Application Note

MASTERING BINARY DATA TRANSFERS WITH YOUR INSTRUMENT

Dr. Mathias Hellwig | 1SL381 | Version 1e | 03.2022

http://www.rohde-schwarz.com/appnote/1SL381

ROHDE&SCHWARZ
Make ideas real

Contents

1 Overview

Automating measurement setups is advantageous for multiple reasons. It saves time in case of repeated measurements and in hazardous environments, instruments can be operated from a distance. The measurements are repeatable because they are always performed with a defined procedure, leading to higher test confidence.

Automated systems are scalable. If structured correctly, the user can perform the same measurement on multiple devices under test (DUT) simultaneously. Recording and documenting the test results and test setups allow accurate comparison of different test scenarios.

The equipment used to perform these automated tasks is known as automated test equipment (ATE). It consists of:

- Controller (usually a PC or industrial computer)
- One or more test instruments capable of remote-control operation
- Communications interface between the controller, test instrument(s) and if necessary the DUT (usually Ethernet, USB or GPIB)
- Remote control software running on the control computer
- Device under test (DUT)

In remote control applications, users often perceive synchronization and binary transfers as challenging. Therefore, the focus in this application note is on binary transfer of data to and from the instrument. The basic parameters are discussed in chapter 3. Chapter 4 then guides you through the detailed setup. The code is also available for download.

For every example, two contrasting implementations are provided: a high-level approach in Python and a low-level approach in C++ 11. Both demonstrate the necessary configuration and transfer as well as the mapping to an appropriate data structure in particular a vector. For other programming languages like MATLAB and C#, a suitable solution can be derived from one or the other implementation example.

Four examples are presented in chapter 4. These examples differ in terms of the direction (upload or download) as well as the memory allocation. The first two examples (chapters 4.2 and 4.3) demonstrate a segmented operation (i.e. data is not fully available) with sequential processing. The given examples just write or read data from the disk. Other possible operations might involve a linear operation with the scale and offset parameters in case the format is integer only (8 bits or 16 bits), or power and mean calculations.

The final two examples (chapters 4.4 and 4.5) illustrate a complete data transfer for postprocessing. This might be required for a more complex analysis, for example a FFT. The right choice depends on the application and the available host memory. For example, an acquisition performed by an oscilloscope may require transfer of over 10 GB of data.

2 Prerequisites

In order to run these examples, the latest version of R&S®Visa should be downloaded and installed. Python version 3.6 or higher is required along with Visual Studio 2017. This corresponds to the used test setup.

The examples have been tested in C++ 11 as a 64-bit application (x64) and also as a 32-bit application (x86). The Python examples were only tested in a 64-bit environment.

Remote control is implemented using an ASCII-string based operation via the SCPI interface VXI-11, or HiSLIP (1). Using a raw socket connection is not considered in this article because this approach is not advised if another session like HiSLIP or VXI-11 is available. This is due to the fact that it is not available for USB and GPIB and may lead to a lockup since the device clear command is not available under all circumstances. Furthermore, it does not offer any performance advantage. IVI-COM or .NET are also not considered here.

For the C/C++ examples, viRead() and viWrite() were used instead of the viScanf() or viPrintf() functions. There are several reasons for this decision. Flexible buffer management is difficult using viScanf() or viPrintf() and portability between different R&S®Visa implementations is not ensured.

3 Basic parameters

Typically, there are two cases when a binary transfer is required. The first case involves file transfers to or from the instrument. This can include setup files, screenshots, waveform and trace files, generator files and so on. The second case concerns direct transfer of waveform data and traces without an intermediate step through a file system.

One question that might come up is why someone should use the SCPI protocol for the transfer? There are some scenarios where a file transfer via SCPI could make sense. Typically, remote file access is supported very well by state-of-the-art operating systems via the server message block protocol (SMB). Sometimes, however, no file export service is available on the instrument. This is either because the instrument is a value instrument with a low-performance CPU or a less advanced operating system, or because Ethernet access is not allowed for security reasons. In addition, the host OS does not support file access via USB (USB-TMC) or GPIB.

For direct transfers, there is no alternative for remote access and the lower latency compared to the file transfer might be a good reason for doing this. Moreover, the download or upload can be implemented in a segmented manner so it does not have to be handled in one chunk, especially for large sets of data. Instead, interleaved processing can be used. This is advantageous for older systems with a 32-bit architecture.

In order to make such a transfer work, a few details must be considered and configured appropriately.

Communications parameter		SCPI example
Format	uint, int, float	`FORMat:DATA REAL,32`
Size [bit]	8,16,32,64	`FORMat:DATA REAL,32`
Endianness	LSBF / MSBF	`FORMat:BORDer MSBF`
EOI char required for binary transfer	`0x0A`	`VI_ATTR_TERMCHAR_EN (visa attribute)`
Number of transmitted bytes	<length information>	See IEC60488-2 indefinite length arbitrary block response data

The first four items must match on both sides of the communication channel (host/instrument) in order to successfully establish communication. Depending on the instrument, the current setting of the parameter can be queried or looked up in the manual.

The format specifies the format of the individual data point for the transfer. The float format (32 bit) and double (64 bit) follow the IEEE 754 standard (2). The integer (int) and unsigned integer (uint) formats typically correspond to ADC raw data and must be scaled and shifted to obtain correct values. Using these integer formats for 8 bits and 16 bits has the advantage that the data volume per transfer is further reduced compared to the IEEE 754 format.

The endianness is an important parameter to ensure the correct byte order in transfers between the host and instrument. It applies to all transfers except those with a size of 8 bits. It is not specified in IEEE 488.2 (3) and must be determined by the application. Though the majority of hosts (especially those with an x86 architecture) implement LSBF or little endian, value instruments or older, PPC based, instruments may support MSBF by default. It is recommended in general to perform the conversion on the host since the host typically has a more powerful CPU than the instrument.

From a standard point of view, all transfers should be terminated with a termination character. There are a few instruments that do not require this at the end of binary transfers. The manual should provide this information. In the given examples, it is assumed that all transfers are terminated with this character.

All binary transfers have a header defined in IEEE 488.2 as **definite length arbitrary block response data** for transfers smaller than 1 GB and **indefinite length arbitrary block response data** for larger transfers.

Instruments from Rohde & Schwarz comply with the standard for transfers smaller than 1 GB (definite length). However, for larger transfers they deviate from the indefinite length arbitrary block response data and indicate the length in parentheses.

Type	Size	Field				
		1st	2nd	3rd	4th	5th
Definite	< 1 GB	'#'	Nonzero digit <field>	<length information>	<data>	

Type	Size	Field				
		1st	2nd	3rd	4th	5th
Indefinite	≥ 1 GB	'#'	Zero digit '0'	<data>		
R&S	≥ 1 GB	'#'	'('	<length information>	')'	<data>

For example, comparing the header for definite length arbitrary block response data with 250 kB with the indefinite length arbitrary block response data with 2.5 GB for the IEEE and R&S formats, we see the following:

- Definite 250 kB -- `#6250000<data>`
- Indefinite 2.5 GB -- `#0<data>`
- R&S 2.5 GB -- `#(2500000000)<data>`

4 Implementation

4.1 Initialization

During initialization, the resource manager is created and the instrument is connected (lines 21-22). A function `viCheckStatus()` checks the status and STB and displays the error code and code line in a text message. This C++ 11 function is part of the download package for this application note.

It is important for the buffer size to align with the size of the data. If a transfer is expected to handle 4-byte data such as float, the buffer size must be divisible by 4 (line 3).

In a second step, the Visa manufacturer, Visa version ID and termination character are queried from the Visa library (lines 24-27) and asserted to check which Visa library is used.

```
ViRsrc      visaResource = const_cast<ViRsrc>("TCPIP::R-RTA4004-02221::INSTR");

int32_t     bufferLength = 4096;
int32_t     nLengthField, nAccLength, nActRead, endianess, line;
int64_t     nDataLength;
bool        bigEndian; // MSBF is true
char        sCommand[1024];
char        *data, *bufferPtr;
char        *buffer = new char[bufferLength];

// Erase the buffer
// std::fill(sCommand, sCommand + sizeof(sCommand), '\0');
std::fill(buffer, buffer + bufferLength, '\0');

ViSession   defaultRM, instrument;
ViUInt8     viTermChar;
ViUInt32    retCount, viTimeout;
ViStatus    viStatus;
ViVersion   viVersion;

// Initialize VISA
viStatus = viOpenDefaultRM(&defaultRM);
viStatus = viOpen(defaultRM, visaResource, VI_NULL, VI_NULL, &instrument);
viCheckStatus(instrument, viStatus, __LINE__);

viStatus = viGetAttribute(instrument, VI_ATTR_TERMCHAR, &viTermChar);
viStatus = viGetAttribute(instrument, VI_ATTR_RSRC_MANF_NAME, buffer);
viStatus = viGetAttribute(instrument, VI_ATTR_RSRC_IMPL_VERSION, &viVersion);
printf("visa lib manufacturer\t:: %s\nVisa version\t\t:: %06x\n", buffer, viVersion);
```

For the high-level Python implementation, the initialization looks similar. The resource manager is created and the instrument is connected (lines 14-15).

In a second step, the Visa manufacturer and Visa version ID are queried from the Visa library (lines 17-20) and asserted. There is no need to retrieve the termination character since this is handled by the high-level implementation.

Only line 12 is different. Since the pyvisa package handles all transfers up to a size of 1 GB, it fails for transfers over 1 GB due to a change in the header for the indefinite length arbitrary block data (see chapter 3). For this reason, the function `parse_ieee_block_header()` was copied from the library and patched. In order to still use the library with the patched function `parse_rs_block_header()`, line 12 patches the package to use the modified function.

```
1  import numpy as np
2  import pyvisa.util
3  from pyvisa.constants import *
4  import util_rs
5
6
7  def test_bed(name):
8
9      visaResource = 'TCPIP::R-RTA4004-02221::INSTR'
10
11     # Monkey patch to cover the binary transfer (> 1 GB) for the R&S header
12     pyvisa.util.parse_ieee_block_header = util_rs.parse_rs_block_header
13
14     rm = pyvisa.ResourceManager()
15     instrument = rm.open_resource(resource_name=visaResource)
16
17     visaManufacturer = instrument.get_visa_attribute(VI_ATTR_RSRC_MANF_NAME)
18     visaVersion =  instrument.get_visa_attribute(VI_ATTR_RSRC_IMPL_VERSION)
19     print("visa lib manufacturer\t:: %s\nVisa version\t\t:: %06x\n"
20          % (visaManufacturer,visaVersion))
```

The patch of the original function `parse_ieee_block_header()` uses the initial segment in `block[]` and checks for an opening parenthesis (line 1) after the pound sign '#('. If this sequence is found, the two return values (offset and data length) are calculated. The data offset starts right after the closing parenthesis (line 4) and the data length is the value between the parentheses (line 4). Placing this code at the end of the function creates the working patch.

```
1  # R&S header for large transfer size (>1 GB)
2  #  "#(data_length)<data>"
3  if chr(block[begin+1]) == "(":
4      offset = block.find(b")") + 1
5      data_length = int(block[begin + 2: offset-1])
```

4.2 Downloading binary character data

The first binary download is a screenshot stored as a file “screenshot.png”. The code reads the header for the *definite length arbitrary block response data*. This ensures that the subsequent download of the data is correctly aligned no matter what format is used. In the initial lines (1-5), the format of the image data is set and the SCPI command string is created. Before the command is sent to the instrument, the timeout value is temporarily adjusted by a factor of 5 (lines 6-7) since image processing on the instrument takes longer than the VISA layer timeout. Then, the command is sent to the instrument and the first two bytes are read and examined (lines 4-7).

Based on the second byte, it is determined whether a *definite length arbitrary block* (lines 37-51), *indefinite length arbitrary block* (lines 52-56) or an R&S format (lines 26-36) is involved and the associated length information for the data block is read.

```
1   sprintf_s(sCommand, "HCOPy:LANGuage PNG");
2   viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
3   viCheckStatus(instrument, viStatus, __LINE__);
4
5   sprintf_s(sCommand, "HCOPy:DATA?");
6   viStatus = viGetAttribute(instrument, VI_ATTR_TMO_VALUE, &viTimeout);
7   viStatus = viSetAttribute(instrument, VI_ATTR_TMO_VALUE, 5 * viTimeout);
8   viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
9   viCheckStatus(instrument, viStatus, __LINE__);
10  viStatus = viRead(instrument, (ViBuf)buffer, 2, &retCount);
11  viCheckStatus(instrument, viStatus, __LINE__);
12  viStatus = viSetAttribute(instrument, VI_ATTR_TMO_VALUE, viTimeout);
13  viCheckStatus(instrument, viStatus, __LINE__);
14  *(buffer + 2) = '\0'; // Add termination char for str processing
15  bufferPtr = buffer;
16
17  if (*buffer != '#')
18  {
19      printf("error %s\n", buffer); exit(1);
20  }
21
22  bufferPtr++; // Examine next character
23
24  switch (*bufferPtr)
25  {
26  // R&S specific format "#(<value>)" for more than 1 GB data
27  case '(':
28      {
29          std::stringstream sstream;
30          do {
31              viRead(instrument, (ViBuf)buffer, 1, &retCount);
32              sstream << buffer[0];
```

```
33          } while (buffer[0] != ')');
34          sstream >> nDataLength;
35      }
36      break;
37  // IEEE 488.2 defined format for less than 1 GB data <definite length arbitrary block data>
38  case '1':
39  case '2':
40  case '3':
41  case '4':
42  case '5':
43  case '6':
44  case '7':
45  case '8':
46  case '9':
47      nLengthField = std::stoi(buffer + 1);
48      viStatus = viRead(instrument, (ViBuf)buffer, nLengthField, &retCount);
49      *(buffer + nLengthField) = '\0'; // add termination char str processing
50      nDataLength = std::stoi(buffer);
51      break;
52  // IEEE 488.2 defined format for more than 1 GB data <indefinite length arbitrary block data>
53  case '0':
54      nLengthField = 0;
55      nDataLength = 0;
56      break;
57  default:
58      printf("error %s\n", buffer);
59      break;
60  }
```

In a next step, the data is downloaded in a segmented manner. This means that the allocated buffer can be smaller than the size of the data for downloading.

The while loop condition (line 5) is based on the viRead() return code that indicates there is more data to be read. Instead of the regular VI_SUCCESS return code, the loop condition is based on the VI_SUCCESS_MAX_CNT return code. Any failure will abort the loop. Inside the loop, the buffer is immediately processed after the read operation and written to a file. After the loop has finished and the status is checked, the file is closed (lines 14-15).

This example illustrates the use of a data buffer with an independent length. It does not require buffer allocation in advance. A later example with a binary float download will show an alternative for large data without this preallocation.

```
1  // Example of segmented download, no intermediate storage and small memory footprint
2  auto myfile = std::fstream("screenshot.png", std::ios::out | std::ios::binary);
3  nAccLength = 0; // Accumulated length
4  retCount = bufferLength;
5  while (viStatus == VI_SUCCESS_MAX_CNT) // END indicator was not received
6  {
```

```
7       viStatus = viRead(instrument, (ViBuf)buffer, bufferLength, &retCount); line = __LINE__;
8       nAccLength += retCount;
9       // Ignore the pending END indicator of visa transmission
10      if (viStatus == VI_SUCCESS && *(buffer + retCount - 1) == viTermChar)
11          retCount--;
12      myfile.write(buffer, retCount);
13  }
14  viCheckStatus(instrument, viStatus, line);
15  myfile.close();
```

The result file from the binary download can be displayed with an image application.

For the high-level Python implementation, this operation is straightforward. Similar to the C++ example, the timeout value is temporarily adjusted by a factor of 5 (lines 2-3) since image processing on the instrument takes longer than the VISA layer timeout. Then, the command is sent to the instrument and the data is received (line 4). Then, the timeout is set back to the previous value (line 5).

The query delivers a binary list which is converted into an array that can be used to display the screenshot directly (lines 7-8).

```
1  instrument.write('HCOPy:LANGuage PNG')
2  timeout_prev = instrument.get_visa_attribute(VI_ATTR_TMO_VALUE)
3  instrument.set_visa_attribute(VI_ATTR_TMO_VALUE,timeout_prev*5)
4  out = instrument.query_binary_values('HCOPy:DATA?', datatype='B')
5  instrument.set_visa_attribute(VI_ATTR_TMO_VALUE,timeout_prev)
6  outData = bytearray(out)
7  image = Image.open(io.BytesIO(outData))
8  image.show()
```

4.3 Uploading binary character data

For the upload in the low-level implementation, a setup file is read in a segmented manner and transferred back to the instrument with a different name.

The binary header for the transfer requires the number of bytes to be transferred in advance. Accordingly, the file size is determined using the fstream library call (lines 5-9). Then, the SCPI command is assembled depending on the file size (lines 14-18).

For the low-level implementation below, it is important to understand that in a segmented transfer, the termination character for the segments must be suppressed (line 20). Otherwise, the instrument would stop receiving after the transfer of the first segment. In a second step, the SCPI command is written to the instrument together with the IEEE 488.2 header (line 21).

The condition for the while loop (lines 25-34) checks whether there is still data to transmit. Inside the loop, the data with the size of a segment is read (line 27) and transmitted to the instrument (line 32). Lines 30-31 make sure that for the final remaining data, the termination character is turned back on again. This ensures the instrument will finish receiving the data correctly.

Lines 38-44 read the directory on the instrument and print it out as confirmation of the transfer.

```
1  // Now transfer the setup file "DWNLTEST.SET" to the instrument
2  myfile = std::fstream("DWNLTEST.SET", std::ios::in | std::ios::binary);
3  // The file size is required to prepare the header for binary transmission
4
5  const auto begin = myfile.tellg();
6  myfile.seekg(0, std::ios::end);
7  const auto end = myfile.tellg();
8  nDataLength = (end - begin);
9  myfile.seekg(0);
10
11 // --- Upload (char) ------------------------------------------------------------------
12 // The assumption is a IEEE488.2 compliant header
13 // Now write it back to the instrument and change the name
14 if ( nDataLength < 1e9)
15     sprintf_s(sCommand, "MMEMory:DATA \"%s\", #%1d%lld", "DWNLTSNW.SET", \
16     (int)(ceil(log10((double)nDataLength))), nDataLength);
17 else // R&S instrument example
18     sprintf_s(sCommand, "MMEMory:DATA \"%s\", #(%lld)", "DWNLTSNW.SET", nDataLength);
19 // Turn off the write end char for the operation
20 viStatus = viSetAttribute(instrument, VI_ATTR_SEND_END_EN, VI_FALSE);
21 viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
22 viCheckStatus(instrument, viStatus, __LINE__);
23
24 nAccLength = 0; // Accumulated length
25 while (nDataLength > nAccLength)
26 {
27     myfile.read(buffer, bufferLength);
```

```
28      nActRead = myfile.gcount();
29      // Turn on the write end char for the last
30      if ( nDataLength <= nAccLength + nActRead)
31          viStatus = viSetAttribute(instrument, VI_ATTR_SEND_END_EN, VI_TRUE);
32      viStatus = viWrite(instrument, (ViBuf)buffer, nActRead, &retCount); line = __LINE__;
33      nAccLength += nActRead;
34  }
35  viCheckStatus(instrument, viStatus, line);
36  myfile.close();
37
38  sprintf_s(sCommand, "MMEMory:CATalog?");
39  viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
40  viCheckStatus(instrument, viStatus, __LINE__);
41  viStatus = viRead(instrument, (ViBuf)buffer, bufferLength, &retCount);
42  viCheckStatus(instrument, viStatus, __LINE__);
43  *(buffer + retCount) = '\0';
44  printf("DIR :: %s\n", buffer);
```

For the high-level Python implementation, this operation is even more straightforward. After reading in the local version of the setup file, the buffer inData is sent together with the command (line 3). Then, the directory content is read out.

The datatype parameter is set to unsigned binary.

```
1  with open('DWNLTEST.SET', 'rb') as f:
2      inData = f.read()
3  instrument.write_binary_values('MMEMory:DATA "DWNLTSNW.SET",', inData, datatype='B')
4
5  out = instrument.query('MMEMory:CATalog?')
6  print(out)
```

The result of the binary upload can be verified by loading the save set.

4.4 Downloading binary float data

Transfer of structured binary data such as float or int16 works similar to the described solutions. However, three additional things are required: the data format, the endianness and the mapping to a data structure. All three additions are demonstrated in the following two chapters.

For the download in the low-level implementation, the preparation for the transfer starts with the setting of the data format (lines 1-3). It is a 4-byte float in this example. In the next step, the endianness is determined. Although it is most likely little-endian, older instruments may possibly use big-endian. Some instruments have a SCPI command for reading this out (lines 5-10), but other instruments may require this detail to be looked up in the manual. Finally, it is stored in a boolean variable (line 11).

Lines 12-29 show an alternative method for dealing with the Visa layer timeout for an operation (`RUNSingle`) that exceeds this limit. Raising the timeout may not be practical since the time to complete the operation depends on the record length and this is undetermined. Instead of trying to adjust the Visa timeout, the operation complete bit (OPC) in the event status register (ESR) is indicated as the fifth bit in the status byte (STB), which is polled (lines 17-20). Once the bit is set in the STB, the ESR is read and it is checked whether the operation complete bit is set (lines 21-29).

This implementation works without any timing limitations and is thus suitable for any processing time.

```
sprintf_s(sCommand, "FORMat:DATA REAL,32");
viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
viCheckStatus(instrument, viStatus, __LINE__);

sprintf_s(sCommand, "FORMat:BORDer?");
viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
viCheckStatus(instrument, viStatus, __LINE__);
viStatus = viRead(instrument, (ViBuf)buffer, bufferLength, &retCount);
viCheckStatus(instrument, viStatus, __LINE__);
*(buffer + retCount) = '\0'; // Add termination char for str processing
bigEndian = (std::string(buffer) == "MSBF\n");

sprintf_s(sCommand, "*ESE 255; *CLS"); // Enable ESTR indication in STB and clear both
viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
sprintf_s(sCommand, "RUNSingle; *OPC"); // Note: *OPC (!) to set bit one in ESR
viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
statusByte = 0;
while (!(statusByte & (1 << 5))) // Poll the STB and check for the fifth bit in STB
{
    viStatus = viReadSTB(instrument, &statusByte);
}
sprintf_s(sCommand, "*ESR?"); // Read the EventStatusRegister for OPerationComplete
viStatus = viWrite(instrument, (ViBuf)sCommand, (ViUInt32)strlen(sCommand), &retCount);
viCheckStatus(instrument, viStatus, __LINE__);
viStatus = viRead(instrument, (ViBuf)buffer, bufferLength, &retCount);
viCheckStatus(instrument, viStatus, __LINE__);
if (!*buffer & 1) // Only the first ESR bit shows OPC!
{
    printf("error %x\n", *buffer);
}
```

Once the format parameter is set and the endianness is determined, binary header processing is the same as described in chapter 4.2. In other words, once the header is processed, the data download can start.

A robust implementation is still required for the byte swap, but it can be found on the internet. It uses C++ function templates, standard arrays and the built-in reverse_copy() function. The nice part is that it is independent of the input data type (uint16, uint32, uint64, float or double).

```
template <typename T>
void swapEndian(T &val) {
    union U {
        T val;
        std::array<std::uint8_t, sizeof(T)> raw;
    } src, dst;

    src.val = val;
    std::reverse_copy(src.raw.begin(), src.raw.end(), dst.raw.begin());
```

```
10      val = dst.val;
11  }
```

For the download functionality, the loop does not use the length of the binary data since in the IEEE 488 compliance header format, the data length indicator is missing for downloads ≥ 1 GB. If it is available, it is used to compared the announced data length with the transferred data length (line 25).

Since the length may be unknown, there is no way to allocate the buffer in advance. Thus, each single transfer is appended (after a potential byte swap) to the previously downloaded data. To make this convenient, the C++ 11 vector is used (line 1). For developers who are C purists, a linked list holding a buffer may be a better choice here.

The loop works similar to the loop described in chapter 4.2. The differences are in the removal of the term character (lines 8-10) before the data is converted from big to little endian (lines 12-20). The conversion is handled in a loop (lines 13-17) going to the character array with the step size equal to the data.

In the loop, the buffer data is cast to the correct data format – in this case, float (line 17) – and passed as a reference to the swap function.

When the transferred data buffer is byte-swapped, it is appended to the result vector end (lines 21-22) and the accumulated data is updated (line 23).

```
1   std::vector<float> datav;
2
3   nAccLength = 0; // Accumulated length
4   retCount = bufferLength;
5   while (viStatus == VI_SUCCESS_MAX_CNT) // END indicator was not received
6   {
7       viStatus = viRead(instrument, (ViBuf)buffer, bufferLength, &retCount);
8       // EOI reached, should not be included in the conversion
9       if (viStatus == VI_SUCCESS && *(buffer + retCount - 1) == viTermChar)
10          retCount--;
11      if (bigEndian)
12      {
13          for (  bufferPtr = buffer;
14                 (bufferPtr - buffer) < retCount;
15                 bufferPtr = bufferPtr + sizeof(datav[0])
16              )
17          {
18              swapEndian(reinterpret_cast<float &>(bufferPtr));
19          }
20      }
21      datav.insert(datav.end(), (float *)buffer, \
22                   ((float *)buffer) + (retCount / sizeof(datav[0])));
23      nAccLength += retCount;
24  }
25  if (nDataLength != 0 && nAccLength != nDataLength)
```

```
26      printf("error mismatch in download length\nData length ::\t%016lld,\naccumulated length
27  ::\t%016d\n", nDataLength, nAccLength);
28  viCheckStatus(instrument, viStatus, __LINE__);
29  viStatus = viSetAttribute(instrument, VI_ATTR_TMO_VALUE, viTimeout);
```

For the high-level Python implementation, this operation is much easier thanks to pyvisa. Before the data is queried, the format is set to a 32-bit IEEE 754 float format (line 1). The endianness is queried from the instrument (lines 2-5).

Similar to the C++ example, the status byte is polled in order to avoid a Visa timeout after the **RUNSingle** operation (lines 6-11).

Then, the data is queried again with two new parameters (line 8). One is the endianness 'is_big_endian' and the other is the mapping container 'container=np.array'. The result yields a numpy array with floats that can be used to directly display the transferred waveform (line 8).

```
1   instrument.write('FORMat:DATA REAL,32')
2   if instrument.query('FORMat:BORDer?') == 'MSBF\n':
3       endianess = True
4   else:
5       endianess = False
6   instrument.write('*ESE 255; *CLS') # Clear the status bytes
7   instrument.write('RUNSingle; *OPC')
8   esrInSTB = instrument.read_stb()
9   while ( not (esrInSTB & (1<<5))):
10      esrInSTB = instrument.read_stb()
11  esr = instrument.query('*ESR?')
12  ooo = instrument.query_binary_values('CHANnel1:DATA?', datatype='f', container=np.array,
13  is_big_endian=endianess)
14  plt.plot(ooo)
```

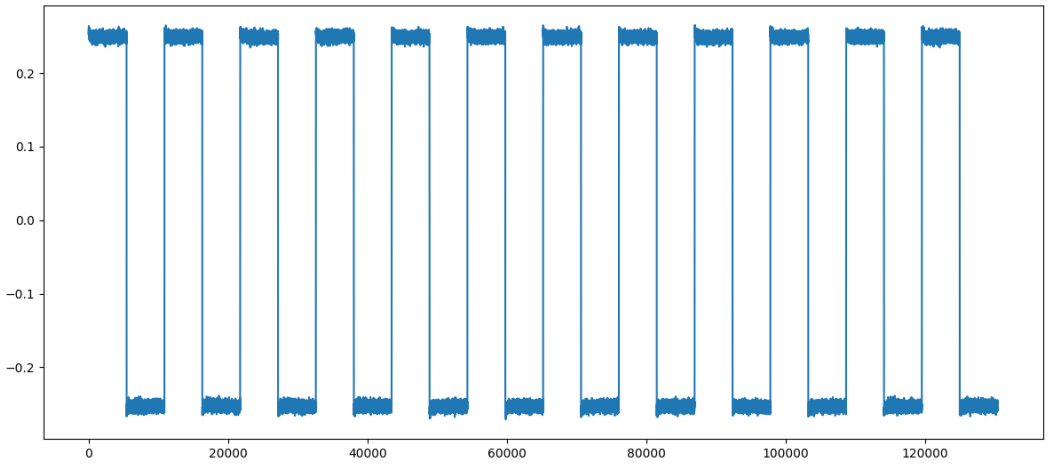

Figure 1 -- Captured waveform

The same waveform is captured with a screenshot that was downloaded as described in chapter 4.2.

Figure 2 -- Captured waveform as a screenshot

If it is necessary to run the Python script with a segmented transfer, the pyvisa package provides the `read_raw()` and `write_raw()` functions. The Python code should implement these functions similar to how the C code uses `viRead()` and `viWrite()`.

4.5 Uploading binary float data

The binary upload is not very different. For the C implementation of the binary character upload, the header format is already part of the SCPI format. The swap function that must be added in the loop is symmetrical, i.e. it operates both ways. Thus, any header format can be implemented.

For the high-level implementation, no change is required. The function definition is shown below. Setting the parameter `header_fmt` to `none`, the header can be part of the message string. The data type and endianness can be configured as needed.

```
def write_binary_values(
    self,
    message: str,
    values: Sequence[Any],
    datatype: util.BINARY_DATATYPES = "f",
    is_big_endian: bool = False,
    termination: Optional[str] = None,
    encoding: Optional[str] = None,
    header_fmt: util.BINARY_HEADERS = "ieee",
):
```

5 Summary

Binary transfer of data to and from an instrument is an important part of remote control. Although it can be confusing for unexperienced users, this task can be simplified and a robust implementation can be obtained by paying attention to the relevant configuration parameters.

The customer's choice of programming environment (high-level or low-level) should not represent an obstacle of any sort. Custom changes to the protocol can be added to both the low-level and the high-level implementation. The techniques that were demonstrated are directly usable in customer applications. It should also be a simple matter to convert the examples to other programming languages like MATLAB® and C#.

6 Literature

1. ***High-Speed LAN Instrument Protocol (HiSLIP).*** **Santa Rosa, CA 95402 : IVI Foundation, 2011. IVI-6.1.**

2. Committee, Microprocessor Standards. ***IEEE Standard for Floating-Point Arithmetic.*** **New York : IEEE Computer Society, 2019.**

3. ***Standard digital interface for programmable instrumentation – Part 2: Codes, formats, protocols and common commands.*** **Geneva : IEC, 2004.**

Rohde & Schwarz

The Rohde & Schwarz electronics group offers innovative solutions in the following business fields: test and measurement, broadcast and media, secure communications, cybersecurity, monitoring and network testing. Founded more than 80 years ago, the independent company which is headquartered in Munich, Germany, has an extensive sales and service network with locations in more than 70 countries.

www.rohde-schwarz.com

Certified Quality Management
ISO 9001

Rohde & Schwarz training

www.training.rohde-schwarz.com

PAD-T-M: 3572.7186.00/01.00/EN

Rohde & Schwarz customer support

www.rohde-schwarz.com/support

R&S® is a registered trademark of Rohde & Schwarz GmbH & Co. KG
Trade names are trademarks of the owners.
1SL381 | Version 1e | 03.2022
Application Note | Mastering binary data transfers with your instrument
Data without tolerance limits is not binding | Subject to change
© 2022 Rohde & Schwarz GmbH & Co. KG | 81671 Munich, Germany
www.rohde-schwarz.com