Digital I-Q Data Capture and Correction

Abstract: The RSA3408A Opt. 05 provides real-time digital I and Q outputs on the RSA3408A. Data of bandwidths up to 36 MHz span may be collected continuously at these ports. This **white paper** explains the theory of operation of the internal acquisition memory and data correction methods of the RSA3408A, and offers a method by which these corrections may be applied to data captured using an external PC.

1.0 RSA3408A Internal Acquisition, Storage, and Data Correction (Overview)

A brief description of the internal signal acquisition and correction process is provided as an introduction to the operations required when performing these functions on an external PC. Detailed descriptions of the internal operation of the RSA3408A can be found in the RSA3408A User Manual, P/N 071161701.

1.1 Internal Signal Acquisition:

The RSA3408A digitizes the incoming signal at an intermediate frequency of 76 MHz. This data path is 36 MHz wide, digitized at 102.4 MSamples/sec with 14 bits resolution. This data is then digitally downconverted to I and Q samples at 51.2 MSamples/sec for both the I and Q signals. This information is then decimated to produce the desired span of analysis, ranging from 36 MHz span to as narrow as 100 Hz. This uncorrected data is then stored in the acquisition memory of the RSA3408A.

1.2 Internal Post Acquisition Data Correction:

The data stored in the RSA3408A acquisition memory must be corrected for errors caused by the frequency response/group delay of the RF. Correction is performed by application of the Flatness Correction Data followed by the Calibration Data. These files are unique to each RSA3408A and are created during factory calibration. These corrections are performed automatically each time the data is used to produce measurement results in the RSA and the process is not seen by the user of the instrument. This is true even if the data has been stored to the hard drive.

2.0 RSA3408A Option 05 External Storage and Data Correction (Overview)

When external storage is used in conjunction with the Option 05 Digital IQ outputs, the correction process normally performed by the RSA software must be performed externally. A diagram of the hardware required and the data flow for external capture and correction is shown in Figure 1 below.

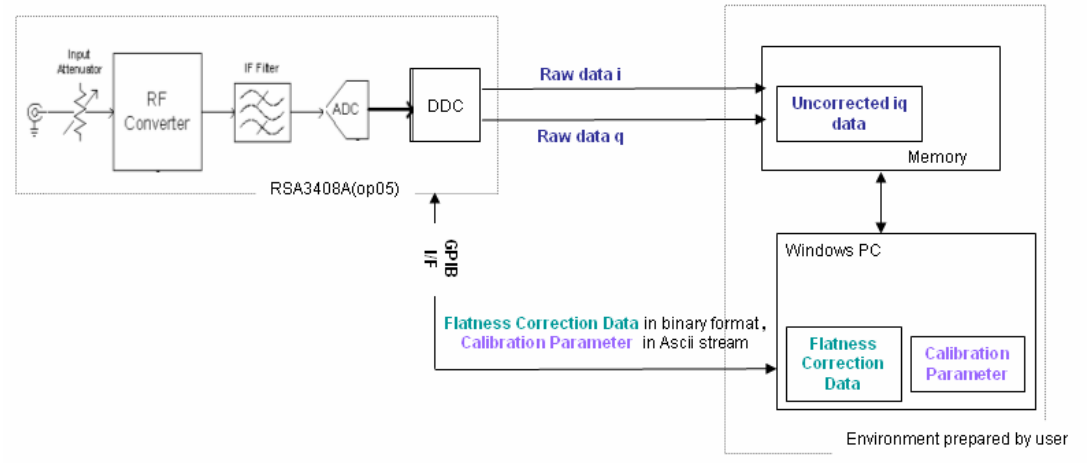


Figure 1: Hardware Wiring and Data Flow, RSA3408A Option 05

Figure 1 shows that the Flatness Correction Data and Calibration Parameter files are collected from the RSA3408A via GPIB commands sent from the data collection PC. Uncorrected (Raw) I and Q samples are collected into the PC memory via an interface cable and data collection card. Applying the calibration and correction files to the raw IQ samples results in a corrected IQ file ready for further processing. Each step of this process will now be considered in detail.

2.1 IQ Data Structure and Correction Factors in Detail

2.1.1 Raw IQ Data

Raw (uncorrected) IQ data points, are sent from the connector on the rear panel of the RSA3408A. 16 bit LVDS signals are used for both I and Q. Although the A/D converters are 14 bits in resolution, up to 16 bits of data may be available on I and Q as a result of span-dependent decimation. Since these signals are the same as data tapped at the output from the DDC of the RSA3408A, this is time-domain data. The data rate of the I and Q outputs is defined by the span setting of the RSA3408A. Data is sent from the option 05 outputs as shown in Figure 2.

Type : signed integer 16bit

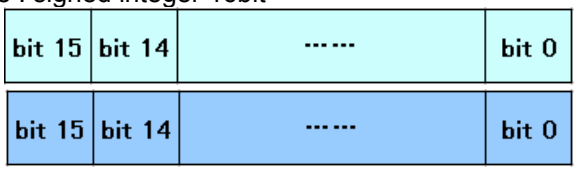


Figure 2: Option 05 Output Data Format

Figure **3** shows the relationship of span setting, data rate and data size of the captured signals. There is no limit to the data size that may be sent by these outputs, but memory sizes will limit the practical length of the collection time.

Span	Data rate	Data size for 10sec	Data size for 1 min
36 MHz	51.2 Msps	1.901GB	11.44 GB
20 MHz	25.6 Msps	0.95 GB	5.72 GB
10 MHz	12.8 Msps	488.28 MB	2.86 GB
5 MHz	6.4 Msps	244.14 MB	1.43 GB
2 MHz	2.56 Msps	97.66 MB	585.94 MB
1 MHz	1.28 Msps	48.83 MB	292.97 MB
500 kHz	640 ksps	24.41 MB	146.48 MB
200 kHz	256 ksps	9.77 MB	58.59 MB
100 kHz	128 ksps	4.88 MB	29.30 MB
50 kHz	64 ksps	2.44 MB	14.65 MB
20 kHz	25.6 ksps	0.98 MB	5.86 MB
10 kHz	12.8 ksps	500 KB	2.93 MB
5 kHz	6.4 ksps	250 KB	1.46 MB
2 kHz	2.56 ksps	100 KB	600 KB
1 kHz	1.28 ksps	50 KB	300 KB
500 Hz	640 sps	25 KB	150 KB
200 Hz	256 sps	10 KB	60 KB
100 Hz	128 sps	5 KB	30 KB

Figure 3 : Relation of span, data rate and data size

For compensation of the raw IQ daa stream using the SW algorithm in this documentation, a data file that has the following data structure is required. The user should prepare their HW to capture raw IQ data into PC memory in accordance to the following structured data file (Figure 4). A method for transferring raw IQ data into a PC memory is described in Section 3**.** The raw IQ file should be stored on the controller PC as “RAW_IQ.DAT

File contents: IQ data to be converted in [Voltage] and to be corrected.

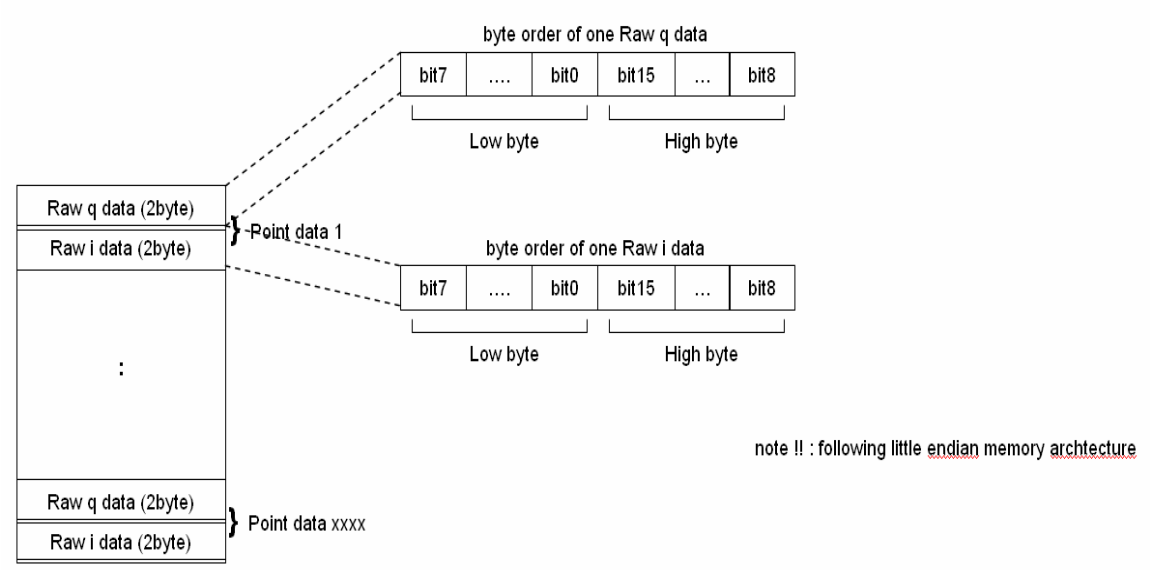


Figure 4: data structure of whole Raw iq data

2.1.2 Flatness Correction Data (Gain and Phase)

This data is used for compensating flatness errors in the uncorrected IQ data file and consists of gain and phase components. The GPIB command used to retrieve the gain components of the flatness correction data is:

":CALibration:IQ:CORRection:MAGNitude? "

The Data format of the response from the RSA3408A is shown in Figure 5.

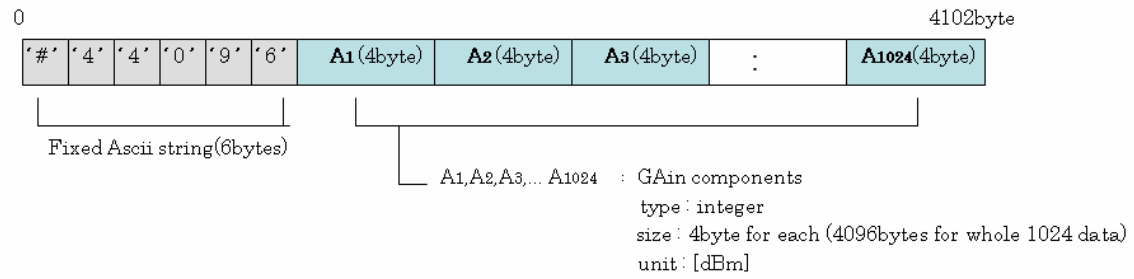


Figure 5 : data format of Gain components

Similarly, the command for retrieving phase correction data is:

" :CALibration:IQ:CORRection:PHASe? "

The Data format of the response from the RSA3408A is shown in Figure 6.

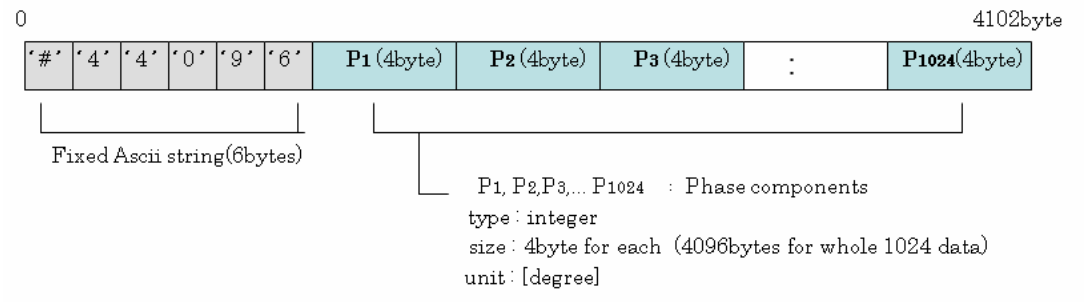


Figure 6: data format of Phase components

The Gain and Phase components correspond to a frequency bin and both are expressed in the frequency domain. The formulas for producing Flatness Correction data from the files retrieved from the RSA3408A as I and Q in floating point is shown below.

1) a pair of Gain and Phase corresponds to a frequency bin
2) data under frequency domain
3) formula to see Flatness Correction Data, just retrieved from RSA3408A, as IQ factor counted by floating point

Amplitude n = $\sqrt{}$ 10 (An/32768 x 10)
An : A1, A2,…..A1024 (As in Fig: 5)
Phase n [radian] = **Pn** / 32768.0 * PI / 180
Pn : P1, P2,…..P1024 (As in Fig. 6)

Therefore, the formula to provide IQ values of the Flatness Correction table from ***Amplitude*** and ***Phase*** is as below.

Flatness Correction data I**n** = 1.0 / ***Amplitude n*** * cos(***Phase n***)
Flatness Correction data Q**n** = 1.0 / ***Amplitude n*** * sin(***Phase n***)

2.1.3 Calibration Parameter file

A calibration parameter file is generated in the RSA3408A during setup of the instrument for data acquisition. The calibration parameter file corrections must be performed on externally stored raw IQ files. These parameters are:

i) GainOffset [dB] : gain offset for amplitude value
ii) MaxInputLevel [dBm] : reference level set by RSA3408A when acquiring data
iii) LevelOffset [dB]: level offset set by RSA3408A when acquiring data
iv) IOffset : Offset value of I data
v) QOffset : Offset value of Q data

All of the parameters listed above can be retrieved via GPIB command as shown below (Figure 7).

GPIB Command: ***“ :CALibration:IQ:HEADer? “***
Data format of response from RSA3408A (note: data is in ASCII code):

```
"Type=RSA3408AIQT
FrameReverse=Off
FramePadding=Before
Band=RF1
MemoryMode=Zoom
FFTPoints=1024
Bins=721
MaxInputLevel=0
LevelOffset=0
CenterFrequency=1.5G
FrequencyOffset=0
Span=36M
BlockSize=2
ValidFrames=3730
FramePeriod=20u
UnitPeriod=20u
FrameLength=20u
DateTime=2005/01/24@14:56:42
GainOffset=-82.2601145991602
MultiFrames=1
MultiAddr=0
IOffset=0.0361328125
QOffset=-0.01800537109375
“
```

Figure 7: Response of GPIB command ***:CALibration:IQ:HEADer?***

2.2 Calibration and Corrections: Time and Frequency Domains

As is seen in Figure 8 (Below), the calibration factors (amplitude scaling) are applied to the raw time domain I and Q samples. The frequency-domain flatness correction data is applied following an FFT of the amplitude scaled I and Q samples. After the Flatness Correction data is applied, an IFFT is performed, resulting in scaled, corrected I and Q data.

Details on the frequency domain processing of the flatness correction data are found in Appendix **C and D**

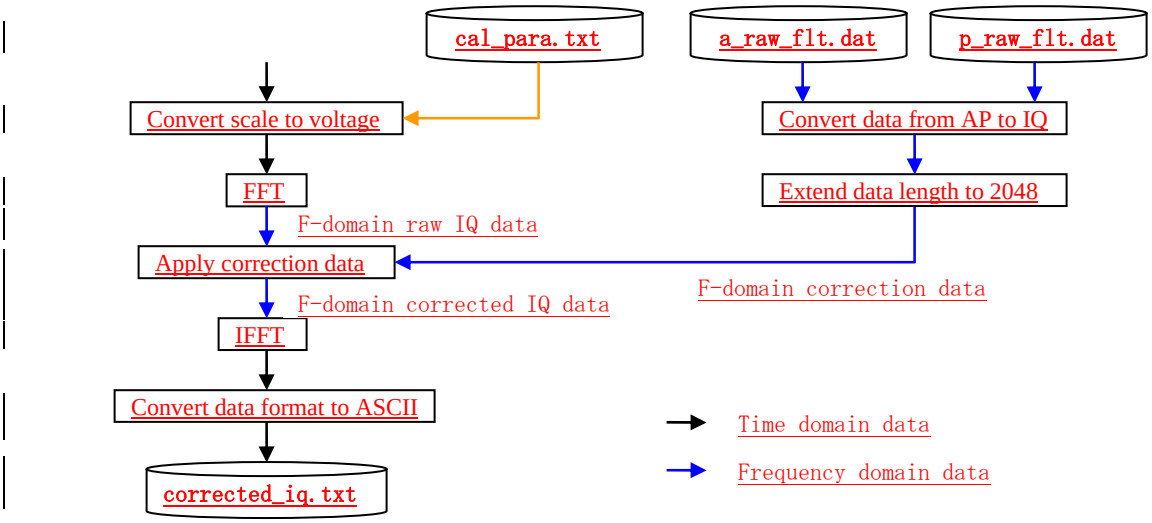


Figure 8: Corrections Applied in Time and Frequency Domains

3.0 An Application Example

This section will demonstrate the collection of raw IQ files into an external PC, and provides software (See Appendix A) implementing the described IQ correction process. Using these techniques, corrected data from the RSA3408A can be used by external data analysis software with confidence in the accuracy of the result.

3.1 Setup Description

The setup required for capture of raw IQ from the RSA3408A is shown in Figure 9 below.

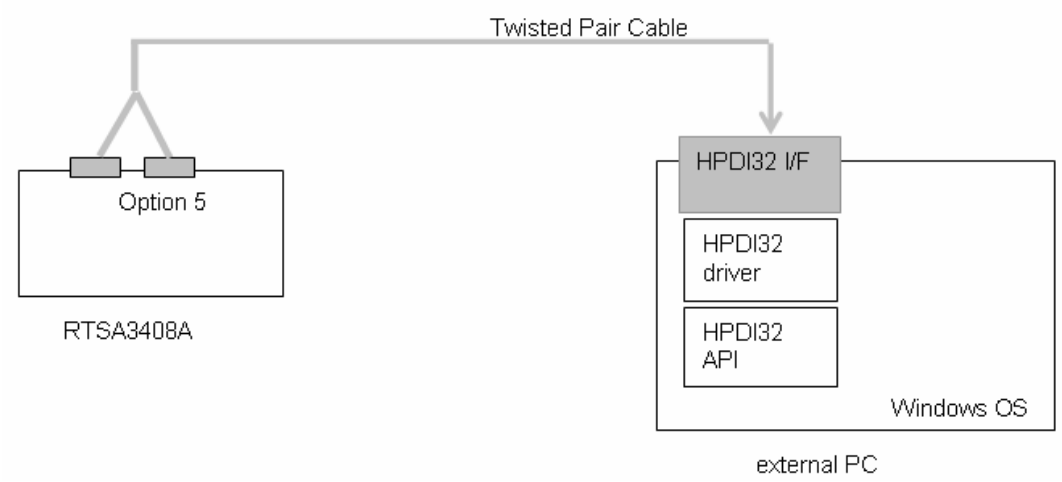


Figure 9: Hardware and SW Diagram for Raw IQ Capture

3.1.1 Preparation – 32bit Parallel Digital I/O Card

Tektronix recommends the hardware and software described below for data capture from the RSA3408A Option 05.

Digital I/O card : PCI64-HPDI32A
Cable : twisted pair cable (see 3.1.3)
sw : HPDI32 driver/API Release 4.0.0.7, 2004-12-29
vendor: General Standards Corporation (http://www.generalstandards.com)

Note: When RSA3408A option 5 operates at 36MHz span, the rate of IQ data out become 204.8 Mbytes/sec. (16 bits X 2 X 51.2 Mbits/sec) Tektronix recommends to use 64 bits PCI interface.

3.1.2 Installation – 32 bit Parallel Digital I/O Card

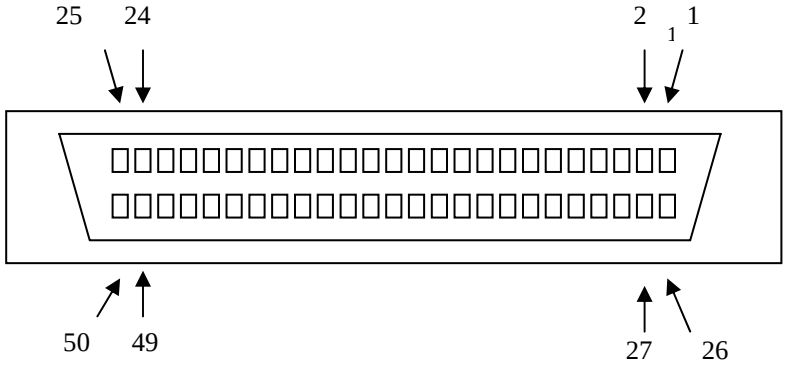
The process to setup is the following.

1) prepare an external PC running Windows.
2) Install PCI64-HPDI32A DIO card into external PC
3) install HPDI32 driver in external PC.
4) connect Option 5 on rear panel of RSA3408A to external PC via 32bit HPDI32A I/F using the twisted pair cable (see 3.1.3)

3.1.3 Connecter on the rear of RSA3408A

RSA3408A has two connecters for Digital IQ output on the rear panel, one is for “I” out and the other one is for “Q” out. The connecter specification is as following.

Pin assignment (Rear view)



Connector P/N 10250-1210VE
Vender of connecter: 3M corp.

The connection details between RSA3408A rear connecters and I/F card are shown in Figure 2

Note:
Regarding the Both connecters for raw I and Q data
Pin1 and Pin26 should be connected to ground at the RSA3408A side
Pin25 and Pin50 are clock output from RSA3408A.

3.1.4 Twisted Pair Interface

The cable connecting the RSA3408A to the digital I/O card can be either made by the user or purchased.

If you wish to produce the cable in-house, connection details are shown in Figure 10.

Pin assignment for connecting to Digital IO card.

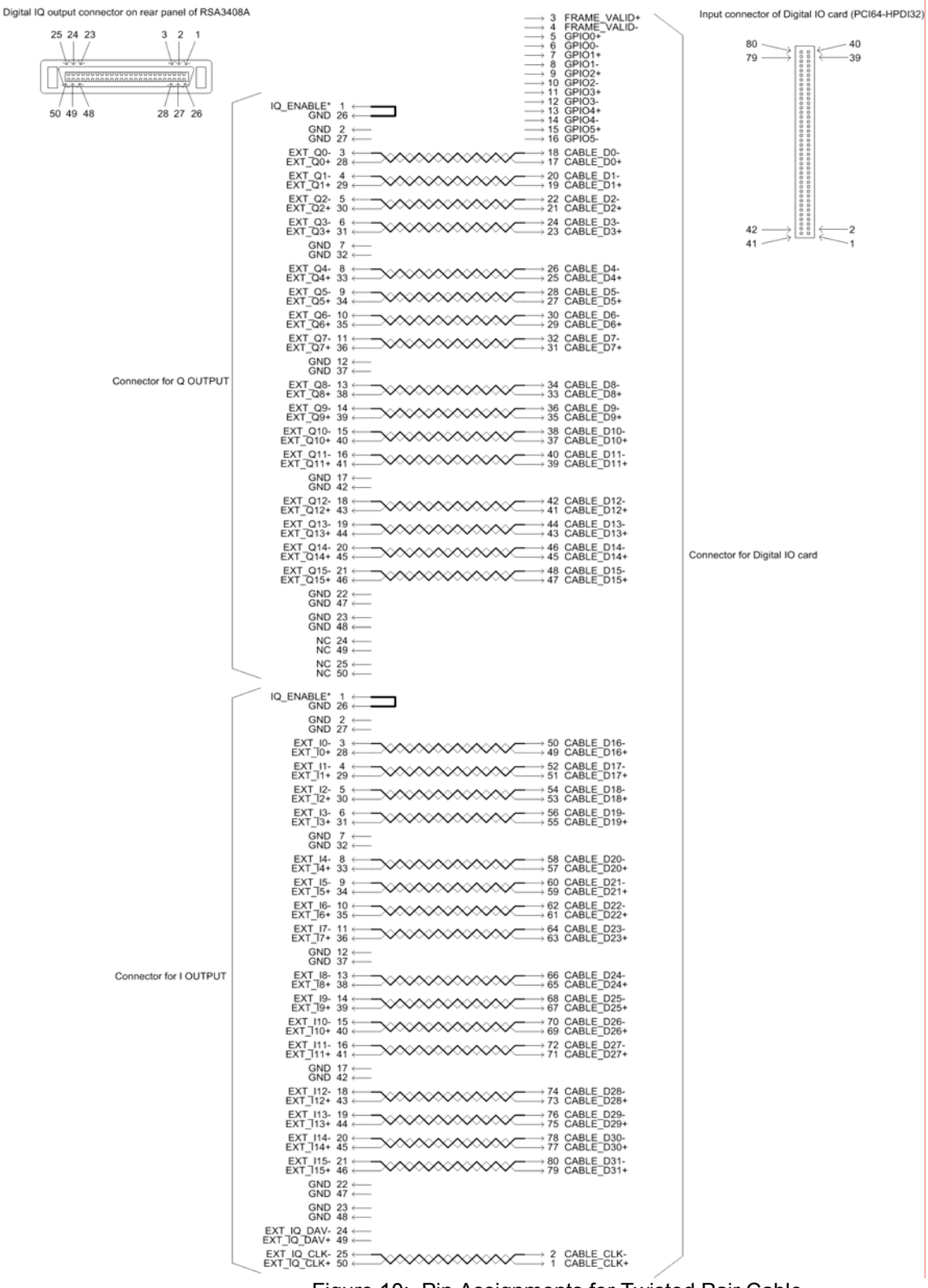


Figure 10: Pin Assignments for Twisted Pair Cable

3.2 Collect raw IQ data from RSA3408A, and Data correction

The steps for retrieving raw IQ data from Option 5 of RSA3408A are the following.

1) Setup environment described in 3.1
2) Enable Option 5 on RSA3408A, see user manual.
3) On external PC, invoke sample program explained in 4.1 to capture data from RSA3408A.
4) On external PC used in step 3 or other, prepare of flatness correction file and calibration parameter file described in 4.2.1.
5) On either external PC used in above step 3 or other, invoke sample flatness correction program explained in 4.2 together with input file, RAW_IQ.dat, just created in previous step 3). See 4.2.
6) corrected_iq.txt, described in 4.2.2, will be created. See 4.2 and Appendix D

4.0 Sample programs

This application note has sample source programs to install the following four commands.

- captureiq:
  This command is used for data gathering from HPDI32A interface board and creates a raw IQ data file (raw_iq.dat) for data recording.

- makecaldata:
  This command creates three calibration files from an IQT file. The calibration files are as follows.

  1) Amplitude flatness correction data file (a_raw_flat.dat)
  2) Phase flatness correction data file (p_raw_flat.dat)
  3) Calibration parameter file (cal_para.txt)

  The contents of these files can be received by using GPIB commands also (See 3.1).

- makeiqt:
  This command creates an IQT format file (captured.iqt) from the raw IQ data file and the calibration files.

- correctiq:
  This command corrects IQ data in the raw IQ data file by using the calibration files and writes the corrected IQ data into an ASCII IQ data file (corrected_iq.txt).

The relation between these commands and files is shown in Figure11.

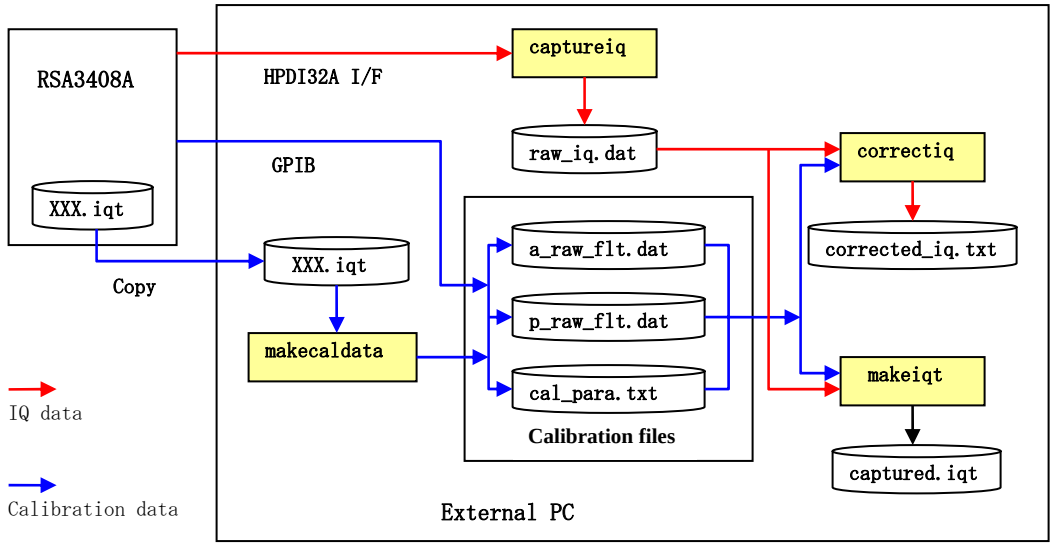


Figure 11 Relation between commands and files

Microsoft Visual C++ v6.0 is used to build execution modules of these commands from sample programs. All execution modules are executed as DOS commands.

4.1 Functional Description, Captureiq

4.1.1 Installation

The following files are needed to install captureiq command. Put these files into your working folder (Header files and library file are provided by General Standard Corporation).

- Source file : captureiq.c (Appendix-A)
- Header files : gsc_common.h, gsc_pci9080.h, gsc_pci9656.h, hpdi32_api.h
- Library file : hpdi32_api.lib

The execution module (captureiq.exe) is created by the following command.

- > cl captureiq.c hpdi32_api.lib

4.1.2 Synopsys

- captureiq [sampling-points]
- One sampling data point is 4 bytes and one data sample data interval is depends on the RSA3408A SPAN setting.

4.1.3 Input, output

- Input : Raw IQ data from HPDI32A interface board.
- Output file : raw_iq.dat (see 3.1.1)

4.1.4 Description

Captureiq command captures IQ data from HPDI32A interface board and saves the data into a raw IQ data file. The raw IQ data file name is raw_iq.dat. The command argument ‘sampling-points’ is an integer value and captureiq captures and saves specified points of IQ data. If no argument is given, captureiq captures 10G points data.
(This command requires 1GB memory on PC as software FIFO buffer.)

4.2 Functional Description, Makecaldata

4.2.1 Installation

The following files are needed to install makecaldata command. Put these files into your working folder.

- Source file : makecaldata.c (Appendix-B)
- Header file : rtsa_iqt.h (Appendix-E)

The execution module (makecaldata.exe) is created by the following command.

- > cl makecaldata.c

4.2.2 Synopsys

- makecaldata IQT-file

4.2.3 Input, output

- Input file : IQT format file
- Output files : a_raw_flt.dat, p_raw_flt.dat, cal_para.txt (see 3.1.2 and 3.1.3)

4.2.4 Description

Makecaldata command reads data for calibration from an IQT file and creates three calibration files. The IQT file name should be specified by the command argument ‘IQT-file’.

4.3 Functional Description, Makeiqt

4.3.1 Installation

The following files are needed to install makeiqt command. Put these files into your working folder.

- Source file : makeiqt.c (Appendix-C)
- Header file : rtsa_iqt.h (Appendix-E)

The execution module (makeiqt.exe) is created by the following command.

- > cl makeiqt.c

4.3.2 Synopsys

- makeiqt

4.3.3 Input, output

- Input files : raw_iq.dat, a_raw_flt.dat, p_raw_flt.dat, cal_para.txt
- Output file : captured.iqt

4.3.4 Description

Makeiqt command reads IQ data, file header information and flatness correction data from the raw IQ data file and the calibration files, then creates IQT format file according to the data. The IQT format file name is captured.iqt.

4.4 Functional Description, Correctiq

4.4.1 Installation

The following file is needed to install correctiq command. Put this file into your working folder.

- Source file : correctiq.c (Appendix-D)

The execution module (correctiq.exe) is created by the following command.

- > cl correctiq.c

4.4.2 Synopsys

- correctiq

4.4.3 Input, output

- Input files : raw_iq.dat, a_raw_flt.dat, p_raw_flt.dat, cal_para.txt
- Output file : corrected_iq.txt

4.4.4 Description

Correctediq command reads IQ data and calibration data from the raw IQ data file and the calibration files, corrects IQ data according to the calibration data, converts the data format from binary to ASCII, then writes the data into an ASCII IQ data file. The method of data correction is described in 4.6. The ASCII IQ data file name is corrected_iq.txt.

4.5 Data files

4.5.1 Raw IQ data file (raw_iq.dat)

Raw IQ data file has a sequence of IQ data captured from HSDP32A interface, which is described in 3.1.1. One point data in this file is composed by two short integers which correspond to I and Q value, so the size of a point data is 4 bytes. If the number of points captured by cuptureiq command is N, the size of raw IQ data file is 4N bytes. A point data is read from HSDP32A interface as an integer (4 bytes) which has both I and Q data, and the byte order of the Windows PC is little endian. So the byte order of IQ values in this file is shown in Figure12.

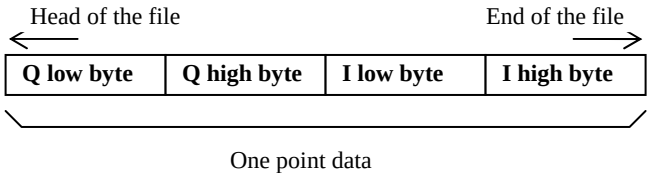


Figure 12: Byte order of IQ in raw IQ data file

4.5.2 Amplitude flatness correction data file (a_raw_flt.dat)

Amplitude flatness correction data file has 1024 integers (4096 bytes) which are described in 3.1.2 (A1, A2, …, A1024). This file is created by makecaldata command, otherwise the contents of this file can be gotten from the response of GPIB command. The byte order of an integer is little endian in this file (Low byte first).

4.5.2 Phase flatness correction data file (p_raw_flt.dat)

Phase flatness correction data file has 1024 integers (4096 bytes) which are described in 3.1.2 (P1, P2, …, P1024). This file is created by makecaldata command, otherwise the contents of this file can be gotten from the response of GPIB command. The byte order of an integer is little endian in this file (Low byte first).

4.5.3 Calibration parameter file (cal_para.txt)

Calibration parameter file has measurement parameters as text data, which are described in 3.1.3. This file is created by makecaldata command, otherwise the contents of this file can be gotten from the response of GPIB command.

Correctiq command converts raw IQ data to IQ values in voltage using some calibration parameters (GainOffset, MaxInputLevel, LevelOffset, IOffset, QOffset). The following formulas are applied for the conversion (Iraw and Qraw are IQ values in raw IQ data file).

IQScale = sqrt(Power(10,(GainOffset + MaxInputLevel + LevelOffset)/10/20*2))

I [V] = (Iraw – IOffset) * IQScale
Q [V]= (Qraw – QOffset) * IQScale

4.5.4 Corrected ASCII IQ data file (corrected_iq.txt)

Corrected ASCII IQ data file has a sequence of IQ data in ASCII format. In this file, one line has two real values which correspond to I and Q. The two values are separated by comma (,) as shown below.

-1.520849e-002,2.921454e-003
-1.520110e-002,7.701263e-004
1.432840e-002,-1.596774e-003
.
.
.

4.6 Details of frequency-domain flatness correction process

1) It needs to extend flatness correction data with 1024bins to 2048bins. Following is the process of extension.

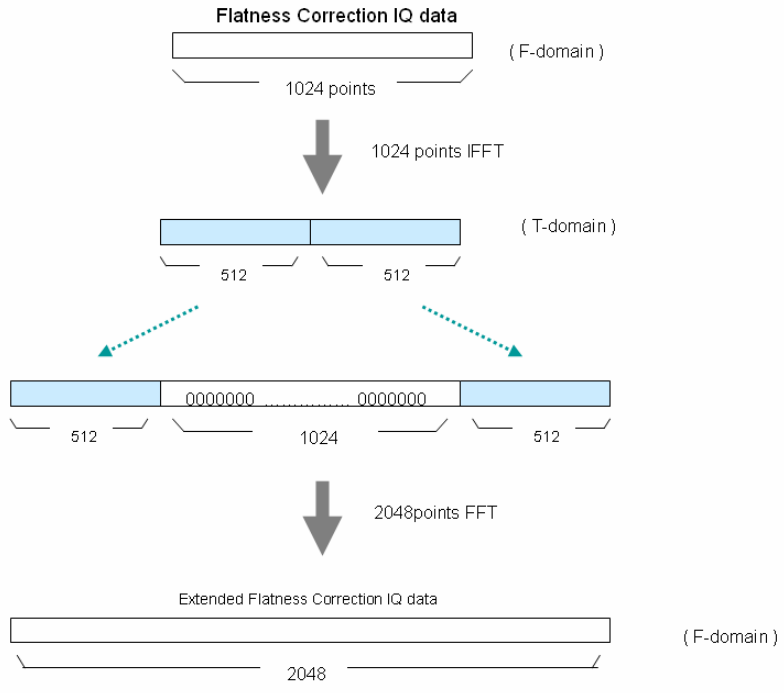


Figure 13: Extension of flatness correction data

2) Compensating IQ data with N points length : framing
(Making one data block, named “frame” as one data block in this case)
Assume IQ data to be corrected has **N** points length of itself, making each data frame whole IQ data into **n** frames under the below condition.

N = 1024 * (**n** -1) + **m** (**n**, **m** : integer, 0 < **m** <= 1024)

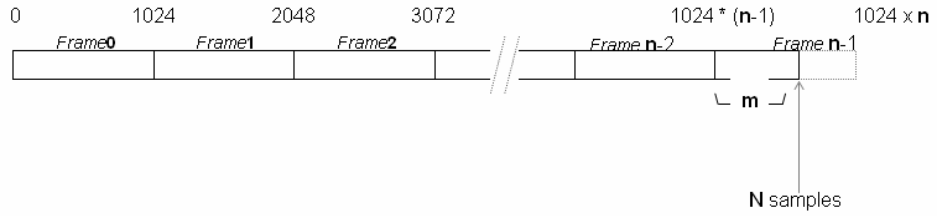


Figure 14: Before filling up to 1024 points with “0” at last frame as a dummy data

3) Compensating IQ data with N points length : padding with “0” data as a dummy

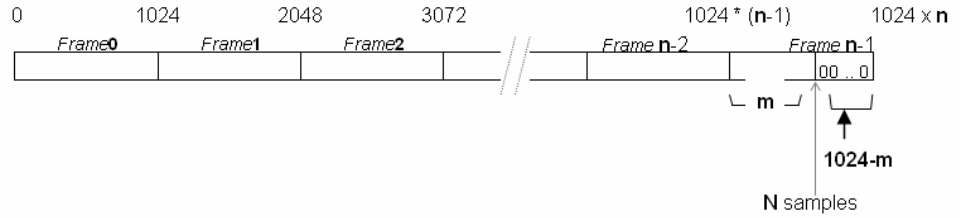


Figure 15: After filling up to 1024 points with “0” at last frame as a dummy data

4) Compensating IQ data with N points length: apply flatness correction data compensating each frame, called Frame **i**, where **i** will be 0 through **n**-1

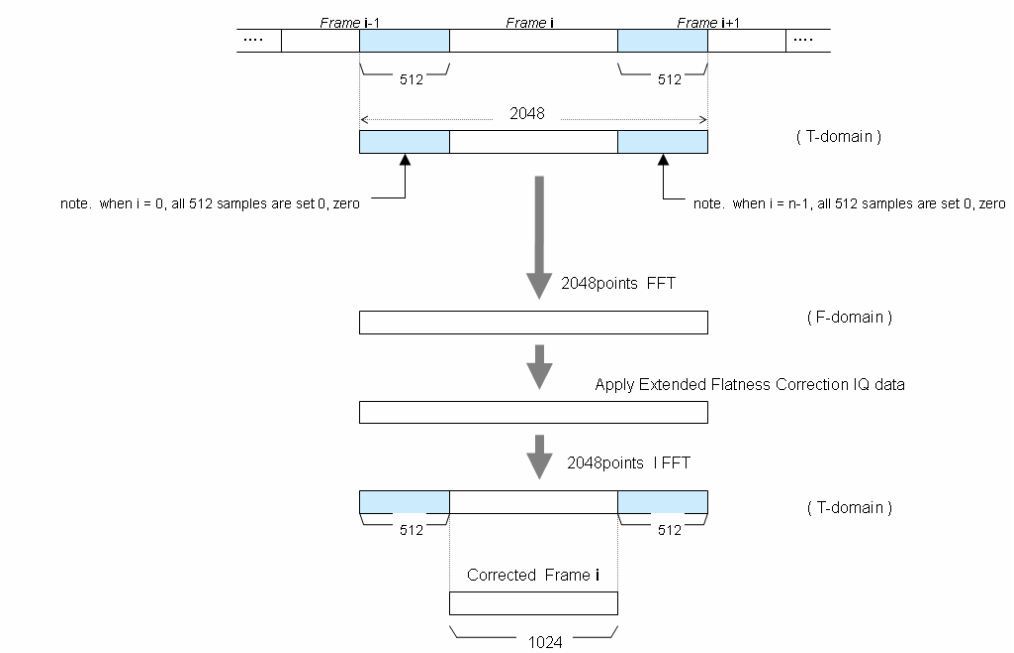


Figure 16: Application of flatness correction data

Appendix A: *captureiq.c*

```
/* Copyright (C) Tektronix */

#include <windows.h>
#include <stdio.h>
#include <fcntl.h>

#define MIN_FIFO_BYTES        64000000
#define REPORT_POINTS         256000000
#define OPEN_MODE   (O_WRONLY | O_TRUNC | O_CREAT | O_BINARY | O_SEQUENTIAL)

#define READ_WAIT     1
#define READ_SUSPEND          2
#define READ_STOP     3

#define USAGE\
        fprintf(stderr,\
 "captureiq [-f FIFO-size(bytes)] [-s sample-points] [-t sample-time(sec)] [output-file]\n")

#define GET_FIFO_DISTANCE()\
        (fifoBufSize * (readCycle - writeCycle) + readPosition - writePosition)

char    *outputFile = "raw_iq.dat";

int     fifoBytes = 1000000000;
double  samplePoints = 10e9;
double  sampleSeconds = 0.0;

int     *fifoBuffer;

int     readBufSize = 48000;
int     writeBufSize = 0;
int     fifoBufSize;

int     readCycle;
int     readPosition;
int     writeCycle;
int     writePosition;
int     maxDistance;

int     readStatus;
HANDLE          readThread;

int     errorCount;
double  spentTime;
double  readTime;

int     hpdi32InitializeDevice(void);
int     hpdi32ReadReset(void);
int     hpdi32ReadData(int *, int);

void    readData(void);
int     writeData(int, int);

void    printTime(char *, double, double);
double  getTime(void);

main(int argc, char *argv[])
{
        int     i, size, wn, id, fd, rtv;
```

```
char	*cp;
double	wpoints, tsum, wsum, floor();

while(argc > 2 && argv[1][0] == '-'){
	switch(argv[1][1]){
		case 'f' :
			if(sscanf(argv[2], "%d", &fifoBytes) != 1){
				USAGE;
				exit(0);
			}
			break;
		case 'o' :
			outputFile = argv[1];
			break;
		case 't' :
			if(sscanf(argv[2], "%lg", &sampleSeconds) != 1){
				USAGE;
				exit(0);
			}
			break;
		default :
			USAGE;
			exit(1);
	}
	argc -= 2; argv += 2;
}
if(argc > 1){
	if(sscanf(argv[1], "%lg", &samplePoints) != 1){
		USAGE;
		exit(0);
	}
}
readBufSize = readBufSize / 16 * 16;
if(fifoBufSize < MIN_FIFO_BYTES)
	fifoBufSize = MIN_FIFO_BYTES;
fifoBufSize = fifoBytes / (4 * readBufSize * 10) * readBufSize * 10;
if(writeBufSize <= 0)
	writeBufSize = fifoBufSize / 10;
maxDistance = fifoBufSize - readBufSize;
samplePoints = floor(samplePoints);
wpoints = samplePoints;
size = fifoBufSize * 4;
printf("FIFO buffer size: %d bytes\n", size);
if((fifoBuffer = (int *)malloc(size)) == NULL){
	fprintf(stderr, "Error: malloc %d bytes\n", size);
	exit(1);
}
if(sampleSeconds > 0.0){
	samplePoints = 1e12;
	printf("%g sec ", sampleSeconds);
}
else
	printf("%g samples", samplePoints);
printf(" -> '%s'\n", outputFile);
tsum = 0.0;
errorCount = 0;
readCycle = writeCycle = readPosition = writePosition = 0;
readStatus = READ_WAIT;
if((fd = open(outputFile, OPEN_MODE, 0xff)) < 0){
	perror(outputFile);
	exit(1);
```

```
	}

	if((readThread = CreateThread(NULL, 0,
	 (LPTHREAD_START_ROUTINE)readData, NULL, 0, &id)) == NULL){
		fprintf(stderr, "Error: CreateThread\n");
		exit(1);
	}
	if(SetThreadPriority(readThread, THREAD_PRIORITY_HIGHEST) == 0)
		fprintf(stderr, "Error: SetThreadPriority\n");
	while(wpoints >= 1.0){
		wn = wpoints > (double)REPORT_POINTS ? REPORT_POINTS : (int)wpoints;
		rtv = writeData(fd, wn);
		tsum += spentTime;
		if(rtv < 0)
			break;
		wpoints -= (double)wn;
		if(wpoints >= 1.0)
			printTime(" Write", wn, spentTime);
	}
	close(fd);
	wsum = (double)fifoBufSize * writeCycle + writePosition;
	printTime("Total", wsum, tsum);
	if(errorCount > 0){
		for(i = 0; i < 100 && readStatus != READ_STOP; i++)
			Sleep(10);
		printf("Error %d: %g sec/error, %gM sample/error\n",
		 errorCount, readTime / errorCount, wsum * 1e-6/ errorCount);
	}
	else
		printf("No error\n");
}

void
readData(void)
{
	int	bn, pn, rtv, *bufp;
	double	t0, t1;

	bn = 0;
	if(hpdi32InitializeDevice() < 0)
		exit(1);
	readTime = 0.0;
	while(readStatus == READ_WAIT);
	t0 = getTime();
	hpdi32ReadReset();
	while(samplePoints > 0){
		if(GET_FIFO_DISTANCE() > maxDistance){
			printf("FIFO full\n");
			while(GET_FIFO_DISTANCE() > maxDistance)
				Sleep(1);
		}
		pn = samplePoints > (double)readBufSize ? readBufSize : (int)samplePoints;
		bufp = fifoBuffer + readPosition;
		rtv = hpdi32ReadData(bufp, pn);
		if(rtv < 0){
			if(++errorCount >= 10)
				break;
			readTime += getTime() - t0;
			readStatus = READ_SUSPEND;
			SuspendThread(readThread);
			readStatus = 0;
```

```
                        t0 = getTime();
                        hpdi32ReadReset();
                }
                else{
                        readPosition += pn;
                        if(readPosition >= fifoBufSize){
                                readCycle++;
                                readPosition = 0;
                        }
                        samplePoints -= pn;
                }
                if(sampleSeconds > 0.0 && getTime() - t0 >= sampleSeconds)
                        break;
        }
        readTime += getTime() - t0;
        readStatus = READ_STOP;
        ExitThread((DWORD)0);
}

int
writeData(int fd, int points)
{
        int     n, rtv;
        double  t;


        t = getTime();
        if(readStatus == READ_WAIT)
                readStatus = 0;
        rtv = 0;
        while(points > 0){
                n = fifoBufSize - writePosition;
                if(n > writeBufSize)
                        n = writeBufSize;
                if(n > points)
                        n = points;
                if(readStatus == READ_SUSPEND
                 && GET_FIFO_DISTANCE() <= readBufSize)
                        ResumeThread(readThread);
                else{
                        while(GET_FIFO_DISTANCE() < n){
                                if(readStatus == READ_STOP){
                                        if(GET_FIFO_DISTANCE() < n){
                                                rtv = -1;
                                                goto loop_end;
                                        }
                                        break;
                                }
                                if(readStatus == READ_SUSPEND)
                                        ResumeThread(readThread);
                                Sleep(20);
                        }
                }
                if(write(fd, fifoBuffer + writePosition, n * 4) < 0){
                        perror("Data write");
                        return(-1);
                }
                writePosition += n;
                if(writePosition >= fifoBufSize){
                        writeCycle++;
                        writePosition = 0;
```

```
                }
                points -= n;
        }
loop_end:
        spentTime = getTime() - t;
        if(spentTime < 1e-3)
                spentTime = 1e-3;
        return(rtv);
}

void
printTime(char *msg, double points, double sec)
{
        points *= 1e-6;
        printf("%s %gM samples, %g sec: %gM sample/sec, FIFO: %d %%\n",
         msg,  points, sec, points / sec,
         (int)(GET_FIFO_DISTANCE() * 100.0 / fifoBufSize));
}

#include <sys/types.h>
#include <sys/timeb.h>

double
getTime(void)
{
        struct timeb       tbuf;

        ftime(&tbuf);
        return(tbuf.time + tbuf.millitm * 1e-3);
}

/* Functions for HPDI32A interface */

#include "hpdi32_api.h"

static void         *hpdiDevice;

static U32          setConfig(void);
static U32          setParameter(U32 arg1, U32 arg2, U32 arg3);

int
hpdi32InitializeDevice(void)
{
        int         index;
        U32         status;
        U32         arg, stat, ret;

        index = 0;
        status = hpdi32_api_status(&stat, &arg, HPDI32_API_VERSION);
        if (status != GSC_SUCCESS || stat != GSC_SUCCESS){
                fprintf(stderr, "API failed\n");
                return(-1);
        }
        if (hpdi32_open((U8)index, &hpdiDevice) != GSC_SUCCESS){
                fprintf(stderr, "Unable to access device %d.\n", index);
                return(-1);
        }
        if (setConfig() != GSC_SUCCESS){
                fprintf(stderr, "Configuration error\n");
                return(-1);
        }
```

```
	return(0);
}

int
hpdi32ReadReset(void)
{
	HPDI32_RX_ENABLE__YES(hpdiDevice);
	HPDI32_FIFO_RESET__RX_YES(hpdiDevice);
	HPDI32_RX_OVERRUN__CLEAR(hpdiDevice);
	return(0);
}

int
hpdi32ReadData(int *bufp, int points)
{
	U32	bytes, status, xfer, overrun, ret;

	bytes = points * 4;
	status = hpdi32_read(hpdiDevice, bufp, bytes, &xfer);
	if (status == GSC_SUCCESS && bytes == xfer){
		ret = HPDI32_RX_OVERRUN__GET(hpdiDevice, &overrun);
		if(ret == GSC_SUCCESS ){
			if(overrun != HPDI32_RX_OVERRUN_YES)
				return(points);
			fprintf(stderr, "Buffer overrun\n");
		}
		else
			fprintf(stderr, "Can not get overrun status\n");
	}
	else if (status == GSC_WAIT_TIMEOUT)
		fprintf(stderr, "Read time out\n");
	else
		fprintf(stderr, "Read error: status %d\n", status);
	return(-1);
}

static U32
setConfig(void)
{
	U32	status;

	//AlmostEmpty:16
	if((status = setParameter(HPDI32_FIFO_ALMOST_LEVEL,
	 HPDI32_WHICH_TX_RX | HPDI32_WHICH_AE, 16)) != GSC_SUCCESS)
		return(status);
	//AlmostFull:16
	if((status = setParameter(HPDI32_FIFO_ALMOST_LEVEL,
	 HPDI32_WHICH_TX_RX | HPDI32_WHICH_AF, 16)) != GSC_SUCCESS)
		return(status);
	//DataSize:32
	if((status = setParameter(HPDI32_IO_DATA_SIZE,
	 HPDI32_WHICH_TX_RX, HPDI32_IO_DATA_SIZE_32_BITS)) != GSC_SUCCESS)
		return(status);
	//FrameValid : "FlowControl"
	if((status = setParameter(HPDI32_CABLE_COMMAND_MODE,
	 HPDI32_WHICH_FRAME_VALID_,
	 HPDI32_CABLE_COMMAND_MODE_FLOW_CONTROL)) != GSC_SUCCESS)
		return(status);
	//LineValid : "FlowControl"
	if((status = setParameter(HPDI32_CABLE_COMMAND_MODE,
	 HPDI32_WHICH_LINE_VALID_,
```

```
 HPDI32_CABLE_COMMAND_MODE_FLOW_CONTROL)) != GSC_SUCCESS)
                return(status);
        //LineValidOffCount : 0x0
        if((status = setParameter(HPDI32_TX_LINE_VALID_OFF_COUNT, 0, 0))
         != GSC_SUCCESS)
                return(status);
        //LineValidOnCount : 0x0
        if((status = setParameter(HPDI32_TX_LINE_VALID_ON_COUNT, 0, 0))
         != GSC_SUCCESS)
                return(status);
        //Mode : DMDMA
        if((status = setParameter(HPDI32_IO_MODE,
         HPDI32_WHICH_TX_RX, HPDI32_IO_MODE_DMDMA)) != GSC_SUCCESS)
                return(status);
        //RxEnabled : In
        if((status = setParameter(HPDI32_CABLE_COMMAND_MODE,
         HPDI32_WHICH_RX_ENABLED_,
         HPDI32_CABLE_COMMAND_MODE_GPIO_IN)) != GSC_SUCCESS)
                return(status);
        //RxReady : In
        if((status = setParameter(HPDI32_CABLE_COMMAND_MODE,
         HPDI32_WHICH_RX_READY_,
         HPDI32_CABLE_COMMAND_MODE_GPIO_IN)) != GSC_SUCCESS)
                return(status);
        //StatusValid : "FlowControl"
        if((status = setParameter(HPDI32_CABLE_COMMAND_MODE,
         HPDI32_WHICH_STATUS_VALID_,
         HPDI32_CABLE_COMMAND_MODE_FLOW_CONTROL)) != GSC_SUCCESS)
                return(status);
        //StatusValidCount : 0x0
        if((status = setParameter(HPDI32_TX_STATUS_VALID_COUNT, 0, 0))
         != GSC_SUCCESS)
                return(status);
        //StatusValidMirror : "Yes"
        if((status = setParameter(HPDI32_TX_STATUS_VALID_MIRROR,
         HPDI32_WHICH_TX_RX,
         HPDI32_TX_STATUS_VALID_MIRROR_ENABLE)) != GSC_SUCCESS)
                return(status);
        //Timeout : 20sec
        status = setParameter(HPDI32_IO_TIMEOUT, HPDI32_WHICH_TX_RX, 20);
        return(status);
}

static U32
setParameter(U32 arg1, U32 arg2, U32 arg3)
{
        U32     dummy;

        return(hpdi32_parm_config(hpdiDevice, arg1, arg2, arg3, &dummy, NULL));
}
```

Appendix B: *makecaldata.c*

```
/* Copyright (C) Tektronix */

#include <stdio.h>
#include "rtsa_iqt.h"

#define USAGE fprintf(stderr,\
 "makecaldata [-a amp-flatness-file] [-p phase-flatness-file] [-c cal-param-file] IQT-file\n")

char    *ampFlatnessFile = "a_raw_flat.dat";      // Flatness Correction data : Amplitude
char    *phaseFlatnessFile = "p_raw_flat.dat";    // Flatness Correction data : Phase
char    *calParameterFile = "cal_para.txt";       // Calibration parameter

int     flatnessA[FLATNESS_POINTS];
int     flatnessP[FLATNESS_POINTS];

int     getFileHeader(FILE *, char *);
int     getValidFrames(char *, char *);
int     getCorrectionTable(FILE *, int, int *, int *);
int     writeFile(char *, void *, int, int);

main(int argc, char *argv[])
{
        int     n, vfn, offs;
        char    header[MAX_HEADER_SIZE];
        FILE    *fp;

        while(argc > 2 && argv[1][0] == '-'){
                switch(argv[1][1]){
                        case 'a' :
                                ampFlatnessFile = argv[2];
                                break;
                        case 'p' :
                                phaseFlatnessFile = argv[2];
                                break;
                        case 'c' :
                                calParameterFile = argv[2];
                                break;
                        default :
                                USAGE;
                                exit(1);
                }
                argc -= 2; argv += 2;
        }
        if(argc < 2){
                USAGE;
                exit(1);
        }
        if((fp = fopen(argv[1], "rb")) == NULL){
                perror(argv[1]);
                exit(1);
        }
        if((n = getFileHeader(fp, header)) < 0)
                exit(1);
        if((vfn = getValidFrames(header, "ValidFrames=")) < 0)
                exit(1);
        offs = (sizeof(struct frameHeader_st) + FRAME_SIZE) * vfn;
        if(getCorrectionTable(fp, offs, flatnessA, flatnessP) < 0)
                exit(1);
        fclose(fp);
```

```
	if(writeFile(calParameterFile, header, n, 0) < 0)
		exit(1);
	if(writeFile(ampFlatnessFile, flatnessA, sizeof(int) * FLATNESS_POINTS, 1) < 0)
		exit(1);
	if(writeFile(phaseFlatnessFile, flatnessP, sizeof(int) * FLATNESS_POINTS, 1) < 0)
		exit(1);
}

int
getFileHeader(FILE *fp, char *hdr)
{
	int	n;
	char	buf[10];

	if(fread(buf, 1, 1, fp) != 1 || (n = buf[0] - '0') < 1 || n > 9)
		goto err_header;
	if(fread(buf, 1, n, fp) != n)
		goto err_header;
	buf[n] = '\0';
	if(sscanf(buf, "%d", &n) != 1 || n > MAX_HEADER_SIZE)
		goto err_header;
	if(fread(hdr, n, 1, fp) == 1){
		hdr[n] = '\0';
		return(n);
	}
err_header:
	fprintf(stderr, "Illegal file header\n");
	return(-1);
}

int
getValidFrames(char *hdr, char *str)
{
	int	len, val;

	len = strlen(str);
	while(*hdr){
		if(strncmp(hdr, str, len) == 0){
			if(sscanf(hdr + len, "%d", &val) == 1)
				return(val);
		}
		while(*hdr >= ' ')
			hdr++;
		while(*hdr && *hdr <= ' ')
			hdr++;
	}
	fprintf(stderr, "Parameter '%s' was not found\n", str);
	return(-1);
}

int
getCorrectionTable(FILE *fp, int offs, int *afp, int *pfp)
{
	int	i;
	char	buf[10];
	struct frameHeader_st	fhdr;
	struct apBin_st	apbuf[FRAME_POINTS];
	struct extendedCorrectionData_st	exbuf;

	if(fseek(fp, offs, SEEK_CUR) < 0
	 || fread(&fhdr, sizeof(struct frameHeader_st), 1, fp) != 1
```

```
	 || fread(apbuf, FRAME_SIZE, 1, fp) != 1){
		for(i = 0; i < FRAME_POINTS; i++){
			afp[i] = 1 << 8;
			pfp[i] = 0;
		}
		return(0);
	}
	for(i = 0; i < FRAME_POINTS; i++){
		afp[i] = (apbuf[i].a << 8) & 0xffffff00;
		pfp[i] = (apbuf[i].p << 8) & 0xffffff00;
	}
	if(fread(buf, 5, 1, fp) != 1 || strncmp(buf, "40000", 5) != 0
	 || fread(&exbuf, sizeof(struct extendedCorrectionData_st), 1, fp) != 1)
		return(0);
	for(i = 0; i < FRAME_POINTS; i++){
		afp[i] |= exbuf.a[i] & 0xff;
		pfp[i] |= exbuf.p[i] & 0xff;
	}
	return(1);
}

int
writeFile(char *fname, void *buf, int size, int bin)
{
	FILE	*fp;

	if((fp = fopen(fname, bin ? "wb" : "w")) == NULL){
		perror(fname);
		return(-1);
	}
	if(fwrite((char *)buf, size, 1, fp) != 1){
		fprintf(stderr, "%s: Data write error\n", fname);
		fclose(fp);
		return(-1);
	}
	fclose(fp);
	return(0);
}
```

Appendix C: *makeiqt.c*

```
/* Copyright (C) Tektronix */

#include <stdio.h>
#include <sys/stat.h>
#include "rtsa_iqt.h"

#define MAX_PARAM_CNT       128

#define USAGE fprintf(stderr,\
 "makeiqt [-a amp-flatness-file] [-p phase-flatness-file] [-c cal-param-file] [-o otuput-file] [raw-IQT-file]\n")

char    *inputFile = "raw_iq.dat";                  // Input raw IQ data file
char    *ampFlatnessFile = "a_raw_flat.dat";        // Flatness Correction data : Amplitude
char    *phaseFlatnessFile = "p_raw_flat.dat";      // Flatness Correction data : Phase
char    *calParameterFile = "cal_para.txt";         // Calibration parameter
char    *outputFile = "captured.iqt";               // Output IQT file

short   flatnessA[FLATNESS_POINTS];
short   flatnessP[FLATNESS_POINTS];
struct extendedCorrectionData_st        flatnessExt;

char    fileHeader[MAX_HEADER_SIZE];

struct frameHeader_st   frameHeader = {
        0,      0,      0,      -1,     -1,
        1024,   0,      0,      0,      0,      0
};

int     getCalibrationParameters(char *, char **);
int     getCorrectionTable(char *, short *, unsigned char *);
int     searchParameter(char **, char *, int);
int     writeFileHeader(FILE *, char **, int, int);
int     writeIQData(FILE *, char *, int);
int     writeFlatnessData(FILE *, short *, short *, struct extendedCorrectionData_st *);

main(int argc, char *argv[])
{
        int     n, pn, fn, bpos;
        char    header[MAX_HEADER_SIZE], *params[MAX_PARAM_CNT];
        struct stat     stbuf;
        FILE    *fp;

        while(argc > 2 && argv[1][0] == '-'){
                switch(argv[1][1]){
                        case 'a' :
                                ampFlatnessFile = argv[2];
                                break;
                        case 'p' :
                                phaseFlatnessFile = argv[2];
                                break;
                        case 'c' :
                                calParameterFile = argv[2];
                                break;
                        case 'o' :
                                outputFile = argv[2];
                                break;
                        default :
                                USAGE;
                                exit(1);
```

```
                }
                argc -= 2; argv += 2;
        }
        if(argc > 1)
                inputFile = argv[1];
        if((pn = getCalibrationParameters(calParameterFile, params)) < 0)
                exit(1);
        if(getCorrectionTable(ampFlatnessFile, flatnessA, flatnessExt.a) < 0)
                exit(1);
        if(getCorrectionTable(phaseFlatnessFile, flatnessP, flatnessExt.p) < 0)
                exit(1);
        if(stat(inputFile, &stbuf) < 0){
                perror(inputFile);
                exit(1);
        }
        fn = stbuf.st_size / (sizeof(short) * 2 * FRAME_POINTS);
        if((fp = fopen(outputFile, "wb")) == NULL){
                perror(outputFile);
                exit(1);
        }
        printf("%s: %d frames\n", outputFile, fn);
        n = fn * sizeof(short) * 2 * FRAME_POINTS;
        if(n != stbuf.st_size){
                printf("Last %d points are truncated\n",
                 (stbuf.st_size - n) / (sizeof(short) * 2));
        }
        if((bpos = searchParameter(params, "Bins=", pn)) >= 0)
                sscanf(params[bpos] + 5, "%d", &(frameHeader.bins));
        if(writeFileHeader(fp, params, pn, fn) < 0)
                exit(1);
        if(writeIQData(fp, inputFile, fn) < 0)
                exit(1);
        if(writeFlatnessData(fp, flatnessA, flatnessP, &flatnessExt) < 0)
                exit(1);
        fclose(fp);
}

int
getCalibrationParameters(char *fname, char **pp)
{
        int     i, c;
        char    *hp;
        FILE    *fp;

        if((fp = fopen(fname, "rb")) == NULL){
                perror(fname);
                return(-1);
        }
        for(i = 0; i < MAX_HEADER_SIZE && (c = getc(fp)) != EOF; i++)
                fileHeader[i] = c;
        if(c != EOF){
                fprintf(stderr, "%s: Too large file\n", fname);
                return(-1);
        }
        hp = fileHeader;
        for(i = 0; i < MAX_PARAM_CNT && *hp; i++){
                pp[i] = hp;
                while(*hp >= ' ')
                        hp++;
                if(*hp)
                        *hp++ = '\0';
```

```
                while(*hp && *hp <= ' ')
                        hp++;
        }
        fclose(fp);
        if(i >= MAX_PARAM_CNT){
                fprintf(stderr, "%s: Too many header parameters\n", fname);
                return(-1);
        }
        return(i);
}

int
getCorrectionTable(char *fname, short *fltp, unsigned char *exfp)
{
        int     i, rtv, fbuf[FLATNESS_POINTS];
        FILE    *fp;

        if((fp = fopen(fname, "rb")) == NULL){
                perror(fname);
                return(-1);
        }
        rtv = fread(fbuf, sizeof(int), FLATNESS_POINTS, fp);
        fclose(fp);
        if(rtv  != FLATNESS_POINTS){
                fprintf(stderr, "%s: Illegal flatness correction data file\n", fname);
                return(-1);
        }
        for(i = 0; i < FLATNESS_POINTS; i++){
                fltp[i] = (fbuf[i] >> 8) & 0xffff;
                exfp[i] = fbuf[i] & 0xff;
        }
        return(0);
}

int
searchParameter(char **pp, char *str, int n)
{
        int     i, len;

        len = strlen(str);
        for(i = 0; i < n; i++){
                if(strncmp(pp[i], str, len) == 0)
                        return(i);
        }
        return(-1);
}

int
writeFileHeader(FILE *fp, char **pp, int pn, int frmn)
{
        int     i, vfpos, bytes;
        char    vfbuf[128], buf[16], *vfstr;

        vfstr = "ValidFrames=";
        if((vfpos = searchParameter(pp, vfstr, pn)) < 0){
                fprintf(stderr, "Parameter '%s' not found\n", vfstr);
                return(-1);
        }
        sprintf(vfbuf, "%s%d", vfstr, frmn);
        pp[vfpos] = vfbuf;
        for(i = bytes = 0; i < pn; i++)
```

```
                bytes += strlen(pp[i]) + 2;
        sprintf(buf, "%04d", bytes);
        fprintf(fp, "%d%s", strlen(buf), buf);
        for(i = 0; i < pn; i++)
                fprintf(fp, "%s\r\n", pp[i]);
        return(0);
}

int
writeIQData(FILE *wfp, char *fname, int frmn)
{
        int     i, rtv;
        struct frameHeader_st   frmhdr;
        struct iqBin_st   iqbuf[FRAME_POINTS];
        FILE    *rfp;

        if((rfp = fopen(fname, "rb")) == NULL){
                perror(fname);
                return(-1);
        }
        rtv = 0;
        for(i = 0; i < frmn; i++){
                if(i == frmn - 1)
                        frameHeader.lastFrame = -1;
                if(fread(iqbuf, sizeof(struct iqBin_st), FRAME_POINTS, rfp) != FRAME_POINTS){
                        fprintf(stderr, "%s: Data read error\n", fname);
                        rtv = -1;
                        break;
                }
                if(fwrite(&frameHeader, sizeof(struct frameHeader_st), 1, wfp) != 1
                 || fwrite(iqbuf, sizeof(struct iqBin_st), FRAME_POINTS, wfp) != FRAME_POINTS){
                        fprintf(stderr, "%s: Data write error\n", outputFile);
                        rtv = -1;
                        break;
                }
                frameHeader.ticks++;
        }
        fclose(rfp);
        return(rtv);
}

int
writeFlatnessData(FILE *fp, short *ap, short *pp, struct extendedCorrectionData_st *extp)
{
        int     i;
        struct apBin_st  apbuf[FLATNESS_POINTS];

        for(i = 0; i < FLATNESS_POINTS; i++){
                apbuf[i].a = *ap++;
                apbuf[i].p = *pp++;
        }
        frameHeader.validA = frameHeader.validP = -1;
        frameHeader.validI = frameHeader.validQ = 0;
        if(fwrite(&frameHeader, sizeof(struct frameHeader_st), 1, fp) != 1
         || fwrite(apbuf, sizeof(struct apBin_st), FLATNESS_POINTS, fp) != FLATNESS_POINTS
         || fwrite("40000", 1, 5, fp) != 5
         || fwrite(extp, sizeof(struct extendedCorrectionData_st), 1, fp) != 1){
                fprintf(stderr, "%s: Data write error\n", outputFile);
                return(-1);
        }
        return(0);
```

```
}
```

Appendix D: *correctiq.c*

```
/* Copyright (C) Tektronix */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define FLATNESS_SIZE          1024
#define PI          3.141592653589793

#define USAGE fprintf(stderr,\
        "correctiq [-a amp-flatness-file] [-p phase-flatness-file] [-c cal-param-file] [-o output-file] [-s IQ-separator] [input-file]\n")

char    *ampFlatnessFile = "a_raw_flat.dat";      // Flatness Correction data : Amplitude
char    *phaseFlatnessFile = "p_raw_flat.dat";    // Flatness Correction data : Phase
char    *calParameterFile = "cal_para.txt";       // Calibration parameter
char    *inputFile = "raw_iq.dat";                // Raw IQ data to be corrected
char    *outputFile = "corrected_iq.txt";  // Corrected ASCII data

char    *iqSeparator = ",";

double  iOffset;
double  qOffset;
double  gainOffset;
double  maxInputLevel;
double  levelOffset;

double  flatnessI[FLATNESS_SIZE];
double  flatnessQ[FLATNESS_SIZE];

int     readFlatnessData(char *, char *, double *, double *);
int     readCalParameters(char *);
int     getCalParameterValue(char *, char *, double * );

int     flatnessOverlapProcess(char *, char *, double *, double *);
void    flatnessSetTable(double *, double *, double *, double *);
void    flatnessCompensation(double *, double *, double *, double *, int);

int     readIQData(FILE *, double *, double *, int);
int     writeAsciiData(FILE *, double *, double *, int );

void    fft(int, double *, double *);
void    makeSineTable(int, double *);
void    makeBitReverse(int, int *);

main(int argc, char *argv[])
{
        int     n;

        while(argc > 2 && argv[1][0] == '-'){
                switch(argv[1][1]){
                        case 'a' :
                                ampFlatnessFile = argv[2];
                                break;
                        case 'p' :
                                phaseFlatnessFile = argv[2];
                                break;
```

```
                        case 'c' :
                                calParameterFile = argv[2];
                                break;
                        case 'o' :
                                outputFile = argv[2];
                                break;
                        case 's' :
                                iqSeparator = argv[2];
                                break;
                        default :
                                USAGE;
                                exit(1);
                }
                argc -= 2; argv += 2;
        }
        if(argc > 1)
                inputFile = argv[1];
        if(readCalParameters(calParameterFile) < 0)
                exit(1);
        if(readFlatnessData(ampFlatnessFile, phaseFlatnessFile, flatnessI, flatnessQ) < 0)
                exit(1);
        printf("%s -> %s\n", inputFile, outputFile);
        if((n = flatnessOverlapProcess(inputFile, outputFile, flatnessI, flatnessQ)) < 0)
                exit(1);
        printf(" %d points\n", n);
}

int
readCalParameters(char *fname)
{
        if(getCalParameterValue(fname, "MaxInputLevel=", &maxInputLevel) < 0)
                return(-1);
        if(getCalParameterValue(fname, "LevelOffset=", &levelOffset) < 0)
                return(-1);
        if(getCalParameterValue(fname, "GainOffset=", &gainOffset) < 0)
                return(-1);
        if(getCalParameterValue(fname, "IOffset=", &iOffset) < 0)
                return(-1);
        if(getCalParameterValue(fname, "QOffset=", &qOffset) < 0)
                return(-1);
        return(0);
}

int
getCalParameterValue(char *fname, char * str, double* val)
{
        int     len;
        char    buf[128];
        FILE    *fp;

        if((fp = fopen(fname, "r")) == NULL){
                perror(fname);
                return(-1);
        }
        len = strlen(str);
        while(fgets(buf, 127, fp) != NULL){
                if(strncmp(buf, str, len) == 0){
                        if(sscanf(buf + len, "%lg", val) == 1){
                                fclose(fp);
                                return(0);
                        }
```

```
                }
        }
        fprintf(stderr, "%s: Parameter '%s' was not found\n", fname, str);
        fclose(fp);
        return(-1);
}

int
readFlatnessData(char *afname, char *pfname, double *ip, double *qp)
{
        int     i, val;
        double  amp, phase;
        FILE    *afp, *pfp;

        if((afp = fopen(afname, "rb")) == NULL){
                perror(afname);
                return(-1);
        }

        if((pfp = fopen(pfname, "rb")) == NULL){
                perror(pfname);
                return(-1);
        }

        for(i = 0; i < FLATNESS_SIZE; i++){
                if(fread(&val, sizeof(int), 1, afp) != 1){
                        fclose(afp);
                        fprintf(stderr, "%s: Illegal flatness data\n", afname);
                        return(-1);
                }
                // Calculation for converting dBm to Volatge
                amp = 1.0 / sqrt(pow(10.0, (val / (128.0 * 256.0)) / 10.0));

                if(fread(&val, sizeof(int), 1, pfp) != 1){
                        fclose(pfp);
                        fprintf(stderr, "%s: Illegal flatness data\n", pfname);
                        return(-1);
                }
                // Convert degree to radian about phase component
                phase = val /32768.0 * PI / 180.0;
                // calculate IQ value of Flatness Correction data
                ip[i] = amp * cos(phase);
                qp[i] = amp * sin(phase);
        }
        fclose(afp);
        fclose(pfp);
        return(FLATNESS_SIZE);
}

/* Flatness */

#define TABLE_SIZE    (FLATNESS_SIZE * 2)

int
flatnessOverlapProcess(char *rfname, char *wfname, double *fip, double *fqp)
{
        int     i, s_2, s_4, s3_4, n, wn, points;
        double  bufi[TABLE_SIZE], bufq[TABLE_SIZE];
        double  flati[TABLE_SIZE], flatq[TABLE_SIZE];
        FILE    *rfp, *wfp;
```

```
	if((rfp = fopen(rfname, "rb")) == NULL){
		perror(rfname);
		return(-1);
	}
	if((wfp = fopen(wfname, "w")) == NULL){
		perror(wfname);
		fclose(rfp);
		return(-1);
	}
	flatnessSetTable(fip, fqp, flati, flatq);
	s_2 = TABLE_SIZE / 2;
	s_4 = TABLE_SIZE / 4;
	s3_4 = s_4 * 3;
	for(i = 0; i < s_4; i++)
		bufi[i] = bufq[i] = 0.0;
	points = 0;
	while((n = readIQData(rfp, bufi + s_4, bufq + s_4, s3_4)) > 0){
		for(i = s_4 + n; i < TABLE_SIZE; i++)
			bufi[i] = bufq[i] = 0.0;
		fft(TABLE_SIZE, bufi, bufq);
		flatnessCompensation(bufi, bufq, flati, flatq, TABLE_SIZE);
		fft(-TABLE_SIZE, bufi, bufq);
		if(n < s_2)
			wn = n;
		else
			wn = s_2;
		writeAsciiData(wfp, bufi + s_4, bufq + s_4, wn);
		points += wn;
		if(n <= s_2)
			break;
		else{
			fseek(rfp, -s_2 * sizeof(short) * 2, SEEK_CUR);
			readIQData(rfp, bufi, bufq, s_4);
		}
	}
	fclose(rfp);
	fclose(wfp);
	if(n < 0)
		return(-1);
	return(points);
}

void
flatnessSetTable(double *ip, double *qp, double *fip, double *fqp)
{
	int	i, pos, fs_2;
	double	mul;

	fs_2 = FLATNESS_SIZE / 2;
	mul = (double)TABLE_SIZE / FLATNESS_SIZE;
	for(i = 0; i < FLATNESS_SIZE; i++){
		fip[i] = *ip++ * mul;
		fqp[i] = *qp++ * mul;
	}
	fft(-FLATNESS_SIZE, fip, fqp);
	pos = TABLE_SIZE - fs_2;
	memcpy(fip + pos, fip + fs_2, sizeof(double) * fs_2);
	memcpy(fqp + pos, fqp + fs_2, sizeof(double) * fs_2);
	for(i = fs_2; i < pos; i++)
		fip[i] = fqp[i] = 0.0;
	fft(TABLE_SIZE, fip, fqp);
```

```
}

void
flatnessCompensation(double *ip, double *qp, double *fip, double *fqp, int n)
{
	int	i;
	double	ix, qx;

	for(i = 0; i < n; i++){
		ix = ip[i]; qx = qp[i];
		ip[i] = *fip * ix - *fqp * qx;
		qp[i] = *fip * qx + *fqp * ix;
		fip++; fqp++;
	}
}

int
readIQData(FILE *fp, double *ip, double *qp, int n)
{
	int	i;
	short	iqbuf[2];
	double	scale;

	scale = sqrt(pow(10.0, (gainOffset + maxInputLevel + levelOffset)
	 /10) / 20 * 2);
	for(i = 0; i < n; i++){
		if(fread(iqbuf, sizeof(short), 2, fp) != 2)
			break;
		ip[i] = (iqbuf[1] - iOffset) * scale;
		qp[i] = (iqbuf[0] - qOffset) * scale;
	}
	return(i);
}

int
writeAsciiData(FILE *fp, double *ip, double *qp, int n)
{
	int	i;

	for(i = 0; i < n; i++)
		fprintf(fp, "%e%s%e\n", ip[i], iqSeparator, qp[i]);
	return(n);
}

/* FFT */

#define MAX_FFT_SIZE		TABLE_SIZE

void
fft(int n, double *x, double *y)
{
	static double	sintbl[MAX_FFT_SIZE + MAX_FFT_SIZE / 4];
	static int	bitrev[MAX_FFT_SIZE], last_n = 0;
	double	t, s, c, dx, dy;
	int		i, j, k, ik, h, d, k2, n_4, inverse;

	if(n < 0){			// IFFT
		n = -n;
		inverse = 1;
	}
	else
```

```
		inverse = 0;
	n_4 = n / 4;
	if(n != last_n){
		last_n = n;
		makeSineTable(n, sintbl);
		makeBitReverse(n, bitrev);
	}
	for(i = 0; i < n; i++){
		j = bitrev[i];
		if (i < j) {
			t = x[i]; x[i] = x[j]; x[j] = t;
			t = y[i]; y[i] = y[j]; y[j] = t;
		}
	}
	for(k = 1; k < n; k = k2){
		h = 0;
		k2 = k + k;
		d = n / k2;

		for(j = 0; j < k; j++){
			c = sintbl[h + n_4];
			s = inverse ? -sintbl[h] : sintbl[h];
			for(i = j; i < n; i += k2){
				ik = i + k;
				dx = s * y[ik] + c * x[ik];
				dy = c * y[ik] - s * x[ik];
				x[ik] = x[i] - dx;
				x[i] += dx;
				y[ik] = y[i] - dy;
				y[i] += dy;
			}
			h += d;
		}
	}
	if(!inverse){
		for(i = 0; i < n; i++){
			x[i] /= n;
			y[i] /= n;
		}
	}
}

void
makeSineTable(int n, double *sintbl)
{
	double  c, s, dc, ds, t;
	int	i, n_2, n_4, n_8;

	n_2 = n / 2;
	n_4 = n / 4;
	n_8 = n / 8;
	t = sin(PI / n);
	dc = 2 * t * t;
	ds = sqrt(dc * (2 - dc));
	t = 2 * dc;
	c = sintbl[n_4] = 1;
	s = sintbl[0] = 0;

	for(i = 1; i < n_8; i++){
		c -= dc;
		dc += t * c;
```

```
                s += ds;
                ds -= t * s;
                sintbl[i] = s;
                sintbl[n_4 - i] = c;
        }
        if(n_8 != 0)
                sintbl[n_8] = sqrt(0.5);

        for(i = 0; i < n_4; i++)
                sintbl[n_2 - i] = sintbl[i];

        for(i = 0; i < n_2 + n_4; i++)
                sintbl[i + n_2] = -sintbl[i];
}


void
makeBitReverse(int n, int *bitrev)
{
        int     i, j, k, n_2;

        n_2 = n / 2;
        for(i = j = 0; ; ){
                bitrev[i] = j;
                if(++i >= n)
                        break;
                k = n_2;
                while(k <= j){
                        j -= k;
                        k /= 2;
                }
                j += k;
        }
}
```

Appendix E: *rtsa_iq.h*

```
/* Copyright (C) Tektronix */

#define FRAME_POINTS          1024
#define FLATNESS_POINTS     1024
#define FRAME_SIZE    (sizeof(struct iqBin_st) * FRAME_POINTS)
#define MAX_HEADER_SIZE    4096

struct frameHeader_st {
        short    dataShift;
        short    validA;
        short    validP;
        short    validI;
        short    validQ;
        short    bins;
        short    frameError;
        short    triggered;
        short    overLoad;
        short    lastFrame;
        long     ticks;
};

struct apBin_st {
        short    a;
        short    p;
};

struct iqBin_st {
        short    q;
        short    i;
};

struct extendedCorrectionData_st {
   unsigned char a[1024];
   unsigned char p[1024];
};
```

TEKTRONIX, ITS RESELLERS OR OTHERS FROM WHOM TEKTRONIX MAY HAVE OBTAINED A LICENSING RIGHT DO NOT WARRANT THE PROGRAM OR OPERATION THEREOF, DO NOT ASSUME ANY LIABILITY WITH RESPECT TO ITS USE, AND DO NOT UNDERTAKE TO FURNISH ANY SUPPORT OR INFORMATION RELATING THERETO.

Note:
This white paper represents the sample IQ data capturing system using sample software. The sample software is made for the capturing system represented in this white paper and customers and user may use these samples software with no charge and with no support.