/\*\*

\* The `dgram` module provides an implementation of UDP datagram sockets.

\*

\* ```js

\* import dgram from 'dgram';

\*

\* const server = dgram.createSocket('udp4');

\*

\* server.on('error', (err) => {

\* console.log(`server error:\n${err.stack}`);

\* server.close();

\* });

\*

\* server.on('message', (msg, rinfo) => {

\* console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);

\* });

\*

\* server.on('listening', () => {

\* const address = server.address();

\* console.log(`server listening ${address.address}:${address.port}`);

\* });

\*

\* server.bind(41234);

\* // Prints: server listening 0.0.0.0:41234

\* ```

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/dgram.js)

\*/

declare module 'dgram' {

import { AddressInfo } from 'node:net';

import \* as dns from 'node:dns';

import { EventEmitter, Abortable } from 'node:events';

interface RemoteInfo {

address: string;

family: 'IPv4' | 'IPv6';

port: number;

size: number;

}

interface BindOptions {

port?: number | undefined;

address?: string | undefined;

exclusive?: boolean | undefined;

fd?: number | undefined;

}

type SocketType = 'udp4' | 'udp6';

interface SocketOptions extends Abortable {

type: SocketType;

reuseAddr?: boolean | undefined;

/\*\*

\* @default false

\*/

ipv6Only?: boolean | undefined;

recvBufferSize?: number | undefined;

sendBufferSize?: number | undefined;

lookup?: ((hostname: string, options: dns.LookupOneOptions, callback: (err: NodeJS.ErrnoException | null, address: string, family: number) => void) => void) | undefined;

}

/\*\*

\* Creates a `dgram.Socket` object. Once the socket is created, calling `socket.bind()` will instruct the socket to begin listening for datagram

\* messages. When `address` and `port` are not passed to `socket.bind()` the

\* method will bind the socket to the "all interfaces" address on a random port

\* (it does the right thing for both `udp4` and `udp6` sockets). The bound address

\* and port can be retrieved using `socket.address().address` and `socket.address().port`.

\*

\* If the `signal` option is enabled, calling `.abort()` on the corresponding`AbortController` is similar to calling `.close()` on the socket:

\*

\* ```js

\* const controller = new AbortController();

\* const { signal } = controller;

\* const server = dgram.createSocket({ type: 'udp4', signal });

\* server.on('message', (msg, rinfo) => {

\* console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);

\* });

\* // Later, when you want to close the server.

\* controller.abort();

\* ```

\* @since v0.11.13

\* @param options Available options are:

\* @param callback Attached as a listener for `'message'` events. Optional.

\*/

function createSocket(type: SocketType, callback?: (msg: Buffer, rinfo: RemoteInfo) => void): Socket;

function createSocket(options: SocketOptions, callback?: (msg: Buffer, rinfo: RemoteInfo) => void): Socket;

/\*\*

\* Encapsulates the datagram functionality.

\*

\* New instances of `dgram.Socket` are created using {@link createSocket}.

\* The `new` keyword is not to be used to create `dgram.Socket` instances.

\* @since v0.1.99

\*/

class Socket extends EventEmitter {

/\*\*

\* Tells the kernel to join a multicast group at the given `multicastAddress` and`multicastInterface` using the `IP\_ADD\_MEMBERSHIP` socket option. If the`multicastInterface` argument is not

\* specified, the operating system will choose

\* one interface and will add membership to it. To add membership to every

\* available interface, call `addMembership` multiple times, once per interface.

\*

\* When called on an unbound socket, this method will implicitly bind to a random

\* port, listening on all interfaces.

\*

\* When sharing a UDP socket across multiple `cluster` workers, the`socket.addMembership()` function must be called only once or an`EADDRINUSE` error will occur:

\*

\* ```js

\* import cluster from 'cluster';

\* import dgram from 'dgram';

\*

\* if (cluster.isPrimary) {

\* cluster.fork(); // Works ok.

\* cluster.fork(); // Fails with EADDRINUSE.

\* } else {

\* const s = dgram.createSocket('udp4');

\* s.bind(1234, () => {

\* s.addMembership('224.0.0.114');

\* });

\* }

\* ```

\* @since v0.6.9

\*/

addMembership(multicastAddress: string, multicastInterface?: string): void;

/\*\*

\* Returns an object containing the address information for a socket.

\* For UDP sockets, this object will contain `address`, `family` and `port`properties.

\*

\* This method throws `EBADF` if called on an unbound socket.

\* @since v0.1.99

\*/

address(): AddressInfo;

/\*\*

\* For UDP sockets, causes the `dgram.Socket` to listen for datagram

\* messages on a named `port` and optional `address`. If `port` is not

\* specified or is `0`, the operating system will attempt to bind to a

\* random port. If `address` is not specified, the operating system will

\* attempt to listen on all addresses. Once binding is complete, a`'listening'` event is emitted and the optional `callback` function is

\* called.

\*

\* Specifying both a `'listening'` event listener and passing a`callback` to the `socket.bind()` method is not harmful but not very

\* useful.

\*

\* A bound datagram socket keeps the Node.js process running to receive

\* datagram messages.

\*

\* If binding fails, an `'error'` event is generated. In rare case (e.g.

\* attempting to bind with a closed socket), an `Error` may be thrown.

\*

\* Example of a UDP server listening on port 41234:

\*

\* ```js

\* import dgram from 'dgram';

\*

\* const server = dgram.createSocket('udp4');

\*

\* server.on('error', (err) => {

\* console.log(`server error:\n${err.stack}`);

\* server.close();

\* });

\*

\* server.on('message', (msg, rinfo) => {

\* console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);

\* });

\*

\* server.on('listening', () => {

\* const address = server.address();

\* console.log(`server listening ${address.address}:${address.port}`);

\* });

\*

\* server.bind(41234);

\* // Prints: server listening 0.0.0.0:41234

\* ```

\* @since v0.1.99

\* @param callback with no parameters. Called when binding is complete.

\*/

bind(port?: number, address?: string, callback?: () => void): this;

bind(port?: number, callback?: () => void): this;

bind(callback?: () => void): this;

bind(options: BindOptions, callback?: () => void): this;

/\*\*

\* Close the underlying socket and stop listening for data on it. If a callback is

\* provided, it is added as a listener for the `'close'` event.

\* @since v0.1.99

\* @param callback Called when the socket has been closed.

\*/

close(callback?: () => void): this;

/\*\*

\* Associates the `dgram.Socket` to a remote address and port. Every

\* message sent by this handle is automatically sent to that destination. Also,

\* the socket will only receive messages from that remote peer.

\* Trying to call `connect()` on an already connected socket will result

\* in an `ERR\_SOCKET\_DGRAM\_IS\_CONNECTED` exception. If `address` is not

\* provided, `'127.0.0.1'` (for `udp4` sockets) or `'::1'` (for `udp6` sockets)

\* will be used by default. Once the connection is complete, a `'connect'` event

\* is emitted and the optional `callback` function is called. In case of failure,

\* the `callback` is called or, failing this, an `'error'` event is emitted.

\* @since v12.0.0

\* @param callback Called when the connection is completed or on error.

\*/

connect(port: number, address?: string, callback?: () => void): void;

connect(port: number, callback: () => void): void;

/\*\*

\* A synchronous function that disassociates a connected `dgram.Socket` from

\* its remote address. Trying to call `disconnect()` on an unbound or already

\* disconnected socket will result in an `ERR\_SOCKET\_DGRAM\_NOT\_CONNECTED` exception.

\* @since v12.0.0

\*/

disconnect(): void;

/\*\*

\* Instructs the kernel to leave a multicast group at `multicastAddress` using the`IP\_DROP\_MEMBERSHIP` socket option. This method is automatically called by the

\* kernel when the socket is closed or the process terminates, so most apps will

\* never have reason to call this.

\*

\* If `multicastInterface` is not specified, the operating system will attempt to

\* drop membership on all valid interfaces.

\* @since v0.6.9

\*/

dropMembership(multicastAddress: string, multicastInterface?: string): void;

/\*\*

\* This method throws `ERR\_SOCKET\_BUFFER\_SIZE` if called on an unbound socket.

\* @since v8.7.0

\* @return the `SO\_RCVBUF` socket receive buffer size in bytes.

\*/

getRecvBufferSize(): number;

/\*\*

\* This method throws `ERR\_SOCKET\_BUFFER\_SIZE` if called on an unbound socket.

\* @since v8.7.0

\* @return the `SO\_SNDBUF` socket send buffer size in bytes.

\*/

getSendBufferSize(): number;

/\*\*

\* By default, binding a socket will cause it to block the Node.js process from

\* exiting as long as the socket is open. The `socket.unref()` method can be used

\* to exclude the socket from the reference counting that keeps the Node.js

\* process active. The `socket.ref()` method adds the socket back to the reference

\* counting and restores the default behavior.

\*

\* Calling `socket.ref()` multiples times will have no additional effect.

\*

\* The `socket.ref()` method returns a reference to the socket so calls can be

\* chained.

\* @since v0.9.1

\*/

ref(): this;

/\*\*

\* Returns an object containing the `address`, `family`, and `port` of the remote

\* endpoint. This method throws an `ERR\_SOCKET\_DGRAM\_NOT\_CONNECTED` exception

\* if the socket is not connected.

\* @since v12.0.0

\*/

remoteAddress(): AddressInfo;

/\*\*

\* Broadcasts a datagram on the socket.

\* For connectionless sockets, the destination `port` and `address` must be

\* specified. Connected sockets, on the other hand, will use their associated

\* remote endpoint, so the `port` and `address` arguments must not be set.

\*

\* The `msg` argument contains the message to be sent.

\* Depending on its type, different behavior can apply. If `msg` is a `Buffer`,

\* any `TypedArray` or a `DataView`,

\* the `offset` and `length` specify the offset within the `Buffer` where the

\* message begins and the number of bytes in the message, respectively.

\* If `msg` is a `String`, then it is automatically converted to a `Buffer`with `'utf8'` encoding. With messages that

\* contain multi-byte characters, `offset` and `length` will be calculated with

\* respect to `byte length` and not the character position.

\* If `msg` is an array, `offset` and `length` must not be specified.

\*

\* The `address` argument is a string. If the value of `address` is a host name,

\* DNS will be used to resolve the address of the host. If `address` is not

\* provided or otherwise nullish, `'127.0.0.1'` (for `udp4` sockets) or `'::1'`(for `udp6` sockets) will be used by default.

\*

\* If the socket has not been previously bound with a call to `bind`, the socket

\* is assigned a random port number and is bound to the "all interfaces" address

\* (`'0.0.0.0'` for `udp4` sockets, `'::0'` for `udp6` sockets.)

\*

\* An optional `callback` function may be specified to as a way of reporting

\* DNS errors or for determining when it is safe to reuse the `buf` object.

\* DNS lookups delay the time to send for at least one tick of the

\* Node.js event loop.

\*

\* The only way to know for sure that the datagram has been sent is by using a`callback`. If an error occurs and a `callback` is given, the error will be

\* passed as the first argument to the `callback`. If a `callback` is not given,

\* the error is emitted as an `'error'` event on the `socket` object.

\*

\* Offset and length are optional but both \_must\_ be set if either are used.

\* They are supported only when the first argument is a `Buffer`, a `TypedArray`,

\* or a `DataView`.

\*

\* This method throws `ERR\_SOCKET\_BAD\_PORT` if called on an unbound socket.

\*

\* Example of sending a UDP packet to a port on `localhost`;

\*

\* ```js

\* import dgram from 'dgram';

\* import { Buffer } from 'buffer';

\*

\* const message = Buffer.from('Some bytes');

\* const client = dgram.createSocket('udp4');

\* client.send(message, 41234, 'localhost', (err) => {

\* client.close();

\* });

\* ```

\*

\* Example of sending a UDP packet composed of multiple buffers to a port on`127.0.0.1`;

\*

\* ```js

\* import dgram from 'dgram';

\* import { Buffer } from 'buffer';

\*

\* const buf1 = Buffer.from('Some ');

\* const buf2 = Buffer.from('bytes');

\* const client = dgram.createSocket('udp4');

\* client.send([buf1, buf2], 41234, (err) => {

\* client.close();

\* });

\* ```

\*

\* Sending multiple buffers might be faster or slower depending on the

\* application and operating system. Run benchmarks to

\* determine the optimal strategy on a case-by-case basis. Generally speaking,

\* however, sending multiple buffers is faster.

\*

\* Example of sending a UDP packet using a socket connected to a port on`localhost`:

\*

\* ```js

\* import dgram from 'dgram';

\* import { Buffer } from 'buffer';

\*

\* const message = Buffer.from('Some bytes');

\* const client = dgram.createSocket('udp4');

\* client.connect(41234, 'localhost', (err) => {

\* client.send(message, (err) => {

\* client.close();

\* });

\* });

\* ```

\* @since v0.1.99

\* @param msg Message to be sent.

\* @param offset Offset in the buffer where the message starts.

\* @param length Number of bytes in the message.

\* @param port Destination port.

\* @param address Destination host name or IP address.

\* @param callback Called when the message has been sent.

\*/

send(msg: string | Uint8Array | ReadonlyArray<any>, port?: number, address?: string, callback?: (error: Error | null, bytes: number) => void): void;

send(msg: string | Uint8Array | ReadonlyArray<any>, port?: number, callback?: (error: Error | null, bytes: number) => void): void;

send(msg: string | Uint8Array | ReadonlyArray<any>, callback?: (error: Error | null, bytes: number) => void): void;

send(msg: string | Uint8Array, offset: number, length: number, port?: number, address?: string, callback?: (error: Error | null, bytes: number) => void): void;

send(msg: string | Uint8Array, offset: number, length: number, port?: number, callback?: (error: Error | null, bytes: number) => void): void;

send(msg: string | Uint8Array, offset: number, length: number, callback?: (error: Error | null, bytes: number) => void): void;

/\*\*

\* Sets or clears the `SO\_BROADCAST` socket option. When set to `true`, UDP

\* packets may be sent to a local interface's broadcast address.

\*

\* This method throws `EBADF` if called on an unbound socket.

\* @since v0.6.9

\*/

setBroadcast(flag: boolean): void;

/\*\*

\* \_All references to scope in this section are referring to [IPv6 Zone Indices](https://en.wikipedia.org/wiki/IPv6\_address#Scoped\_literal\_IPv6\_addresses), which are defined by [RFC

\* 4007](https://tools.ietf.org/html/rfc4007). In string form, an IP\_

\* \_with a scope index is written as `'IP%scope'` where scope is an interface name\_

\* \_or interface number.\_

\*

\* Sets the default outgoing multicast interface of the socket to a chosen

\* interface or back to system interface selection. The `multicastInterface` must

\* be a valid string representation of an IP from the socket's family.

\*

\* For IPv4 sockets, this should be the IP configured for the desired physical

\* interface. All packets sent to multicast on the socket will be sent on the

\* interface determined by the most recent successful use of this call.

\*

\* For IPv6 sockets, `multicastInterface` should include a scope to indicate the

\* interface as in the examples that follow. In IPv6, individual `send` calls can

\* also use explicit scope in addresses, so only packets sent to a multicast

\* address without specifying an explicit scope are affected by the most recent

\* successful use of this call.

\*

\* This method throws `EBADF` if called on an unbound socket.

\*

\* #### Example: IPv6 outgoing multicast interface

\*

\* On most systems, where scope format uses the interface name:

\*

\* ```js

\* const socket = dgram.createSocket('udp6');

\*

\* socket.bind(1234, () => {

\* socket.setMulticastInterface('::%eth1');

\* });

\* ```

\*

\* On Windows, where scope format uses an interface number:

\*

\* ```js

\* const socket = dgram.createSocket('udp6');

\*

\* socket.bind(1234, () => {

\* socket.setMulticastInterface('::%2');

\* });

\* ```

\*

\* #### Example: IPv4 outgoing multicast interface

\*

\* All systems use an IP of the host on the desired physical interface:

\*

\* ```js

\* const socket = dgram.createSocket('udp4');

\*

\* socket.bind(1234, () => {

\* socket.setMulticastInterface('10.0.0.2');

\* });

\* ```

\* @since v8.6.0

\*/

setMulticastInterface(multicastInterface: string): void;

/\*\*

\* Sets or clears the `IP\_MULTICAST\_LOOP` socket option. When set to `true`,

\* multicast packets will also be received on the local interface.

\*

\* This method throws `EBADF` if called on an unbound socket.

\* @since v0.3.8

\*/

setMulticastLoopback(flag: boolean): boolean;

/\*\*

\* Sets the `IP\_MULTICAST\_TTL` socket option. While TTL generally stands for

\* "Time to Live", in this context it specifies the number of IP hops that a

\* packet is allowed to travel through, specifically for multicast traffic. Each

\* router or gateway that forwards a packet decrements the TTL. If the TTL is

\* decremented to 0 by a router, it will not be forwarded.

\*

\* The `ttl` argument may be between 0 and 255\. The default on most systems is `1`.

\*

\* This method throws `EBADF` if called on an unbound socket.

\* @since v0.3.8

\*/

setMulticastTTL(ttl: number): number;

/\*\*

\* Sets the `SO\_RCVBUF` socket option. Sets the maximum socket receive buffer

\* in bytes.

\*

\* This method throws `ERR\_SOCKET\_BUFFER\_SIZE` if called on an unbound socket.

\* @since v8.7.0

\*/

setRecvBufferSize(size: number): void;

/\*\*

\* Sets the `SO\_SNDBUF` socket option. Sets the maximum socket send buffer

\* in bytes.

\*

\* This method throws `ERR\_SOCKET\_BUFFER\_SIZE` if called on an unbound socket.

\* @since v8.7.0

\*/

setSendBufferSize(size: number): void;

/\*\*

\* Sets the `IP\_TTL` socket option. While TTL generally stands for "Time to Live",

\* in this context it specifies the number of IP hops that a packet is allowed to

\* travel through. Each router or gateway that forwards a packet decrements the

\* TTL. If the TTL is decremented to 0 by a router, it will not be forwarded.

\* Changing TTL values is typically done for network probes or when multicasting.

\*

\* The `ttl` argument may be between between 1 and 255\. The default on most systems

\* is 64.

\*

\* This method throws `EBADF` if called on an unbound socket.

\* @since v0.1.101

\*/

setTTL(ttl: number): number;

/\*\*

\* By default, binding a socket will cause it to block the Node.js process from

\* exiting as long as the socket is open. The `socket.unref()` method can be used

\* to exclude the socket from the reference counting that keeps the Node.js

\* process active, allowing the process to exit even if the socket is still

\* listening.

\*

\* Calling `socket.unref()` multiple times will have no addition effect.

\*

\* The `socket.unref()` method returns a reference to the socket so calls can be

\* chained.

\* @since v0.9.1

\*/

unref(): this;

/\*\*

\* Tells the kernel to join a source-specific multicast channel at the given`sourceAddress` and `groupAddress`, using the `multicastInterface` with the`IP\_ADD\_SOURCE\_MEMBERSHIP` socket

\* option. If the `multicastInterface` argument

\* is not specified, the operating system will choose one interface and will add

\* membership to it. To add membership to every available interface, call`socket.addSourceSpecificMembership()` multiple times, once per interface.

\*

\* When called on an unbound socket, this method will implicitly bind to a random

\* port, listening on all interfaces.

\* @since v13.1.0, v12.16.0

\*/

addSourceSpecificMembership(sourceAddress: string, groupAddress: string, multicastInterface?: string): void;

/\*\*

\* Instructs the kernel to leave a source-specific multicast channel at the given`sourceAddress` and `groupAddress` using the `IP\_DROP\_SOURCE\_MEMBERSHIP`socket option. This method is

\* automatically called by the kernel when the

\* socket is closed or the process terminates, so most apps will never have

\* reason to call this.

\*

\* If `multicastInterface` is not specified, the operating system will attempt to

\* drop membership on all valid interfaces.

\* @since v13.1.0, v12.16.0

\*/

dropSourceSpecificMembership(sourceAddress: string, groupAddress: string, multicastInterface?: string): void;

/\*\*

\* events.EventEmitter

\* 1. close

\* 2. connect

\* 3. error

\* 4. listening

\* 5. message

\*/

addListener(event: string, listener: (...args: any[]) => void): this;

addListener(event: 'close', listener: () => void): this;

addListener(event: 'connect', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'listening', listener: () => void): this;

addListener(event: 'message', listener: (msg: Buffer, rinfo: RemoteInfo) => void): this;

emit(event: string | symbol, ...args: any[]): boolean;

emit(event: 'close'): boolean;

emit(event: 'connect'): boolean;

emit(event: 'error', err: Error): boolean;

emit(event: 'listening'): boolean;

emit(event: 'message', msg: Buffer, rinfo: RemoteInfo): boolean;

on(event: string, listener: (...args: any[]) => void): this;

on(event: 'close', listener: () => void): this;

on(event: 'connect', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'listening', listener: () => void): this;

on(event: 'message', listener: (msg: Buffer, rinfo: RemoteInfo) => void): this;

once(event: string, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'connect', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'listening', listener: () => void): this;

once(event: 'message', listener: (msg: Buffer, rinfo: RemoteInfo) => void): this;

prependListener(event: string, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'connect', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'listening', listener: () => void): this;

prependListener(event: 'message', listener: (msg: Buffer, rinfo: RemoteInfo) => void): this;

prependOnceListener(event: string, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'connect', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'listening', listener: () => void): this;

prependOnceListener(event: 'message', listener: (msg: Buffer, rinfo: RemoteInfo) => void): this;

}

}

declare module 'node:dgram' {

export \* from 'dgram';

}