The Generic Recording Format (GRF)

An Open, Device and Platform Independent, Packet Oriented Digital Waveform Data Recording Format

Revision 1.2.0

May 2003

Robert Banfill

Banfill Software Engineering

Valdez Alaska
(907) 835-4122

http://www.banfill.net/

Table of Contents

Table of Figures

Chapter 1

Introduction

1.1 Overview

This document describes and defines the Generic Recording Format (GRF). The GRF is a completely open, device and platform independent way of transporting and storing digital waveform data created by field digitizer hardware. GRF includes various command, control, and connection management features in a device independent manner allowing different GRF compatible hardware from various manufacturers to be integrated into a unified system.

Although GRF is geared toward network oriented digital seismic data acquisition systems, it is by no means limited to that particular application. GRF can easily be extended to accommodate almost any conceivable data acquisition application.

GRF is based on small (less than two kilobytes) self-identifying packets that can be moved through network connections and/or stored in binary files on a filesystem. GRF packet images are written in a serialized fashion so that the media sees only a simple stream of bytes. This allows GRF data to move freely between platforms without the usual byte order and/or alignment issues.

1.2 Design Goals

We set out to design a relatively simple and open recording format that overcomes the shortcomings of the various existing recording formats that were available to us.

The primary design criteria specify that GRF must be:

- **Network oriented** – GRF is based on small self-identifying packets. Each packet is made up of a common header that encapsulates a particular type of GRF packet. This encapsulation contains a signature and GRF version number, a unit identifier that identifies the creator of this packet, a sequence number, the type of packet encapsulated, and its length.

  These packets are moved through standard TCP socket connections and/or stored in a filesystem in binary packet image files. We do not intend GRF to be a full communications protocol. As such, no error correction mechanism is provided by GRF itself and non-error-correcting transports such as UDP should not be used for bulk data transfer.

- **Platform independent** – All binary data within a packet are stored in *network byte order*. This is also called *big-endian* byte order, as the most significant byte is stored at the higher numbered memory address. This is the byte order employed by the TCP/IP protocol family. All 8, 16, 32, and 64-bit, signed and unsigned, integral types are stored as two's complement integers. All 32 and 64-bit real types are stored as IEEE 754 floating-point numbers. Any padding fields added for alignment on any particular platform are removed before the packet image is written to the media. An ANSI C library is provided which illustrates how to handle all of these device dependencies as GRF packets are written to (serialized) or read from (deserialized) the media. This approach allows data written on one platform to be read on another without regard to these issues.
- **Device independent** – Because GRF strives to be independent of any particular hardware platform or digitizer, it is fully generic in the way in which digital waveform data, and its associated ancillary data, is stored. We attempt to follow current best practices for time keeping, naming of channels, and so forth, so that any modern system can employ GRF.
- **Self-contained** – Digitizers create digital waveforms that represent a voltage over time. By self-contained we mean that a single GRF data packet that contains a waveform segment must contain all of the information needed to view that waveform segment as a voltage over time. One does not need to look elsewhere for such information such as the initial sample time, the sampling rate, the name of the channel, and so on.
- **Extensible** – Because we know that we cannot possibly anticipate every possible application and requirement that will present itself in future, GRF is easily extended to include new packet types. Currently only 10 of the 256 possible packet types are used.

  Existing packet types can also be extended to include new fields without jeopardizing older applications ability to also read those packets even though they will not be able to understand or use these new additional fields.

1.3 The GRF Home Page

Banfill Software Engineering maintains the GRF home page at http://www.banfill.net/grf.html. Go to this page to:

- Get the latest news and information about the GRF.
- Get the latest GRF Tools Suite and other software and/or services.
- Make suggestions, submit software, and/or otherwise participate in the development of the GRF.
- Get GRF unit identifier number assignments.

Contacting Us

Our offices are located in Valdez Alaska. We can be reached at:

Banfill Software Engineering

P.O. Box 462
Valdez, AK 99686-0462 USA
(907) 835-4122 Voice
http://www.banfill.net
mailto:info@banfill.net

Chapter 2

GRF Encapsulation

2.1 Overview

Every GRF packet is encapsulated by a 13 byte common header. This encapsulation serves several purposes:

- It identifies this packet as a GRF packet and specifies the major version of GRF to which it conforms.
- It provides the unit identifier number for the creator of this packet and gives the packet sequence number of this packet.
- It provides the packet type and overall length of the packet.

2.2 The Common Header

The following figure shows the layout of the common header fields.

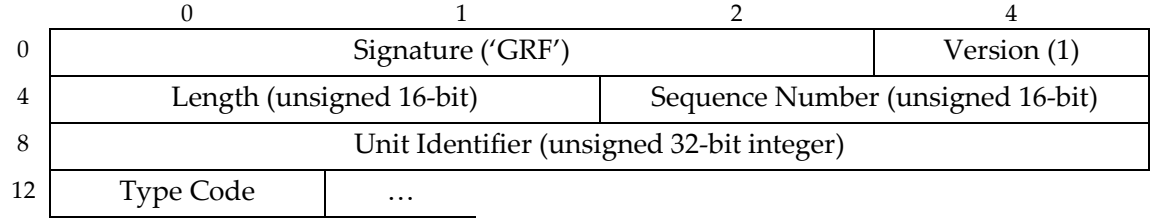

Figure 1 GRF 13-byte common header.

Below we will cover each field within the common header in more detail.

The Signature Field

The common header starts at offset 0 with the GRF signature, which consists of three consecutive bytes containing the ASCII characters: 'G' (0x47), 'R' (0x52), and 'F' (0x46). The main purpose of this field is to identify this as a GRF packet. This signature is used when reading packets to check for byte stream synchronization.

The Version Field

The signature is followed by the unsigned 8-bit integer GRF version number at offset 3. This is the version number of the GRF to which this packet conforms. Currently there is a single version of GRF and that is version 1. In the future, this field will be used as the GRF is

extended to accommodate unforeseen applications and requirements in a backward compatible manner.

The Length Field

The version field is followed by the unsigned 16-bit integer packet length at offset 4. This is the overall length of the packet including the 13-byte common header. GRF packets are limited in overall length to 2048 (2k) bytes.

The Sequence Field

The sequence field contains the unsigned 16-bit integer sequence number for this packet at offset 6. As GRF packets are created, they are sequentially numbered using this field. When this number reaches its maximum value (65535 decimal), it simply wraps back to zero and starts again. A reader of these packets can check sequence numbers as packets are read to determine if packets are missing. Note that sequence numbers are meaningful only within the scope of a particular GRF unit identifier.

The Unit Identifier Field

The unit identifier field contains an unsigned 32-bit integer that uniquely identifies the creator of this packet at offset eight. For field digitizers, this number is typically derived from the serial number of the hardware. In other cases, this number is simply assigned.

It is of utmost importance that this number be unique for every source of GRF packets. We maintain a list of all GRF assigned numbers (http://www.banfill.net/doc/grf_assigned_numbers.txt) and coordinate this information between all GRF users. Please contact us if you would like to have a range of values assigned for your use.

The Type Field

At offset 12, the type field contains the unsigned 8-bit integer packet type identifier. This field can be thought of as either the last field of the common header or the first byte of the actual packet. However, because all GRF packets must have a type, the minimum length is the 13 bytes of the common header including the type field.

Figure 2 shows the currently defined GRF packet type identifiers

Type	Name	Description
0x00	No Packet	Undefined, null, or void packet type.
0x01	Data	Waveform data packet.
0x02	ConnectReq	Client connection request.
0x03	ConnectAck	Client connection positive acknowledgement.
0x04	ConnectNak	Client connection negative acknowledgement.
0x05	Command	Command packet.
0x06	CommandAck	Command positive acknowledgement.
0x07	CommandNak	Command negative acknowledgement.
0x08	Information	General informational message packet.
0x09	Disconnect	Client disconnect request.
0x0A – 0xFF	Reserved	These type codes are reserved for future use.

Figure 2 GRF packet types.

The most common GRF packet by far is the data (0x01) packet because it carries the waveform sample data. The next most common packet is the information (0x08) packet. This packet contains general informational messages along with waveform data. Typically, information packets contain such information as GPS positions and clock lock/unlock messages.

The other packet types are used to manage connections between GRF servers and clients and perform command and control operations.

2.3 The GRF port number

The Internet Assigned Numbers Authority (IANA) has assigned both the TCP and the UDP registered port number 3757 for use by GRF server applications. All GRF servers should listen on this port for client connections by default. The server listen endpoint interface and port number should also be user configurable.

Chapter 3

GRF Packets

3.1 Overview

As stated in the previous section, all GRF packets begin with the common header and the last field in that header defines the packet type that it encapsulates. In this section, we define these packet types and their fields.

3.2 Data Packet

The data packet, type code 0x01, is at the heart of the GRF as it is the workhorse packet used to transport waveform segments. GRF data packets are completely self-contained, that is, along with the waveform sample data; the various other data needed to identify and view the waveform segment are also contained within the packet. This information is contained in the various fields within the packet structure, which are then followed by the waveform sample data.

GRF Time Values

Before delving into the details of the GRF data packet, let's quickly cover how time information is stored in GRF.

All absolute time values within GRF packets are represented as a signed 64-bit integer number of microseconds since January 1, 1970, 00:00:00.000000 UTC. This is an efficient way to store high precision time values and because there is only a single integer value, date/time arithmetic is trivial and can be performed without loss of precision.

For a particular time, say the initial sample time for the waveform segment within a GRF data packet, three values are stored: the absolute time, a correction to that time, and a quality code indicating the quality of the time after the correction is applied. There are five possible time qualities:

Code	Quality	Description
0	Unknown	The quality of the time is completely unknown.
1	Bad	The time is known to be invalid.
2	Poor	The time quality has been good but the timing system is currently free running. This typically indicates that a GPS is currently unlocked.

3	Good	The timing system is actively correcting time relative to its reference. This typically means that a GPS is locked and that time is actively being corrected to the best of the system's ability.
4	Very Good	The time is known to be within +/- 500 microseconds of UTC.
5-255	–	Reserved codes

Figure 3 GRF time qualities.

A simple ANSI C library, USTime, is provided to facilitate dealing with the GRF time values. This library provides various functions to encode, decode, format, and parse these microsecond time values within your applications.

Data Packet Structure

Like all GRF packets, each data packet begins with the common header (shown shaded in the figure below) and can be up to 2048 bytes (2 kb) overall in length. The following figure shows the various fields that makeup the GRF data packet.

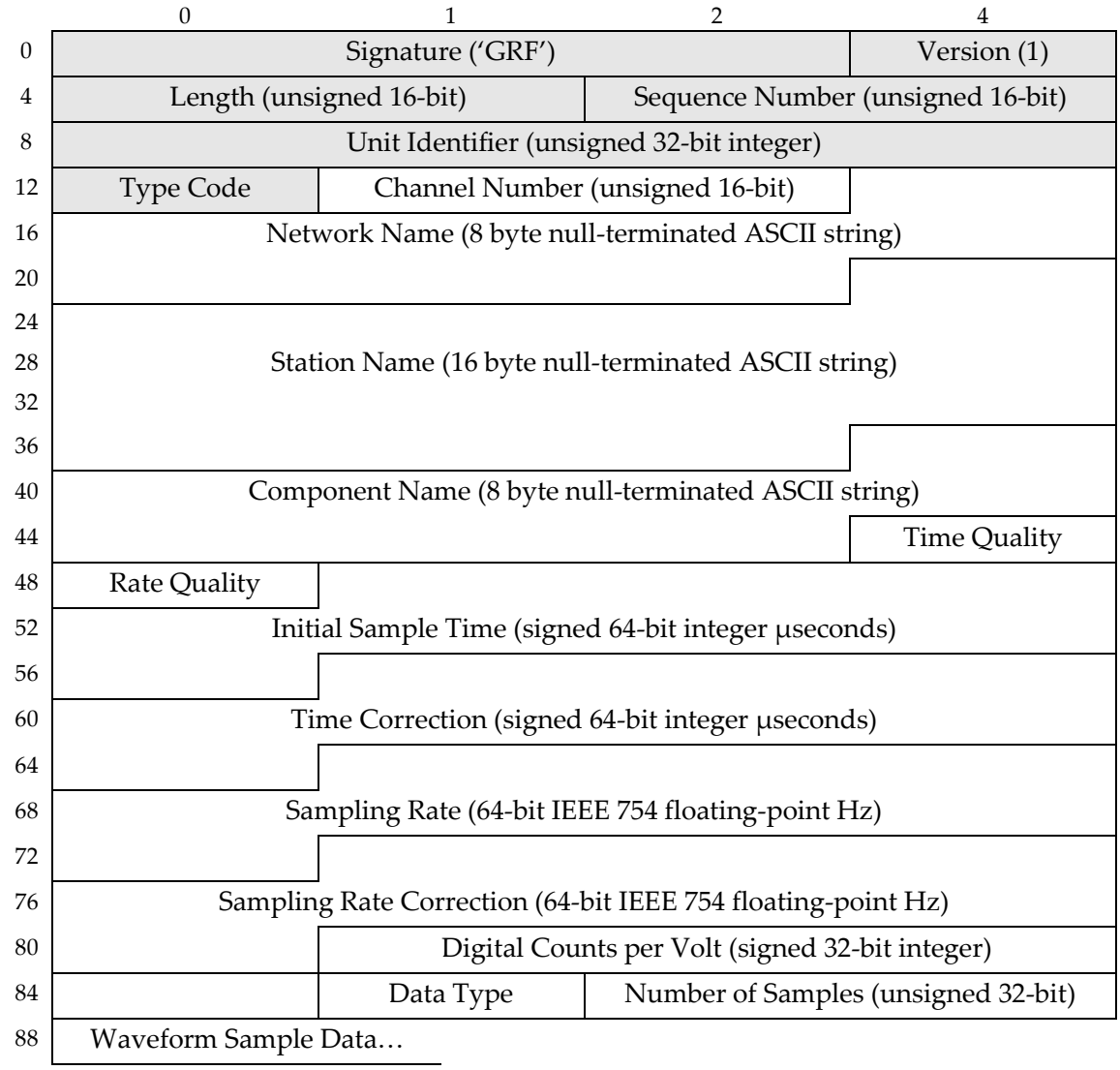

Figure 4 GRF data packet structure.

The fields within the data packet can be divided into three basic categories. The first category includes fields used to identify the source of the data. The second includes timing information. The third includes information needed to view the waveform segment itself.

Identification Fields

There are two different ways that one can identify the source of a GRF waveform segment. The most general way is to use the Unit Identifier field from the common header combined with the Channel Number field above. This unit:channel pair uniquely identifies the source of this waveform segment and is generally how software deals with waveform data internally.

The second way that a waveform segment might be identified is to give it a name. This is how we humans generally deal with waveform data. This name is made up of three parts, the network name, the station name, and the component identifier. The lengths of these fields allow use of the naming conventions in use in most common processing systems available today.

The Channel Number Field

This is the number of the channel on the digitizer that recorded this waveform segment. GRF channel numbers are stored as unsigned 16-bit integers at offset 13 and are one based, that is to say, channel numbers start from one, not zero and range up to 65535. Channel number zero (0) is reserved and should not be used.

The Network Name Field

This field begins at offset 15 and contains an eight-byte null terminated ASCII string. As the string is null terminated, the length of the string is limited to seven characters. Note that most software will treat this field as case-sensitive.

The name in this field should uniquely identify the network of which this particular station is a part. This is the highest-level name. In most applications, this field should contain a two upper-case character network code that has been assigned by the FDSN[1] archive (IRIS DMC[2]).

The Station Name Field

This field begins at offset 23 and contains a sixteen-byte null terminated ASCII string. As the string is null terminated, the length of the string is limited to fifteen characters. Note that most software will treat this field as case-sensitive.

The name in this field should uniquely identify the station of the network at which this waveform was recorded. This is the mid-level name. Typically, these names are administered by the network operator and only need to be unique within the context of the particular network.

The Component Name Field

This field begins at offset 39 and contains an eight-byte null terminated ASCII string. As the string is null terminated, the length of the string is limited to seven characters. Note that most software will treat this field as case-sensitive.

---

[1] Federation of Digital Seismographic Networks (FDSN)

[2] Incorporated Research Institutions for Seismology (IRIS) Data Management Center (DMC)

The name in this field should uniquely identify the component, or channel, at this station. This is the lowest-level name. In most applications, these names should follow the conventions set forth by IRIS for use with the SEED[3] format as described in detail in Appendix A of the SEED Reference Manual.

Timing Fields

This group of six fields contains two basic pieces of information: the initial sample time and the sampling rate. As stated above, the time consists of the absolute time, a correction to that time, and a quality indicator. Likewise, the sampling rate consists of the rate, a correction to that rate, and a quality indicator.

The Time Quality Field

This is an unsigned 8-bit integer at offset 47 containing a value between 0 and 4 inclusive. The meanings of these values are shown in figure 3. This field indicates the time quality after the time correction is added to the initial sample time.

The Rate Quality Field

This is an unsigned 8-bit integer at offset 48 containing a value between 0 and 4 inclusive. The values used in this field are the same as those used in the time quality field but with somewhat different meanings. Generally, the sampling rate is known to at least some nominal accuracy and so the rate correction field is set to zero and the quality is set to 'Poor' (2).

On some systems, the sampling rate is observed over relatively long periods of time and corrections are computed. Once a correction is computed and stored in the rate correction field, the quality may be promoted to 'Good' (3) or even 'Very Good' (4) as appropriate.

The Initial Sample Time Field

This field contains the uncorrected signed 64-bit integer microsecond time of the first sample of the waveform segment at offset 49. Add the value contained in the time correction field to get the corrected initial sample time.

All time values stored in the GRF are stored as the number of microseconds that have elapsed since UTC zero hours on 1 January 1970. See section 3.1 for more about GRF times, corrections, and qualities.

The Time Correction Field

This field begins at offset 57 and contains the signed 64-bit integer microseconds to add to the value contained in the initial sample time field to correct it to the quality contained in the time quality field.

$$\text{initial sample time} + \text{time correction} = \text{initial sample time}^{\text{time quality}}$$

The Sampling Rate Field

This field begins at offset 65 and contains the uncorrected 64-bit IEEE 754 floating-point sampling rate as samples per second (Hz). Add the value contained in the sampling rate correction field below to get the corrected sampling rate.

[3] Standard for the Exchange of Earthquake Data (SEED)

The Sampling Rate Correction Field

This field begins at offset 73 and contains the 64-bit IEEE 754 floating-point sampling rate correction to add to the sampling rate field to correct it to the quality indicated by the rate quality field.

$$\text{sampling rate} + \text{sampling rate correction} = \text{sampling rate}^{\text{rate quality}}$$

Waveform Information Fields

These fields contain information about the waveform segment itself.

The Counts per Volt Field

This field contains the signed 32-bit integer number of digital counts per volt at offset 81. This is sometimes called the digitizer, or A/D, constant. This value is needed in order to view waveform samples in volts.

The Data Type Field

This field contains the unsigned 8-bit integer data type code at offset 85. This data type code specifies the method used to encode the waveform sample data contained in this packet. Currently, there are three different techniques used to store waveform data.

Code	Name	Description
0x00	INT32	Each sample is stored as a signed 32-bit integer occupying four bytes of storage. This is the least efficient data type in terms of storage.
0x01	INT24	Each sample is stored as a signed 24-bit integer occupying three bytes of storage. This is a useful format as most digitizers generate samples 24 or less significant bits. This format is sometimes called 'byte-packed'.
0x02	CM8	This is a standard data encoding technique used by the GSE2.0[4] format. Compression ratios as high as 3.5:1 can be achieved with quiet data, but on average, ratios run from 2.0:1 to 2.5:1. This is the most efficient data type in terms of storage and is the most commonly used.
0x03-0xFF	–	Reserved codes.

Figure 5 GRF data type codes.

The Number of Samples Field

This field begins at offset 86 and contains the unsigned 16-bit integer number of samples that are contained within this packet.

[4] The Conference on Disarmament (CD) established the Ad Hoc Group of Scientific Experts (GSE) in July 1976 with a mandate to consider international cooperative measures to detect and identify seismic events to facilitate the monitoring of a Comprehensive Test Ban Treaty. Since then, the GSE has developed several generations of data exchange systems. For more information go to http://www.pidc.org/

Data Encoding

The waveform sample data begins at byte offset 88 from the beginning of the packet. The samples are encoded as indicated by the data type and the number of samples (*N*) fields in the packet.

INT32 Encoding

When the data type field indicates the INT32 data type, the waveform samples are simply stored as an array of signed 32-bit integers containing *N* elements. The size of this array is simply *N* × 4 bytes. The length field in the common header will set to 88 + (*N* × 4) bytes on full packets.

INT24 Encoding

When the data type field indicates the INT24 data type, each sample is stored as a signed 24-bit integer that occupies three bytes. *N* values are stored as a packed array of *N* × 3 bytes in size. On most platforms, these 24-bit values will need to be decoded into signed 32-bit integers using bit-wise operators. The length field in the common header will be set to 88 + (*N* × 3) bytes on full packets.

CM8 Encoding

When the data type field indicates the CM8 data type, the samples are compressed using the CM8 compression algorithm and a CCITT 16-bit CRC is appended. The CM8 algorithm is defined in GSE Conference Room Paper 243 (GSE/CRP/243), Annex 3 - Formats and Protocols for Data Exchange, Section 4.5 - GSE2.0 Waveform Segments - Sub-format CM8. This document is available at http://www.pidc.org/web-gsett3/CRP-243. Here we present a brief description of the CM8 algorithm.

We start with an array, *x*, containing waveform samples as signed 32-bit, two's complement, integers in the range of -2^{31} to $+2^{31}-1$ inclusive. The second difference, *d*, that is, the difference between the first differences is computed:

$$d_i = x_i - 2x_{i-1} + x_{i-2}$$

where zero and negative indices are ignored. Thus, the second difference data in *d* for *N* samples are:

$$x_1, x_2 - 2x_1, x_3 - 2x_2 + x_1, \ldots, x_N - 2x_{N-1} + x_{N-2}$$

To compress these second difference values, each element of the array *d* are converted from two's complement to sign and magnitude. These values are then encoded into a variable number of bytes. The most significant bit of each byte is a flag, or control bit, which, if set, signifies that the following byte contains bits of the current sample. The second most significant bit is used as the sign bit on the first byte pertaining to the current sample. The remaining bits are used to store the value of the second difference.

MSB LSB

control	sign/ data bit	data bit	data bit	data bit	data bit	data bit	data bit

Figure 6 CM8 encoded byte layout.

As many bytes as are needed are used to encode the second difference value. If the magnitude of the second difference can be contained in six bits or less (<64), a single byte stores the entire difference with the control bit cleared. If the magnitude is greater than six bits (≥64) but less than thirteen bits (<8192), then the second difference is stored using two

bytes with the first byte containing the sign and the six most significant bits and a set control bit, and the second with the seven least significant bits and a cleared control bit. As many as five bytes can be required to encode a second difference when its magnitude is greater than 27 bits, however this very rarely happens with real data and most often only one or two bytes are required to encode each second difference.

Once the second differences are encoded, a 16-bit CCITT CRC is computed over the encoded bytes and is appended as the last two bytes of data (see below). The length field in the common header will equal 88 + *n* + 2, where 88 is the length of the common and data packet headers, *n* is the number of bytes used to store the CM8 encoded samples, and 2 is the length of the 16-bit CRC value appended to the end of the CM8 data.

Decoding the CM8 data is very straightforward. The CRC is computed over the encoded data including the two CRC bytes and the result must equal zero to pass the CRC check; otherwise, the encoded data are corrupt. For each difference, the sign is stored and the magnitude is built up as dictated by the control bit on each byte. Each second difference is stored into an array and double integrated to restore the original sample values.

CCITT 16-bit CRC Algorithm

The cyclic redundancy code (CRC) used within the GRF to perform error checking on encoded waveform data is a standard CCITT[5] CRC algorithm. Below we provide a simple and very fast table-driven C language implementation of this algorithm.

In the GRF, we always use an initial value of zero and the result is not exclusive or'd (XOR) with any value. The advantage of this is that if the two CRC bytes are simply appended to the data, the check can be performed by computing the CRC over the data and the two stored CRC bytes. If the result is zero, the data are error free.

crc16.c

```
typedef unsigned char UINT8;
typedef unsigned short UNIT16;

/* CRC table parameters:
 *   Width      = 16 bits
 *   Polynomial = 0x1021 (x^16+x^12+x^5+1, standard CCITT)
 *   Reflection = no
 */
static const UINT16 crc_table[256] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
    0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
    0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
    0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
    0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
    0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
    0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
    0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
    0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
    0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
    0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
    0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
    0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
    0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
    0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
```

[5] Commite' Consultatif International de Telecommunications et Telegraphy (CCITT) – A committee of the International Telecommunications Union (ITU) responsible for making technical recommendations about telephone and data communication systems.

```
    0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
    0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
    0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
    0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
    0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
    0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
    0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
    0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0
};

/*--------------------------------------------------------------------*/
UINT16 ComputeCRC16(VOID *bytes, size_t length)
{
    UINT16 crc;

    crc = 0;
    while (length--) {
        crc = (crc << 8) ^ crc_table[(crc >> 8) ^ *bytes++];
    }

    return crc;
}
```

crc16.c

Figure 7 C language implementation of the CRC-16 algorithm.

Note the `typedef` statements above defining the types: `UINT16`, and `UINT8`. These refer to unsigned 16-bit and 8-bit integers respectively and may need to be adjusted for your particular platform.

3.3 Connection Packets

In this section, we define four connection management packets: ConnectReq (0x02), ConnectAck (0x03), ConnectNak (0x04), and Disconnect (0x09). These packets are used by client and server applications to establish or terminate a connection through a TCP socket.

Packet Structure

These four packet types share a common packet structure:

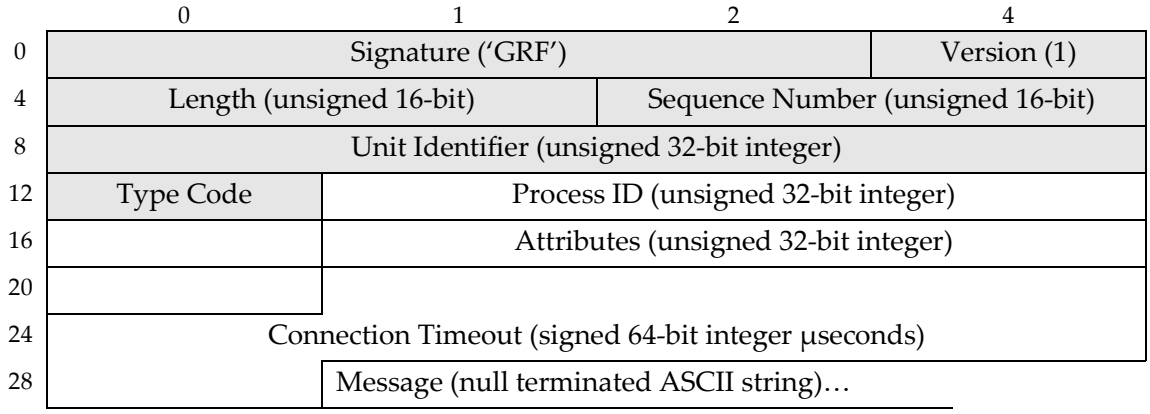

Figure 8 GRF connection packet structure.

Like all GRF packets, the connection packets begin with the common header (shown shaded in the figure above) that is defined in section 2. The exact meaning of each field within the packet is determined by the type code field at offset 12.

ConnectReq Packets

This packet is sent from a client application to a server application requesting that a connection by established.

The Process ID Field

This field contains the unsigned 32-bit integer process identifier at offset 13. This value must be unique for the client host system, as the server will use this value along with the address of the client to identify this particular connection.

The Attributes Field

This field contains an unsigned 32-bit integer that is used as a bit field. The bits within this field are used to request certain connection attributes:

Bit	Hex value	Name	Description
0	0x00000001	GRF_ATTRIB_DATA	Waveform data access
1	0x00000002	GRF_ATTRIB_CMD	Command and control access
2–31	–	–	Reserved

Figure 9 GRF connection attributes.

Note that attributes can be combined, for example, to request both data and command access, you would specify the attribute as 0x00000003 (both bits 0 and 1 set).

The Connection Timeout Field

This field contains the signed 64-bit integer microsecond time value that the server should set as the connection timeout for this connection. See section 3.1 for an explanation how time values are stored in the GRF.

The Message Field

This field contains a null-terminated ASCII string. This string is always empty in ConnectReq packets. The empty string is specified by a null (0) byte at offset 29.

ConnectAck and ConnectNak Packets

One of these packets is sent by a server application in response to a received ConnectReq packet. The ConnectReq packet is simply copied and sent back to the client with the type code field at offset 12 set appropriately and the message field filled with an appropriate message.

If the type code returned is ConnectAck, the server is positively acknowledging the client connection and will immediately begin sending waveform data if the data bit was set in the attribute field. The message field should contain the 'friendly name' of the server.

If the type code returned is ConnectNak, the server is negatively acknowledging the connection and the message field will contain an error message indicating why this connection is not being allowed. Typical reasons that a client would receive a ConnectNak in response to a connection request include:

- The client lacks permissions necessary for the requested access.
- The server has too many active connections.
- The client is being denied access to the server host.

These are only a few of the many possible reasons that a particular client might fail to connect to a particular server. The message field in the ConnectNak should contain a clear and complete error message describing the particular reason that the connection failed.

Disconnect Packet

The Disconnect packet is used by a client application to notify a connected server that it is about to close the socket connection. This notification allows the server to perform an orderly and permanent shutdown of the connection. If a connection is lost without the server receiving a Disconnect packet, the server should be prepared to reestablish the lost connection for some period without loss of data.

The client application prepares a Disconnect packet by setting the Type Code field to 0x09 and setting the Process ID field to the same value used in the ConnectReq that established the connection. The other fields are ignored by the server however it is good practice to set the Attributes and Connection Timeout fields to 0 and set the first byte (offset 29) of the message field to null (0) indicating an empty string.

3.4 Command Packets

In this section, we define the command packet and the positive and negative command response packets. These packets are used to perform command and control operations on GRF enabled systems in a device independent manner. These systems may be field digitizer units, communications infrastructure, and/or data processing systems.

Packet Structure

The Command (0x05) packet and its responses, CommandAck (0x06), and CommandNak (0x07) share a common structure:

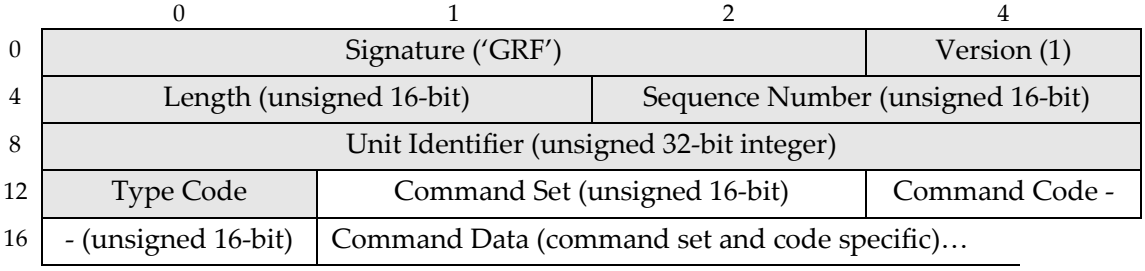

Figure 10 GRF command packet structure.

Like all GRF packets, command packets begin with the common header (shown shaded in the figure above). The Type Code field identifies whether this is a command (0x05), and positive command response (0x06), or a negative command response (0x07).

Command Sets and Codes

In order to accommodate the many different devices and systems that currently exist, as well as those the will exist in the future, the command packet contains at 'Command Set' field and a 'Command Code' field. These fields allow for 65536 sets of 65535 commands each.

In practice, the command set will be specific to a particular unit, for example, a particular type of field digitizer unit. The command codes for this set are all specific to this set and the Command Data will be specific to this command code for this command set.

When you plan to define command codes for your system, please contact us to have a command set identifier assigned for your particular system. Command set identifiers, as all GRF assigned numbers, must be coordinated between all users of the GRF. Please visit the GRF home page or contact us directly for current information about GRF assigned numbers.

Command and Response Sequences

Commands are targeted to a GRF unit identifier and that unit will respond with a command acknowledgement. This will be a CommandAck if it is a positive response or a CommandNak if it is a negative response.

The sequence number field in the common header should be maintained separately for command packets

A separate sequence number should be maintained for commands. Command responses should simply echo the command packet up to the command data field at offset 17 and only change the packet type code to either CommandAck or CommandNak as appropriate. This allows the sender of the command to match up the response by comparing the unit ID, sequence number, command set, and command code fields.

The command data in the command packet may or may not be used as is appropriate for the particular GRF unit. For immediate action commands, this extended data is generally not required and is simply not present. For complex commands, configuration commands for example, the command data might contain ASCII strings or even binary data as appropriate.

3.5 Information Packet

In this section, we define the GRF information packet. This packet is used to transport general informational messages throughout a system. The Unit ID field in the common header identifies the source of the message.

Packet structure

The information packet has the simplest structure of ant GRF packet type. It simply contains a null-terminated ASCII string that is the informational message.

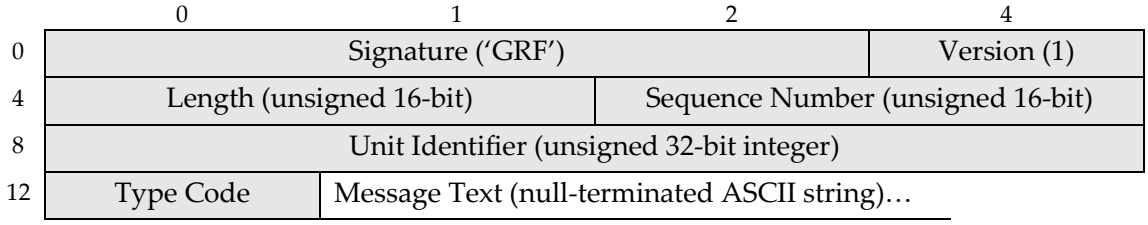

Figure 11 GRF information packet structure.

The type code for the Information packet is 0x08.

The Message Text Field

This field contains a null-terminated ASCII string starting at offset 13 that is the informational message. An empty string is represented by a null (0) byte at offset 13.

When appropriate, a message should begin with an ISO9660 compatible timestamp followed by the message text:

Timestamps using ordinal time representations should be of the form:

```
yyyy-dddThh:mm:ss.ssssssZ: message text
```

Timestamps using calendar time representations should be of the form:

```
yyyy-MM-DDThh:mm:ss.ssssssZ: message text
```

Where `yyyy` is the four digit year, `ddd` is the three digit ordinal day number (day-of-year), `MM` is the two digit month, `DD` is the two digit calendar day (day-of-month), the '`T`' character is the time field designator, `hh` is the two digit hour of day, `mm` is the two digit minute of hour, `ss.ssssss` is the two digit second of minute with up to six digits to the right of the decimal point showing fractional seconds, and the trailing '`Z`' character indicates that times are UTC.

Index

T

U

V

Banfill Software Engineering
http://www.banfill.net/
(907) 835-4122