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.
Handles for this library are always integer numbers greater than 0, so you can use a simple conditional check to find out if e.g. a function returned a handle or an error code:

```
newsock = socket_accept();
if(newsock) {
    // Handle the new connection
}
```

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(host, port) : tcpSocket

host 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).

udp_bind(port) : udpSocket

port is an integer in the range of 0 to 65535.
Creates a new UDP socket and binds it to the specified port. The returned socket can be used for receiving datagrams sent to this port. Also, datagrams sent from this socket will use this port as their source port. If *port* is 0, the socket will be bound to a random unused port. The actual port number can be determined using the function *socket_local_port(socket)*.
Note: On Windows versions before Vista, the "random unused port" feature can select a port which is free for IPv4, but in use for IPv6. In that situation, the created socket will only

work with IPv4.

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. Always returns false for UDP sockets.

tcp_listen(port) : acceptor

port is an integer in the range of 0 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.
If the port number is 0, the socket will be bound to a random unused port. The port number can be determined using the function *socket_local_port(socket)*.
Note: On Windows versions before Vista, the “random unused port” feature can select a port which is free for IPv4, but in use for IPv6. In that situation, the created acceptor will only work with IPv4.

socket_accept(acceptor) : tcpSocket | errorcode

Accept a connection from an acceptor. If a connection is available, a 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 handle and 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 socket is a TCP socket and 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 over TCP, you need to

write data to the socket's send buffer and then call socket_send(tcpSocket). Sending datagrams over UDP is similar: You write the content of the datagram to the socket's sendbuffer and then call udp_send(udpSocket, host, port).
When you receive data, it will be stored in the socket's receive buffer, and you can read it from there.
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 prepare your data in a normal buffer first and then use write_buffer to copy that data to the send buffer. Analogous to this, you can copy the contents of a receive buffer into a normal buffer with the same function.

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(tcpSocket) : 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(tcpSocket) : 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 buffered for sending? This will return the approximate number of memory bytes used for this socket to buffer outgoing data, which includes both the actual sendbuffer that you can write to, and the data already queued for sending.
Note that this is an estimate of the actual memory requirement, so it can be larger than the number of bytes actually written to the socket. For example, if you try to send a lot of zero-length UDP datagrams, they will still take up memory in the queue which will be visible here.
If the socket can't push data out to the network as fast as the application demands, the ammount of queued data will grow. This can be used to detect slow connections, so that you can e.g. reduce the data rate. However, the result should be taken with a grain of salt. The number returned does not include data already handed to the operating system, which also manages buffers for outgoing data. This is usually just a few kilobytes, but if you try to send a large block of data at once (up to a few megabytes), Windows might simply accept it so you won't see this data reflected in the result of this function, even though it is still being sent. If you send data in small chunks though, Windows will buffer less of data and you should see a growing backlog pretty quickly.

socket_sendbuffer_limit(socket, size) : void

Limit the memory (in bytes) used by the socket to buffer outgoing data. 0 means no limit. Analogous to socket_sendbuffer_size, this function does not only apply to the actual sendbuffer of the socket, but also to the data already queued for sending.
If a TCP socket would need to buffer more data than this, the connection will be closed and an appropriate error will be indicated. There is no default limit for TCP sockets since a reasonable value depends heavily on the application. To prevent "out of memory" problems it is recommended to set this limit on every TCP socket.
UDP sockets will continue working when the limit is reached, but they will discard datagrams. UDP sockets have a default limit of two megabytes.

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.
You should use this function to determine the size of a UDP packet after receiving it.

udp_send(udpSocket, host, port) : bool

host is a string containing an IPv4 or IPv6 address or a hostname.
port is an integer in the range of 1 to 65535
Send the current content of the UDP socket's sendbuffer as a datagram to the indicated host and port. The sendbuffer will be cleared by calling this function. If the content of the

sendbuffer is too large to send as a single datagram (larger than 65507 Bytes), nothing at all is sent, but the sendbuffer will still be cleared.
While using a hostname for the host parameter is possible, keep in mind that this leads to a hostname resolution every time you send a datagram, and no datagrams are sent while this resolution is running. Therefore, it is strongly recommended to use ip_lookup_create() and related functions to find an IP for that hostname and use that IP directly, unless you only need to send a small number of datagrams to this host and you don't mind that it can delay other datagrams on this socket from being sent.
It is important to note that this function will also send a datagram if the sendbuffer is empty. In this case, a datagram of length 0 will be sent.
If the function returns true, this indicates that datagrams had to be discarded because the send queue has reached its memory limit. You can check and control the size of the send queue with the functions socket_sendbuffer_limit() and socket_sendbuffer_size().

udp_send(buffer, host, port) : bool

host is a string containing an IPv4 or IPv6 address or a hostname.
port is an integer in the range of 1 to 65535
This convenience function allows you to send a buffer as a single datagram via UDP directly, instead of creating a socket first. Internally, this function creates a default UDP socket on a random port when it is first called, and uses that to send the data contained in the buffer. Therefore, the remarks made in the udp_send socket function above apply here as well, and the return value of this function has the same meaning, but you cannot control the send queue for the default socket which is used here.
You can use this function if you only need to send a datagram but don't need to listen for a reply.
Yes, this function looks identical to the one that uses a socket. The two are only distinguished by whether the first argument is a socket handle or a buffer handle.

udp_broadcast(buffer|socket, port) : bool

port is an integer in the range of 1 to 65535
This function is similar to udp_send (both the buffer and socket variant), but instead of sending a datagram to a particular IP, it will send a broadcast that can be received by all computers on the local network. This allows things like automatically discovering LAN servers: Let the LAN server listen on a known port, and let a client search by sending a broadcast to that port. If the server is on the same network as the client, he will receive the packet and can reply with server information.

An alternative way to get a behavior like this would be to send a UDP packet to the "local broadcast" address 255.255.255.255. However, Windows does not handle this well if multiple network interfaces are connected. Depending on the Windows version, sending to 255.255.255.255 will either only send the datagram out of the primary interface, or will send them out all the interfaces but with a wrong source address for all but the primary interface.

This function will try to figure out all connected networks and then sends a directed broadcast to each of them individually. This includes the loopback network (127.0.0.0). Please note that as a result, it is probable that you will receive this broadcast at least twice on the local machine.

A return value of true indicates that datagrams had to be discarded to fit the new datagrams into the send queue.

udp_receive(udpSocket) : bool

Receive a datagram and place its content into the socket's receive buffer. The previous content of the receive buffer is discarded, even if no new datagram is received. Returns true if a datagram was received, false if not.
You can query the remote port and IP of the received datagram by using the functions *socket_remote_port* and *socket_remote_ip*.

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 | socketReceiveBuf) : size

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

buffer_set_readpos(buffer | socketReceiveBuf, 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_hex(buffer | socketSendBuf, hexString) : errorcode

Decode a hexadecimal string and append its binary contents to the buffer, or to the socket's send buffer. For example, *write_hex(buf, "466175636574")* would append the ascii string "Faucet" to the end of *buf*.

hexString must consist of an even number of hexadecimal digits. Both uppercase and lowercase hex digits are accepted. If *hexString* has an odd number of characters or contains characters that are not hex digits, the buffer contents will not be changed.

The return value indicates whether the data was successfully written. A return value of 1 indicates that everything worked, a negative value indicates an error (bad buffer/socket handle or bad hex string).

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 and always copies the entire buffer content.

write_buffer_part(target, source, size) : size

Append *size* bytes from the *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.
In contrast to write_buffer, this function starts reading from the source at the current read position, and advances the read position accordingly. If less than *size* bytes are left to read in the source buffer, only the remaining data is appended to the target buffer.
The number of bytes actually copied is returned.

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.

read_hex(buffer | socketReceiveBuf, size) : string

Read the given number of bytes from the buffer and return them as a lowercase hexadecimal string. For example, if *buf* is a buffer that contains the bytes 123, 231 and 132, then *read_hex(buf, 3)* will return "7be784".
The reading starts at the buffer's current read position. The read position will be advanced by *size* bytes. 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. Otherwise, the resulting string will consist of *size*2* hex digits. 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.

Hostname/IP-Lookup

The basic problem is simple: You have a hostname, and you want an IP to go with that. However, looking a bit closer this problem has a few interesting aspects. One of them is that there might be several IP addresses associated with a single hostname. Some of those might be IPv4 addresses, and some IPv6. For example, google.com has six different IPv4 addresses at the moment. heise.de has one IPv4 and one IPv6 address. So, a simple function that takes a hostname and returns the IP is not sufficient.

A function like that would have another problem as well: Resolving a hostname into an IP can take time, so the function would have to block until the result is available. Just as with blocking socket calls, this would possibly freeze your game for a short time, which looks unprofessional on a game client and has disastrous effects on a game server.

This is why the following functions work differently: They view a lookup (the process of finding the IPs to a hostname) as a resource with a handle. When starting a lookup, you get a handle that you can query regularly until the name resolution is finished. After that, you can query the handle for all results.

Example use:

```
lookup = ip_lookup_create("google.com");

// Wait until the lookup is ready. You can check this once per step
// instead, so your game can continue running normally.
while(!ip_lookup_ready(lookup)) {
    sleep(1);
}

// Show all IPs that were found. If none were found,
// no message is shown.
while(ip_lookup_has_next(lookup)) {
    show_message(ip_lookup_next_result(lookup));
}
ip_lookup_destroy(lookup);
```

ip_lookup_create(hostname) : lookup
ipv4_lookup_create(hostname) : lookup
ipv6_lookup_create(hostname) : lookup

Start a new lookup to find the IP address or addresses associated with the hostname. The first variant will look for both IPv4 and IPv6 addresses, the other two only look for one specific type of address. The return value is a handle to a lookup object which can be used in the functions below, and which has to be destroyed after use.

ip_lookup_ready(lookup) : bool

Returns true if the lookup is finished and the results can be queried with the functions below, false if the lookup is still running. Note that calling this function on an invalid handle will return true, to prevent erroneous code from waiting indefinitely for a nonexistent lookup to finish.

ip_lookup_has_next(lookup) : bool

Returns true if calling ip_lookup_next_result() on this lookup will return another IP, false if all IPs found have already been read or if the lookup is not ready.

ip_lookup_next_result(lookup) : ip

Returns the next IP address found by the lookup. Calling this function repeatedly on a ready lookup will return all IPs found, one at a time, and then return an empty string for every further call. If the lookup is not ready, this function returns the empty string.

ip_lookup_destroy(lookup) : void

Destroys the lookup object and releases the handle.

ip_is_v4(string) : bool
ip_is_v6(string) : bool

Returns whether the string is interpreted as an IPv4/IPv6 literal by the operating system. This can e.g. be used to find out if an IP returned by ip_lookup_next_result() or by

socket_remote_ip() is an IPv4 address or an IPv6 address, or if a user-provided string can be interpreted as an IP at all. Please note though that Windows can interpret strings as IP that you might not expect. For example, simple numbers up to 2^{32} are interpreted as numeric IPv4 literals.
ip_is_v6() will always return false if Windows IPv6 support is not installed.
Both functions return false if an empty string is provided.
IPv6 literals that represent mapped IPv4 addresses (e.g. ::ffff:192.0.2.128) are interpreted as IPv6 by these functions.

Miscellaneous

socket_remote_ip(socket) : string

For a TCP socket, this 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. Sockets returned by socket_accept have the remote IP available unless they fail during accepting.
For a UDP socket, this returns the remote (source) IP of the datagram which is currently stored in the socket's receive buffer, or an empty string if no datagram is currently stored there.
Please note that the result can be either an IPv4 address or an IPv6 address, depending on the protocol of the connection/datagram.

socket_local_port(socket) : port

Returns the local port (1-65535) that the provided socket is bound to. Returns 0 if the socket has an error or does not exist.

socket_remote_port(socket) : port

For a TCP socket, this returns the port number which the socket is or has been connected to, or 0 if the socket has never been connected. The remote port 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. Sockets returned by socket_accept have the remote port available unless they fail during accepting.
For a UDP socket, this returns the remote (source) port number of the datagram which is currently stored in the socket's receive buffer, or 0 if no datagram is currently stored there.

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 or UDP 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.

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.

build_ubyte(bit7, bit6, bit5, bit4, bit3, bit2, bit1, bit0) : real

Construct an unsigned byte out of eight boolean values. Every parameter corresponds to a bit in the resulting byte, starting from the most significant bit (Called bit 7, for its place value of 2^7). A bit in the output will be 1 iff the corresponding parameter evaluates to true (using Game Maker convention for truth, i.e. value>=0.5).

Example: build_ubyte(1,1,0,0,1,0,0,1) will return 201.

bit_set(value, bitnum, bitval) : real

Sets or clears a single bit in *value* and returns the result. This function will modify the bit at place *bitnum* (the one with place value 2^{bitnum}). The bit will be set if *bitval* evaluates to true (using Game Maker convention for truth), it will be cleared otherwise.

It is important to understand that Game Maker's "real" data type is a floating point type, which can exactly represent integer values with up to 53 bits (bit 0 to 52). This means the following constraints must be observed for the parameters:

bitnum must be in the range of 0 to 52. If it is not an integer, it will be truncated.

bitval must be an integer greater or equal to -2^{53} and smaller than 2^{53}.

Using parameters outside this range might give unexpected results. For negative values, two's complement representation is used.

Examples:

bit_set(8, 0, true) will return 9.

bit_set(-8, 0, true) will return -7.

bit_get(value, bitnum) : bool

Checks whether the bit at place *bitnum* in *value* is set.
bitnum must be in the range of 0 to 52. If it is not an integer, it will be truncated.
bitval must be an integer greater or equal to -2^{53} and smaller than 2^{53}.
Using parameters outside this range might give unexpected results. For negative values, two's complement representation is used.
Example: bit_get(10, 1) will return 1 (true).

append_file_to_buffer(buffer | socketSendBuf, filename) : errorcode

Reads the indicated file and appends its entire content to the end of buffer. Returns 1 on success, or a negative error code on failure. If a negative value is returned, the contents of the buffer are not changed. Note that reading very large files (hundreds of megabytes) is not recommended and can cause "out of memory" errors, crashing your game.

write_buffer_to_file(buffer | socketReceiveBuf, filename) : errorcode

Overwrites/creates the file with the provided name and writes the entire buffer content to this file. Returns 1 on success, a negative number on failure.

mac_addrs() : string

Returns a comma-separated string containing the physical addresses (MAC addresses) of the network interfaces on the local machine, e.g. ethernet cards, wifi and similar. Tunnel interfaces, loopbacks and PPP modems are excluded, since those tend not to have interesting physical addresses. This is similar to getmacaddress() in 39dll, except that getmacaddress() only returns the address of the first network interface.
If an error occurs (or no interface is found), an empty string is returned. Otherwise, the returned string consists of a comma-separated list of addresses, where each address is represented as a set of (uppercase) hexadecimal bytes separated by hyphens.
An example return value looks like this: "12-34-56-78-9A-BC,00-81-54-71-DE-CA"
Note that there is no guarantee that the addresses will always be six bytes long, or that they will always be globally unique.