libgcp Programmer's Guide

Jonathan Lamothe

May 21, 2012

Contents

1 Introduction

GCP (the Generic Communications Protocol) is intended to send messages of an arbitrary number of octets over a network in a simple, open manner. While it can be used on may different types of networks, it was specifically designed for serial networks, such as RS232, RS485, or other networks designed to send a stream of octets. The protocol provides its own error detection, making such a service unnecessary at a lower layer.

This protocol does not implicitly provide addressing, message acknowledgement, or streaming services, although, such services can be implemented on top of it. The protocol also places no restrictions on the content or format of its messages' payloads.

libgcp does not actually *send* or *receive* data. It merely assembles and processes frames. It is up to the programmer to send and receive the frames to and from what ever data stream they're using for transport.

Note: This document is intended to give an overview of how to use libgcp. For a more detailed description on the various functions and structures, please refer to the reference manual.

2 GCP Structures and Functions

2.1 The `GCPConn` Structure

Before any data can be processed, a `GCPConn` object needs to be created and initialized. The object is initialized by passing a pointer to it to the `gcp_init()` function, such as in the following example:

```
#include <gcp.h>

int main()
{
    GCPConn conn;
    gcp_init(&conn);

    /* do stuff here */

    return 0;
}
```

2.2 Allocating Buffers

After the connection has been initialized, send and receive buffers need to be allocated. These buffers contain the payload data for an outgoing or incoming message. If a connection is going to be one way (i.e: send or receive only) only one buffer needs to be allocated. **It is up to the programmer to free these buffers when they are no longer required.**

A buffer is simply an array of type `uint8_t`. These buffers are then pointed to by the `recv_buf` and `send_buf` fields of the `GCPConn` object. Also, the size of the receive buffer needs to be set in the `recv_size` field. The `send_size` value is set to the size of the message to be sent, rather than the size of the entire buffer. This is typically done just before sending. An example follows:

```
#include <gcp.h>

#define SIZE 1024

GCPConn conn;
uint8_t send[SIZE], recv[SIZE];

int main()
{
    gcp_init(&conn);
    conn.send_buf = send;
    conn.recv_buf = recv;
    conn.recv_size = SIZE;

    /* do stuff here */

    return 0;
}
```

2.3 Sending Messages

libgcp does not actually send data; it merely assembles frames so they can be sent over a stream of some sort. First, the message payload must be placed in the send buffer. Then, the size needs to be set. Finally, the `gcp_send_byte()` function needs to be called until the `GCPConn`'s `send_lock` flag returns to a value of 0. Every time this function is called, it will return the next octet to be sent to the outgoing stream.

Once the `send_lock` flag returns to 0, the buffer can be modified to contain the payload of the next message. **Do not change the send buffer's contents, or the value of `send_size`, while the `send_lock` flag's value is 1 unless you are *absolutely certain* you know what you're doing.**

The following is an example of a program that sends a series of messages:

```
#include <stdio.h>
#include <string.h>
#include <gcp.h>

#define SIZE 1024

GCPConn conn;
uint8_t send[SIZE];
```

```
void send_byte(uint8_t byte)
{
    /* send byte to the stream */
}

void send_message()
{
    do
        gcp_send_byte(&conn);
    while(conn.send_lock);
}

int main()
{
    int i = 0, byte;
    gcp_init(&conn);
    conn.send_buf = send;
    for(byte = getchar(); byte != EOF; byte = getchar())
    {
        if(byte == '\n')
        {
            conn.size = (i > SIZE) ? SIZE : i;
            i = 0;
            send_message();
        }
        else
        {
            if(i < SIZE)
                send[i] = byte;
            i++;
        }
    }
    return 0;
}
```

2.4 Receiving Messages

libgcp does not actually receive data, it merely parses octets passed to it from a stream and extracts the message payload whenever a valid message is found. In order to process a message, simply call the `gcp_recv_byte()` function with a pointer to the `GCPConn` object, and the next octet read from the stream, until the `GCPConn`'s `recv_lock` flag changes to a value of 0.

Once the `recv_lock` flag changes to 0, it means that a message has been successfully read. The message payload will be stored in the buffer pointed to by the `GCPConn`'s `recv_buf` field. This message is not terminated with an ASCII

0 value, however the length will be stored in the `GCPConn`'s `data_size` field. **Do not read from the receive buffer while the `recv_lock` flag's value is 1 unless you are *absolutely certain* you know what you're doing.**

It is important to note that if the message's payload is larger than the size of the receive buffer, it will be truncated to fit. This can be checked by comparing `data_size` to `recv_size`. If the former is larger than the latter, the message has been truncated. For this reason, it is important to select an appropriate size for the receive buffer.

The following is an example of a program that reads GCP messages:

```
#include <stdio.h>
#include <gcp.h>

#define SIZE 1024

GCPConn conn;
uint8_t recv[SIZE];

int get_byte()
{
    /* return an octet from the stream */
}

void process_message()
{
    int i;
    unsigned msg_size = (conn.data_size > conn.recv_size)
                        ? conn.recv_size : conn.data_size;
    for(i = 0; i < msg_size; i++)
        putchar((char)recv[i]);
    putchar('\n');
}

int main()
{
    int byte;
    gcp_init(&conn)
    conn.recv_buf = recv;
    conn.recv_size = SIZE;
    for(byte = get_byte(); byte != EOF; byte = get_byte())
    {
        gcp_recv_byte(&conn, byte);
        if(!conn.recv_lock)
            process_message();
    }
    return 0;
```

```
}
```

3 CRC-16 Structures and Functions

The `gcp_send_byte()` and `gcp_recv_byte()` functions automatically perform CRC-16 calculations, however the CRC-16 functions can be accessed directly as well.

3.1 The `CRC16Params` Structure

Since there are several ways of implementing a CRC-16 checksum, the `CRC16Params` structure was created to define the manner in which the calculation is performed. It contains the following fields:

`prefix` This is an unsigned 16-bit integer (`uint16_t`) which is appended to the beginning of the data before performing the checksum. GCP uses a value of 0. Setting it to a non-zero value is useful for making sure that leading zeroes affect the checksum calculation.

`poly` The 16-bit polynomial (`uint16_t`) to be used in the CRC-16 calculation. The polynomial used by GCP is `0x8005` ($x^{16} + x^{15} + x^2 + 1$).

`flip_bits` A single bit flag which, when set, causes the most significant bit of each octet to be processed first. This bit is set to 0 in a GCP checksum.

`flip_bytes` A single bit flag which, when set, causes the last octet in the buffer to be processed first. This bit should be set to 0 for a GCP checksum.

`flip_output` A single bit flag which, when set, reverses the bits of the checksum after calculation. This bit should be set to 1 for a GCP checksum.

See the reference manual for more details.

3.2 The `crc16_gen()` Function

The `crc16_gen()` function is used to generate a CRC16 checksum. It takes three parameters:

`data` A pointer to the buffer containing the data to be processed.

`size` The size of the buffer (number of octets).

`params` A pointer to the `CRC16Params` which describes how the checksum is to be performed (see section 3.1).

This function will return the calculated CRC-16 checksum. See the reference manual for more details.

3.3 The `crc16_check()` Function

The `crc16_check()` function checks the validity of a CRC-16 checksum. The first three parameters for this function are the same as for the `crc16_gen()` function (see section 3.2), the fourth parameter is the checksum being checked. If the checksums match, it will return 0, otherwise it will return a non-zero value. See the reference manual for more details.