Faucet Networking

About handles

Analogous to Game Maker's built in data structure functions, this extension uses handles to allow games to refer to sockets, buffers and other objects.
Passing an invalid handle to a function will generate reasonable behaviour. For example, socket functions will pretend that the handle belongs to a socket in error state, and reading the error string will return an appropriate message.

About integers and rounding

For some function parameters, only integer values make sense. For example, you can't read 2.5 bytes, or connect to port 91.6. In cases where integer values are expected, they are rounded toward zero (truncated) before use, except where otherwise noted. The most important exception are the write_xxx functions for appending values to a buffer or socket, those always round toward the nearest integer.

Connecting and disconnecting

tcp_connect(server, port) : tcpSocket

server is a string containing an IPv4 or IPv6 address or a hostname.
port is an integer in the range of 1 to 65535.
The returned socket can instantly be used in sending and receiving operations and will buffer sent data internally until the connection is actually established.
If the connection attempt fails, the socket will eventually report an error (see socket_has_error).

socket_connecting(socket) : bool

Returns true if the socket is currently performing a connection attempt. Returns false once the connection is established or if the connection failed.

socket_has_error(socket | acceptor) : bool

When this function returns true the socket or acceptor is no longer able to perform its function.
For a TCP socket, this means that it can no longer send or receive data over the network. Trying to send additional data is useless (but harmless), and trying to receive anything will fail as if there was simply no data available.
For an acceptor it means that no new connections will be accepted.
The function socket_error() can be used to get a description of the error.

tcp_listen(port) : acceptor

port is an integer in the range of 1 to 65535.
Create a new acceptor to listen for incoming TCP connections on the indicated port.

Both IPv4 and IPv6 connections to that port will be accepted. If an error occurs, the returned acceptor will indicate the failure. Please note that acceptors will only flag an error if both IPv4 and IPv6 connections can't be accepted anymore.

socket_accept(acceptor) : socket | errorcode

Accept a connection from an acceptor. If a connection is available, a (non-negative) TCP socket handle is returned. If no connection is available, a negative value is returned.

socket_destroy_abortive(socket | acceptor) : void

socket_destroy(socket | acceptor) : void

Closes the connection and destroys the socket. All sockets and acceptors have to be destroyed, even if they are already closed or in an error state, to release the associated resources. The handle will become invalid immediately and shouldn't be used again afterwards.

socket_destroy_abortive will close the connection immediately without trying to send any remaining data. *socket_destroy* will attempt to send all data from the socket's send buffer before closing (graceful close). In general, use a graceful close if you are closing the connection because there is nothing left to send or receive, and use an abortive close if you want to cut off a connection. If the socket is already in error state, then the connection is already closed, so there is no difference between the two functions.

If the remote end of the connection is still sending data when you close the local socket, his socket will probably receive an error so that he will not be able to read all data that you sent, even when you use a graceful close.

For acceptors, it doesn't matter which of the functions you use.

Sending and receiving information

To understand how sending and receiving work, you first need to know that a socket has two internal buffers: The send buffer and the receive buffer. To send data, you need to add data to the socket's send buffer and then call socket_send(socket). When you receive data, it will be written to the receive buffer.

To access these buffers, you can use some of the normal buffer functions described in the Buffers section and pass them a socket handle instead of a buffer handle. In general, buffer functions that read data will refer to the receive buffer, and those that write data will refer to the send buffer. However, not all buffer functions make sense for the internal buffers of sockets, so check the descriptions of the individual functions to find out more.

If you need more flexibility, you can use write_buffer to copy existing normal buffers to the send buffer, or to copy the receive buffer into a normal buffer.

tcp_receive(tcpSocket, size) : bool

Try reading *size* bytes into the socket's receive buffer. The previous contents of the receive buffer will be discarded. If the requested number of bytes is not available, a receive operation is started in the background to read at least as many bytes as requested, so that calling this function again later can succeed.

Returns true if the operation was successful. The receive buffer now contains exactly the

number of bytes requested. If false is returned, the receive buffer will be empty.
If *size* is negative or far too large (>=2^{32}), the function will return false without starting a background read.
If the connection is closed and more bytes are requested than are remaining to be read, the requested number of bytes can never become available. In this case the socket will transition to an error state.

tcp_receive_available(tcpSocket) : size

Read all data currently available from the socket into the socket's receive buffer. The previous contents of the receive buffer will be discarded. The function returns the number of bytes read, which is the same as the new size of the receive buffer.

tcp_eof(socket) : bool

Determine if there is no more data to receive. This means that the connection is closed in the receiving direction, either because the sender has closed it or because of an error, and all internally buffered data has been received (though there might still be unread data in the receive buffer). In that case, attempting to receive any more data on this socket will cause the socket to report an error.

socket_send(socket) : void

Try to send data from the internal sendbuffer out through the socket. If you call this function too often, it might result in many small TCP packets being sent, which causes a lot of overhead and thus needs more bandwidth. If you call it too seldom, the data will sit in the sendbuffer for a longer time before being sent, so you get larger delays. A good time to call this is right after you're done writing things to the sendbuffer of this socket for this step.

socket_sendbuffer_size(socket) : size

How many bytes are in the sendbuffer?
If the connection can't send data as fast as the application demands, the sendbuffer will grow. This function can be used to detect slow connections and reduce the data rate. However, the result should be taken with a large grain of salt. The number returned does not include data already queued for sending in the network layer, and this can be a lot of data (in the order of megabytes). If you send data in many small chunks though, there should be less of that queued data and you should see a growing backlog in the sendbuffer pretty quickly.

socket_receivebuffer_size(socket) : size

Returns the ammount of data in the receive buffer, analogous to buffer_size for normal buffers. The reason why this is a seperate function is to avoid confusion, because buffer_size(socket) doesn't make clear whether the send or receive buffer is meant, and the function would make sense for both.

socket_sendbuffer_limit(socket, size) : void

Prevent the sendbuffer from growing larger than this value (in bytes). 0 means no limit. If the sendbuffer would exceed this capacity the connection will be closed and

an appropriate error will be indicated. There is no default limit since a reasonable value depends heavily on the application. To prevent “out of memory” problems it is recommended to set this limit on every socket.

udp_send(buffer, host, port) : void

Warning: This function has been included for preliminary UDP support, to make it possible to port Gang Garrison 2 to this library. It might be changed or removed when a proper UDP api is designed.
host is a string containing an IPv4 or IPv6 address or a hostname.
port is an integer in the range of 1 to 65535
Make sure that the buffer is smaller than the maximum UDP packet size (roughly 64kb).

Buffers

Buffers typically contain data that has been received or should be sent.
Handing an invalid buffer handle to a buffer function will cause the function to behave as if it was called on a constantly empty buffer. Some of these functions will act on a socket’s send or receive buffer if a socket handle is passed to them. The parameter names socketReceiveBuf and socketSendBuf indicate which of the two buffers the parameter refers to.

buffer_create() : buffer

Create a new, empty buffer.

buffer_destroy(buffer) : void

Destroy a buffer and release its handle. This function should be called on all buffers that are no longer needed to release the memory allocated to them.

buffer_clear(buffer) : void

Remove all content from the buffer, so that it behaves exactly like a buffer that has only just been created.

buffer_size(buffer) : size

Returns the number of bytes in the buffer.

buffer_bytes_left(buffer) : size

Returns the number of bytes that can still be read from the buffer.

buffer_set_readpos(buffer, pos) : void

Set the read pointer of the buffer to a new position (given in bytes). The given position will be clipped to the beginning or the end of the buffer if it is below 0 or above buffer_size(buffer). There is no corresponding buffer_get_readpos(), if you really need this you can calculate it as buffer_size(buffer)-buffer_bytes_left(buffer).

write_[xxx](buffer | socketSendBuf, real) : void

Append the given value to the end of the buffer, or to the socket’s send buffer. The following functions are supported:
write_ubyte 8 bit unsigned integer

write_byte	8 bit signed integer
write_ushort	16 bit unsigned integer
write_short	16 bit signed integer
write_uint	32 bit unsigned integer
write_int	32 bit signed integer
write_float	32 bit floating point value
write_double	64 bit floating point value

The value of the provided real will be clipped to fit the range of the target type. For example, attempting to write anything above 127 as a (signed) byte will always write 127.

When converting the real to an integer type, the value is rounded to the nearest integer, with the halfway point being rounded away from 0.

To give some examples of this: write_short(buffer, 42.5) and write_short(buffer, 43.499) will both write 43, and write_short(buffer, -2.5) will write -3.

write_string(buffer | socketSendBuf, string) : void

Append a string in Game Maker's 8 bit character encoding to the buffer, or to the socket's send buffer. This will only write the raw characters, no length information or delimiter is included, so you will want to do that yourself.

write_buffer(target, source) : void

Append entire *source* buffer to the end of *target*. Both *source* and *target* may be either sockets or buffers. If *target* is a socket, it refers to the socket's send buffer. If *source* is a socket, it refers to the socket's receive buffer. This operation doesn't affect the buffers' read positions.

read_[xxx](buffer | socketReceiveBuf) : real

Read a value of the expected type from the buffer, starting at its current read position. The read position will be advanced to the next byte after the read value. If there are not enough bytes left in the buffer, the returned value is undefined and the read position is set to the end of the buffer. If the buffer does not exist, 0 is returned. See write_[xxx] for possible types.

read_string(buffer | socketReceiveBuf, size) : string

Read the given number of characters from the buffer and return them as a string. The reading starts at the buffer's current read position. The read position will be advanced to the next byte after the read characters. If there are not enough bytes left in the buffer, the returned string will be shorter than requested and the read position is set to the end of the buffer. If *size* is negative, an empty string is returned and the buffer's read position will be unchanged.

Endianness and compatibility

Data is always sent over the network as a bunch of bytes. For data types that consist of several bytes (e.g. int), that creates the question in which order they are transmitted. Most networking applications use the big-endian format, where the most significant byte

of an integer is sent first (we'll get to floating point values further below.) This is also the default byte order for this extension. However, some applications use the little-endian format where the bytes are sent in the opposite order. Perhaps most important in the context of GM networking, 39dll uses little-endian. In order to support both formats, the functions in this chapter were introduced.

In this library, endianness is an attribute of buffers and sockets. Whenever you write a value to a buffer or socket, it will be converted to a sequence of bytes using the endianness set for this target. The endianness also determines how values are read. Buffers and strings are already sequences of bytes, so writing those is not affected by the endianness of the target.

The previous sentence is pretty important, so let's consider what it means in practice: Setting a socket to little-endian does *not* ensure everything sent or received through it will be little-endian - only values written/read *directly* to/from the socket's send and receive buffer will be converted. If you write your values to a big-endian buffer first and then write that to the little-endian socket, the data will still be sent as big-endian.

Another important point is about floating point values. First off, these are converted to the requested endianness as well. However, there are more uncertainties with floats and doubles than just the byte order, because they can have different formats on different kinds of computers, and there is no single exact standard that everyone uses. This library sends them in the native format used by x86 PCs, and you will not run into problems as long as you communicate with games on other PCs which also use this library. However, if you want to communicate with different software or different kinds of computers, using float and double might give you unexpected results.

set_little_endian_global(bool) : void

If the parameter is true, all new buffers and sockets are created with little-endian byte order, if it is false they will use big-endian. Already created buffers and sockets are not affected. The default byte order is big-endian.

set_little_endian(buffer|socket, bool) : void

Set the socket or buffer to use little-endian (*bool*=true) or big-endian (*bool*=false) byte order.

Miscellaneous

socket_remote_ip(socket) : string

Returns a string representation of the IP address of the remote host that the socket is or has been connected to, or an empty string if the socket has never been connected.

The remote IP becomes available once a connection is established and can be queried until the socket is destroyed, even if the connection has already been closed or if the socket has failed with an error.

Please note that the result can be either an IPv4 address or an IPv6 address, depending on the protocol of the connection.

socket_error(socket | acceptor) : string

Returns a string description of the current error status. This function can return a message even if socket_has_error() returns false. This is particularly the case for acceptors, which can fail partially (for one protocol) but won't flag an error while there's still a usable protocol. For acceptors the error messages for all protocols are returned.

debug_handles() : numHandles

Returns the number of valid handles. This can be used in debug output to find out if some part of your code is leaking buffer or socket handles.

socket_handle_io() : void

This function is deprecated and will be removed in a later version of this extension. Calling it has no effect.