAV files)
};

struct struChunkHeader
{
	char	cChunkID[4];
	unsigned unChunkSize;	// Chunk length in bytes excluding header
};
struct struFormatChunkBody
{
	char		ChunkID[4];	// 'fmt '
	unsigned	fmtLength;
	short		sFormatTag;
	unsigned short	usChannels;
	unsigned	unSamplesPerSec;
	unsigned	unAvgBytesPerSec;
	short		sBlockAlign;
	short		sBitsPerSample;
};
```

```
#define WAV_HEADER ((sizeof(struRIFFHeader)+sizeof(struChunkHeader)+sizeof(struFormatChunkBody))-8)
/*******************************************************************/
```

1.8.3 Streaming

```
/***************************streaming*****************************/
void streaming(SOCKET UDPsock, SOCKET TCPsock)
{
       /* create File */
       FILE * stream;
       /* open File */
       if((stream = fopen("testfile_FScan.wav","w+b"))==NULL)
       {
              printf("\nCould not open File");
              exit(1);
       }
       /* create buffer for receiving */
       char buffer[0x80000];
       /* IP address of the PC */
       const char *pcPCAddress = "172.25.10.27";
       /* bring the IP address in the right format for WinSock.h functions */
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       /* port for UDP Mass Data Output */
       unsigned short usUDPPort = 9000;
       /* fill a struct which is defined in WinSock.h with information
       for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
       /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
       /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port    = htons(usUDPPort);
       /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = ulPCAddress;
       /* size of the socket address struct */
       int remlen = sizeof(sockaddr_in);

       /* write a WAV Header "dummy" to the beginning of the file */
       fseek(stream, 0, SEEK_SET);
       /* RIFF Header */
       struRIFFHeader ctRIFFHeader;
       strncpy(ctRIFFHeader.cFormatID, "WAVE",4);
       strncpy(ctRIFFHeader.cRIFFID,"RIFF",4);
       ctRIFFHeader.unLength               = WAV_HEADER;
       /* write the RIFF Header into the file */
       if (fwrite(&ctRIFFHeader, sizeof(struRIFFHeader), 1, stream) != 1)
              printf("Error: received less bytes than expected!");
       /* the settings of the Format Header depend on the chosen audio mode */
       /* Format Header */
       struFormatChunkBody ctFmtChunkBody;
       strncpy(ctFmtChunkBody.ChunkID, "fmt ",4);
       ctFmtChunkBody.fmtLength          =     16;
       ctFmtChunkBody.sFormatTag       = 0x0001;
       ctFmtChunkBody.usChannels       = 1; // 1-Kanal
       ctFmtChunkBody.unSamplesPerSec  = 32000;
       ctFmtChunkBody.sBlockAlign      = 2;
       ctFmtChunkBody.unAvgBytesPerSec = 64000;
       ctFmtChunkBody.sBitsPerSample   = 16;
```

```
/* write the Format Header into the file */
if (fwrite(&ctFmtChunkBody, sizeof(struFormatChunkBody), 1, stream) != 1)
       printf("Error: received less bytes than expected!");
/* Chunk Header */
struChunkHeader ctChunkHeader;
strncpy(ctChunkHeader.cChunkID, "data",4);
ctChunkHeader.unChunkSize = 0;
/* write the Chunk Header into the file */
if (fwrite(&ctChunkHeader, sizeof(struChunkHeader), 1, stream) != 1)
       printf("Error: received less bytes than expected!");

/* counter variable for the amount of packages */
int packages = 0;
/* sum of the sizes of the headers */
int headersize = 0;
/* rawAudio counts the received raw Audio Bytes per package */
int rawAudio = 0;
/*rawAudioSum counts the totally received raw Audio Bytes */
int rawAudioSum = 0;
/* Tag from the UDP packet */
int tag = 0;
/* start the FScan */
sendSCPI(TCPsock, "init\n");
Sleep(2000);
int trace;
/* receiving loop in this example for receiving 20 UDP packages */
while(packages < 3000)
{
       /* The function recvfrom(socket, buffer, size of buffer, flags,
       socket address struct, size of socket address) is defined in WinSock.h
       It receives the UDP packet from the receiver and returns the number of
 received bytes */
       int reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
              (struct sockaddr *)&addrDevice, &remlen);
       if(reclen==SOCKET_ERROR)
       {
              printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
       }
       /* compare the length of the received data with the datasize in the
 EB200 Header */
       EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
       int datasize = ntohl(pEb200Header->DataSize);
       if(reclen != datasize)
       {
              printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
       }
       UDP_DATAGRAM_ATTRIBUTE_TYPE *pAttr = (UDP_DATAGRAM_ATTRIBUTE_TYPE *)(buffer+16);
       /* get the Tag of the received UDP packet */
       tag = ntohs(pAttr->Tag);
```

```
        /* if the Tag is 401 which stands for audio, then store the raw audio
     data into the file */
        if(tag == 401)
        {
               /* always read out the length of the Optional Header from the Attribute
           Header because in case something is changing in the Opt. Header it fits
           automatically */
           /* calculate the size ot the Headers */
               headersize = EB200_HEADER_SIZE + ATTRIBUTE_HEADER_SIZE +
                   (pAttr->OptHeaderLength);
               /* pointer to the beginning of the raw Audio Data */
               char *audio = buffer + headersize;
               /* calculate the length of the raw Audio Data */
               rawAudio = reclen - headersize;
               /* calculate the sum of the received raw Audio Data */
               rawAudioSum = rawAudioSum + rawAudio;
               /* store the raw Audio Data in the File */
               if(fwrite(audio, rawAudio, 1, stream)!=1)
                      printf("Error: received less bytes than expected!");
        }
        /* increase the number of received packages */
        packages++;
    }
    /* stop the FScan */
    sendSCPI(TCPsock, "abort\n");
    /* fill the WAV Header with the information about the Audio Data */
    fseek(stream, 0, SEEK_SET);
    /* RIFF Header */
    strncpy(ctRIFFHeader.cFormatID, "WAVE",4);
    strncpy(ctRIFFHeader.cRIFFID,"RIFF",4);
    ctRIFFHeader.unLength = WAV_HEADER + rawAudioSum;
    /* write the RIFF Header into the file */
    if (fwrite(&ctRIFFHeader, sizeof(struRIFFHeader), 1, stream) != 1)
        printf("Error: received less bytes than expected!");
    /* the settings of the Format Header depend on the chosen audio mode */
    /* Format Header */
    strncpy(ctFmtChunkBody.ChunkID, "fmt ",4);
    ctFmtChunkBody.fmtLength          =      16;
    ctFmtChunkBody.sFormatTag       = 0x0001;
    ctFmtChunkBody.usChannels       = 1;
    ctFmtChunkBody.unSamplesPerSec  = 32000;
    ctFmtChunkBody.sBlockAlign      = 2;
    ctFmtChunkBody.unAvgBytesPerSec = 64000;
    ctFmtChunkBody.sBitsPerSample   = 16;
    /* write the Format Header into the file */
    if (fwrite(&ctFmtChunkBody, sizeof(struFormatChunkBody), 1, stream) != 1)
        printf("Error: received less bytes than expected!");
    /* Chunk Header */
    strncpy(ctChunkHeader.cChunkID, "data",4);
    ctChunkHeader.unChunkSize = rawAudioSum;
    /* write the Chunk Header into the file */
    if (fwrite(&ctChunkHeader, sizeof(struChunkHeader), 1, stream) != 1)
        printf("Error: received less bytes than expected!");
    /* close the File */
    fclose(stream);
}
/******************************************************************/
```

1.9 Level data

1.9.1 Settings

```
/***************************settings******************************/
void settings(SOCKET TCPsock)
{
    sendSCPI(TCPsock, "*idn?\n");
    char pRxBuf[256];
    int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
    if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
    puts( pRxBuf );
    /* sending SCPI commands for the needed settings */
    /* set the frequency mode to fixed */
    sendSCPI(TCPsock, "frequency:mode fixed\n");
    /* set the frequency to 300 MHz */
    sendSCPI(TCPsock, "frequency 95 MHz\n");
    /* set the bandwidth to 120 kHz */
    sendSCPI(TCPsock, "bandwidth 120 kHz\n");
    /* set the demodulation mode to fm */
    sendSCPI(TCPsock, "demodulation fm\n");
    /* set the measurement time to default */
    sendSCPI(TCPsock, "measure:time default\n");
    /* set the measurement mode to periodic */
    sendSCPI(TCPsock, "measure:mode periodic\n");
    /* switch attenuation off*/
    sendSCPI(TCPsock, "input:attenuation:state off\n");
    /* switch the automatic frequency control off */
    sendSCPI(TCPsock, "sense:frequency:afc off\n");
    /* switch the automatic gain control off */
    sendSCPI(TCPsock, "gcontrol:mode mgc\n");
    sendSCPI(TCPsock, "gcontrol 0\n");
    /* switch tone off */
    sendSCPI(TCPsock, "output:tone off\n");
    /* select measure root-mean-square value */
    sendSCPI(TCPsock, "sense:detector rms\n");
    /* GPS and compass settings */
    sendSCPI(TCPsock, "system:gpscompass:source gps, aux1\n");
    sendSCPI(TCPsock, "system:gpscompass:source compass, aux1\n");
    sendSCPI(TCPsock, "system:gpscompass:aux:configuration 1, 4800, 8, none, 1\n");
    sendSCPI(TCPsock, "system:gpscompass on\n");
    /* Setting the needed Tags and Flags for the Level and GPS Streaming */
    sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, ifpan, gpsc\n");
    sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'volt:ac',
    'opt', ‘swap’\n");
}
/*****************************************************************/
```

1.9.2 Optional header

```
/*************************Optional Header***********************/
typedef struct OptHeaderIFPan
{
	unsigned long		Freq_low;
	unsigned long		FSpan;
	short			AvgTime;
	short			AvgType;
	unsigned long		MeasureTime;
	unsigned long		Freq_high;
	signed long		DemodFreqChannel;
	unsigned long		DemodFreq_low;
	unsigned long		DemodFreq_high;
	DWORDLONG		OutputTimestamp; /* nanoseconds since Jan 1st, 1970, without leap sec-
onds */
} OPT_HEADER_IFPAN_TYPE;

typedef struct GPSHeader
{
  signed short   bValid;         /* denotes whether GPS data are to be
                                    considered valid                              */
  signed short   iNoOfSatInView; /* number of satellites in view 0-12;
                                    only valid, if GGA msg is received,
                                    else -1 (GPS_UNDEFINDED)                      */
  signed short   iLatRef;        /* latitude direction ('N' or 'S')               */
  signed short   iLatDeg;        /* latitude degrees                              */
  float         fLatMin;        /* geographical latitude: minutes                */
  signed short   iLonRef;        /* longitude direction ('E' or 'W')              */
  signed short   iLonDeg;        /* longitude degrees                             */
  float         fLonMin;        /* geographical longitude: minutes               */
  float         fPdop;          /* Mean (Position) Dilution Of Precision; */

} GPSHEADER_TYPE;
/*******************************************************************/
```

1.9.3 Streaming

```
/*************************streaming*****************************/
void streaming(SOCKET usock)
{
	/* create new File */
	FILE * stream;
	/* open File */
	if((stream = fopen("testfile_GPS.rtr","w+b"))==NULL)
	{
		printf("\nCould not open File");
		exit(1);
	}

	/* create buffer for receiving */
	char buffer[0x80000];
	/* IP address of the PC */
	const char *pcPCAddress = "172.25.10.27";
	/* bring the IP address in the right format for WinSock.h functions */
	unsigned long ulPCAddress = inet_addr(pcPCAddress);
	/* port for UDP Mass Data Output */
	unsigned short usUDPPort = 9000;
```

```
        /* fill a struct which is defined in WinSock.h with information
        for the socket address format */
        sockaddr_in addrDevice;
        /* reserve memory space for the address struct */
        memset(&addrDevice, 0, sizeof(addrDevice));
        /* setting for TCP/UDP address family */
        addrDevice.sin_family = AF_INET;
        /* put the information for the UDP Mass Data Output port into the struct */
        addrDevice.sin_port   = htons(usUDPPort);
        /* put the information for socket address into the struct */
        addrDevice.sin_addr.s_addr = ulPCAddress;
        /* size of the socket address struct */
        int remlen = sizeof(sockaddr_in);
        /* counter variable for the amount of packages */
        int packages = 0;

        /* receiving loop in this example for receiving 100 UDP packages */
        while(packages < 15)
        {
               /* The function recvfrom(socket, buffer, size of buffer, flags,
               socket address struct, size of socket address) is defined in WinSock.h
         It receives the UDP packet from the receiver and returns the number of received bytes */
               int reclen = recvfrom(usock, buffer,sizeof(buffer),0,
                      (struct sockaddr *)&addrDevice, &remlen);
               if(reclen==SOCKET_ERROR)
               {
                      printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
               }
               /* compare the length of the received data with the datasize
         in the EB200 Header */
               EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
               int datasize = ntohl(pEb200Header->DataSize);
               if(reclen != datasize)
               {
                      printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
               }
               /* write the buffer content into the File */
               if(fwrite(buffer, reclen, 1, stream)!=1)
                      printf("Error: received less bytes than expected!");

               /* increase the number of received packages */
               packages++;
        }
        /* close the File */
        fclose(stream);
}
/*******************************************************************/
```

1.10 MScan

1.10.1 Settings

```
/*************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* store frequencies for this example: 92.4 MHz, 93.3 MHz, 98.5 MHz */
       sendSCPI(TCPsock, "memory:contents 2, 92.4 MHz, 0, fm, 30000, 0,
            off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 3, 93.3 MHz, 0, fm, 30000, 0,
            off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 4, 98.5 MHz, 0, fm, 30000, 0,
            off, off, off, off, on\n");
       /* set the frequency mode to memory scan */
       sendSCPI(TCPsock, "frequency:mode mscan\n");
       /* starting position of the MScan to 1 */
       sendSCPI(TCPsock, "mscan:list:start 2\n");
       /* stopping position of the MScan to 4 */
       sendSCPI(TCPsock, "mscan:list:stop 4\n");
       /* infinite number of scans */
       sendSCPI(TCPsock, "mscan:count inf\n");
       /* set the dwel time to 0 s */
       sendSCPI(TCPsock, "mscan:dwel 0 s\n");
       /* set the hold time to 0 s */
       sendSCPI(TCPsock, "mscan:hold:time 0 s\n");
       /* switch off the squelch */
       sendSCPI(TCPsock, "output:squelch off\n");
       /* set the bandwidth to 120 kHz */
       sendSCPI(TCPsock, "bandwidth 120 kHz\n");
       /* set the demodulation mode to fm */
       sendSCPI(TCPsock, "demodulation fm\n");
       /* select measure root-mean-square value */
       sendSCPI(TCPsock, "sense:detector rms\n");
       /* set the measurement time to 0.5 s */
       sendSCPI(TCPsock, "measure:time 0.5 s\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode periodic\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch the automatic frequency control off */
       sendSCPI(TCPsock, "sense:frequency:afc off\n");
    /* set gcontrol mode to agc */
       sendSCPI(TCPsock, "gcontrol:mode agc\n");
    /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
       /* Setting the needed Tags and Flags for the MScan Streaming */
       sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, mscan\n");
       sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000,  'volt:ac',
            'opt', 'swap', 'chan', 'freq:low:rx', 'freq:high:rx'\n");
}
/******************************************************************/
```

1.10.2 Optional header

The optional header for the PScan streaming is only necessary if the information inside the header is needed or if the headers must be parsed because only the raw data is needed. The structure of the optional header is shown below.

```
/***************************Optional Header***********************/
typedef struct OptHeaderMSC
{
    unsigned short              CycleCount;
    unsigned short              HoldTime;
    unsigned short              DwellTime;
    unsigned short              DirectionUp;
    unsigned short              StopSignal;
    char                                      reserved[6];
    DWORDLONG                   OutputTimestamp; /* nanoseconds since Jan 1st, 1970,
                                              without leap seconds */
} OPTHEADER_MSC_TYPE;
/*******************************************************************/
```

1.10.3 Streaming

Otherwise it is possible to store the whole UDP package (without parsing any headers) in the file, like in this example:

```
/***************************streaming****************************/
void streaming(SOCKET UDPsock, SOCKET TCPsock)
{
       FILE * stream;
       /* open File */
       if((stream = fopen("testfile_MScan.rtr","w+b"))==NULL)
       {
              printf("\nCould not open File");
              exit(1);
       }

       /* create buffer for receiving */
       char buffer[0x80000];
       /* IP address of the PC */
       const char *pcPCAddress = "172.25.10.27";
       /* bring the IP address in the right format for WinSock.h functions */
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       /* port for UDP Mass Data Output */
       unsigned short usUDPPort = 9000;

       /* fill a struct which is defined in WinSock.h with information
       for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
       /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
       /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port   = htons(usUDPPort);
       /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = ulPCAddress;
       /* size of the socket address struct */
       int remlen = sizeof(sockaddr_in);
    /* number of received data */
    int reclen;
```

```
    /* counter variable for the amount of packages */
    int packages = 0;
    /* start the MScan */
    sendSCPI(TCPsock, "init\n");
       Sleep(2000);
       /* receive the udp packages, in this example 10 */
       while(packages < 10)
       {
              /* The function recvfrom(socket, buffer, size of buffer, flags,
              socket address struct, size of socket address) is defined in WinSock.h
              It receives the UDP packet from the receiver and returns the number
        of received bytes */
              reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
                 (struct sockaddr *)&addrDevice, &remlen);
              if(reclen==SOCKET_ERROR)
              {
                     printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
              }
              /* compare the length of the received data with the datasize in the
        EB200 Header */
              EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
              int datasize = ntohl(pEb200Header->DataSize);
              if(reclen != datasize)
              {
                     printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
              }

              /* write the buffer content into the File */
              if(fwrite(buffer, reclen, 1, stream)!=1)
                     printf("Error: received less bytes than expected!");

              packages++;
       }
       /* stop the MScan */
       sendSCPI(TCPsock, "abort\n");
       fclose(stream);
}
/********************************************************************/
```

1.11 Main

```
/****************************main***********************************/
void main(void)
{
       /* declare two variables of the type SOCKET, which is defined
    in WinSock.h */
       SOCKET TCPsock = INVALID_SOCKET;
       SOCKET UDPsock = INVALID_SOCKET;
       /* call the functions */
       connect(&TCPsock, &UDPsock);
       settings(TCPsock);
       streaming(UDPsock, TCPsock);

    printf("done!");
    return 0;
}
/********************************************************************/
```

2 Store into SD card

2.1 Connect

```
/*****************************************************************/
/* Connecting to the Receiver                                    */
/*****************************************************************/
/***************************Header-Files**************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>
/* winsock.h has to be included to connect with the receiver,
additionally the library wsock32.lib has to be included, for
Visual Studio:
Project -> test Properties -> Configuration Properties ->
Linker -> Input, then type in wsock32.lib to the other libraries
(other development environments may need a different workflow)*/

/***************************Functions*****************************/
/***************************Network-Connection-Test***************/
int checkWinSock()
{
	int iRet = 0;
	/* type WSADATA is defined in WinSock.h*/
	WSADATA wsaData;

	/*Testing if Network-Connection is ok */
	if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)
	/*Returning 0 if Connection is ok*/
	{
	/* Tell the user that we couldn't find a usable */
	/* WinSock DLL by returning 1                    */
		iRet = 1;
	}
	/* Confirm that the WinSock DLL supports 2.0.*/
	/* Note that if the DLL supports versions greater  */
	/* than 2.0 in addition to 2.0, it will still return */
	/* 2.0 in Version since that is the version we    */
	/* requested.                         */

	if (LOBYTE(wsaData.wVersion) != 2 ||
		HIBYTE(wsaData.wVersion) != 0 )
	{	/* Tell the user that we couldn't find a usable */
		/* WinSock DLL by returning 2                    */
		iRet = 2;
	}
	/* The WinSock DLL is acceptable. Proceed. */
	return iRet;
}
/*****************************************************************/
```

```
/**************************Create TCP Socket*********************/
int createTCPSocket(SOCKET *TCPsock, unsigned long *pulRemoteAddress,
                            unsigned short *pusPort)
{
       /* Socket Initialization*/
       int iReturn = 0;
       /* Test the Network Connection (see Network-Connection-Test)*/
       iReturn = checkWinSock();

       /* Create a receiver control TCP socket */
       /* The function socket(address family, socket type, protocol) is defined
       in WinSock.h
       It builds up a socket for the connection between the receiver and the PC*/
       *TCPsock = socket(AF_INET, SOCK_STREAM, 0);
       /*Testing if there is an error*/
       if (*TCPsock == SOCKET_ERROR)
       {
              iReturn = 3;
       }
       /* Set the socket options */
       int sopt = 1;
       /* The function setsockopt(socket, level, optname, optval, optlen)
       is defined in WinSock.h
       It sets the current value for a socket option associated with a socket of
       any type, in any state*/
       if ( setsockopt( *TCPsock, IPPROTO_TCP, TCP_NODELAY,
              (char *)&sopt, sizeof( sopt ) ) == SOCKET_ERROR )
       {
              iReturn = 4;
       }
       /* Create socket address structure for INET address family */
       /* The struct sockaddr_in(family, type, address, port)
       is defined in WinSock.h
       This struct defines the structure of the socket, it consists of
       the address family, the address of the server you are connecting to
       and the port you want to connect*/
       struct sockaddr_in addrDevice;
       memset(&addrDevice, 0, sizeof(addrDevice));
       addrDevice.sin_family      = AF_INET;
       addrDevice.sin_addr.s_addr = *pulRemoteAddress;
       addrDevice.sin_port        = htons(*pusPort);
       /* Connect socket to receiver device */
       /* The function connect(socket, socket_addr, size of struct socket_addr)
       is defined in WinSock.h
       It is connecting the receiver and the PC over the socket */
       if (connect(*TCPsock, (struct sockaddr *)&addrDevice,
                 sizeof(addrDevice)) != 0)
       {
              iReturn = 5;
       }
       return iReturn;
}
/********************************************************************/
```

```
/***************************connect SCPI***************************/
void connectSCPI(SOCKET *TCPsock)
{
	const char *pcDeviceAddress = "172.25.9.88"; // IP-address of the receiver
	unsigned long ulRemoteAddress = inet_addr(pcDeviceAddress);
	unsigned short usPort = 5555; // Port number (always 5555)

	printf("Connecting to PR100...\n");
	int iReturn = createTCPSocket(TCPsock, &ulRemoteAddress, &usPort);
	if (iReturn != 0)
	{	/* Error returning:*/
		switch(iReturn)
		{
			case 1:
				printf("Error retrieving windows socket dll.\n");
			case 2:
				printf("Error retrieving correct winsock version 2.0.\n");
			case 3:
				printf("Error creating SCPI socket.\n");
			case 4:
				printf("setsockopt (TCP_NODELAY) failed - errno %d\n",
						WSAGetLastError());
			case 5:
				printf("Error connecting to %s [SCPI port = %d]\n", pcDeviceAddress,
usPort);
		}
	}
	else
	{
		printf("PR100 is connected!\n");
	}
}
/*******************************************************************/
/***************************send string***************************/
int sendSCPI(int TCPsock, char *pBuffer)
{
	unsigned int nLen;
	/* The function send(socket, buffer, size of buffer, flags)
	is defined in WinSock.h
	It sends bytes to the receiver over the socket and returns the
	number of bytes sent */
	nLen = send(TCPsock, pBuffer, strlen(pBuffer), 0);
	/* Testing if all bytes are sent */
	if (nLen != strlen(pBuffer))
	{
		printf("Error writing to socket. Len = %d\n", nLen);
	}
	return nLen;
}
/*******************************************************************/
```

2.2 Settings

2.2.1 IFPAN

```
/***************************settings*******************************/
void settings(SOCKET TCPsock)
{
	/* Get the basic information about the receiver with the
	SCPI command *idn? */
	sendSCPI(TCPsock, "*idn?\n");
	char pRxBuf[256];
	int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
	if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
	puts( pRxBuf );
	/* sending SCPI commands for the needed settings */
	/* set the frequency mode to fixed */
	sendSCPI(TCPsock, "frequency:mode fixed\n");
	/* set the frequency to 98.5 MHz */
	sendSCPI(TCPsock, "frequency 98.5 MHz\n");
	/* set the IF spectrum to 10 MHz */
	sendSCPI(TCPsock, "sense:freq:span 10 MHz\n");
	/* set the IF spectrum mode to average */
	sendSCPI(TCPsock, "calculate:ifpan:average:type scalar\n");
	/* set the measurement time to 50 ms */
	sendSCPI(TCPsock, "measure:time 50 ms\n");
	/* set the measurement mode to periodic */
	sendSCPI(TCPsock, "measure:mode periodic\n");
	/* switch attenuation off*/
	sendSCPI(TCPsock, "input:attenuation:state off\n");
	/* switch the automatic frequency control off*/
	sendSCPI(TCPsock, "sense:frequency:afc off\n");
}
/******************************************************************/
```

2.2.2 PScan

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to PScan */
       sendSCPI(TCPsock, "frequency:mode pscan\n");
       /* set the start frequency to 880 MHz */
       sendSCPI(TCPsock, "frequency:pscan:start 880 MHz\n");
       /* set the stop frequency to 960 MHz*/
       sendSCPI(TCPsock, "frequency:pscan:stop 960 MHz\n");
       /* set the number of Scans to infinite */
       sendSCPI(TCPsock, "pscan:count infinity\n");
       /* set the step size of the PScan to 50 kHz */
       sendSCPI(TCPsock, "sense:pscan:step 50 kHz\n");
       /* set the measurement time to 1 ms */
       sendSCPI(TCPsock, "measure:time 1 ms\n");
       /* set the measurement mode to periodic */
       sendSCPI(TCPsock, "measure:mode periodic\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
}
/****************************************************************/
```

2.2.3 Audio stream

```
/***************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to fixed */
       sendSCPI(TCPsock, "frequency:mode fixed\n");
       /* set the frequency to 95 MHz */
       sendSCPI(TCPsock, "frequency 95 MHz\n");
       /* set the bandwidth to 120 kHz */
       sendSCPI(TCPsock, "bandwidth 120 kHz\n");
       /* set the demodulation mode to fm */
       sendSCPI(TCPsock, "demodulation fm\n");
       /* set the measurement time to default */
       sendSCPI(TCPsock, "measure:time default\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch the automatic frequency control on */
       sendSCPI(TCPsock, "sense:frequency:afc on\n");
    /* set gcontrol to automatically generated */
       sendSCPI(TCPsock, "sense:gcontrol:mode agc\n");
    /* set audio mode to 2, see table 6-8 in the manual */
       sendSCPI(TCPsock, "system:audio:remote:mode 2\n");
    /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
}
/*****************************************************************/
```

2.2.4 I/Q stream

```
/**************************settings*****************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to fixed */
       sendSCPI(TCPsock, "frequency:mode fixed\n");
       /* set the frequency to 390 MHz */
       sendSCPI(TCPsock, "frequency 390 MHz\n");
       /* set the bandwidth to 500 kHz */
       sendSCPI(TCPsock, "bandwidth 500 kHz\n");
       /* set the demodulation mode to iq */
       sendSCPI(TCPsock, "demodulation iq\n");
       /* set the measurement time to default */
       sendSCPI(TCPsock, "measure:time def\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch the automatic frequency control off */
       sendSCPI(TCPsock, "sense:frequency:afc off\n");
       /* switch the automatic gain control off */
       sendSCPI(TCPsock, "gcontrol:mode mgc\n");
       sendSCPI(TCPsock, "gcontrol 0\n");
       /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
}
/*****************************************************************/
```

2.2.5 FScan

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* set the frequency mode to sweep */
       sendSCPI(TCPsock, "frequency:mode sweep\n");
       /* number of sweeps is set to 10 */
       sendSCPI(TCPsock, "sweep:count 10\n");
       /* set the start frequency to 118 MHz */
       sendSCPI(TCPsock, "frequency:start 118 MHz\n");
       /* set the stop frequency to 138 MHz */
       sendSCPI(TCPsock, "frequency:stop 138 MHz\n");
       /* set the frequency stepwidth to 25 kHz*/
       sendSCPI(TCPsock, "sweep:step 25 kHz\n");
       /* switch on sweep control */
       sendSCPI(TCPsock, "sweep:control on\n");
       /* set dwel time to 3 s */
       sendSCPI(TCPsock, "sweep:dwel 3 s\n");
       /* set hold time to 1 s */
       sendSCPI(TCPsock, "sweep:hold:time 1 s\n");
       /* switch on squelch */
       sendSCPI(TCPsock, "output:squelch on\n");
       /* set squelch threshold to 10 dbuV */
       sendSCPI(TCPsock, "output:squelch:threshold 10 dbuV\n");
       /* set the bandwidth to 9 kHz */
       sendSCPI(TCPsock, "bandwidth 9 kHz\n");
       /* set the demodulation mode to am */
       sendSCPI(TCPsock, "demodulation am\n");
       /* set the measurement time to default */
       sendSCPI(TCPsock, "measure:time default\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch automatic frequency control off */
       sendSCPI(TCPsock, "sense:frequency:afc off\n");
       /* set gcontrol to automatically generated */
       sendSCPI(TCPsock, "sense:gcontrol:mode agc\n");
    /* set audio mode to 2, see table 6-8 in the manual */
       sendSCPI(TCPsock, "system:audio:remote:mode 2\n");
       /* switch tone off */
       sendSCPI(TCPsock, "output:tone off\n");
}
/*****************************************************************/
```

2.2.6 Store level

```
/**************************settings*****************************/
void settings(SOCKET TCPsock)
{
	/* Get the basic information about the receiver with the
	SCPI command *idn? */
	sendSCPI(TCPsock, "*idn?\n");
	char pRxBuf[256];
	int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
	if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
	puts( pRxBuf );
	/* sending SCPI commands for the needed settings */
	/* set the frequency mode to fixed */
	sendSCPI(TCPsock, "frequency:mode fixed\n");
	/* set the frequency to 300 MHz */
	sendSCPI(TCPsock, "frequency 95 MHz\n");
	/* set the bandwidth to 120 kHz */
	sendSCPI(TCPsock, "bandwidth 120 kHz\n");
	/* set the demodulation mode to fm */
	sendSCPI(TCPsock, "demodulation fm\n");
	/* set the measurement time to default */
	sendSCPI(TCPsock, "measure:time default\n");
	/* set the measurement mode to periodic */
	sendSCPI(TCPsock, "measure:mode periodic\n");
	/* switch attenuation off*/
	sendSCPI(TCPsock, "input:attenuation:state off\n");
	/* switch the automatic frequency control off */
	sendSCPI(TCPsock, "sense:frequency:afc off\n");
	/* switch the automatic gain control off */
	sendSCPI(TCPsock, "gcontrol:mode mgc\n");
	sendSCPI(TCPsock, "gcontrol 0\n");
	/* switch tone off */
	sendSCPI(TCPsock, "output:tone off\n");
	/* select measure root-mean-square value */
	sendSCPI(TCPsock, "sense:detector rms\n");
	/* GPS and compass settings */
	sendSCPI(TCPsock, "system:gpscompass:source gps, aux1\n");
	sendSCPI(TCPsock, "system:gpscompass:source compass, aux1\n");
	sendSCPI(TCPsock, "system:gpscompass:aux:configuration 1, 4800, 8, none, 1\n");
	sendSCPI(TCPsock, "system:gpscompass on\n");
}
/******************************************************************/
```

2.2.7 MScan

```
/**************************settings******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* store frequencies for this example: 50MHz, 92.4 MHz, 93.3 MHz, 98.5 MHz */
       sendSCPI(TCPsock, "memory:contents 1, 50 MHz, 30,
            fm, 300, 0, off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 2, 92.4 MHz, 30,
            fm, 300, 0, off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 3, 93.3 MHz, 30,
            fm, 300, 0, off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 4, 98.5 MHz, 30,
            fm, 300, 0, off, off, off, off, on\n");
       /* set the frequency mode to memory scan */
       sendSCPI(TCPsock, "frequency:mode mscan\n");
       /* starting position of the MScan to 1 */
       sendSCPI(TCPsock, "mscan:list:start 1\n");
       /* stopping position of the MScan to 4 */
       sendSCPI(TCPsock, "mscan:list:stop 4\n");
       /* infinite number of scans */
       sendSCPI(TCPsock, "mscan:count INF\n");
       /* switch on MScan control */
       sendSCPI(TCPsock, "mscan:control on\n");
       /* set the dwel time to 0 s */
       sendSCPI(TCPsock, "mscan:dwel 0 s\n");
       /* set the hold time to 0 s */
       sendSCPI(TCPsock, "mscan:hold:time 0 s\n");
       /* switch off the squelch */
       sendSCPI(TCPsock, "output:squelch off\n");
       /* set the bandwidth to 120 kHz */
       sendSCPI(TCPsock, "bandwidth 120 kHz\n");
       /* set the demodulation mode to fm */
       sendSCPI(TCPsock, "demodulation fm\n");
       /* select measure root-mean-square value */
       sendSCPI(TCPsock, "sense:detector rms\n");
       /* set the measurement time to 1 s */
       sendSCPI(TCPsock, "measure:time 1 s\n");
       /* set the measurement mode to continuous */
       sendSCPI(TCPsock, "measure:mode continuous\n");
       /* switch attenuation off*/
       sendSCPI(TCPsock, "input:attenuation:state off\n");
       /* switch the automatic frequency control off */
       sendSCPI(TCPsock, "sense:frequency:afc off\n");
       sendSCPI(TCPsock, "gcontrol:mode auto\n");
       /* switch tone off*/
       sendSCPI(TCPsock, "output:tone off\n");
       /* GPS and compass settings */
       sendSCPI(TCPsock, "system:gpscompass:source gps, aux1\n");
       sendSCPI(TCPsock, "system:gpscompass:source compass, aux1\n");
       sendSCPI(TCPsock, "system:gpscompass:aux:configuration 1, 4800, 8, none, 1\n");
       sendSCPI(TCPsock, "system:gpscompass on\n");
}
/****************************************************************/
```

2.3 Store to SD card

```
/***************************store into SDcard*********************/
void storeSD(SOCKET sock)
{
       /* SCPI command for storing into SDcard */
       sendSCPI(sock, "trace:record:storage file\n");
       /* SCPI command to choose the recording mode, in this example it is
    trace but if Audio or IQ data is recorded then it has to be changed */
       sendSCPI(sock, "trace:record:source trace\n");

       /*This command is just necessary to start a PScan, FScan or MScan */
       sendSCPI(sock, "init\n");

       /* Start recording */
       sendSCPI(sock, "trace:record:start\n");
       /* Recording time in ms, here 5 s */
       Sleep(5000);
       /* Stop recording */
       sendSCPI(sock, "trace:record:stop\n");

       /*This command is just necessary to stop a PScan, FScan or MScan */
       sendSCPI(sock, "abort\n");

       sendSCPI(sock, "mmemory:cdirectory 'PR100'\n");
       /* Show the recorded Files, this command is just optional to
    show the Files in the SD card*/
       sendSCPI(sock, "mmemory:catalog?\n");
       char psRxBuf[10000];
       int len = recv(sock, psRxBuf, sizeof(psRxBuf),0);
       if( len < 0 ) len = 0; psRxBuf[ len ] = '\0';
       puts( psRxBuf );
}
/*******************************************************************/
```

2.4 Copy from receiver to PC

```
/***************************transfer to the PC********************/
void transferPC(SOCKET TCPsock)
{
       FILE * stream;
       /* open File, in this example we have trace data and save it in the
       project folder of the program, the name and the folder can be changed
       and if there is Audio or IQ data then the format has to be .wav or .riq*/
       if((stream = fopen("file0.rtr","w+b"))==NULL)
       {
              printf("\nCould not open File");
              exit(1);
       }

       /* open the folder in the SD card */
       sendSCPI(TCPsock, "mmemory:cdirectory 'PR100'\n");
       /* SCPI command to get the recorded data in this case trace data,
       in other cases it can also be Audio or IQ data, then the name of
       the file is different, for example RecAudio_000.wav or Recxxx_000.riq */
       sendSCPI(TCPsock, "mmemory:data? 'RecTrace_000.rtr'\n");

       /* create buffer for receiving */
       char pRxBuf[1000];
       int buffersize = sizeof(pRxBuf);
       /* read the first 1000 bytes of the record */
       int len = recv(TCPsock, pRxBuf, buffersize,0);
       if(len != buffersize)
       {
              printf("Error: received less bytes than expected!");
       }

       /* number of digits: if you receive the data, than
       the second character is the number of digits for the size
       of the file, for example: #530984......
       here 5 is the number of digits and then 30984 (5 characters)
       is the size of the file. This means that you have to store
       your file after the file size, in this case in the 7th
       position of the buffer */
       char c = pRxBuf[1]; // number of digits
       int i = (c - '0')+2; // starting position

       /* getting the size of the file */
       char string [50];
       int z = 0;
       while(z<=i-2)
       {
              string[z] = pRxBuf[z+2];
              z++;
       }
       string[z+1] = '/n';
       int size = atoi(string);

    /* Variable count is the already received bytes in
       the buffer */
       int rec_bytes = buffersize-i;
       /* Variable buf_bytes is the number of still expected bytes from
       the receiver */
       int buf_bytes = size-rec_bytes;
```

```
        /* storing the first received bytes */
        int ret = fwrite((void *)&pRxBuf[i], rec_bytes, 1, stream);
        if(ret!=1) printf("Error: received less bytes than expected!");

        /* Variable received is the number of received bytes */
        int received = 0;
        /* Variable stored is the number of stored bytes */
        int stored = 0;

        /* Loop reading bytes into the file */
        while(rec_bytes < size)     {
                /* Decide if the still expected bytes are more than the size of the buffer */
                if(buf_bytes > buffersize)
                {
                        /* Receiving the bytes into the buffer */
                        received = recv(TCPsock, pRxBuf, buffersize, 0);
                        /* Storing the received bytes into the File */
                        stored = fwrite(pRxBuf, received, 1, stream);
                        if(stored!=1) printf("Error: received less bytes than expected!");
                }
                /* still expected bytes are less than the size of the buffer, so just
                store the expected bytes */
                else
                {
                        /* Receiving the bytes into the buffer */
                        received = recv(TCPsock, pRxBuf, buf_bytes, 0);
                        /* Storing the received bytes into the File */
                        stored = fwrite(pRxBuf, received, 1, stream);
                        if(stored!=1) printf("Error: received less bytes than expected!");
                }
                buf_bytes = buf_bytes - received; // decrease the number of expected bytes
                rec_bytes = rec_bytes + received; // increase the number of bytes sent
        }
        fclose(stream); // close and save the file
}
///*********************************************************************/
```

2.5 Main

```
/****************************main************************************/
int main(void)
{
    /* declare a variable of the type SOCKET, which is defined
    in WinSock.h */
        SOCKET TCPsocket = INVALID_SOCKET;

        connectSCPI(&TCPsocket);
        settings(TCPsocket);
        storeSD(TCPsocket);
        transferPC(TCPsocket);

    printf("done!");
    return 0;
}
/********************************************************************/
```

3 Memory scan and direction finding

For applications such as air traffic control (ATC), it can be very useful to combine different modes – for example, memory scan with direction finding data. In this case, the programming example shows how you can monitor up to 1024 predefined channels which use different demodulation modes and bandwidths. If there is a signal over squelch, the receiver will also store the direction data of the signal.

3.1 Connecting to the receiver

The connection between the receiver and the PC is over LAN and based on Windows sockets. To communicate with the receiver, you have to send SCPI commands over a TCP socket. For receiving the data stream, it is necessary to create an UDP socket. The following code shows an example therefore.

3.1.1 Headers

```
/*****************************************************************/
/* Connecting to the Receiver                                    */
/*****************************************************************/
/**************************Header-Files*************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <winsock.h>
/* winsock.h has to be included to connect with the receiver,
additionally the library wsock32.lib has to be included, for
Visual Studio:
Project -> test Properties -> Configuration Properties ->
Linker -> Input, then type in wsock32.lib to the other libraries
(other development environments may need a different workflow)*/
/*****************************************************************/
/**************************Basic Header*************************/
typedef struct Eb200Header
{
       unsigned long  MagicNumber;
       unsigned short VersionMinor;
       unsigned short VersionMajor;
       unsigned short SeqNumber;
       unsigned short Reserved;
    unsigned long     DataSize;
} EB200_HEADER_TYPE;
```

```
#define EB200_HEADER_SIZE  (sizeof( EB200_HEADER_TYPE ))

typedef struct UdpDatagramAttribute
{
        unsigned short Tag;
        unsigned short Length;
        unsigned short NumItems;
        unsigned char  ChannelNumber;
        unsigned char  OptHeaderLength;
        unsigned long  SelectorFlags;
        unsigned char  OptHeader[22];
} UDP_DATAGRAM_ATTRIBUTE_TYPE;
#define ATTRIBUTE_HEADER_SIZE  (sizeof( UDP_DATAGRAM_ATTRIBUTE_TYPE ))
/*******************************************************************/
```

```
/**************************Optional Header***********************/
/* Optional Header for a Memory Scan package */
typedef struct OptHeaderMSC
{
    unsigned short              CycleCount;
    unsigned short              HoldTime;
    unsigned short              DwellTime;
    unsigned short              DirectionUp;
    unsigned short              StopSignal;
    char                                      reserved[6];
    DWORDLONG                   OutputTimestamp; /* nanoseconds since Jan 1st, 1970,
                                      without leap seconds */
} OPTHEADER_MSC_TYPE;
/* Optional Header for a Direction Finding package */
typedef struct OptHeaderDFPan
{
       unsigned long                           Freq_low;
       unsigned long                           Freq_high;
       unsigned long                           FreqSpan;
       signed long                                    DFThresholdMode;
       signed long                                    DFThresholdValue;
    unsigned long               DfBandwidth;
    unsigned long               StepWidth;
    unsigned long               DFMeasureTime;
    signed long                 DFOption;
    unsigned short              CompassHeading;
    signed short                CompassHeadingType;
    signed long                 AntennaFactor;
    signed long                 DemodFreqChannel;
       unsigned long                           DemodFreq_low;
       unsigned long                           DemodFreq_high;
    DWORDLONG                   OutputTimestamp; /* nanoseconds since Jan 1st, 1970,
                                                                              without leap
    seconds */
} OPT_HEADER_DFPAN_TYPE;
/*****************************************************************/
/**************************Data struct***************************/
/* Structure of the incoming data for a Memory Scan package */
typedef struct Data
{
```

```
        unsigned short level;
        unsigned short channel;
        unsigned long  FreqLow;
        unsigned long  FreqHigh;
} DATA_HEADER_TYPE;
/*******************************************************************/
```

3.1.2 Windows sockets

```
/*************************Functions****************************/
/*************************Network-Connection-Test***************/
int checkWinSock()
{
        /* type WSADATA is defined in WinSock.h*/
        WSADATA wsaData;

        /* Testing if Network-Connection is ok */
        if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)
                /*Returning 0 if Connection is ok*/
        {
                /* Tell the user that we couldn't find a usable */
                /* WinSock DLL by returning 1                    */
                return 1;
        }
        /* Confirm that the WinSock DLL supports 2.0.*/
        /* Note that if the DLL supports versions greater  */
        /* than 2.0 in addition to 2.0, it will still return */
        /* 2.0 in wVersion since that is the version we    */
        /* requested.                      */

        if (LOBYTE(wsaData.wVersion) != 2 ||
                HIBYTE(wsaData.wVersion) != 0 )
        {       /* Tell the user that we couldn't find a usable */
                /* WinSock DLL by returning 2                    */
                return 2;
        }
        /* The WinSock DLL is acceptable. Proceed. */
        return 0;
}
/*************************************************************/
```

3.1.3 TCP socket

```
/**************************Create TCP Socket**********************/
int createTCPSocket(SOCKET *pSock, unsigned long *pulRemoteAddress,
                              unsigned short *pusPort)
{
        /* Socket Initialization*/
        int iRet = 0;
        /* Test the Network Connection (see Network-Connection-Test)*/
        iRet = checkWinSock();
        /* Create a receiver control TCP socket */
        /* The function socket(address family, socket type, protocol) is defined
        in WinSock.h
        It builds up a socket for the connection between the receiver and the PC*/
        *pSock = socket(AF_INET, SOCK_STREAM, 0);
        /*Testing if there is an error*/
        if (*pSock == SOCKET_ERROR)
        {
                iRet = 3;
        }
        /* Set the socket options */
        int sopt = 1;
        /* The function setsockopt(socket, level, optname, optval, optlen)
        is defined in WinSock.h
        It sets the current value for a socket option associated with a socket of
        any type, in any state*/
        if ( setsockopt( *pSock, IPPROTO_TCP, TCP_NODELAY,
                (char *)&sopt, sizeof( sopt ) ) == SOCKET_ERROR )
        {
                iRet = 4;
        }
        /* Create socket address structure for INET address family */
        /* The struct sockaddr_in(family, type, address, port)
        is defined in WinSock.h
        This struct defines the structure of the socket, it consists of
        the address family, the address of the server you are connecting to
        and the port you want to connect*/
        sockaddr_in addrDevice;
        /* reserve memory space for the address struct */
        memset(&addrDevice, 0, sizeof(addrDevice));
        /* setting for TCP/UDP address family */
        addrDevice.sin_family = AF_INET;
```

```
	/* put the information for the UDP Mass Data Output port into the struct */
	addrDevice.sin_port    = htons(*pusPort);
	/* put the information for socket address into the struct */
	addrDevice.sin_addr.s_addr = *pulRemoteAddress;
	/* Connect socket to receiver device */
	/* The function connect(socket, socket_addr, size of struct socket_addr)
	is defined in WinSock.h
	It is connecting the receiver and the PC over the socket */
	if (connect(*pSock, (struct sockaddr *)&addrDevice, sizeof(addrDevice)) != 0)
	{
		iRet = 5;
	}
	return iRet;
}
/*******************************************************************/
```

3.1.4 UDP socket

```
/**************************Create UDP Socket*********************/
int createUDPSocket(SOCKET *sock, unsigned long *pulRemoteAddress,
                                    unsigned short *pusPort)
{
        /* Socket Initialization*/
        int iRet = 0;
        /* Test the Network Connection (see Network-Connection-Test)*/
        iRet = checkWinSock();
        if (iRet != 0) return iRet;
        /* Create a receiver control TCP socket */
        /* The function socket(address family, socket type, protocol) is defined
        in WinSock.h
        It builds up a socket for the connection between the receiver and the PC*/
        *sock = socket(AF_INET, SOCK_DGRAM, 0);
        /*Testing if there is an error*/
        if (*sock == SOCKET_ERROR)
        {
                iRet = 3;
        }
        /* Create socket address structure for INET address family */
        /* The struct sockaddr_in(family, type, address, port)
        is defined in WinSock.h
        This struct defines the structure of the socket, it consists of
        the address family, the address of the server you are connecting to
        and the port you want to connect*/
        sockaddr_in addrDevice;
        /* reserve memory space for the address struct */
        memset(&addrDevice, 0, sizeof(addrDevice));
        /* setting for TCP/UDP address family */
        addrDevice.sin_family = AF_INET;
        /* put the information for the UDP Mass Data Output port into the struct */
        addrDevice.sin_port   = htons(*pusPort);
        /* put the information for socket address into the struct */
        addrDevice.sin_addr.s_addr = *pulRemoteAddress;
        /* The function bind(socket, pointer to the address structure, size of the struct) is
        defined in WinSock.h
        It associates a local address with a socket */
        if ( bind(*sock, (struct sockaddr *)&addrDevice, sizeof(addrDevice)) == SOCKET_ERROR)
        {
```

```
            printf("Couldn't bind TRACe socket - errno = %d\n", WSAGetLastError());
            closesocket(*sock), *sock = INVALID_SOCKET;
            iRet = 4;
        }
        return iRet;
}
/*******************************************************************/
```

3.1.5 Connect

```
/**************************Connect********************************/
void connect(SOCKET *pSock, SOCKET *uSock)
{
	/* IP-address of the receiver (for the TCP socket) */
	const char *pcDeviceAddress = "172.25.10.19";
	unsigned long ulRemoteAddress = inet_addr(pcDeviceAddress);
	unsigned short usSCPIPort = 5555;
	// Port number for the receiver is always 5555

	/* IP-address of the PC (for the UDP streaming) */
	const char *pcPCAddress = "172.25.10.27";
	unsigned long ulPCAddress = inet_addr(pcPCAddress);
	unsigned short usUDPPort = 9000;
	printf("Connecting to PR100...\n");
	/* Creating a TCP socket */
	int iRet = createTCPSocket(pSock, &ulRemoteAddress, &usSCPIPort);
	if (iRet != 0)
	{	/* Error returning:*/
		switch(iRet)
		{
		case 1:
			printf("Error retrieving windows socket dll.\n");
		case 2:
			printf("Error retrieving correct winsock version 2.0.\n");
		case 3:
			printf("Error creating TCP socket.\n");
		case 4:
			printf("setsockopt (TCP_NODELAY) failed - errno %d\n",
				WSAGetLastError());
		case 5:
			printf("Error connecting to %s [SCPI port = %d]\n", pcDeviceAddress, usSCPIPort);
		}
	}
	else
	{
		printf("TCP Socket created!\n");
	}
	/* Creating an UDP Socket */
	int iUDP = createUDPSocket(uSock, &ulPCAddress, &usUDPPort);
	if (iUDP != 0)
```

```
	{	/* Error returning:*/
		switch(iUDP)
		{
		case 1:
			printf("Error retrieving windows socket dll.\n");
		case 2:
			printf("Error retrieving correct winsock version 2.0.\n");
		case 3:
			printf("Error creating UDP socket.\n");
		case 4:
			printf("\n");
		}
	}
	else
	{
		printf("UDP Socket created!\n");
	}
}
/*******************************************************************/
```

3.1.6 Sending SCPI commands

```
/**************************send string****************************/
int sendSCPI(int sd, char *pBuffer)
{
        unsigned int nLen;
        /* The function send(socket, buffer, size of buffer, flags)
        is defined in WinSock.h
        It sends bytes to the receiver over the socket and returns the
        number of bytes sent */
        nLen = send(sd, pBuffer, strlen(pBuffer), 0);
        /* Testing if all bytes are sent */
        if (nLen != strlen(pBuffer))
        {
                printf("Error writing to socket. Len = %d\n", nLen);
        }
        return nLen;
}
/*******************************************************************/
```

3.2 Settings

All the settings for the receiver can be made with SCPI commands over the TCP socket. This means you have the full functionality of the R&S®PR100 and R&S®EM100 via remote control.

```
/****************************Settings*******************************/
void settings(SOCKET TCPsock)
{
       /* Get the basic information about the receiver with the
       SCPI command *idn? */
       sendSCPI(TCPsock, "*idn?\n");
       char pRxBuf[256];
       int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
       if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
       puts( pRxBuf );
       /* sending SCPI commands for the needed settings */
       /* store frequencies for this example: 50.0 MHz, 92.4 MHz, 93.3 MHz,
    98.5 MHz, 92.0 MHz, 100.3 MHz, 40.0 MHz */
       sendSCPI(TCPsock, "memory:contents 1, 50.0 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 2, 92.4 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 3, 93.3 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 4, 98.5 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 5, 92.0 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 6, 100.3 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");
       sendSCPI(TCPsock, "memory:contents 7, 40.0 MHz, 0, fm, 30000, 0,
                       off, off, off, off, on\n");

       /* starting position of the MScan to 1 */
       sendSCPI(TCPsock, "mscan:list:start 1\n");
       /* stopping position of the MScan to 7 */
       sendSCPI(TCPsock, "mscan:list:stop 7\n");
       /* infinite number of scans */
       sendSCPI(TCPsock, "mscan:count inf\n");
       /* set the dwel time to 0.1 s */
       sendSCPI(TCPsock, "mscan:dwel 0.1 s\n");
       /* set the hold time to 0 s */
       sendSCPI(TCPsock, "mscan:hold:time 0 s\n");
       /* switch off the squelch */
```

```
sendSCPI(TCPsock, "output:squelch on\n");
/* set squelch to 20 dBuV, just for this example */
sendSCPI(TCPsock, "output:squelch:threshold 20 dBuV\n");
/* set the bandwidth to 120 kHz */
sendSCPI(TCPsock, "bandwidth 120 kHz\n");
/* set the demodulation mode to fm */
sendSCPI(TCPsock, "demodulation fm\n");
/* select measure root-mean-square value */
sendSCPI(TCPsock, "sense:detector rms\n");
/* set the measurement time to 1 s */
sendSCPI(TCPsock, "measure:time 0.5\n");
/* set the measurement mode to continuous */
sendSCPI(TCPsock, "measure:mode periodic\n");
/* switch attenuation off */
sendSCPI(TCPsock, "input:attenuation:state off\n");
/* switch the automatic frequency control off */
sendSCPI(TCPsock, "sense:frequency:afc off\n");
/* switch automatically generated gain control on */
sendSCPI(TCPsock, "gcontrol:mode agc\n");
/* switch off the tone */
sendSCPI(TCPsock, "output:tone off\n");
```

```
	/* DF Settings */
	sendSCPI(TCPsock, "bandwidth:dfinder 60000Hz\n");
	sendSCPI(TCPsock, "measure:dfinder:mode continuous\n");
	sendSCPI(TCPsock, "measure:dfinder:time 1s\n");

	/* Setting the needed Tags and Flags for the MScan and DF Streaming */
	sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, mscan, dfpan\n");
	sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000,  'volt:ac', 'opt',
	                'swap', 'chan', 'freq:low:rx', 'freq:high:rx', 'dflevel',
	                'azimuth', 'squelch'\n");
}
/*******************************************************************/
/***************************MScan mode****************************/
void MScanSettings(SOCKET TCPsock)
{
	/* set the frequency mode to memory scan */
	sendSCPI(TCPsock, "frequency:mode mscan\n");
	Sleep(2000); // otherwise the program is too fast
}
/*******************************************************************/
/***************************DFPan mode****************************/
void DFPanSettings(SOCKET TCPsock, int frequency)
{
	/* set the frequency mode to direction finding */
	sendSCPI(TCPsock, "frequency:mode df\n");
	/* set the frequency where a signal over squelch was detected */
	char *freq;
	freq = (char*) malloc(13+sizeof(frequency));
	sprintf(freq, "frequency %d Hz\n", frequency);
	sendSCPI(TCPsock, freq);
	/* start direction finding */
	sendSCPI(TCPsock, "init\n");
	Sleep(2000); // otherwise the program is too fast
}
/*******************************************************************/
```

3.3 Data stream

```
/**************************streaming****************************/
void streaming(SOCKET UDPsock, SOCKET TCPsock)
{
       FILE * stream;
       /* open File */
       if((stream = fopen("testfile_MScan_DF.rtr","w+b"))==NULL)
       {
              printf("\nCould not open File");
              exit(1);
       }

       /* create buffer for receiving */
       char buffer[0x80000];
       /* IP address of the PC */
       const char *pcPCAddress = "172.25.10.27";
       /* bring the IP address in the right format for WinSock.h functions */
       unsigned long ulPCAddress = inet_addr(pcPCAddress);
       /* port for UDP Mass Data Output */
       unsigned short usUDPPort = 9000;

       /* fill a struct which is defined in WinSock.h with information
       for the socket address format */
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
       /* setting for TCP/UDP address family */
       addrDevice.sin_family = AF_INET;
       /* put the information for the UDP Mass Data Output port into the struct */
       addrDevice.sin_port   = htons(usUDPPort);
       /* put the information for socket address into the struct */
       addrDevice.sin_addr.s_addr = ulPCAddress;
       /* size of the socket address struct */
       int remlen = sizeof(sockaddr_in);

       /* variable for the number of received packages */
       int packages = 0;
       /* variable for the number of received data */
       int reclen = 0;
       /* variable for the received tag */
       int tag = 0;
```

```
/* variable for the datasize */
int datasize = 0;
/* variables for the received frequency */
int freql = 0;
int freqh = 0;
uint64_t frequency;
/* receive the UDP packages, in this example 20 */
while(packages < 20)
{
        /* for the first start of the MScan */
        if(packages == 0)
        {
                MScanSettings(TCPsock);
                /* start MScan */
                sendSCPI(TCPsock, "init\n");
        }
        /* to continue the MScan */
        else
        {
                MScanSettings(TCPsock);
                /* continue MScan */
                sendSCPI(TCPsock, "init:conm\n");
        }
```

```
	/* do while loop to wait for the package with the right tag, in this
case 201 which stands for MScan */
	do{
		/* The function recvfrom(socket, buffer, size of buffer, flags,
		socket address struct, size of socket address) is defined in WinSock.h
		It receives the UDP packet from the receiver and returns the number
	of received bytes */
		reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
		(struct sockaddr *)&addrDevice, &remlen);
		if(reclen==SOCKET_ERROR)
		{
			printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
		}
		/* control if the received data is complete */
		EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
		datasize = ntohl(pEb200Header->DataSize);
		if(reclen != datasize)
		{
			printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
		}
		/* get the tag from the received package, for that put the data in the
		Attribute Header with an Offset of the EB200 Header Size */
		UDP_DATAGRAM_ATTRIBUTE_TYPE *attrHeaderMScan =
		    (UDP_DATAGRAM_ATTRIBUTE_TYPE*)(buffer + EB200_HEADER_SIZE);
		tag = ntohs(attrHeaderMScan->Tag);
	}while(tag != 201);
	/* stop MScan */
	sendSCPI(TCPsock, "abort\n");
	/* write the package into the File */
	if(fwrite(buffer, reclen, 1, stream)!=1)
		printf("Error: received less bytes than expected!");
/* getting the size of the Optional Header */
	UDP_DATAGRAM_ATTRIBUTE_TYPE *attrHeader =
(UDP_DATAGRAM_ATTRIBUTE_TYPE*)(buffer + EB200_HEADER_SIZE);
	/* always read out the length of the Optional Header from the Attribute
Header because in case something is changing in the Opt. Header it fits
automatically */
	/* Get the frequency from the UDP package */
```

```
DATA_HEADER_TYPE *Data = (DATA_HEADER_TYPE*)(buffer+ (EB200_HEADER_SIZE +
                     ATTRIBUTE_HEADER_SIZE + (attrHeader->OptHeaderLength)));
      freql = Data->FreqLow;
      freqh = Data->FreqHigh;
      frequency = freqh;
      frequency=frequency<<32;
      frequency=frequency|freql;

      /* settings for DFPan */
      DFPanSettings(TCPsock, frequency);
      /* do while loop to wait for the package with the right tag, in this
case 1401 which stands for DFPan */
      do{
             /* The function recvfrom(socket, buffer, size of buffer, flags,
             socket address struct, size of socket address) is defined in WinSock.h
             It receives the UDP packet from the receiver and returns the number
    of received bytes */
             reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
             (struct sockaddr *)&addrDevice, &remlen);
             if(reclen==SOCKET_ERROR)
             {
                    printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
             }
             /* control if the received data is complete */
             EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
             datasize = ntohl(pEb200Header->DataSize);
             if(reclen != datasize)
             {
                    printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
             }
```

```
            /* get the tag from the received package, for that put the data in the
                    Attribute Header with an Offset of 16 because of the EB200 Header */
                    UDP_DATAGRAM_ATTRIBUTE_TYPE *attrHeader =
                    (UDP_DATAGRAM_ATTRIBUTE_TYPE*)(buffer + EB200_HEADER_SIZE);
                    tag = ntohs(attrHeader->Tag);
            }while(tag != 1401);
            /* stop direction finding mode */
            sendSCPI(TCPsock, "abort\n");
            /* write the package into the File */
            if(fwrite(buffer, reclen, 1, stream)!=1)
                    printf("Error: received less bytes than expected!");
            /* increase the number of received packages */
            packages++;
        }
        fclose(stream);
}
/*******************************************************************/
```

3.4 Main

```
/***************************main***********************************/
void main(void)
{
        /* declare two variables of the type SOCKET, which is defined
     in WinSock.h */
        SOCKET TCPsock = INVALID_SOCKET;
        SOCKET UDPsock = INVALID_SOCKET;
        /* call the functions */
        connect(&TCPsock, &UDPsock);
        settings(TCPsock);
        streaming(UDPsock, TCPsock);

     printf("done!");
}
/*******************************************************************/
```

4 Panorama scan and direction finding

If you want to know which signals exist in an unknown area and where they come from, you can combine the panorama scan with the direction finding mode. The panorama scan itself gives you a quick overview of the spectrum occupancy. If any signal appears which is over the squelch level you set before, the scan will give you the direction information too. It is important to know that the signal has to be there for some time, because the program first scans the entire chosen spectrum and then switches to direction finding mode.

4.1 Connecting to the receiver

The connection between the receiver and the PC is over LAN and based on Windows sockets. To communicate with the receiver, you have to send SCPI commands over a TCP socket. For receiving the data stream, it is necessary to create an UDP socket. The following code shows an example of this.

4.1.1 Headers

```
/*******************************************************************/
/* Connecting to the Receiver                                      */
/*******************************************************************/
/*************************Header-Files*************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <winsock.h>
/* winsock.h has to be included to connect with the receiver,
additionally the library wsock32.lib has to be included, for
Visual Studio:
Project -> test Properties -> Configuration Properties ->
Linker -> Input, then type in wsock32.lib to the other libraries
(other development environments may need a different workflow)*/
/*******************************************************************/
/*************************Basic Header*************************/
typedef struct Eb200Header
{
       unsigned long MagicNumber;
       unsigned short      VersionMinor;
       unsigned short      VersionMajor;
       unsigned short      SeqNumber;
       unsigned short      Reserved;
       unsigned long DataSize;
```

```
} EB200_HEADER_TYPE;
#define EB200_HEADER_SIZE (sizeof( EB200_HEADER_TYPE ))

typedef struct UdpDatagramAttribute
{
	unsigned short	Tag;
	unsigned short	Length;
	unsigned short	NumItems;
	unsigned char ChannelNumber;
	unsigned char OptHeaderLength;
	unsigned long SelectorFlags;
	unsigned char OptHeader[22];
} UDP_DATAGRAM_ATTRIBUTE_TYPE;
#define ATTRIBUTE_HEADER_SIZE (sizeof( UDP_DATAGRAM_ATTRIBUTE_TYPE ))
/******************************************************************/
```

```
/**************************Optional Header***********************/
/* Optional Header for a Panorama Scan package */
typedef struct OptHeaderPScan
{
	unsigned long	StartFreq_low;
	unsigned long	StopFreq_low;
	unsigned long	StepFreq;
	unsigned long	StartFreq_high;
	unsigned long	StopFreq_high;
	char		reserved[4];
	DWORDLONG	OutputTimestamp; /* nanoseconds since Jan 1st, 1970, without leap sec-
onds */
} OPT_HEADER_PSCAN_TYPE;

/* Optional Header for a Direction Finding package */
typedef struct OptHeaderDFPan
{
	unsigned long			Freq_low;
	unsigned long			Freq_high;
	unsigned long			FreqSpan;
	signed long				DFThresholdMode;
	signed long				DFThresholdValue;
	unsigned long		DfBandwidth;
	unsigned long		StepWidth;
	unsigned long		DFMeasureTime;
	signed long		DFOption;
	unsigned short		CompassHeading;
	signed short		CompassHeadingType;
	signed long		AntennaFactor;
	signed long		DemodFreqChannel;
	unsigned long			DemodFreq_low;
	unsigned long			DemodFreq_high;
	DWORDLONG		OutputTimestamp; /* nanoseconds since Jan 1st, 1970, with-
out leap seconds */
} OPT_HEADER_DFPAN_TYPE;
/*****************************************************************/
```

4.1.2 Windows sockets

```
/**************************Functions*****************************/
/**************************Network-Connection-Test***************/
int checkWinSock()
{
       /* type WSADATA is defined in WinSock.h*/
       WSADATA wsaData;

       /* Testing if Network-Connection is ok */
       if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)
              /*Returning 0 if Connection is ok*/
       {
              /* Tell the user that we couldn't find a usable */
              /* WinSock DLL by returning 1                   */
              return 1;
       }
       /* Confirm that the WinSock DLL supports 2.0.*/
       /* Note that if the DLL supports versions greater  */
       /* than 2.0 in addition to 2.0, it will still return */
       /* 2.0 in wVersion since that is the version we    */
       /* requested.                      */

       if (LOBYTE(wsaData.wVersion) != 2 ||
              HIBYTE(wsaData.wVersion) != 0 )
       {      /* Tell the user that we couldn't find a usable */
              /* WinSock DLL by returning 2                   */
              return 2;
       }
       /* The WinSock DLL is acceptable. Proceed. */
       return 0;
}
/*****************************************************************/
```

4.1.3 TCP socket

```
/*************************Create TCP Socket*********************/
int createTCPSocket(SOCKET *pSock, unsigned long *pulRemoteAddress,
                            unsigned short *pusPort)
{
       /* Socket Initialization*/
       int iRet = 0;
       /* Test the Network Connection (see Network-Connection-Test)*/
       iRet = checkWinSock();
       /* Create a receiver control TCP socket */
       /* The function socket(address family, socket type, protocol) is defined
       in WinSock.h
       It builds up a socket for the connection between the receiver and the PC*/
       *pSock = socket(AF_INET, SOCK_STREAM, 0);
       /*Testing if there is an error*/
       if (*pSock == SOCKET_ERROR)
       {
              iRet = 3;
       }
       /* Set the socket options */
       int sopt = 1;
       /* The function setsockopt(socket, level, optname, optval, optlen)
       is defined in WinSock.h
       It sets the current value for a socket option associated with a socket of
       any type, in any state*/
       if ( setsockopt( *pSock, IPPROTO_TCP, TCP_NODELAY,
              (char *)&sopt, sizeof( sopt ) ) == SOCKET_ERROR )
       {
              iRet = 4;
       }
       /* Create socket address structure for INET address family */
       /* The struct sockaddr_in(family, type, address, port)
       is defined in WinSock.h
       This struct defines the structure of the socket, it consists of
       the address family, the address of the server you are connecting to
       and the port you want to connect*/
       sockaddr_in addrDevice;
       /* reserve memory space for the address struct */
       memset(&addrDevice, 0, sizeof(addrDevice));
```

```
	/* setting for TCP/UDP address family */
	addrDevice.sin_family = AF_INET;
	/* put the information for the UDP Mass Data Output port into the struct */
	addrDevice.sin_port   = htons(*pusPort);
	/* put the information for socket address into the struct */
	addrDevice.sin_addr.s_addr = *pulRemoteAddress;
	/* Connect socket to receiver device */
	/* The function connect(socket, socket_addr, size of struct socket_addr)
	is defined in WinSock.h
	It is connecting the receiver and the PC over the socket */
	if (connect(*pSock, (struct sockaddr *)&addrDevice, sizeof(addrDevice)) != 0)
	{
		iRet = 5;
	}
	return iRet;
}
/*******************************************************************/
```

4.1.4 UDP socket

```
/*************************Create UDP Socket********************/
int createUDPSocket(SOCKET *sock, unsigned long *pulRemoteAddress,
                                   unsigned short *pusPort)
{
        /* Socket Initialization*/
        int iRet = 0;
        /* Test the Network Connection (see Network-Connection-Test)*/
        iRet = checkWinSock();
        if (iRet != 0) return iRet;
        /* Create a receiver control TCP socket */
        /* The function socket(address family, socket type, protocol) is defined
        in WinSock.h
        It builds up a socket for the connection between the receiver and the PC*/
        *sock = socket(AF_INET, SOCK_DGRAM, 0);
        /*Testing if there is an error*/
        if (*sock == SOCKET_ERROR)
        {
                iRet = 3;
        }
        /* Create socket address structure for INET address family */
        /* The struct sockaddr_in(family, type, address, port)
        is defined in WinSock.h
        This struct defines the structure of the socket, it consists of
        the address family, the address of the server you are connecting to
        and the port you want to connect*/
        sockaddr_in addrDevice;
        /* reserve memory space for the address struct */
        memset(&addrDevice, 0, sizeof(addrDevice));
        /* setting for TCP/UDP address family */
        addrDevice.sin_family = AF_INET;
        /* put the information for the UDP Mass Data Output port into the struct */
        addrDevice.sin_port   = htons(*pusPort);
        /* put the information for socket address into the struct */
        addrDevice.sin_addr.s_addr = *pulRemoteAddress;
        /* The function bind(socket, pointer to the address structure, size of the struct) is
        defined in WinSock.h
        It associates a local address with a socket */
        if ( bind(*sock, (struct sockaddr *)&addrDevice, sizeof(addrDevice)) == SOCKET_ERROR)
        {
```

```
            printf("Couldn't bind TRACe socket - errno = %d\n", WSAGetLastError());
            closesocket(*sock), *sock = INVALID_SOCKET;
            iRet = 4;
        }
        return iRet;
}
/*******************************************************************/
```

4.1.5 Connect

```
/*************************Connect*****************************/
void connect(SOCKET *pSock, SOCKET *uSock)
{
	/* IP-address of the receiver (for the TCP socket) */
	const char *pcDeviceAddress = "172.25.10.19";
	unsigned long ulRemoteAddress = inet_addr(pcDeviceAddress);
	unsigned short usSCPIPort = 5555;
	// Port number for the receiver is always 5555

	/* IP-address of the PC (for the UDP streaming) */
	const char *pcPCAddress = "172.25.10.27";
	unsigned long ulPCAddress = inet_addr(pcPCAddress);
	unsigned short usUDPPort = 9000;
	// Port for UDP Mass Data Output to be sure that its not already
	// used, choose number above xxxxx
	printf("Connecting to PR100...\n");
	/* Creating a TCP socket */
	int iRet = createTCPSocket(pSock, &ulRemoteAddress, &usSCPIPort);
	if (iRet != 0)
	{	/* Error returning:*/
		switch(iRet)
		{
		case 1:
			printf("Error retrieving windows socket dll.\n");
		case 2:
			printf("Error retrieving correct winsock version 2.0.\n");
		case 3:
			printf("Error creating TCP socket.\n");
		case 4:
			printf("setsockopt (TCP_NODELAY) failed - errno %d\n",
				WSAGetLastError());
		case 5:
			printf("Error connecting to %s [SCPI port = %d]\n", pcDeviceAddress, usSCPI-
Port);
		}
	}
	else
	{
		printf("TCP Socket created!\n");
	}
```

```
        /* Creating an UDP Socket */
        int iUDP = createUDPSocket(uSock, &ulPCAddress, &usUDPPort);
        if (iUDP != 0)
        {       /* Error returning:*/
                switch(iUDP)
                {
                case 1:
                        printf("Error retrieving windows socket dll.\n");
                case 2:
                        printf("Error retrieving correct winsock version 2.0.\n");
                case 3:
                        printf("Error creating UDP socket.\n");
                case 4:
                        printf("\n");
                }
        }
        else
        {
                printf("UDP Socket created!\n");
        }
}
/*******************************************************************/
```

4.1.6 Sending SCPI commands

```
/***************************send string***************************/
int sendSCPI(int sd, char *pBuffer)
{
        unsigned int nLen;
        /* The function send(socket, buffer, size of buffer, flags)
        is defined in WinSock.h
        It sends bytes to the receiver over the socket and returns the
        number of bytes sent */
        nLen = send(sd, pBuffer, strlen(pBuffer), 0);
        /* Testing if all bytes are sent */
        if (nLen != strlen(pBuffer))
        {
                printf("Error writing to socket. Len = %d\n", nLen);
        }
        return nLen;
}
/*******************************************************************/
```

4.2 Settings

All the settings for the receiver can be made with SCPI commands over the TCP socket. This means you have the full functionality of the R&S®PR100 and R&S®EM100 via remote control.

```
/***************************settings*******************************/
void settings(SOCKET TCPsock)
{
	/* Get the basic information about the receiver with the
	SCPI command *idn? */
	sendSCPI(TCPsock, "*idn?\n");
	char pRxBuf[256];
	int len = recv(TCPsock, pRxBuf, sizeof(pRxBuf),0);
	if( len < 0 ) len = 0; pRxBuf[ len ] = '\0';
	puts( pRxBuf );
	/* sending SCPI commands for the needed settings */
	/* set the frequency mode to PScan */
	sendSCPI(TCPsock, "frequency:mode pscan\n");
	/* set the start frequency to 880 MHz */
	sendSCPI(TCPsock, "frequency:pscan:start 118 MHz\n");
	/* set the stop frequency to 960 MHz*/
	sendSCPI(TCPsock, "frequency:pscan:stop 138 MHz\n");
	/* set the number of Scans to infinite */
	sendSCPI(TCPsock, "pscan:count infinity\n");
	/* set the step size of the PScan to 50 kHz */
	sendSCPI(TCPsock, "sense:pscan:step 25 kHz\n");
	/* set the measurement time to 1 ms */
	sendSCPI(TCPsock, "measure:time 1 ms\n");
	/* set the measurement mode to periodic */
	sendSCPI(TCPsock, "measure:mode periodic\n");
    /* switch attenuation off*/
    sendSCPI(TCPsock, "input:attenuation:state off\n");
	/* DF Settings */
	sendSCPI(TCPsock, "bandwidth:dfinder 60000Hz\n");
	sendSCPI(TCPsock, "measure:dfinder:mode continuous\n");
	sendSCPI(TCPsock, "measure:dfinder:time 1s\n");

	/* Set the needed tags and flags for the PScan and DF streaming */
	sendSCPI(TCPsock, "trace:udp:tag:on '172.25.10.27', 9000, pscan, dfpan\n");
	sendSCPI(TCPsock, "trace:udp:flag:on '172.25.10.27', 9000, 'freq:low:rx',
	     'freq:high:rx', 'volt:ac', 'swap', 'opt', 'dflevel', 'azimuth',\n");
}
/*******************************************************************/
```

```
/**************************DFPan mode****************************/

void DFPan(SOCKET TCPsock, int frequency)
{
	sendSCPI(TCPsock, "frequency:mode df\n");
	/* set the frequency mode to Direction Finding */
	char *freq;//[13+sizeof(frequency)];//= "frequency ";
	freq = (char*) malloc(13+sizeof(frequency));
	sprintf(freq, "frequency %d Hz\n", frequency);

	sendSCPI(TCPsock, freq);

	sendSCPI(TCPsock, "init\n");
	Sleep(2000); // otherwise the program is too fast
}
/*****************************************************************/
```

4.3 Data stream

```
/*************************streaming***************************/
void streaming(SOCKET UDPsock, SOCKET TCPsock)
{
	/* create new File */
	FILE * stream;
	/* open File */
	if((stream = fopen("testfile_PScan_DF.rtr","w+b"))==NULL)
	{
		printf("\nCould not open File");
		exit(1);
	}

	/* create buffer for receiving */
	char buffer[0x80000];
	/* IP address of the PC */
	const char *pcPCAddress = "172.25.10.27";
	/* bring the IP address in the right format for WinSock.h functions */
	unsigned long ulPCAddress = inet_addr(pcPCAddress);
	/* port for UDP Mass Data Output */
	unsigned short usUDPPort = 9000;

	/* fill a struct which is defined in WinSock.h with information
	for the socket address format */
	sockaddr_in addrDevice;
	/* reserve memory space for the address struct */
	memset(&addrDevice, 0, sizeof(addrDevice));
	/* setting for TCP/UDP address family */
	addrDevice.sin_family = AF_INET;
	/* put the information for the UDP Mass Data Output port into the struct */
	addrDevice.sin_port    = htons(usUDPPort);
	/* put the information for socket address into the struct */
	addrDevice.sin_addr.s_addr = ulPCAddress;
	/* size of the socket address struct */
	int remlen = sizeof(sockaddr_in);
	/* counter variable for the amount of packages */
	int packages = 0;
	/* variable for the returning value of the fwrite function */
	int stor = 0;
	/* start the PScan */
```

```
sendSCPI(TCPsock, "init\n");
Sleep(2000); // otherwise the program is too fast
/* total number of signals over squelch */
int freqs_over_sql = 0;
/* number of signals over sql per package */
int number_of_freq_over_sql = 0;
/* variable for the frequency */
uint64_t frequency;
/* buffer to store the signals over sql */
uint64_t freqs_over_sql_buffer[1000];
/* variable to get the tag from the package */
int tag;
/* variable to get the number of traces from the package */
int number_of_traces;
/* loop count variable */
int loop_counter;
/* variables to compare received data with datasize */
int reclen;
int datasize;


/* receiving one complete PScan spectrum */
do
{
	/* The function recvfrom(socket, buffer, size of buffer, flags,
	socket address struct, size of socket address) is defined in WinSock.h
	It receives the UDP packet from the receiver and returns the number of
 received bytes */
	reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
	  (struct sockaddr *)&addrDevice, &remlen);
	if(reclen==SOCKET_ERROR)
	{
		printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
	}
	/* stop the Scan */
	sendSCPI(TCPsock, "abort\n");
	/* compare the length of the received data with the datasize in the
 EB200 Header */
	EB200_HEADER_TYPE *pEb200Header = (EB200_HEADER_TYPE*)(buffer);
	datasize = ntohl(pEb200Header->DataSize);
	if(reclen != datasize)
```

```
	{
		printf("Error: received bytes: %d != data size %d\n", reclen, datasize);
		exit(1);
	}
	/* store the whole UDP package for PScan into the file */
	if(fwrite(buffer, reclen, 1, stream)!=1)
		printf("Error: received less bytes than expected!");
	/* get the tag and the number of traces in the UDP package */
	UDP_DATAGRAM_ATTRIBUTE_TYPE *attHeader =
		(UDP_DATAGRAM_ATTRIBUTE_TYPE *)(buffer+ EB200_HEADER_SIZE);
	tag = ntohs(attHeader->Tag);
	number_of_traces = ntohs(attHeader->NumItems);
/* always read out the length of the Optional Header from the Attribute
Header because in case something is changing in the Opt. Header it fits
automatically */
	/* get the required information about to define which signals are over squelch */
	short *levelbuf = (short *)(buffer + (EB200_HEADER_SIZE +
			ATTRIBUTE_HEADER_SIZE + (attrHeader->OptHeaderLength)));
	long *freqlbuf = (long*)(buffer + (EB200_HEADER_SIZE +
					ATTRIBUTE_HEADER_SIZE + (attrHeader->OptHeaderLength))
			+ (2*number_of_traces));
	long *freqhbuf = (long*)(buffer+(EB200_HEADER_SIZE +
					ATTRIBUTE_HEADER_SIZE + (attrHeader->OptHeaderLength))
			+ (2*number_of_traces) + (4*number_of_traces));
	/* set variable for the number of frequencys over squelch per package to 0 */
	number_of_freq_over_sql = 0;
	/* loop to store frequencys over Squelch */
	for(loop_counter = 0; loop_counter <= number_of_traces; loop_counter++)
	{
		/* the PC is getting more frequencys than needed because of the resolution
		bandwidth and the start and stop parameters. So it is necessary to stop the
		Scanning manually at the required frequency. In this case at 138 MHz. */
		if(freqlbuf[loop_counter]<= 138000000)
		{
			/* test if the signal is over the required level, in this case 18 dBuV
*/
			short lev = levelbuf[loop_counter];
			if(lev>180)
```

```
                {
                        /* get the frequency and change it to the correct format */
                        frequency = freqhbuf[loop_counter];
                        frequency = frequency<<32;
                        frequency = frequency|freqlbuf[loop_counter];
                        /* store the frequency in a buffer */
                        freqs_over_sql_buffer[freqs_over_sql] = frequency;
                        /* increase the total number of signals over squelch */
                        freqs_over_sql++;
                        /* increase the number of signals over squelch (per package) */
                        number_of_freq_over_sql++;
                }
                /* increase the counter variable for the loop */
                loop_counter++;
        }
    }
    /* increase the number of received packages */
    packages++;
}while(frequency!=0);// until the end of one PScan
/* loop to store the DF packages for the signals over squelch */
for(loop_counter=0;loop_counter<freqs_over_sql;loop_counter++)
{
    /* get the frequencys from the buffer */
    frequency = freqs_over_sql_buffer[loop_counter];
    if(frequency > 0)
    {
        printf("%d\n", frequency);
        /* switch to DF mode */
        DFPan(TCPsock, frequency);
        /* do while loop to wait for the package with the right tag, in
   this case 1401 which stands for DFPan */
        do{
            /* The function recvfrom(socket, buffer, size of buffer, flags,
            socket address struct, size of socket address) is defined in WinSock.h
            It receives the UDP packet from the receiver and returns the number
      of received bytes */
            reclen = recvfrom(UDPsock, buffer,sizeof(buffer),0,
          (struct sockaddr *)&addrDevice, &remlen);
            if(reclen==SOCKET_ERROR)
```

```
                    {
                            printf("Error: recvfrom, error code: %d\n",WSAGetLastError());
                    }
                    UDP_DATAGRAM_ATTRIBUTE_TYPE *attrHeader =
                                    (UDP_DATAGRAM_ATTRIBUTE_TYPE*)(buffer+16);
                    tag = ntohs(attrHeader->Tag);
            }while(tag != 1401);
            /* stop direction finding */
            sendSCPI(TCPsock, "abort\n");
            /* store the DF data into the file */
            if(fwrite(buffer, reclen, 1, stream)!=1)
                    printf("Error: received less bytes than expected!");
        }
    }
    /* close the File */
    fclose(stream);
}
/*******************************************************************/
```

4.4 Main

```
/**************************main**********************************/
void main(void)
{
       /* declare two variables of the type SOCKET, which is defined
    in WinSock.h */
       SOCKET TCPsock = INVALID_SOCKET;
       SOCKET UDPsock = INVALID_SOCKET;
       /* call the functions */
       connect(&TCPsock, &UDPsock);
       settings(TCPsock);
       streaming(UDPsock, TCPsock);

    printf("done!");
}
/****************************************************************/
```

Service that adds value

- Worldwide
- Local and personalized
- Customized and flexible
- Uncompromising quality
- Long-term dependability

About Rohde & Schwarz

The Rohde & Schwarz electronics group is a leading supplier of solutions in the fields of test and measurement, broadcasting, secure communications, and radiomonitoring and radiolocation. Founded more than 80 years ago, this independent global company has an extensive sales network and is present in more than 70 countries. The company is headquartered in Munich, Germany.

Sustainable product design

- Environmental compatibility and eco-footprint
- Energy efficiency and low emissions
- Longevity and optimized total cost of ownership

Certified Quality Management
ISO 9001

Certified Environmental Management
ISO 14001

Rohde & Schwarz GmbH & Co. KG
www.rohde-schwarz.com

Regional contact

- Europe, Africa, Middle East | +49 89 4129 12345
  customersupport@rohde-schwarz.com
- North America | 1 888 TEST RSA (1 888 837 87 72)
  customer.support@rsa.rohde-schwarz.com
- Latin America | +1 410 910 79 88
  customersupport.la@rohde-schwarz.com
- Asia/Pacific | +65 65 13 04 88
  customersupport.asia@rohde-schwarz.com
- China | +86 800 810 8228/+86 400 650 5896
  customersupport.china@rohde-schwarz.com

R&S® is a registered trademark of Rohde & Schwarz GmbH & Co. KG
Trade names are trademarks of the owners

PD 3607.0098.92 | Version 01.00 | May 2014 (sk)
Remote control programming examples for R&S®PR100/R&S®EM100
Data without tolerance limits is not binding | Subject to change

© 2014 Rohde & Schwarz GmbH & Co. KG | 81671 Munich, Germany

3607.0098.92 01.00 PDP 1 en