NAME

netgraph - graph based kernel networking subsystem

DESCRIPTION

The **netgraph** system provides a uniform and modular system for the implementation of kernel objects which perform various networking functions. The objects, known as *nodes*, can be arranged into arbitrarily complicated graphs. Nodes have *hooks* which are used to connect two nodes together, forming the edges in the graph. Nodes communicate along the edges to process data, implement protocols, etc.

The aim of **netgraph** is to supplement rather than replace the existing kernel networking infrastructure. It provides:

- A flexible way of combining protocol and link level drivers.
- A modular way to implement new protocols.
- A common framework for kernel entities to inter-communicate.
- A reasonably fast, kernel-based implementation.

Nodes and Types

The most fundamental concept in **netgraph** is that of a *node*. All nodes implement a number of predefined methods which allow them to interact with other nodes in a well defined manner.

Each node has a *type*, which is a static property of the node determined at node creation time. A node's type is described by a unique ASCII type name. The type implies what the node does and how it may be connected to other nodes.

In object-oriented language, types are classes, and nodes are instances of their respective class. All node types are subclasses of the generic node type, and hence inherit certain common functionality and capabilities (e.g., the ability to have an ASCII name).

Nodes may be assigned a globally unique ASCII name which can be used to refer to the node. The name must not contain the characters '.' or ':', and is limited to NG_NODESIZ characters (including the terminating NUL character).

Each node instance has a unique *ID number* which is expressed as a 32-bit hexadecimal value. This value may be used to refer to a node when there is no ASCII name assigned to it.

Hooks

Nodes are connected to other nodes by connecting a pair of *hooks*, one from each node. Data flows bidirectionally between nodes along connected pairs of hooks. A node may have as many hooks as it

needs, and may assign whatever meaning it wants to a hook.

Hooks have these properties:

- A hook has an ASCII name which is unique among all hooks on that node (other hooks on other nodes may have the same name). The name must not contain the characters '.' or ':', and is limited to NG_HOOKSIZ characters (including the terminating NUL character).

- A hook is always connected to another hook. That is, hooks are created at the time they are connected, and breaking an edge by removing either hook destroys both hooks.

- A hook can be set into a state where incoming packets are always queued by the input queueing system, rather than being delivered directly. This can be used when the data is sent from an interrupt handler, and processing must be quick so as not to block other interrupts.

- A hook may supply overriding receive data and receive message functions, which should be used for data and messages received through that hook in preference to the general node-wide methods.

A node may decide to assign special meaning to some hooks. For example, connecting to the hook named *debug* might trigger the node to start sending debugging information to that hook.

Data Flow

Two types of information flow between nodes: data messages and control messages. Data messages are passed in *mbuf chains* along the edges in the graph, one edge at a time. The first *mbuf* in a chain must have the M_PKTHDR flag set. Each node decides how to handle data received through one of its hooks.

Along with data, nodes can also receive control messages. There are generic and type-specific control messages. Control messages have a common header format, followed by type-specific data, and are binary structures for efficiency. However, node types may also support conversion of the type-specific data between binary and ASCII formats, for debugging and human interface purposes (see the NGM_ASCII2BINARY and NGM_BINARY2ASCII generic control messages below). Nodes are not required to support these conversions.

There are three ways to address a control message. If there is a sequence of edges connecting the two nodes, the message may be "source routed" by specifying the corresponding sequence of ASCII hook names as the destination address for the message (relative addressing). If the destination is adjacent to the source, then the source node may simply specify (as a pointer in the code) the hook across which the message should be sent. Otherwise, the recipient node's global ASCII name (or equivalent ID-based name) is used as the destination address for the message (absolute addressing). The two types of ASCII

addressing may be combined, by specifying an absolute start node and a sequence of hooks. Only the ASCII addressing modes are available to control programs outside the kernel; use of direct pointers is limited to kernel modules.

Messages often represent commands that are followed by a reply message in the reverse direction. To facilitate this, the recipient of a control message is supplied with a "return address" that is suitable for addressing a reply.

Each control message contains a 32-bit value, called a "typecookie", indicating the type of the message, i.e. how to interpret it. Typically each type defines a unique typecookie for the messages that it understands. However, a node may choose to recognize and implement more than one type of messages.

If a message is delivered to an address that implies that it arrived at that node through a particular hook (as opposed to having been directly addressed using its ID or global name) then that hook is identified to the receiving node. This allows a message to be re-routed or passed on, should a node decide that this is required, in much the same way that data packets are passed around between nodes. A set of standard messages for flow control and link management purposes are defined by the base system that are usually passed around in this manner. Flow control message would usually travel in the opposite direction to the data to which they pertain.

Netgraph is (Usually) Functional

In order to minimize latency, most **netgraph** operations are functional. That is, data and control messages are delivered by making function calls rather than by using queues and mailboxes. For example, if node A wishes to send a data *mbuf* to neighboring node B, it calls the generic **netgraph** data delivery function. This function in turn locates node B and calls B's "receive data" method. There are exceptions to this.

Each node has an input queue, and some operations can be considered to be *writers* in that they alter the state of the node. Obviously, in an SMP world it would be bad if the state of a node were changed while another data packet were transiting the node. For this purpose, the input queue implements a *reader/writer* semantic so that when there is a writer in the node, all other requests are queued, and while there are readers, a writer, and any following packets are queued. In the case where there is no reason to queue the data, the input method is called directly, as mentioned above.

A node may declare that all requests should be considered as writers, or that requests coming in over a particular hook should be considered to be a writer, or even that packets leaving or entering across a particular hook should always be queued, rather than delivered directly (often useful for interrupt routines who want to get back to the hardware quickly). By default, all control message packets are considered to be writers unless specifically declared to be a reader in their definition. (See NGM_READONLY in <*netgraph/ng_message.h*>.)

While this mode of operation results in good performance, it has a few implications for node developers:

- Whenever a node delivers a data or control message, the node may need to allow for the possibility of receiving a returning message before the original delivery function call returns.

- **Netgraph** provides internal synchronization between nodes. Data always enters a "graph" at an *edge node*. An *edge node* is a node that interfaces between **netgraph** and some other part of the system. Examples of "edge nodes" include device drivers, the *socket*, *ether*, *tty*, and *ksocket* node type. In these *edge nodes*, the calling thread directly executes code in the node, and from that code calls upon the **netgraph** framework to deliver data across some edge in the graph. From an execution point of view, the calling thread will execute the **netgraph** framework methods, and if it can acquire a lock to do so, the input methods of the next node. This continues until either the data is discarded or queued for some device or system entity, or the thread is unable to acquire a lock on the next node. In that case, the data is queued for the node, and execution rewinds back to the original calling entity. The queued data will be picked up and processed by either the current holder of the lock when they have completed their operations, or by a special **netgraph** thread that is activated when there are such items queued.

- It is possible for an infinite loop to occur if the graph contains cycles.

So far, these issues have not proven problematical in practice.

Interaction with Other Parts of the Kernel

A node may have a hidden interaction with other components of the kernel outside of the **netgraph** subsystem, such as device hardware, kernel protocol stacks, etc. In fact, one of the benefits of **netgraph** is the ability to join disparate kernel networking entities together in a consistent communication framework.

An example is the *socket* node type which is both a **netgraph** node and a socket(2) in the protocol family PF_NETGRAPH. Socket nodes allow user processes to participate in **netgraph**. Other nodes communicate with socket nodes using the usual methods, and the node hides the fact that it is also passing information to and from a cooperating user process.

Another example is a device driver that presents a node interface to the hardware.

Node Methods

Nodes are notified of the following actions via function calls to the following node methods, and may accept or reject that action (by returning the appropriate error code):

Creation of a new node

The constructor for the type is called. If creation of a new node is allowed, constructor method may allocate any special resources it needs. For nodes that correspond to hardware, this is typically done during the device attach routine. Often a global ASCII name corresponding to the device name is assigned here as well.

Creation of a new hook

The hook is created and tentatively linked to the node, and the node is told about the name that will be used to describe this hook. The node sets up any special data structures it needs, or may reject the connection, based on the name of the hook.

Successful connection of two hooks

After both ends have accepted their hooks, and the links have been made, the nodes get a chance to find out who their peer is across the link, and can then decide to reject the connection. Tear-down is automatic. This is also the time at which a node may decide whether to set a particular hook (or its peer) into the *queueing* mode.

Destruction of a hook

The node is notified of a broken connection. The node may consider some hooks to be critical to operation and others to be expendable: the disconnection of one hook may be an acceptable event while for another it may effect a total shutdown for the node.

Preshutdown of a node

This method is called before real shutdown, which is discussed below. While in this method, the node is fully operational and can send a "goodbye" message to its peers, or it can exclude itself from the chain and reconnect its peers together, like the ng_tee(4) node type does.

Shutdown of a node

This method allows a node to clean up and to ensure that any actions that need to be performed at this time are taken. The method is called by the generic (i.e., superclass) node destructor which will get rid of the generic components of the node. Some nodes (usually associated with a piece of hardware) may be *persistent* in that a shutdown breaks all edges and resets the node, but does not remove it. In this case, the shutdown method should not free its resources, but rather, clean up and then call the **NG_NODE_REVIVE**() macro to signal the generic code that the shutdown is aborted. In the case where the shutdown is started by the node itself due to hardware removal or unloading (via **ng_rmnode_self**()), it should set the NGF_REALLY_DIE flag to signal to its own shutdown method that it is not to persist.

Sending and Receiving Data

Two other methods are also supported by all nodes:

Receive data message

A **netgraph** *queueable request item*, usually referred to as an *item*, is received by this function. The item contains a pointer to an *mbuf*.

The node is notified on which hook the item has arrived, and can use this information in its processing decision. The receiving node must always **NG_FREE_M**() the *mbuf chain* on completion or error, or pass it on to another node (or kernel module) which will then be responsible for freeing it. Similarly, the *item* must be freed if it is not to be passed on to another node, by using the **NG_FREE_ITEM**() macro. If the item still holds references to *mbufs* at the time of freeing then they will also be appropriately freed. Therefore, if there is any chance that the *mbuf* will be changed or freed separately from the item, it is very important that it be retrieved using the **NGI_GET_M**() macro that also removes the reference within the item. (Or multiple frees of the same object will occur.)

If it is only required to examine the contents of the *mbufs*, then it is possible to use the **NGI_M**() macro to both read and rewrite *mbuf* pointer inside the item.

If developer needs to pass any meta information along with the *mbuf chain*, he should use mbuf_tags(9) framework. **Note that old netgraph specific meta-data format is obsoleted now.**

The receiving node may decide to defer the data by queueing it in the **netgraph** NETISR system (see below). It achieves this by setting the HK_QUEUE flag in the flags word of the hook on which that data will arrive. The infrastructure will respect that bit and queue the data for delivery at a later time, rather than deliver it directly. A node may decide to set the bit on the *peer* node, so that its own output packets are queued.

The node may elect to nominate a different receive data function for data received on a particular hook, to simplify coding. It uses the **NG_HOOK_SET_RCVDATA**(*hook*, *fn*) macro to do this. The function receives the same arguments in every way other than it will receive all (and only) packets from that hook.

Receive control message

This method is called when a control message is addressed to the node. As with the received data, an *item* is received, with a pointer to the control message. The message can be examined using the **NGI_MSG**() macro, or completely extracted from the item using the **NGI_GET_MSG**() which also removes the reference within the item. If the item still holds a reference to the message when it is freed (using the **NG_FREE_ITEM**() macro), then the message will also be freed appropriately. If the reference has been removed, the node must free the message itself using the **NG_FREE_MSG**() macro. A return address is always supplied, giving the address of the node that originated the message so a reply message can be sent anytime later. The return address is retrieved from the *item*

using the **NGI_RETADDR**() macro and is of type *ng_ID_t*. All control messages and replies are allocated with the malloc(9) type M_NETGRAPH_MSG, however it is more convenient to use the **NG_MKMESSAGE**() and **NG_MKRESPONSE**() macros to allocate and fill out a message. Messages must be freed using the **NG_FREE_MSG**() macro.

If the message was delivered via a specific hook, that hook will also be made known, which allows the use of such things as flow-control messages, and status change messages, where the node may want to forward the message out another hook to that on which it arrived.

The node may elect to nominate a different receive message function for messages received on a particular hook, to simplify coding. It uses the **NG_HOOK_SET_RCVMSG**(*hook*, *fn*) macro to do this. The function receives the same arguments in every way other than it will receive all (and only) messages from that hook.

Much use has been made of reference counts, so that nodes being freed of all references are automatically freed, and this behaviour has been tested and debugged to present a consistent and trustworthy framework for the "type module" writer to use.

Addressing

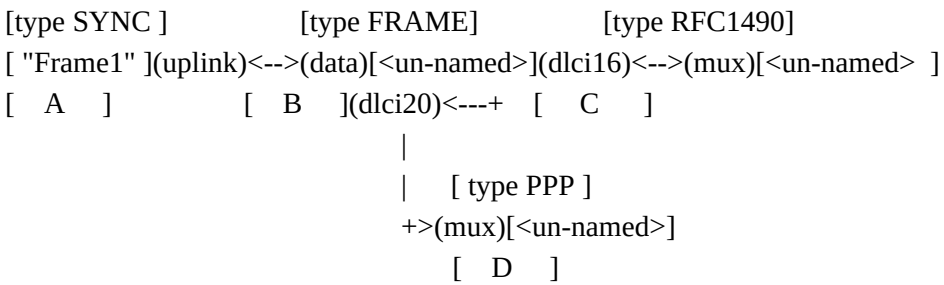
The **netgraph** framework provides an unambiguous and simple to use method of specifically addressing any single node in the graph. The naming of a node is independent of its type, in that another node, or external component need not know anything about the node’s type in order to address it so as to send it a generic message type. Node and hook names should be chosen so as to make addresses meaningful.

Addresses are either absolute or relative. An absolute address begins with a node name or ID, followed by a colon, followed by a sequence of hook names separated by periods. This addresses the node reached by starting at the named node and following the specified sequence of hooks. A relative address includes only the sequence of hook names, implicitly starting hook traversal at the local node.

There are a couple of special possibilities for the node name. The name ‘.’ (referred to as ‘.:’) always refers to the local node. Also, nodes that have no global name may be addressed by their ID numbers, by enclosing the hexadecimal representation of the ID number within the square brackets. Here are some examples of valid **netgraph** addresses:

.:
[3f]:
foo:
.:hook1
foo:hook1.hook2
[d80]:hook1

The following set of nodes might be created for a site with a single physical frame relay line having two active logical DLCI channels, with RFC 1490 frames on DLCI 16 and PPP frames over DLCI 20:



One could always send a control message to node C from anywhere by using the name "Frame1:uplink.dlci16". In this case, node C would also be notified that the message reached it via its hook *mux*. Similarly, "Frame1:uplink.dlci20" could reliably be used to reach node D, and node A could refer to node B as ".:uplink", or simply "uplink". Conversely, B can refer to A as "data". The address "mux.data" could be used by both nodes C and D to address a message to node A.

Note that this is only for *control messages*. In each of these cases, where a relative addressing mode is used, the recipient is notified of the hook on which the message arrived, as well as the originating node. This allows the option of hop-by-hop distribution of messages and state information. Data messages are *only* routed one hop at a time, by specifying the departing hook, with each node making the next routing decision. So when B receives a frame on hook *data*, it decodes the frame relay header to determine the DLCI, and then forwards the unwrapped frame to either C or D.

In a similar way, flow control messages may be routed in the reverse direction to outgoing data. For example a "buffer nearly full" message from "Frame1:" would be passed to node B which might decide to send similar messages to both nodes C and D. The nodes would use *direct hook pointer* addressing to route the messages. The message may have travelled from "Frame1:" to B as a synchronous reply, saving time and cycles.

Netgraph Structures

Structures are defined in <*netgraph/netgraph.h*> (for kernel structures only of interest to nodes) and <*netgraph/ng_message.h*> (for message definitions also of interest to user programs).

The two basic object types that are of interest to node authors are *nodes* and *hooks*. These two objects have the following properties that are also of interest to the node writers.

struct ng_node

Node authors should always use the following **typedef** to declare their pointers, and should never actually declare the structure.

typedef struct ng_node *node_p;

The following properties are associated with a node, and can be accessed in the following manner:

Validity
A driver or interrupt routine may want to check whether the node is still valid. It is assumed that the caller holds a reference on the node so it will not have been freed, however it may have been disabled or otherwise shut down. Using the **NG_NODE_IS_VALID**(*node*) macro will return this state. Eventually it should be almost impossible for code to run in an invalid node but at this time that work has not been completed.

Node ID (*ng_ID_t*)
This property can be retrieved using the macro **NG_NODE_ID**(*node*).

Node name
Optional globally unique name, NUL terminated string. If there is a value in here, it is the name of the node.

```
if (NG_NODE_NAME(node)[0] != '\0') ...

if (strcmp(NG_NODE_NAME(node), "fred") == 0) ...
```

A node dependent opaque cookie
Anything of the pointer type can be placed here. The macros **NG_NODE_SET_PRIVATE**(*node*, *value*) and **NG_NODE_PRIVATE**(*node*) set and retrieve this property, respectively.

Number of hooks
The **NG_NODE_NUMHOOKS**(*node*) macro is used to retrieve this value.

Hooks
The node may have a number of hooks. A traversal method is provided to allow all the hooks to be tested for some condition. **NG_NODE_FOREACH_HOOK**(*node*, *fn*, *arg*, *rethook*) where *fn* is a function that will be called for each hook with the form **fn**(*hook*, *arg*) and returning 0 to terminate the search. If the search is terminated, then *rethook* will be set to the hook at which the search was terminated.

struct ng_hook
Node authors should always use the following **typedef** to declare their hook pointers.

typedef struct ng_hook *hook_p;

The following properties are associated with a hook, and can be accessed in the following manner:

A hook dependent opaque cookie
Anything of the pointer type can be placed here. The macros **NG_HOOK_SET_PRIVATE**(*hook*, *value*) and **NG_HOOK_PRIVATE**(*hook*) set and retrieve this property, respectively.

An associate node
The macro **NG_HOOK_NODE**(*hook*) finds the associated node.

A peer hook (*hook_p*)
The other hook in this connected pair. The **NG_HOOK_PEER**(*hook*) macro finds the peer.

References
The **NG_HOOK_REF**(*hook*) and **NG_HOOK_UNREF**(*hook*) macros increment and decrement the hook reference count accordingly. After decrement you should always assume the hook has been freed unless you have another reference still valid.

Override receive functions
The **NG_HOOK_SET_RCVDATA**(*hook*, *fn*) and **NG_HOOK_SET_RCVMSG**(*hook*, *fn*) macros can be used to set override methods that will be used in preference to the generic receive data and receive message functions. To unset these, use the macros to set them to NULL. They will only be used for data and messages received on the hook on which they are set.

The maintenance of the names, reference counts, and linked list of hooks for each node is handled automatically by the **netgraph** subsystem. Typically a node's private info contains a back-pointer to the node or hook structure, which counts as a new reference that must be included in the reference count for the node. When the node constructor is called, there is already a reference for this calculated in, so that when the node is destroyed, it should remember to do a **NG_NODE_UNREF**() on the node.

From a hook you can obtain the corresponding node, and from a node, it is possible to traverse all the active hooks.

A current example of how to define a node can always be seen in *src/sys/netgraph/ng_sample.c* and should be used as a starting point for new node writers.

Netgraph Message Structure

Control messages have the following structure:

```
#define NG_CMDSTRSIZ    32      /* Max command string (including null) */

struct ng_mesg {
  struct ng_msghdr {
    u_char      version;        /* Must equal NG_VERSION */
    u_char      spare;          /* Pad to 4 bytes */
    uint16_t    spare2;
    uint32_t    arglen;         /* Length of cmd/resp data */
    uint32_t    cmd;            /* Command identifier */
    uint32_t    flags;          /* Message status flags */
    uint32_t    token;          /* Reply should have the same token */
    uint32_t    typecookie;     /* Node type understanding this message */
    u_char      cmdstr[NG_CMDSTRSIZ];  /* cmd string +   */
  } header;
  char  data[];                 /* placeholder for actual data */
};

#define NG_ABI_VERSION  12              /* Netgraph kernel ABI version */
#define NG_VERSION      8               /* Netgraph message version */
#define NGF_ORIG        0x00000000      /* The msg is the original request */
#define NGF_RESP        0x00000001      /* The message is a response */
```

Control messages have the fixed header shown above, followed by a variable length data section which depends on the type cookie and the command. Each field is explained below:

version
: Indicates the version of the **netgraph** message protocol itself. The current version is NG_VERSION.

arglen This is the length of any extra arguments, which begin at *data*.

flags Indicates whether this is a command or a response control message.

token The *token* is a means by which a sender can match a reply message to the corresponding command message; the reply always has the same token.

typecookie
: The corresponding node type’s unique 32-bit value. If a node does not recognize the type cookie it must reject the message by returning EINVAL.

Each type should have an include file that defines the commands, argument format, and cookie for its own messages. The typecookie ensures that the same header file was included by both sender and receiver; when an incompatible change in the header file is made, the typecookie *must* be changed. The de-facto method for generating unique type cookies is to take the seconds from the Epoch at the time the header file is written (i.e., the output of "**date -u** +%s").

There is a predefined typecookie NGM_GENERIC_COOKIE for the *generic* node type, and a corresponding set of generic messages which all nodes understand. The handling of these messages is automatic.

cmd The identifier for the message command. This is type specific, and is defined in the same header file as the typecookie.

cmdstr
Room for a short human readable version of *command* (for debugging purposes only).

Some modules may choose to implement messages from more than one of the header files and thus recognize more than one type cookie.

Control Message ASCII Form

Control messages are in binary format for efficiency. However, for debugging and human interface purposes, and if the node type supports it, control messages may be converted to and from an equivalent ASCII form. The ASCII form is similar to the binary form, with two exceptions:

1. The *cmdstr* header field must contain the ASCII name of the command, corresponding to the *cmd* header field.

2. The arguments field contains a NUL-terminated ASCII string version of the message arguments.

In general, the arguments field of a control message can be any arbitrary C data type. **Netgraph** includes parsing routines to support some pre-defined datatypes in ASCII with this simple syntax:

- Integer types are represented by base 8, 10, or 16 numbers.

- Strings are enclosed in double quotes and respect the normal C language backslash escapes.

- IP addresses have the obvious form.

- Arrays are enclosed in square brackets, with the elements listed consecutively starting at index zero. An element may have an optional index and equals sign ('=') preceding it. Whenever an element

does not have an explicit index, the index is implicitly the previous element’s index plus one.

- Structures are enclosed in curly braces, and each field is specified in the form *fieldname=value*.

- Any array element or structure field whose value is equal to its "default value" may be omitted. For integer types, the default value is usually zero; for string types, the empty string.

- Array elements and structure fields may be specified in any order.

Each node type may define its own arbitrary types by providing the necessary routines to parse and unparse. ASCII forms defined for a specific node type are documented in the corresponding man page.

Generic Control Messages

There are a number of standard predefined messages that will work for any node, as they are supported directly by the framework itself. These are defined in <*netgraph/ng_message.h*> along with the basic layout of messages and other similar information.

NGM_CONNECT
: Connect to another node, using the supplied hook names on either end.

NGM_MKPEER
: Construct a node of the given type and then connect to it using the supplied hook names.

NGM_SHUTDOWN
: The target node should disconnect from all its neighbours and shut down. Persistent nodes such as those representing physical hardware might not disappear from the node namespace, but only reset themselves. The node must disconnect all of its hooks. This may result in neighbors shutting themselves down, and possibly a cascading shutdown of the entire connected graph.

NGM_NAME
: Assign a name to a node. Nodes can exist without having a name, and this is the default for nodes created using the NGM_MKPEER method. Such nodes can only be addressed relatively or by their ID number.

NGM_RMHOOK
: Ask the node to break a hook connection to one of its neighbours. Both nodes will have their "disconnect" method invoked. Either node may elect to totally shut down as a result.

NGM_NODEINFO
: Asks the target node to describe itself. The four returned fields are the node name (if named),

the node type, the node ID and the number of hooks attached. The ID is an internal number unique to that node.

NGM_LISTHOOKS
: This returns the information given by NGM_NODEINFO, but in addition includes an array of fields describing each link, and the description for the node at the far end of that link.

NGM_LISTNAMES
: This returns an array of node descriptions (as for NGM_NODEINFO) where each entry of the array describes a named node. All named nodes will be described.

NGM_LISTNODES
: This is the same as NGM_LISTNAMES except that all nodes are listed regardless of whether they have a name or not.

NGM_LISTTYPES
: This returns a list of all currently installed **netgraph** types.

NGM_TEXT_STATUS
: The node may return a text formatted status message. The status information is determined entirely by the node type. It is the only "generic" message that requires any support within the node itself and as such the node may elect to not support this message. The text response must be less than NG_TEXTRESPONSE bytes in length (presently 1024). This can be used to return general status information in human readable form.

NGM_BINARY2ASCII
: This message converts a binary control message to its ASCII form. The entire control message to be converted is contained within the arguments field of the NGM_BINARY2ASCII message itself. If successful, the reply will contain the same control message in ASCII form. A node will typically only know how to translate messages that it itself understands, so the target node of the NGM_BINARY2ASCII is often the same node that would actually receive that message.

NGM_ASCII2BINARY
: The opposite of NGM_BINARY2ASCII. The entire control message to be converted, in ASCII form, is contained in the arguments section of the NGM_ASCII2BINARY and need only have the *flags*, *cmdstr*, and *arglen* header fields filled in, plus the NUL-terminated string version of the arguments in the arguments field. If successful, the reply contains the binary version of the control message.

Flow Control Messages

In addition to the control messages that affect nodes with respect to the graph, there are also a number of *flow control* messages defined. At present these are *not* handled automatically by the system, so nodes need to handle them if they are going to be used in a graph utilising flow control, and will be in the likely path of these messages. The default action of a node that does not understand these messages should be to pass them onto the next node. Hopefully some helper functions will assist in this eventually. These messages are also defined in <*netgraph/ng_message.h*> and have a separate cookie NG_FLOW_COOKIE to help identify them. They will not be covered in depth here.

INITIALIZATION

The base **netgraph** code may either be statically compiled into the kernel or else loaded dynamically as a KLD via kldload(8). In the former case, include

options NETGRAPH

in your kernel configuration file. You may also include selected node types in the kernel compilation, for example:

options NETGRAPH
options NETGRAPH_SOCKET
options NETGRAPH_ECHO

Once the **netgraph** subsystem is loaded, individual node types may be loaded at any time as KLD modules via kldload(8). Moreover, **netgraph** knows how to automatically do this; when a request to create a new node of unknown type *type* is made, **netgraph** will attempt to load the KLD module *ng_<type>.ko*.

Types can also be installed at boot time, as certain device drivers may want to export each instance of the device as a **netgraph** node.

In general, new types can be installed at any time from within the kernel by calling **ng_newtype**(), supplying a pointer to the type's *struct ng_type* structure.

The **NETGRAPH_INIT**() macro automates this process by using a linker set.

EXISTING NODE TYPES

Several node types currently exist. Each is fully documented in its own man page:

SOCKET
: The socket type implements two new sockets in the new protocol domain PF_NETGRAPH. The new sockets protocols are NG_DATA and NG_CONTROL, both of type SOCK_DGRAM.

Typically one of each is associated with a socket node. When both sockets have closed, the node will shut down. The NG_DATA socket is used for sending and receiving data, while the NG_CONTROL socket is used for sending and receiving control messages. Data and control messages are passed using the sendto(2) and recvfrom(2) system calls, using a *struct sockaddr_ng* socket address.

HOLE
: Responds only to generic messages and is a "black hole" for data. Useful for testing. Always accepts new hooks.

ECHO
: Responds only to generic messages and always echoes data back through the hook from which it arrived. Returns any non-generic messages as their own response. Useful for testing. Always accepts new hooks.

TEE This node is useful for "snooping". It has 4 hooks: *left*, *right*, *left2right*, and *right2left*. Data entering from the *right* is passed to the *left* and duplicated on *right2left*, and data entering from the *left* is passed to the *right* and duplicated on *left2right*. Data entering from *left2right* is sent to the *right* and data from *right2left* to *left*.

RFC1490 MUX
: Encapsulates/de-encapsulates frames encoded according to RFC 1490. Has a hook for the encapsulated packets (*downstream*) and one hook for each protocol (i.e., IP, PPP, etc.).

FRAME RELAY MUX
: Encapsulates/de-encapsulates Frame Relay frames. Has a hook for the encapsulated packets (*downstream*) and one hook for each DLCI.

FRAME RELAY LMI
: Automatically handles frame relay "LMI" (link management interface) operations and packets. Automatically probes and detects which of several LMI standards is in use at the exchange.

TTY This node is also a line discipline. It simply converts between *mbuf* frames and sequential serial data, allowing a TTY to appear as a **netgraph** node. It has a programmable "hotkey" character.

ASYNC
: This node encapsulates and de-encapsulates asynchronous frames according to RFC 1662. This is used in conjunction with the TTY node type for supporting PPP links over asynchronous serial lines.

ETHERNET
: This node is attached to every Ethernet interface in the system. It allows capturing raw Ethernet frames from the network, as well as sending frames out of the interface.

INTERFACE
: This node is also a system networking interface. It has hooks representing each protocol family (IP, IPv6) and appears in the output of ifconfig(8). The interfaces are named "ng0", "ng1", etc.

ONE2MANY
: This node implements a simple round-robin multiplexer. It can be used for example to make several LAN ports act together to get a higher speed link between two machines.

Various PPP related nodes
: There is a full multilink PPP implementation that runs in **netgraph**. The *net/mpd5* port can use these modules to make a very low latency high capacity PPP system. It also supports PPTP VPNs using the PPTP node.

PPPOE
: A server and client side implementation of PPPoE. Used in conjunction with either ppp(8) or the *net/mpd5* port.

BRIDGE
: This node, together with the Ethernet nodes, allows a very flexible bridging system to be implemented.

KSOCKET
: This intriguing node looks like a socket to the system but diverts all data to and from the **netgraph** system for further processing. This allows such things as UDP tunnels to be almost trivially implemented from the command line.

Refer to the section at the end of this man page for more nodes types.

NOTES

Whether a named node exists can be checked by trying to send a control message to it (e.g., NGM_NODEINFO). If it does not exist, ENOENT will be returned.

All data messages are *mbuf chains* with the M_PKTHDR flag set.

Nodes are responsible for freeing what they allocate. There are three exceptions:

1. *Mbufs* sent across a data link are never to be freed by the sender. In the case of error, they should be considered freed.

2. Messages sent using one of **NG_SEND_MSG_*()** family macros are freed by the recipient. As in the case above, the addresses associated with the message are freed by whatever allocated them so the recipient should copy them if it wants to keep that information.

3. Both control messages and data are delivered and queued with a **netgraph** *item*. The item must be freed using **NG_FREE_ITEM(***item***)** or passed on to another node.

FILES

<netgraph/netgraph.h>
Definitions for use solely within the kernel by **netgraph** nodes.

<netgraph/ng_message.h>
Definitions needed by any file that needs to deal with **netgraph** messages.

<netgraph/ng_socket.h>
Definitions needed to use **netgraph** *socket* type nodes.

<netgraph/ng_><type>.h
Definitions needed to use **netgraph** *type* nodes, including the type cookie definition.

/boot/kernel/netgraph.ko
The **netgraph** subsystem loadable KLD module.

/boot/kernel/ng_<type>.ko
Loadable KLD module for node type *type*.

src/sys/netgraph/ng_sample.c
Skeleton **netgraph** node. Use this as a starting point for new node types.

USER MODE SUPPORT

There is a library for supporting user-mode programs that wish to interact with the **netgraph** system. See netgraph(3) for details.

Two user-mode support programs, ngctl(8) and nghook(8), are available to assist manual configuration and debugging.

There are a few useful techniques for debugging new node types. First, implementing new node types in

user-mode first makes debugging easier. The *tee* node type is also useful for debugging, especially in conjunction with ngctl(8) and nghook(8).

Also look in */usr/share/examples/netgraph* for solutions to several common networking problems, solved using **netgraph**.

SEE ALSO

socket(2), netgraph(3), ng_async(4), ng_bluetooth(4), ng_bpf(4), ng_bridge(4), ng_btsocket(4), ng_car(4), ng_cisco(4), ng_device(4), ng_echo(4), ng_eiface(4), ng_etf(4), ng_ether(4), ng_frame_relay(4), ng_gif(4), ng_gif_demux(4), ng_hci(4), ng_hole(4), ng_hub(4), ng_iface(4), ng_ip_input(4), ng_ipfw(4), ng_ksocket(4), ng_l2cap(4), ng_l2tp(4), ng_lmi(4), ng_mppc(4), ng_nat(4), ng_netflow(4), ng_one2many(4), ng_patch(4), ng_ppp(4), ng_pppoe(4), ng_pptpgre(4), ng_rfc1490(4), ng_socket(4), ng_split(4), ng_tee(4), ng_tty(4), ng_ubt(4), ng_UI(4), ng_vjc(4), ng_vlan(4), ngctl(8), nghook(8)

HISTORY

The **netgraph** system was designed and first implemented at Whistle Communications, Inc. in a version of FreeBSD 2.2 customized for the Whistle InterJet. It first made its debut in the main tree in FreeBSD 3.4.

AUTHORS

Julian Elischer <*julian@FreeBSD.org*>, with contributions by Archie Cobbs <*archie@FreeBSD.org*>.