Version 2.1

NeSeL

NETWORK SPECIFICATION LANGUAGE

Users Guide & Reference Manual

City University London

April 1989, Revised November 1996.
Supported by a grant from SERC

Table of Contents

OVERVIEW 1

NODES AND DATA STRUCTURES 2

NODE FUNCTIONS 4

NETS 4

PATHS, MACROS AND LABELS 5

CONNECTIONS 7

Auxiliary Nets 8

SPECIAL NETS: BIN, HEAP AND PILE 9

FLOATING DECLARATIONS 9

LISTS 10

List Evaluation, Selectors and Partitions 10

Variables and Expressions 11

List Operators 11

Function 11

Conditionals and Loops 12

THE NESEL PROGRAMMING ENVIRONMENT 12

NETWORK REDUCTION ORDER 17

NODE DATA STRUCTURES 17

NODE FUNCTIONS 18

MACROS 19

NETS 19

Object Declarations 20

Floated Paths 20

Labels 21
Qualifiers 21
Partitions 21

Connection Lists 22
Connection Operators 22
Auxiliary Nets 23
Label Relative Paths & Global Connections 23
Inline Variable Initialisers 24

Additional declarations 24

SPECIAL NETS: BIN, HEAP & PILE 25

The Bin 25

The Heap and Pile 25

LISTS 25

List evaluation 26

Selectors 26
Path Representation and Selector Co-ordinates 27
Linked Lists and the Data Structure Indicator 28

List Operators 29
User list functions 29

PRAGMAS 30

RESERVED WORDS 30

THE NESEL COMPILER: NSL1 AND NSL2 31

nsl1 31

nsl2 33

Current Compiler Restrictions 33

THE NESEL CONVERTER: NSLTOX 35

THE NESEL LOADER: NSL 37

THE STAND-ALONE MONITOR: NSLW & NSLMS 38

NETWORK SPECIFICATION FILE FORMATS 39

THE NESEL LIBRARY FILES 40

General Files 40

Small Model 40

Large Model 40

STANDARD FUNCTIONS 40

Input Access Functions (nslios.h) 40

Data Structure Access Functions (nslios.h) 40

Simulator Functions (nslios.h) 41

Monitor Functions (nslios.h) 42

Debugger Functions (nslios.h) 43

Window & Graphics Functions (nslwsio.h) 45

List Functions (nslstd.h) 49

Standard List Macros (nslstd.h) 51

Vector Functions (nslios.h) 51

Variables (nslios.h) 51

NeSeL Error Codes 52

NeSeL Types 52

INSTALLING NESEL 53

The Source Disk 53

NeSeL Demonstration 54

Updating User Functions 54

Errors and Debugging Tips 54

SYNTAX NOTATION 57

Chapter 1

NeSeL Language Introduction

Overview

NeSeL is a general purpose language for specifying networks in a concise and natural way. It is primarily an extension of the C language. A network is a set of nodes and arcs. In NeSeL nodes are atomic functions which operate on their respective data structures, and arcs represent communication channels between nodes. Traditional digital and analogue circuits, and networks of high connectivity in which the interconnect is most readily described by distribution functions, for example neural nets, are catered for.

Specifications are based on a simple network model in which a network is reducible to a set of coupled functions. Quantising time, by making functions respond at specific frequencies, completes this basic model. The model can best be explained algebraically. Suppose we have a set of parameterised node functions labelled “A” to “Z”, such that the inputs to a function are expressed as parameters:

$A_{t+1} = A(B_t, F_t, G_t, S_t)$
$B_{t+1} = B(E_t, F_t)$
$C_{t+1} = C(A_t, B_t, C_t)$
etc.,

where the subscript **t** represents a certain time epoch, and the functions **A(...)** map the prior node state to the new one. By definition each application of a function takes exactly one time epoch to complete. To simulate more advanced features, such as delayed inputs or asynchrony, simply requires using delay functions and randomising functions, respectively. For example, if we want to take as inputs states further back in time than the last state, such as **t-1**, we would include a delay function:

$D_{t+1} = D(D_t)$
$E_{t+1} = C(A_t, D(B_{t-1}))$

Here, function **D(x)** simply propagates its input to its output incurring a one epoch delay in the process.

Continuous (analogue) response functions are modelled by choosing a base time quantum that is, say, an order of magnitude smaller than the observed critical dynamic resolution. This is a very primitive model, but it lets the user model exactly what they want. However, the standard NeSeL libraries contain functions and macros for modelling some features in a high level fashion. Armed with a specification in NeSeL it can be fed to the NeSeL compiler, which checks it in various ways as it converts it to standard C, consequently enabling it to be simulated.

A NeSeL specification consists of a hierarchy of nets. Each net specifies the child nets and nodes it contains and how they are connected together. Hence, a particular network can be referred to by the root parent net. For example:

```
net nand( )
{  not( ) output:, input = and/output;
        and( ) input:;
}
```

This describes a net called **nand** which performs the logical nand operation. Its children, **not** and **and** may be either nodes or nets, this distinction is transparent to the user. The **nand** gate may in turn be used by other nets.

The nets and nodes in a network have an associated precedence. This stems from the evaluation order followed when a net is reduced to a set of coupled nodes during compilation. In fact, the evaluation order is very much like that of a sequential procedural language. Beginning at the root net, which is always called **network**, each child net is reduced as it appears in the net declaration. Once all the child nets have been reduced, the connection specifications of the net are evaluated. Data structure initialisation occurs as a separate, distinct, phase in an analogous order to net reduction. Namely, the network is descended to the first terminal node structure or normal structure (see below). It is initialised, and initialisation proceeds with the next lowest structure, and so on. This continues until the root net and its variable initialisers have been evaluated. Initialisation occurs after net reduction because the exact members of a nodes data structure aren’t known until all its connections have been dealt with.

The practical upshot of this reduction order is that later declarations may affect prior ones by, say, overwriting previous connections. This is seen as a virtue of the language, and where necessary is easily countered by “floating” declarations to parental nets. More of this later though. Hence, a NeSeL specification looks very similar to a normal procedural language, except that procedural evaluation is used to produce a network in the form of a set of coupled concurrent functions. Parallel execution of these functions consequently emulates the specified network.

The following sections explain how to specify node data structures, node functions and nets. Finally, the NeSeL programming environment is explained with an example. Further details may be found in the Reference section.

Nodes and Data Structures

The nodes in a network are specified by standard C functions. These functions are called upon at a user defined rate to compute the next state and output values of a node based on its prior inputs and state. Output values, state information and input addresses are stored in a nodes personal data structure. Notice that a node can have many different outputs, in fact a net could be modelled by a single node if required. Thus, each node is described by a function and the data structure it operates on.

The C language **struct** declarator is used to specify node data structures. These structures can contain substructures to an arbitrary level, arrays, other data types and **typedef**s. Substructures might be used, for example, to represent a neurons dendritic and axonic tree. Two sorts of structure types are distinguished in NeSeL, normal C structures and node structures. The later are analogous to normal ones except that they can be given default values and appear as non-terminal unbound arrays. Unbound arrays, i.e. array declarations in which no size is specified, are permitted in a data structure provided either the declaration is a terminal member (i.e. a non-**struct** type or no reference is made to its sub-members if it's a **struct**) or it's a node structure. New node structure definitions are introduced in a similar fashion to a **typedef struct** declaration but by way of the reserved word "**node**". For example:

```
node                          typedef struct
{       short int *input[ ];  {       float gain,offset,gradient;
        float weight;         } parms;
} synapse;
node
{       int output;
        synapse dendrite[ ];
        parms state;
}       discrete_neuron = {0,(synapse){0,0.5},{1.0,0.5,-1.0}};
```

Here a couple of node structures, "synapse" and "discrete_neuron", are introduced along with a normal **typedef struct** declaration. Notice unbound array "dendrite". NeSeL converts this to a pointer type and determines the array size during compilation, this is transparent to the user. The member names "input" and "output" are used here purely for clarity, any valid identifier is allowed.

To clarify the difference between terminal and non-terminal unbound arrays consider the following node structure declaration for a logic gate:

```
node
{       BOOL output;
        int state;
        BOOL *input[ ];
} logic_gate = {FALSE,0,{FALSE}};
```

Here member "input" is a terminal unbound array. In "discrete_neuron", member "dendrite" was a non-terminal unbound array, but it was a node structure type. Although "discrete_neuron" looks like a variable declaration it's actually a type, like **int** or **char***, but NeSeL allows it to be given default values. In addition, structure member names and wild names can be used to specify particular members to be initialised. For example:

```
        node {...} discrete_neuron = {state = {, 0, grad* = 2.0} };
```

which only sets the state parameters "offset" and "gradient". Note the empty field (i.e. spacing comma) and wild name "grad*" used to select "gradient". Wild names are discussed shortly. The type "discrete_neuron" might represent a simple neuron node data structure, such as a Hopfield first order neuron.

Finally, notice that the inputs to a node are pointer types. In general an input is a pointer type to that of the output it's connected to. For example, in the above "logic_gate" structure, member "*input[ ]" refers to an array of BOOL outputs from other node structures.

Another reason for distinguishing between normal and node structures is to identify node function declarations, which will now be described.

Node Functions

A node function declaration is introduced by specifying a pseudo normal C function with a node struct type. For example,

```
logic_gate not(output)
        in input; out output;
{       int i; BOOL output = FALSE;
        for(i=0;i<$.size.input;++i) output |= ini($.input[i]);
        return(!output);
}
```

This defines a node function called "not" which operates on a data structure of type "logic_gate". Parameters to node functions are used to initialise the data structure at compile time. In this case a net might use a "not" gate via "...not(LOW)..." which sets "output" accordingly. Alternatively, explicit members can be given by the net, as in "...not(state=BOOT)...". Where parameter types would normally go in standard C functions optional communication access privileges can be given. These are guards on what members of the data structure are visible to its users. A guard is a guard type followed by a list of names. Again wild names can be used. Permissible guard types are "in", "out" and "inout" which declare an identifier to be an input only, output only or both respectively.

Finally, the node function accesses its data structure through the special dollar node pointer symbol "$". No explicit return value should be given in node functions. NeSeL redeclares these functions to be of type **void** when it translates the pointer symbol and communications into standard C. Default values can also be given to a nodes data structure by following the function body with a variable initialiser in a similar fashion to node data structure declarations. For example:

```
discrete_neuron hopfield(...)
{/* function body */}
 = {output = 0, dendrite = (synapse){weight = 0.5}}
```

Nets

A net is a collection of nodes and subnets connected together in some fashion. To specify such collections a net declaration facility is provided. This is one of the main features of NeSeL, it's also the only other appendage to the C language. For someone already familiar with C this makes NeSeL very easy to pick up. A net declaration is a special kind of macro. Whenever a net is used by another net it's actually a macro expanded copy of the net that's used. Similarly, when a net uses a node it's a copy of that node that's taken. With this in mind an outline syntax of a net declaration is:

```
net ident([[macro_parms]])
```

```
        [[comm_decls]]
        {[[object_decls and additional_decls]] } [[variable_decls]]
```

where the double square brackets indicate optional parts. An object can be either a net or node.

Object declarations specify what nodes or nets a net contains and how they are connected. Additional declarations specify any additional connections that might be needed between objects. The optional communications and variable initialiser declarations are the same as for node functions. However, unlike node struct type initialisers, node function and net initialisers may refer to variables by their path, or wild-paths. NeSeL paths are described in a moment. For now look at the specification of a net for adding two bits, called “a” and “b”, producing a “sum” and “carry”:

```
net halfadder(boot)
   in a,b; out sum,carry;
{  and(boot)[2]
         a: [1]/input[0],
         b: [1]/input[1],
         sum: [0]/output,
         carry: [1]/output,
         [0]=(not(boot))[1]+or;
   or(boot)
         ~/input[?]= >a + >b;
}
```

```
net fulladder(boot)
        in a,b,carryin; out sum,carry;
{       halfadder(boot)[2]
                a: [0]/a,
                b: [0]/b,
                carryin: [1]/b,
                sum: [1]/sum,
                [1]/a = [0]/sum;
        or(boot) carry:,
        input[?] = halfadder[1,0]/carry;
}
```

Net “halfadder” contains two object declarations: “and( )” and “or( )”. Object declarations always begin with the name of a net or node and a pair of brackets enclosing optional arguments. Single dimension arrays of objects are permitted, such as in “and(boot)[2]”. The object stub may then be followed by an optional comma separated list of object elements, which are label and connection specifications, finally terminated by a semicolon.

An outline syntax of an object element is:

```
          in_path = conx_list,
    or    label: in_path = conx_list,
    or    label: path,
```

where “conx_list” is a list of connection elements, each of which comprises connection combination operators and an output path with an optional variable initialiser. An optional auxiliary net may be prefixed to the output path. A label is either an identifier or an identifier post-fixed by a list. Auxiliary nets and lists are described later.

Paths, Macros and Labels

Nets, nodes and data structure members are referred to by paths in a similar fashion to the UNIX path naming convention. Thus, the root net corresponds to, say, the root directory in a UNIX file system. Subnets and nodes correspond to subdirectories as do node structs and normal structs. Basic (non-struct) data variables correspond to terminal file entries in a directory.

The UNIX path reference qualifiers (i.e. backslash or forward slash, double dot and single dot) can be used to signify paths relative to the root net, parent nets and the current net. Wild-path names are permitted, just as in UNIX. There is an additional wild-level operator “**” which matches any number of sublevels.

In large nets and data structures long path names can become tedious to work with. Macros and labels are provided to alleviate this problem. Macros are used in a fairly obvious way - we just define suitable ones for common paths. This can be achieved with either the C “#define” pre-processor directive, or by defining a NeSeL macro with the keyword “macro”. For example:

```
    macro interrupt(type,num) {/cpu/intr_latch[num]/type}
```

Whenever the identifier “interrupt” is seen during compilation it is replaced by the contents of the curly brackets with the appropriate parameter substitutions. Macro expansion in net declarations and macros conforms to the ANSI C standard with some additions such as a special form of the C pre-processor conditional directives. Simply write conditional blocks as normal but replace the hash “#” by a dollar character “$”. For example:

```
    net multilayer(%n,x,s,rest)
    {
    $if(n<2)
        feedforward(x)[s]
    $else
        multilayer(n-1,rest)
    $endif
    }
```

might be used to create a generic multilayered network of feedforward nets. Note the mutually recursive use of “multilayer”. A “$” has to be used instead of the “#” in order to stop the C pre-processor catching the conditional directives. This also highlights the different evaluation times of **#define**’s and **macro**s. The C pre-processor is run on net specifications and its output fed to the NeSeL compiler. Macro expansion occurs in the course of net reduction.

Labels are defined in object declarations. In the net “halfadder” some labels were defined. For example, label “sum” refers to the low order bit of adding the two inputs “a” and “b”. Thus, instead of referring to the output as “halfadder/and[0]/output” we would write “halfadder/sum”, similarly for the inputs and carry. Note the list on “and” for selecting a particular gate. Lists can be used with labels in two ways: in their definition or in a path as above.

```
    net ADD2x2(boot)                    net ADD2(boot)
        in input; out output;               in input; out output;
    {   ADD2(boot)[4]                   {   and(boot)[2]
            input[0,2]: [3]/input[0..1],        input[*]: [1]/input[0..1],
            input[1]: [2]/input[1],             output[*]: [*]/output,
            input[3]: [1]/input[1],             [0]=(not(boot))[1]+or;
            output[0..2]:                   or(boot)
                [3,1,0]/output[0],              ~/input[?]=>input[..];
            [0]=[1]/output[1]           }
                +[2]/output[1],
            [1]=[2]/output[0],
            [2]=[3]/output[1];
    }
```

In “ADD2x2” the same label, in this case “input”, is used to select different paths according to the list index. This presents a uniform naming and index range for inputs to the net as seen by its users. The same label can be used with different objects. If such a label has overlapping indices then it refers to each object according to the reduction order. Now, the list elements in a label-list are mapped to the first list appearing in the defining path. For example, the elements in label "output[0..2]" are mapped linearly to "ADD2[3,1,0]", i.e. "output[0] → ADD2[3]" . Thus, if we referred to "ADD2x2" in a path, such as "ADD2x2/output[0]", this is equivalent to "ADD2x2/ADD2[3]/output[0]".

Lists can be applied in paths to non label-lists or objects. In this case the list is mapped to the first list appearing in the label definition. A default list is added when necessary. Path members in label definitions may themselves refer to labels or label-lists, so a list element may be mapped over may labels. Note the difference in mapping between a label-list and the leftmost list in its definition, which is by position, and a list in a path and the corresponding list in the label definition, which is by index, see Reference section.

Connections

The basic form of a connection specification in an object element is:

```
in_path = out_path,
```

meaning, form connections from the terminal data structure members referred to by the output path to those referred to by the input path. Input paths occurring in object elements are always relative to the object, unless floated. This saves a bit of typing. Output paths are relative to the enclosing net. To refer to other objects or labels in the enclosing net from an input path the UNIX "./" qualifier can be used. For example,

```
net fred( ) { or( ); and( ) ./or=output; }
```

Connections are unidirectional information flows from the source to the destination. Bi-directional connections are modelled by making connections in the reverse direction. Operators can be placed before the output path to achieve other combination actions. These are "+" for addition, "-" for subtraction, "++" for overwriting, "--" an omni form of subtraction, ">" for forking connections, "^" diverting, "|" for grouping connection elements, "<" merging, and "!" for copying. These later two operators are used in association with auxiliary nets and stack operations respectively (described later). Addition is the default operation. NeSeL treats the input and output paths in connection specifications as sequential streams. This will be more apparent later on when the special functions, called selectors, are dealt with. For now consider a simple example:

```
A/B[0..5]/C[8,2..4] = X[0..2]/Y[7..0]
```

The input stream expects 24 connections, which the output supplies. Connections are pulled from the output stream on demand by the input stream.

The right-hand side of a connection specification can contain a list of output paths and their operators. Consecutive output paths are treated as a single group if they do not contain the subtraction operator. So, for example,

```
A[0..5] = X[0..2] + X[3..5] - Z[2,4]
```

is equivalent to the more verbose declaration:

```
A[0..2] = X[0..2], A[3..5] = X[3..5], A[0..5] = -Z[2,4]
```

This default grouping can be overridden with the connection grouping operator "|", see reference section. Now, during evaluation any excess output connections are ignored, and a group is rescanned if the input stream requires more connections.

In the second example, below, connections are forked from the prior connections added to the points indicated by the output path,

```
A/B[0..4] = >X[3,1,0,4,2]
```

All connection paths must ultimately yield terminal data structure members. NeSeL applies default rules to paths which fail to reach such points. It adds the required number of “/input[*]” and “/output[*]” path members to input and output paths respectively. Of course this only works if nets, labels or data structure members of the same name have been defined where necessary. In the above example forked path “X[...]” is treated as an input path if the default rules need to be applied.

A connection path can be post-fixed by a struct initialiser. The initialiser is applied to the parental struct of the input connection variable. These inline initialisers are useful for setting connection weight vectors and so forth when used with a list initialiser. See Reference section for more details on inline initialisers and list initialisers:

```
A[+]/wt = X[0..9]{(float)[vector_func(...)]}
```

where the function “vector_func( )” might return a list of **float**’s.

Auxiliary Nets

Sometimes we may want to filter the signal sent down a connection before it reaches its destination. A simple example would be using a logical **not** gate to invert an input. Consider one way of doing this in NeSeL:

```
net example( ) { A( ); B( ); not( ); A=not; not=B; }
```

This makes a connection to object A from B and inverts any signals received. What if a number of signals needed to be inverted? An array of **not** gates could be used and connections routed through each in turn. However, an easier method is available, namely auxiliary nets. These are specified by enclosing the filter net in brackets and inserting it in the connection specification:

```
net example( ) { A( ); B( ); A=(not( ))B; }
```

An auxiliary net could be a single node or a large net. The only restriction being that its inputs and outputs must be similarly named. They also cannot be accessed externally through paths for example. This means they can only be initialised by their data structure initialisers, or directly with arguments in the connection declaration.

Another example might be a delay line declared as an auxiliary net:

```
net delay(x,boot)
{
$if(x>1)
	epoch(boot)[x] input:[0],output:[x-1],[1..(x-1)]=[0..(x-2)];
$elif(x==1)
	epoch(boot) input:,output:;
$endif
}
```

Function “epoch” would simply copy its input to its output. Notice that if “x<1”, i.e. no delay, the net “delay” won’t contain any declarations. Such nets are called null nets. They can only appear as auxiliary nets. When they do, they are ignored.

A more economical way of specifying “delay” for simulations would be as a node function which queued signals:

```
node {BOOL output,*input,*queue; int delay;}
delay_line={FALSE,,NULL,1};
delay_line delay(delay,output) in input; out output;
{       int nn;
        if($.delay>1)
        {       if(!$.queue)
                {       $.queue=(BOOL *)malloc($.delay*sizeof(BOOL));
                        for(nn=0;nn<$.delay;++nn) $.queue[nn]=$.output;
                }
                for(nn=1;nn<$.delay;++nn) $.queue[nn-1]=$.queue[nn];
                $.queue[$.delay-1]=inx(BOOL,$.input);
                $.output=$.queue[0];
        }
        else return(inx(BOOL,$.input));
}
```

Special Nets: Bin, Heap and Pile

Three special pseudo nets have been predefined in NeSeL. These are a bin, heap and pile. The bin net can only appear as a connection input path. Any connections sent to it are diverted and then deleted. The heap and pile are global and local stacks respectively. Adding connections to them is like pushing onto a stack, removing connections is equivalent to popping. They can be indexed into and copied from as well. For example,

```
heap = + X[...];
A[...] = heap[1,3];
```

See Reference section for more details.

Floating Declarations

A network is compiled by following a pseudo procedural reduction order across the net hierarchy. This means nets, and therefore connection specifications, are reduced in a set order. In some cases, particularly with the fork and divert connection operators, it’s useful to be able to change this order. Consider the halfadder net defined earlier. When connections are added to it after it’s been reduced a potential connection mistake can arise. Inputs are sent to the gate “and[1]”. Now, the “or” gate receives inputs by forking them from this gate. However, if the or’s connection declaration was reduced at the same time as the “halfadder” net, no connections would have been added to the “and[1]” gate yet. So none would be forked. The fork operation must be postponed until all input connections have been added to “halfadder”. The reduction order of a connection specification can be changed by floating it to a higher net.

When a net is reduced subnets are reduced first, then its connection specifications and finally any connection specifications floated to it from its subnets. Floatation is achieved by preceding the connection specification input path by one or two tilde characters (“~”). The declaration is then moved to the parental net indicated by the path reference qualifiers appearing in the path. Prefixing input paths by just one tilde character postpones evaluation until the float destination net evaluates floated declarations. If two tilde characters are used it's as though the floated path was declared in the destination net. This affects the reference net used to evaluate connection output paths. See Reference section for a summary of reference net dependencies and postponement times.

Lists

Lists are used extensively in paths when specifying connections and objects. These sorts of lists always have positive integer elements. The elements of lists used in initialisers or as arguments can contain other types. The simplest lists, those we've seen so far, just contain numbers or number ranges e.g. "[2,5..8] ≡ [2,5,6,7,8]". However, expressions, functions, selectors, variables and loop statements are allowed in lists. These are now described in more detail.

List Evaluation, Selectors and Partitions

Lists are accessed like a sequential stream and are evaluated on a demand basis. When sufficient elements have been determined for the current task evaluation is suspended to be resumed later if necessary. A list is said to be suspended, or only partially evaluated, if the outermost right-hand side square bracket has yet to be encountered. This means infinite lists can be defined as long as they are used by a terminating process.

Information on the current state of a list is maintained internally. Special functions, called selectors, can be used inside lists to access this information. A selector in one list can be used to access information on any other list, or itself, appearing in the input or output paths of a connection or variable declaration. Selectors start with the dollar character, a code indicating the particular selector function required, optional co-ordinates indicating the list to extract information from, and finally an optional application path. For example, "$^" returns the largest element in the current list, "$_" the lowest value, and "$|" the current number of elements in the list. There are selectors for returning the number of inputs and outputs from a specified point, the name of the path a selector's in, and so on. In all there's over a dozen selectors, some of which have a few subfunctions. A more easily remembered macro has been defined for each of the selectors, for example "s_size(x)" can be used instead of "$&x".

Four of the selectors apply specifically to the output path. These are "$!@", "$!#", "$!!" and "$!p". The first two return the absolute and relative number of the connection to be read next from the output stream. The third returns nonzero if the output stream contains more connections i.e. returns true if not end of stream. Finally, the fourth returns the current number of leaves in the output path evaluated to the specified partition level. Partitions haven't been dealt with yet, but they are simply a section of a path. A path may be divided into partitions, for subsequent use with the partition selector, by using a partition separator between the desired path members instead of the normal path separators. A partition can be either hard or soft, indicated by the pipe "$|" and double backslash "\\" or double forward slash "//" separators respectively. Soft partitions in label definitions are ignored if the label is used within another path. Hard partitions are never ignored. Partitions are extremely useful when connections need to be grouped. The append list operator, described later, is based on the partition selector. Finally, most selectors return minus one when applied to an unevaluated list, or when used incorrectly; some return zero, see Reference section.

Variables and Expressions

Standard C expressions are allowed as list elements. Variables with the C base types **double**, **int** etc., can be declared just as in C. For example,

```
[int num,bot=10,top=100; bot..top] ≡ [10..100]
```

The scope of a variable is the list it was declared in and any sublists it may contain. In the event of name clashes the inner declaration has precedence. In fact, a special form of restricted C is permitted inside lists. Think of a list as a list of C statements where most yield a value. In the above example the variable declaration doesn’t return a value, only the range does. Another example:

```
[int size; (size=$&1); 0..(size/10)]
```

Besides commas, semicolons are allowed as separators. Notice the selector assignment to “size”. If the selector equalled 200 the above would be equivalent to “[200,0..20]”. Sometimes we might want to ignore the value returned by a statement. This is achieved by prefixing one or more colons to the statement e.g. “[1,::2,3] ≡ [1,3]”.

List Operators

The list operators are a set of commonly used lists which are denoted in a concise way. List operators are similar to selectors in their notation, but are enclosed by square brackets. For example, the list operator “[&x]” is equivalent to “[0..(($&x)-1)]”, where “x” is a parameter and provided “$&x>0”. The first character in a list operator, “&” in the above example, selects a particular list. Three of the more useful ones are the append list “[+]”, data structure list “[.]” and the fit list “[?]”. The first enables connections to be appended to a net without having to worry about the size of the output stream, or any prior connections added to the net. When connections have been added and deleted to a node gaps may arise, some data structure array members may not have any inputs while their neighbours do. The second list (“[.]”) returns a list of indices of those members with inputs. Note NeSeL compresses the indices of unbound arrays at the end of compilation. The third list returns either the append list or data structure list depending upon whether the current connection specification was used with a minus operator. This allows many connection specifications to be applied to the same input path. For example:

```
A[?] = + X[5,6] - Y[0..3]
```

Apart from the dummy list “[*]”, all list operators can be used either directly or inside other lists. There are about a dozen list operators, see Reference section for more details.

Function

Standard functions, such as those in the maths library or user defined functions, are allowed in list expressions. NeSeL recognises a special type of function which returns a list. Any function returning a pointer to a basic type is treated as such a list function. Arguments to functions may themselves be functions or even a NeSeL list. For example:

```
[lcat([0..10],[11..20])] ≡ [0..20]
```

where “lcat( )” is a list function which appends it’s two list arguments. Typically, distribution functions would be used which return a list of connection indices distributed accordingly. For example, an on--centre off--surround distribution function, or ones to map between one and two dimensional co-ordinates (NeSeL only directly supports single dimensioned arrays). To distinguish a list function from a function which just returns a pointer type, the later must be explicitly cast where required, see Reference section.

Conditionals and Loops

Lists can contain **for**, **while** and **do** loops, plus break statements, as well as **if** and **else** statements. Their syntax and usage is the same as in C. For example:

```
[int num=0; while($!!) [if(num==5) break; num++] ]
```

which produces the list "[0,1,2,3,4]" while the output stream contains connections, or it aborts anyway when "num" equals five. Remember that list evaluation is suspended when enough elements have been produced for the current task, and only resumed later if necessary. Some selectors can cause suspension as well. In the example above the **while** loop is executed and then suspended. This is because a guard is placed on the "$!!" selector which forces suspension until the output stream is accessed. Notice that the list form of C used in lists differs only in that square brackets are used to group statements, rather than curly braces. Actually curly braces can be used, but the intention is considered clearer with square ones.

Statements and so forth can be used to build long lists. Normally the new elements to a list are added to the start, sometimes we may want to add them to the end. For example: "[int i; for(i=0;i<10;++i)[i]]" produces the list "[i]". What may be wanted is the list "[0..i]" which depends on how many times it's used. Two operators are available, namely "+" and "++", for specifying where elements are added to a list. Remembering the stream analogy, the default action is to add elements to the start of the stream; "+" means add to the base of the prior list, and "++" means add to the end of the stream. These operators are used by prefixing them to the list in question. For example:

```
[int i; for(i=0;i<10;++i)++[i]]
```

produces "[0..i]" which is what was wanted. A list operator might have been more appropriate in this particular case.

The NeSeL Programming Environment

NeSeL specifications can be compiled into standard C and consequently simulated. The NeSeL compiler produces a set of coupled functions which is equivalent to the original net. These functions can be executed as either a small or large model. For small models it is assumed that the whole net will fit into the memory of a uniprocessor. A very small simulator is then all that is needed to simulate the net. For large models the net may be spread across many processors and spooled from disk if it won't all fit into memory at once. In either case the NeSeL specification remains the same.

The differences in implementation are hidden from the user by adopting a standard set of functions when accessing and communicating between nodes. Thus, for small models we simply use the NeSeL small model library files, and the large model files for large models. These files chiefly define a set of routines for accessing another nodes data structure.

For the sake of clarity network specifications should be stored in files ending with a “.n” extension, and network header information in files ending with “.nh”. C specific stuff, such as function “main( )”, should be stored in a “.c” file of the same name. The following example net illustrates typical NeSeL header files (note their order is important):

```
#include "math.h"
#include "nsltype.h"
#include "nslstd.h"
#include "nslios.h"

node { int output, *input[ ]; } logic_gate = {0};

logic_gate not(output) in input; out output;
{       int i,output=0;
        for(i=0;i<$.size.input;++i) output|=ini($.input[i]);
        return(!output);
}

net auxnet(boot)
{ not(boot)[2] input[*]: [*]|input[*], output[*]:; }

net network( )
{ not(FALSE)[2] [*]|input[0]=<(auxnet(0))$[*]|output; }
```

The above would be stored in a file called “noddy.n” for example. The main C file, which calls the simulator and prints results, would be called “noddy.c” and contain the following, for example:

```
#include "stdio.h"
#include "nslstd.h"
#include "nslios.h"
#include "noddy.nsh"
#include "noddy.nsc"
#include "noddy.nsd"

void main( )
{       long xx; int nn;
        for(xx=0;xx<20;++xx)
        {       nsl_cycle(network,1L);
                for(nn=1;nn<=network->num_nodes;++nn)
                        printf("(#%d)=%d  ",nn,dsi(nn,logic_gate,output));
                printf("\n");
        }
}
```

Now consider the logic gate node function “not” defined in “noddy.n”. This is a general sort of “not” gate. It can have a number of inputs which are tied together by or’ing. The result is then inverted and stored in the nodes output variable. Notice that inputs are accessed via the function “ini(n)”. This fetches an integer from the node address specified by “n”. The NeSeL library file “nslios.h” includes a number of routines for fetching values from nodes. Notice, as well, that the data structure of the “not” gate was referenced with the special dollar symbol. The gates inputs were declared as an unbound array. The size of this array is determined at compile time, and provided the appropriate compiler option was set, this value will be available in a special struct added to the “not” gate data structure (in this case “$.size.input”). Finally, notice the return statement. The compiler actually converts this to “{$.output=!output;return;}”.

Library file “nslios.h” also declares the types and names of the various simulator and monitor routines. The library file “nsltype.h” defines the various types used by NeSeL, such as “NSLNSLnet”. Library file “nslstd.h” defines some useful macros, such as more easily remembered forms of the selectors and list operators. It also defines some common functions e.g. list catenating and sigmoids. The library files “digital.nh”, “analog.nh” and “nets.nh”, define some popular nodes and nets. For example, various logic gates and circuits, sigmoids, Hopfield nets, BEPs, ARTs etc. The “show.nh” library file defines a node called “show” which simply displays the value of any connections sent to it. This is useful when using the stand-alone monitor “nslw”. A standard set of graphics and i/o routines have been defined. These are accessible via library file “nslwsio.h”

The main file “noddy.c” simulates twenty cycles of the net by calling the simulator, defined in “nslios.h”. After each cycle it prints the output value of each node. Again a set of routines are supplied in “nslios.h” for accessing node data structures in a consistent way across models. In this case function “dsi(n,t,p)” is called to fetch the output value. Incidentally, the rate at which a particular node is called by the simulator can be set by specifying the rate in a nodes data structure. A count of the cycles remaining is maintained for a node, when this reaches zero the node is called. This too can be given an initial value, or changed dynamically by the node.

When a net is converted into C, information about it, such as its name, number of nodes, connections, data structure pointers, function pointers etc., is held in a structure of type “NSLNSLnet”. In the above example the default network was produced and used. Information on this network was held in a “NSLNSLnet” variable pointed to by “network”. A number of independent nets can be loaded at the same time if their information structures have different names. NeSeL provides a couple of ways of doing this. First of all the compiler essentially produces three files. The first, ending in “.nsh”, contains header information such as the types used by the net. One ending in “.nsc” contains C code derived from the “.n” net file. This is often fairly independent of the net specification itself. It contains such things as node function definitions and monitor information about structures and symbols. The third file produced, and ending in “.nso”, is a pseudo object file. It contains information on each of the nodes in a net and their data structures.

The main C file “noddy.c” contained three include files that are derived from “noddy.n”. These are “noddy.nsh”, “noddy.nsc” and “noddy.nsd”. The third is produced by the program “nsltox” which converts NeSeL object files into other standard forms. At present these are either C: “.nsd” or text: “.nst” forms. The text form is useful when debugging a net or comparing the compiled output with a network diagram. The “nsltox” program can be told to give a net a different name to the default (i.e. “network”). So, for example, we could convert a NeSeL object file to many nets with different names. These could then all be loaded and simulated at the same time.

The other way of loading and simulating multiple nets is with the NeSeL monitor functions. These are a set of routines for loading and storing nets. They work on NeSeL object files. A specific net can be loaded into memory and given its own NSLNSLnet structure (like file pointers in C). Similarly, nets can be dumped to an object file with a different name. Since nets are dumped in object format they can be converted to the text form with “nsltox” and so examined.

A net can be compiled with NeSeL and simulated straight away without having to use “nsltox” or a C compiler. This is only possible if a standard environment is established. The C routines and variables commonly used must be compiled and linked with the stand-alone monitor once only when it is initially created. The program “nslw” is a stand-alone monitor for loading, simulating and dumping nets in NeSeL object format, see Reference section.

Chapter 2

NeSeL Reference

Network Reduction Order

Reduction starts at the root net called “network”. Any subnets are descended in turn and reduced. When a net contains no subnets its object and additional connection elements are evaluated in turn. A list of floated elements may be built up. These are evaluated after non floated elements according to float order (FIFO queue). Once all elements have been evaluated reduction moves to the parental net, if any. The next subnet of the parent is likewise reduced and so reduction continues. Note a subnet may float an element to a parental nets float queue.

Data structure initialisation starts after net reduction and proceeds in a similar fashion (but no floating is allowed). Note, this means node struct initialisers are evaluated first, then node function initialisers, followed by subnet initialisers, and finally the root net initialisers. To summarise, reduction order is determined according to depth first, widthwise descent from the root net to its leaf nodes.

Node data structures

A new node type is introduced in a similar way to **typedef** structs. Their declaration is similar to normal structures:

```
node { ds_member_list } node_type_list ;
```

Think of the word “node” being equivalent to “**typedef struct**”.

Unbound arrays are permitted on subnode structures or members used in a terminal sense i.e. no reference made to their submembers if they contain any. For example, two unbound array members:

```
int *member_name[ ];
node_struct_type member_name[ ];
```

Unbound arrays are converted to pointer types.

Node types can be initialised when they are defined. The syntax is as for normal struct initialisers, the new type name is treated as a pseudo variable:

```
node_type_list: new_node_spec | new_node_spec , node_type_list
new_node_spec: node_type_name [[ = { var_decls } ]]
```

Node struct initialiser declarations can be qualified by member names:

```
var_decls: var_decl | var_decl , var_decls
var_decl: [[ [[ wild_path = ]] [[ (type_cast) ]] var_exp ]]
```

```
var_exp: exp | STRING | IDENT | FIDENT | { var_decls } | sslist
```

Qualified initialisers can appear in any order and do not affect the normal assignment order of unnamed initialisers:

```
node { int red,blue,sara,vince;} gate = {1,vince=3,2,4,sara=5}→{1,2,5,3}
```

Note that a "var_decl" can be nothing in which case the obligatory comma is used for skipping a struct member i.e. ",,". Arrays can be initialised by specifying just one value if necessary. This is repeated over the array members. Casts are also allowed on arrays and structs to solve any ambiguities that may arise between the two: "(int [ ]){1,2}" and "(gate){1,2,3,4}". Also, casts to node types need only specify the node type name i.e. "(gate)" instead of "(struct gate)". Lists are allowed as initialisers. *Important note: list functions MUST be enclosed in square brackets when used as an initialiser.* A special cast: "(type [ ])", when used on a list means take the next list value for the current variable and then skip to the next initialiser. This kind of cast is used when, say, a number of structs need to be initialised with one struct member taking a value from the list and the others taking fixed values:

```
node {int red,blue,green;} gate;
gate and( ){...}
net network( ) { and[10]; }
={and[0..9]={40,(int [ ])[50..59],60} }
```

Finally, inputs must be a pointer to the type of value being input e.g.

```
int output; → int *input;.
```

If variables of the node type are required they are declared using the "node_type_name" in the same manner as a typedef:

```
node {int red,blue,green;} gate;
…
gate colour;
…
```

Node Functions

These functions are similar to normal C functions. They must be introduced by a node struct type:

```
node_type_name  node_func_name( [[ node_parms ]] )
        [[ comm_decls ]]
{...function body...} [[ = { var_decls } ; ]]
```

The optional "node_parms" indicates default data structure members to initialise when the node function is used with arguments by a net. Note an argument expression may be qualified by a name which overrides the default parameters. The "comm_decls" specifies the external visibility (scope) of data structure members:

```
comm_decls: comm_elm | comm_decls comm_elm
comm_elm: {{ in | out | inout }} comm_list ;
comm_list: wild_id | comm_list , wild_id
```

where "in" means "wild_id" is for input only etc.; by default all members are in scope.

The function body is normal C statements. The special dollar character "$" is used to refer to a nodes data structure. For example: "$.state=6;". If the node data structure contains a member called "output" in its root node struct, the statement "return(x)" can be used as a shorthand way of setting it. Finally, the optional initialising section "var_decls" has the same meaning and syntax as in data structure declarations. Again empty members can be used for spacing. Function initialisers take priority over node struct initialisers. A node function can specify label relative inputs by prefixing two dollar characters and an optional label to the input variables name, see section on label relative connections.

Outline of how NeSeL converts node functions:

```
node_type func_name(parms)        void func_name(NSLNSLnet *net,node_type *node)
        comm_list
{       ... $.id_list           →  {      node->id_list
        ... user_func($) …      →         ... user_func(node) ...
        ... return(x) ...       →         ... {node->output=(x);return;} ...
} = { var_decls }               }
```

During simulation node functions are called at a user determined rate, see section on compiler options. Reference to external variables and files is permitted but the outcome may not be as expected in large models. Internal static variables should be used with caution, a node function may be called by many different nodes.

Macros

In addition to the C pre-processors **#define** macro directive a more general form is permitted. These are declared as follows:

```
macro mac_name([[mac_parms]])
{ body }
```

Expansion conforms to the ANSI C standard with anything appearing inside the curly braces replacing the original macro call. The standard features are: two hashes "##" before or after a parameter in the macro text cause white space to be removed between the substituted parameter and text; a single hash before a parameter in the text causes the substitution to be stringified; and a single hash before a macro argument causes the initial parameter to be destringified. In addition, a percent character "%" before a macro argument causes the parameter to be evaluated before substitution. Conditional expansion is supported inside the braces. The syntax is the same as for the conditional directives **if**, **elif**, **else** and **endif** except that "$" replaces "#". Any level of embedded conditional directives is allowed. Finally, a form of variable arguments is allowed. If more arguments than actual parameters are given during a macro call the excess arguments are assigned to the last parameter. Note macro expansion occurs when the macro is encountered during net reduction. Whereas, the **#define** directives are expanded by the C pre-processor whose output is then passed to NeSeL at the start of compilation.

Nets

A net declaration has the form:

```
net net_name([[mac_parms]]
        [[comm_decls]]
{ [[net_decls]] } [[ = { var_decls } [[;]] ]]
```

Nets are macros, and expansion extends from the start of the "comm_decls" to the end of the "var_decls" section. The optional "comm_delcs" and "var_decls" have the same syntax and meaning as in node functions. The body of a net is a list of “net_elms”. A null body “{}” is only allowed in auxiliary nets.

```
net_decls: net_elm | net_decls net_elm
net_elm: obj_decl ; | add_decl ;
```

Object Declarations

An object declaration consists of an object specifier followed by an optional list of object elements:

```
obj_decl: net_or_node [[ [exp] ]] [[obj_elms]]
net_or_node: net_name(mac_args) | node_name(var_decls)
```

An array of objects can be declared. obj_elms is a list of comma separated object elements:

```
obj_elms: obj_lab_path [[ = conx_list ]]
obj_lab_path: label obj_path | label | obj_path | vir_innet
```

The “obj_path” is a label definition path or an input path (stream) for connections. It is relative to the object unless overridden with path reference qualifiers.

```
obj_path: bas_path | tildes in_path
bas_path: gen_path | dol_path | dec_path | dodo_path
```

The “gen_path” is a general path, it may be wild and use path reference qualifiers. If qualifiers are used (i.e. “\”, “..\”, “.\”) the object element is moved to the specified parental net and evaluated immediately. A “dol_path” starts with the dollar character, which takes on the name of the current object. It too is relative to the object. The dollar symbol may be followed by a list, for object indexing, and/or a wild path. A “dec_path” is identical to a “dol_path” except that the dollar is left out and a list is always the first element. Note non arrayed objects are treated as arrays of size one, so lists can be applied to them. A “dodo_path” is a label relative path, see section on these.

Floated Paths

Floated input paths are specified by one or two tilde characters “~” prefixed to an “in_path”:

```
in_path: [[abssep]] flo_path | ref_path [[abssep flo_path]]
flo_path: wild_path | dol_path | dec_path
```

Floated object elements are placed in the float queue of the specified net to be evaluated at the appropriate time. One tilde means float the declaration to the specified net and evaluate it in due course but using the original net for reference. Two tildes means move the declaration to the specified net and evaluate it in due course from that net as though it was originally declared in that net. The table below summarises the actions of local, qualified and floated “obj_paths” on the reference net used for evaluation and the time of evaluation:

Path prefix & Eval time	Ref used for Input path				Ref used to eval Output path			
	(c|*|?)	(.\)	(..\)	(\)	(c|*|?)	(.\)	(..\)	(\)
none:	O	F	F	R	D	D	D	R
immediately	O	D	D	D	—	—	—	—
1 tilde (~):	D	D	D	D	D	D	D	R
postponed to:	O	D	F	R	—	—	—	—
2 tilde (~~):	D	D	F	R	F	D	F	R
postponed to:	D	D	F	R	—	—	—	—
O = object element was declared in, or net if "add_decl" D = net element was declared in F = parental net element was moved to or floated to R = root net								

Note ".\..\" etc. is permitted. Since ".\" always refers to the element declaration net, ".\.." can be used to move to parental nets after floatation.

Labels

Labels serve as a shorthand referencing mechanism and as a means of presenting a uniform indexing range over different objects in a net.

label: IDENT : | IDENT list :

When used a label is equivalent to its defining "obj_path". Note the later may be null in which case it refers to the object. If a label is declared with a list, called the label list, its elements are mapped by position to the first leftmost list in the "obj_path". A default path member is added to the end of the "obj_path" when required, i.e. "input[*]" or "output[*]" depending on context. A number of labels with label lists and the same name can be defined many times in a net over the same object and different objects. Subsequent use of the label will select those definitions which include the required index. Multiple occurrences are each taken in turn. When a label is used at some stage in a path it may be post-fixed with a list. In which case the list is mapped, by index, to either the label list or the leftmost list in the "obj_path" as before. Multiple mappings may occur in paths that refer to labels that in turn refer to labels and so on. To clarify this see below:

A**\B ≡ A\B, A*\B, A**\B, A*\...*\B etc.

Uncontrolled use, i.e. using it by itself, should be avoided in large networks to be compiled. However, in specifications it's useful: "** = --~~", this declaration deletes all self referencing connections from a network.

Qualifiers

These are the path prefixes: ".\", "..\", "\", with the same meaning as in UNIX. However, in declarations ".\" always refers to the elements declaration net. NeSeL allows this qualifier to also be followed by a path starting with a number of parent qualifiers "..\". This enables floated elements to access the original declaration net and its parents. For example: ".\..\..\A\B\C".

Partitions

These are special path separators specified in label definitions or normal paths. They section a path for use with the partition selector. There are two sorts:

1. ["|"] hard partition, always obeyed;
2. ["\\" or "//"] soft partition, ignored when label is used in another path.

The partitions in a path are numbered from left to right starting from 1 i.e. cardinal co-ordinates. However, note the partition leaf selector uses relative co-ordinates by default, although this can be overridden with the cardinal indicator. Partitions are often used in conjunction with the append list operator to section the input and output paths.

Connection Lists

The optional connection list part of an object element is:

```
conx_list: conx_elm | conx_elm conx_list
conx_elm: [[cop_list]] [[(aux_obj)]]
            out_path [[ { var_decls } ]]
out_path: bas_path | dodo_path
            | tildes [[sel_path]] | vir_outnet [[list]]
```

This allows a list (of lists) of connections to be added to the object input path ("obj_path"). Note that in output paths tildes refer to the current input path. Two tildes returns the node path, and a single tilde returns the current input connection path. The input and output paths are treated as streams. The input demands connections from the output stream (demand driven). During stream evaluation non-terminating paths have default members added ("input[*]" or "output[*]" depending on context) until they reach a terminal data structure member. For selector operations applicable to connection streams see section on selectors.

A connection is a unidirectional link from a terminal data structure member of the output nodes data structure to a pointer type member in the input node. For instance, if output type equals "t", the input type must equal "t *". NeSeL represents connections by storing a symbolic output address in the input node. Library routines then fetch the required output from a node when called by an input, which provides a connection address.

Connection Operators

Connections are combined with the input according to the specified connection operators ("cop_list"):

1. **+** Add connections to input. The default operation, does not overwrite.
2. **-** Subtract connection from input, which must match the connection address currently stored in the input.
3. **++** Add connection and overwrite any existing connection.
4. **--** Subtract connection from wherever it occurs in the base data structure array referenced by the current input position.
5. **>** Fork connections to input. The "out_path" is actually treated as an input path. The connections so found are copied to the input stream specified by "obj_path".
6. **^** Divert connections. Same as fork operator but doesn't copy connection.
7. **<** Merge connections through an auxiliary net. Only used in conjunction with an auxiliary net.

As an example of the merge operator, consider:

```
A[0..2]\B[+] = X[0..2] + Y[3..5] + Z[2,4];
```

In the above all the output elements are added in order to "A[0]\B[+]", then the whole group is rescanned and the elements added to "A[1]\B[+]". However, if we wanted to add each output element to all of the input stream in turn, then it could either be specified as:

```
A[0..2]\B[+] = X[0..2], A[0..2]\B[+] = Y[3..5], A[0..2]\B[+] = Z[2,4];
```

Or more succinctly using the group break operator:

```
A[0..2]/B[+] = X[0..2] |+ Y[3..5] |+ Z[2,4];
```

Auxiliary Nets

These appear in connection declarations and act as filters between outputs and inputs. Reduction takes place upon the first request for a connection from the input stream of the connection element containing the auxiliary net. During reduction the present net state is saved and the auxiliary net treated as though it was the root net. Requests for inputs by the auxiliary net are handled by restoring the saved net state and fetching a connection from the output stream of the original connection element. This is achieved by adding a special "add_decl" to the end of the auxiliary nets float queue, specifically:

```
(aux float queue) input[+] = (saved connection elements "out_path").
```

Reduction then returns to the auxiliary net until it is fully reduced. The original declaration's input then fetches connections from the reduced auxiliary net by evaluating the output stream "output[.]" on it. Note, to communicate with an auxiliary net it is assumed to contain objects or labels with the names "input" and "output" in the root net.

Variables of the auxiliary net can be initialised either by passing as arguments or in the normal manner in the auxiliary's "var_decls" sections. Note the global heap can be used to pass or return additional connections to and from auxiliary nets. For example:

```
Inpath = < (auxnet(var_decls)) Outpath.
```

Note connections can only be added to or received from auxiliary nets.

Label Relative Paths & Global Connections

When a path is prefixed by two dollar characters and an optional label it is evaluated relative to the label, which is set to "global" by default.

```
dodo_path:  $$ [[IDENT :]] wild_path
```

This label ("IDENT") is searched for starting from the original declaration path position and ending with the root net "network". When found label expansion takes place and evaluation proceeds, for example:

```
Inpath = $$disable:busy,
$$store:members[+] = Outpath.
```

This facility is primarily intended for initialising global variables in a net. For example, suppose all the transfer functions of a net have the same slope specified by a variable of type float. If two or more instances of this net are required with differing slopes then a slope variable would be needed for each. These global variables are accommodated by declaring them in a node structure. Then a node must then be declared of the same node structure type. Any node function that accesses these variables must do so via an input connection from this node. Accordingly it must have an appropriate input variable in its data structure. Node functions can specify label relative connections as opposed to normal connection inputs by prefixing the input variable with two dollar characters and an optional label:

```
node
{       float slope,gain,offset; int number;
}       my_globals;

node
{       BOOL output,*input[ ]; float *slope;
}       and_gate;
```

```
and_gate and(output)
{       int nn=$.size.input;
        float mm=inf($$my_lab:.slope),gg=inf($$slope);
        while(nn--) {if(!inx(BOOL,$.input[nn])) return(FALSE);}
        return(TRUE);
}

my_globals glob( ){}
net network( ){ and( ); glob( ) my_lab:, global:;}
```

Here the “and” gate accesses the global variable “slope” stored in the “glob” node. The optional labelling facility is also demonstrated although in this case both labels point to the same node. Note that in node functions the connection “in_path” is also used as the “out_path” (after prefixing with the label), so the input and output variable names must be the same.

Inline Variable Initialisers

A struct initialiser can be post-fixed to a connection “out_path”. It is then used to initialise the struct containing the input connection variable:

```
node {int red,blue,*input;} gate;
gate and( ){...}
net network( )
{       and[10]
        [*]=[-.]{$#,[0..9]};
}
```

When using LOPs and selectors, such as “$#” above, the current path is the input path positioned at the structures parent net. Remember to use an alignment comma if necessary in the struct initialiser to step over the connection input variable.

Additional declarations

Nets can contain additional declarations. These are like an “obj_elm” except there is no reference object:

```
add_decl: add_lab_path | add_lab_path = conx_list
add_lab_path: label add_path | label | add_path
add_path: [[tildes]] gen_path | vir_innet | dodo_path
```

For example:

```
net adder( )
{       and(FALSE) input = bus[1]; /* an object declaration & element */
        and(FALSE) my_and: ;       /* an object & label */
        my_and\input = bus[1]; /* additional declaration using label */
}
```

Labelled additional declarations are useful for linking inputs and outputs to other nets:

```
net pop_macro( )
{       delay_node( ) input[*]:;
        recognizer( ) output[*]:, input[+]=delay_node+\sym_stack;
        input[*]: \sym_stack[*]; /* labelled additional declaration */
}
```

When the user of this net adds connections to it they are added to the “delay_node” and the structure referenced by the “sym_stk” residing in the root net. This in turn may be another label etc.

Special nets: Bin, Heap & Pile

These nets are grouped as follows:

```
vir_innet: bin | heap | pile
vir_outnet: heap | pile
```

The “vir_innet” nets can be used as input “obj_paths” or “add_paths”, while the “vir_outnet” nets can be used as “out_paths” as well.

The Bin

The bin is a pseudo net into which connections can be diverted and so deleted. Connection operators are forced to the divert operator. For example:

```
bin = fred[*] + sam[1,3,10]
```

The output streams are expanded as though they were input paths. Deleting via the bin saves having to know the source of a connection.

The Heap and Pile

These are really stacks. The heap is a global stack accessible by all nets, even auxiliary nets, onto which connections can be pushed and popped. The pile stack is local to a net and only exists while the net is being reduced. Connections can only be pushed onto the top of these stacks, but the output path may be indexed so this isn’t much of a restriction. Connections can be popped off in any order via indexing:

```
heap = fred[1,6,2]
sam[1,2,4] = heap[3,1,0]
```

The top connection in a stack is the last one pushed. Its index number is zero, the next is one, and so on. Note, however, that indexing is relative and a connections index number changes as soon as another connection is pushed or a connection with a lower numbered index is popped. Function “smap(list)”, defined in the NeSeL library, maps absolute index values to relative ones. For example:

```
heap = fred[0,1,2]                /* fred = {a,b,c} → heap = {c,b,a} */
sam[0,1,2] = heap[0,1,2]          /* sam = {c,a,b} */
sam[0,1,2] = heap[smap(0,1,2)]    /* sam = {c,b,a} */
```

The heap and pile can only be indexed into as output streams.

Lists

A list can be a list operator or a list of elements:

```
list: list_sel | [ lelms ]
lelm: list_sel | range | loop_statement | break_statement | id_decls
range: exp | skips exp | exp .. exp
loop_statement: for ( [[exp]] ; [[exp]] ; [[exp]] ) loop_list
        | while ( exp ) loop_list
        | do loop_list while ( exp )
        | if ( exp ) if_list
if_list: loop_list [[ else loop_list ]]
loop_list: [[ loop_apnd ]] loop_ary | break_statement
loop_ary: list | loop_statement | range | { lelms }
```

Note that “exp” includes functions that return lists. “skips” is a string of one or more colons signifying that the following expression is not to be included in the list. Two colons must be used at the start of a list in order to avoid a conflict with the “[:x]” list operator.

Not all loop elements return a value, those that don't are "id_decls", "break_statements" and any element preceded by a colon. "id_decls" is a list of variable declarations:

```
id_decls: {{ double | float | long | int | char }} id_list ;
id_list: id_elm | id_elm , id_list
id_elm: [[ * ]] IDENT [[ = exp ]]
```

The recognised "break_statements" are: **break**, **continue** and **return**. The return value of "loop_statements" are "loop_lists". The position at which these lists are added to the enclosing list can be specified with a "loop_apnd" operator ("+" or "++"). The default is to add the list to the beginning of the parent list. A single "+" adds to the current base which is the list position before the "loop_statement" was evaluated, and a double "+" adds to the end of the list.

List evaluation

Lists are evaluated on a demand basis. When sufficient elements have been obtained for the current task evaluation is suspended. When the rightmost outermost square bracket is encountered evaluation terminates. Break statements can be used to force various degrees of termination. Selectors and list operators have guards which cause evaluation to be suspended if they are used again before the output stream is accessed. A separate guard is applied to each instance of a particular selector. For example:

```
input[ int i=0; while($!!) i++ ] = output[0..10]
```

When the input stream is accessed and the list evaluated it returns "[0]". The **while** suspends until a connection is fetched from the output stream. Evaluation resumes on the input stream returning "[1]". This continues until the end of the output stream is reached causing the **while** statement to break. Evaluation then terminates when the outermost rightmost square bracket is reached.

Selectors

Selectors are used in lists to access information about lists in the current input and output paths. The syntax of a selector is:

```
select: $ selcode selnum | $! partnum | $!! | $!@ | $!#
selcode: & | "|" | ^ | _ | % | : | @ | # | * | + | $-$
partnum: exp | "|" exp
selnum: [[selrel]] [[exp]]
selrel: selcoor | ds_op | ds_op selcoor
selcoor: @ | "|" | ? | ! | @"|" | @? | @!
ds_op: "."
```

The "selcode" indicates the selector function, and "selnum" the list to apply the function to. For the co-ordinate scheme used to specify a path list see the next section. In the following list of selectors, the parameter "x" is:

```
x: [[selnum]] [[app_path]].
```

The value of some selectors is modified by the "data structure" co-ordinate indicator: "ds_op". The macro form of a selector is also given:

1. **$&x ≡ s_size(x)** Fixed size of array; with "ds_op": size of data structure array. Note unbound arrays return zero, fixed arrays return (maximum element index+1).
2. **$|x ≡ s_num(x)** Current number of elements in list; with "ds_op": current number of elements in data structure array, or zero if applied to net.

3. **$^ x ≡ s_hi(x)** Largest value in list; with "ds_op": largest data structure array value, or (-1) if applied to net.
4. **$_x ≡ s_lo(x)** Lowest value in list; with "ds_op": lowest data structure array value, or zero if applied to net.
5. **$%x ≡ s_in(x)** Number of net or struct subcomponents associated with net, node or data structure member; with "ds_op": number of input connections to branch so far.
6. **$:x ≡ s_out(x)** Number of net or struct subcomponents associated with net, node or data structure member; with "ds_op": number of output connections from branch so far.
7. **$.x ≡ s_node(x)** Returns the current node number (0 if not at a node yet).
8. **$@x ≡ s_pos(x)** Position of current element in list (-1 if not ref'd yet).
9. **$#x ≡ s_val(x)** Value of current element (-1 if not ref"d yet).
10. **$!@ ≡ s_abs( )**
11. **$!# ≡ s_rel( )** Returns absolute or relative number of connection to be read next. The first selector counts the connections produced from the current "conx_list". The second counts the number from the current "conx_elm". These two selectors may be assigned values.
12. **$!!≡ s_eoc( )** Returns non zero if not end of output stream i.e. more connections exist in output stream.
13. **$!p ≡ s_leaf(p)** Returns number of leaves in next "conx_elm" evaluated to specified partition level, (p is a relative or cardinal partnum co-ordinate).
14. **$* x ≡ s_dip(x)** Returns path string in (declaration number:index number) pair format, note caller should copy before using path string selectors again.
15. **$+x ≡ s_path(x)** Returns path string in symbolic format.
16. **$-x ≡ s_type(x)** Returns path string in type format.

Path Representation and Selector Co-ordinates

A path has a relative and absolute representation. Relative paths refer to the original form of a path used in it's object or additional declaration. For example:

```
object( ) A\B\C\D = P\Q\R;
```

where "A\B\C\D" and "P\Q\R" are relative paths. When relative paths are evaluated they are expanded to absolute paths by substituting all labels for their definition. For example:

```
A\B\C\D → a1\a2\a3\b1\c1\c2\d1\d3
```

A selector in the list of a path member can access other path member list information by specifying a co-ordinate scheme and a co-ordinate. The permissible co-ordinate indicators are:

1. **@ x ≡ c_abs(x)** Use absolute path.
2. **| x ≡ c_card(x)** Use cardinal co-ordinates.
3. **? x ≡ c_in(x)** Apply to input path using cardinal co-ordinates.
4. **! x ≡ c_out(x)** Apply to output path using cardinal co-ordinates.
5. **. x ≡ c_base(x)** Take data structure array of base to specified path member.

A co-ordinate indicator is optionally followed by a number indicating the path member list to access. This number depends on the addressing mode used. There are two modes: relative and cardinal co-ordinates. These are illustrated below:

Cardinal:	$-4	$-3	$-2	$-1	$0
or:	$1	$2	$3	$4	$5
Ref path:	\aaa	\bbb	\ccc	\ddd	\eee
			↑		
Relative:	$-1	$0	$1	$2	$3

where path member "ccc" is the current path position. *By default the current relative path is used with relative co-ordinates*. When selectors are used in variable initialisers the current path and position is the input path positioned at the variable (right-hand of path). If the input path is accessed via a co-ordinate indicator, say the "?", in label-mode the current position is the original declaration net. If the input path is accessed in absolute-mode the current position is the root net. For example:

```
net test( )
{       bep( )
                error[l_wild( )] = ctrl_error[l_size(c_in(X))];
}
```

Parameter "X" is a cardinal co-ordinate referring to the input path:

Cardinal:	-1	0
or:	1	2
Input path:	\bep	\error[*]

An application path can follow a selector if another path besides the current input or output path is required as reference:

```
app_path: $$ [[IDENT :]] app_member | $ app_abs
        | $ tildes app_abs  $! app_stem | $? app_stem
```

They start with a dollar character and then either the required path, or another dollar for label relative paths, or a question-mark or an exclamation-mark for selecting the input or output path respectively. For example:

```
A\B\C[$&x$?bus\data] ≡ A\B\C[a_in(s_size(x),bus\data)]
```

Here application path "$?bus\data" causes the selector to be applied as though "cur_out_path\bus\data[s_size(x)]" had appeared. The equivalent form shows the use of the macro:

1. **a_label(s,p) ≡ s $$ p**
2. **a_app(s,p) ≡ s $ p**
3. **a_out(s,p) ≡ s $! p**
4. **a_in(s,p) ≡ s $? p**

Linked Lists and the Data Structure Indicator

A list may be linked to others for mapping purposes when a label path is defined. A chain of linked lists may arise when the path member belonging to the leftmost list in the label definition path is itself a label. The fixed size of a linked list is set to the current number of elements in its child linked list. The size of the base child list is either zero, if it's an unbound array, or the declared array size (see "s_num(x)" and "s_size(x)" selectors). The data structure indicator selects the base list in a chain.

List Operators

The list operators provide a shorthand notation for common lists. Most could be specified as macros, however, a few could not. The syntax of a list operator is:

list_sel: [[[-]] list_opr selnum]
list_opr: & | "|" | ^ | _ | % | : | = | . | ! | + | @ | ?

In the following list of list operators, the optional parameter "x" is a "selnum". Some operators include co-ordinate indicators, if parameter "x" contains indicators as well, they are combined with the former. The macro form of the list operator is also given:

1. **[*]** ≡ **[l_wild( )]** wild list. A dummy list used in label definitions to indicate when a list is expected as a parameter.
2. **[&x]** ≡ **[l_size(x)]** Returns [0..(s_size(x)-1)] if (s_size(x)-1)>0, else [0..0].
3. **[|x]** ≡ **[l_num(x)]** Returns [0..(s_num(x)-1)] if (s_num(x)-1)>0, else [0..0].
4. **[^x]** ≡ **[l_hilo(x)]** Returns [s_hi(x)..s_lo(x)].
5. **[_x]** ≡ **[l_lohi(x)]** Returns [s_lo(x)..s_hi(x)].
6. **[%x]** ≡ **[l_in(x)]** Returns [0..(s_in(x)-1)] if (s_in(x)-1)>0, else [0..0].
7. **[:x]** ≡ **[l_out(x)]** Returns [0..(s_out(x)-1)] if (s_out(x)-1)>0, else [0..0].
8. **[=x]** ≡ **[l_eval(x)]** Takes corresponding list string, evaluates it and returns list as result.
9. **[.x]** ≡ **[l_ds(x)]** Returns data structure array if selected path member is in a node data structure, otherwise returns [l_size(x)].
10. **[!x]** ≡ **[l_copy(x)]** Takes corresponding specified list as result.
11. **[+x]** ≡ **[l_add(x)]** Returns [(1+s_hi(c_base(x))..(1+s_hi(c_base(x))+s_leaf(-1))]. Append connections in output stream to current input stream. Note this operator is re-evaluated for each "conx_elm" in the "conx_list" in order to determine stream size.
12. **[@x]** ≡ **[l_app(x)]** Returns [(1+s_hi(c_base(x))..(2+s_hi(c_base(x))]. Append one element to current input stream. Does not look at output stream to determine size. Useful for adding to unbound arrays one element at a time in cases where the output stream contains elements to be added to a sub-node data-structure.
13. **[?x]** ≡ **[l_fit(x)]** If "conx_elm" was used with a minus operator the [l_ds(x)] list is returned, otherwise the [l_add(x)] list is returned. Again, this operator is re-evaluated for each "conx_elm".

The "list_opr" code may be prefixed by a minus character, in which case the reverse list is returned.

User list functions

Here are some implementation details for defining user functions that take or return lists as arguments. Two kinds of lists are recognised. Pair-lists in which each element is stored as a pair of numbers specifying the start and end of a subrange, and user-lists where each element is stored as a single number and there are no subranges. Evaluated pair-lists are internally stored as an array of integers (of type "mile"). For example:

[6] → {6,6}
[10..6] → {10,6}
[6..10,0] → {6,10,0,0}

Note [6..6] ≡ [6], but the former is treated as a list when determining the leftmost list in a label or object path. The type of a user list can be specified by preceding the list with a cast. Non-cast lists are treated as pair-lists. For example:

```
    (double)[6.12] → {6.12}
    (double)[10..6] → {10.0,6.0}
```

If a cast list contains subranges it is treated as a pair-list of the cast type. When used as an argument to another function, user functions of type "pointer to" are assumed to return a list. If the intention was actually to return a non list pointer the function must be cast accordingly:

```
    func_take_list(lcat([a..b],[c..d]))
    func_take_pt((char *)lcat([1..10],[6]))
```

In the later function, the return value of "lcat( )} is treated as a non list pointer. The reason for requiring an explicit cast is because the NeSeL compiler frees the space used by a list after use. Clearly, attempting to free a non list pointer may cause problems. Space for lists is claimed from memory using the C "malloc( )" function via the user visible functions "lmore( )" and "lnew( )". When a list is freed the user visible function "lfree(list)" is called.

The first few bytes of the list space is reserved for list details (equivalent to five ints of type "mile"). Specifically: current number of elements in list, maximum number of elements allowed by space, the size of an element in storage units, the type code of element, and a check value. A list pointer points to the first element byte after the list details. Library file "nslstd.h" defines the type codes and some useful macros and functions for accessing and creating lists.

Pragmas

Pragmas are compiler directives that occur in the source text. They are much like the "#pragma" directives of ANSI standard C, but without the "#". The pragmas recognised by the NeSeL compiler are:

1. **pragma variable** ***expression*** Depending on the value of the constant expression this pragma is used for turning variable extraction on (expression > 0), off (expression = 0) or restoring it to the prior state (expression < 0), see "nsl1" "-v" option. When switching extraction on or off the prior state is saved on a stack first. So the user should conclude any switch with a restore.
2. **pragma header** ***expression*** Similarly, this pragma turns header information extraction on, off or restores it to the prior state, see "nsl1-h" option.

Reserved Words

In addition to the normal C reserved words the following are also reserved: **macro**, **pragma**, **net**, **node**, **in**, **ins**, **out**, **outs**, **size**, **total**, **NSL***, **nsl***.

Also be careful not to redefine any of the functions or macros defined in the NeSeL library files.

The NeSeL Compiler: nsl1 and nsl2

The NeSeL compiler is split into two parts. Program “nsl1” performs syntax checking and converts the net specification into an internal intermediate form. Program “nsl2” then translates this into a set of functions and data structures, which represent the net, and stores this in a pseudo object format.

nsl1

Command line: **nsl1** *options* inname

Options:

1. **-a** Add all additional information. Same as **-bcgr**.
2. **-b** Add node information to root node struct of data structure. A struct called “node” and of type “NSLNSLid” is added to any node struct that is used as a root node struct. It indicates the node number, as a **long** type member called “name”, and its “dip” path, as a **char*** string member called “path”. Note, if this option is used it is advisable not to use root node structs as substructs.
3. **-c** Add connection size information to data structures. A struct called “size” and of type “NSLparent_type_name” is added to all structures containing members with unbound arrays. For each unbound array a similarly named member of type **int** appears in the “size” struct. The members of the “size” struct are initialised to the size of their respective unbound arrays by nsl2 and can subsequently be accessed by node functions. A member, called “total”, is also added and is the sum of the unbound array sizes. Two other structs are also added. These are called “ins” and “outs” and of the same type as “size”. They indicate the number of inputs and outputs, respectively, to the branches of the data structure with unbound array members as their root. The sum stored in member “total” includes, as well, any connections from regular bound array members and so forth.
4. **-f** Produce user function details for nsl2. Stored in “nslusrfn.c”. Use this option when setting up a new set of user functions for use inside lists. The “nslusrfn.c” file should then be linked with nsl2.
5. **-g** Add connection group information to input data structures. A struct called “group” and of type “NSLNSLid” is added to all structures with a member called “input” that is used for input. The group struct indicates the node number and “dip” path of the connection source.
6. **-h** Extract C header information from input and write to “outname.nsh”. Use this option when the input contains C details that are needed by later files and that isn’t specified in header files.
7. **-l** Log errors to “outname.log”.
8. **-m** Produce structure information for monitor. Added to “outname.nsc”. This information is used to access node data structures by the monitor routines for loading and saving a net.
9. **-n *netname*** Give net this name. Cause nets details to be stored in a unique net structure. Use when multiple net files are to be loaded. If symbol producing options are specified and this option is not used a nets symbol details will be stored in the default net structures.

10. **-o** ***outname*** Set output name. Chops off any extension, then adds set extensions.
11. **-p** Make “outname.nsc” acceptable to the C pre-processor again.
12. **-q** Leave out “#line...” information in output.
13. **-r** Add timing information to root node struct of data structures. A struct called “time” and of type “NSLNSLtime” is added to all root node structs. This struct contains **int** type members called “rate” and “wait”. Member “rate” specifies the number of cycles between calls to the node function. Member “wait” is a count down to the next call. Note, a zero rate disables the node. These members are normally set during compilation (by “var_decls”), but can be set dynamically.
14. **-s** Produce user symbol information for monitor. Added to “outname.nsc”. If function names etc. are passed as arguments or assigned to data structure members this option should be used. It enables the monitor routine for saving nets to convert pointers to symbolic form again.
15. **-U** ***name*** Undefine name i.e. skip its declaration.
16. **-v** Extract information on any global C variables declared. This permits variable values to be accessed symbolically during runtime via the monitor routines. See “nsl_write( )” and “nsl_read( )”.
17. **-x** Extract node function information for the stand-alone monitor. Stored in “nslmonfn.c”. Note when this option is used the main network in the inname net file is suppressed. Use this option when setting up a new set of node functions. The “nslmonfn.c” file should then be linked with nslw.
18. **-y** Add null monitor definitions to “.nsc” file. If the monitor file “nslmonfn.c” is not included in the final linking process and other monitor files are, then this option should be used.

Note:
Input to “nsl1” should be a net specification file that has been run through the C pre-processor and given an extension “.nsp”.

Files:
(input) inname A net specification file that’s been run through cpp.
(output) outname.log Log file of errors.
outname.nss Resource file, intermediate form of net.
outname.nsc C specific details on net.
nslusrfn.c User defined functions etc. extracted from input.
(temporary) outname.nsz.

nsl2

Command line: **nsl2** *options* inname

Options:

1. **-e** Switch off error filtering. Normally related errors are not reported.
2. **-j** Turn on a flashing star which indicates that compiler is active and not stuck in a loop etc.
3. **-k** Switch off connection book-keeping. Normally a record of the number of inputs and outputs to and from the various branches of a nodes data structure is kept. This information is used for consistency checking, data structure pruning and by certain selectors. When this option is used compilation is significantly faster, however node connection output details are no longer available, so the "$$:$" selector returns zero and the "outs" structure members are set to zero.
4. **-l** Log errors to "outname.log".
5. **-o *outname*** Set output name. No extension expected.
6. **-u*x*** Sets the number of node buffers to ***x***. The default is 10. A large value helps prevent thrashing and so speeds up compilation. However, it is currently limited by memory space.
7. **-w** Switch off warnings.
8. **-W*x*** Set warning level:
   x = 1 indicates type mismatches etc.,
   x = 0 these warnings indicate possible errors to fix etc.,
   x = 2 indicates if selector used before defined etc.,
   x = 3 (all warnings) indicates additional selector undefined traps etc.
9. **-z*x*** Sets the number of string buffers to ***x***. The default is 10. The results of commands, such as "printfs( )", which return a character string are stored in a circular queue. If an expression contains more than ten active strings at any one time use this option to increase the queue size.

Note:
When the program nsl2 is initially compiled it has to be linked with the user dependent file "nslusrfn.c". This file is created by compiling the file "nslusrfn.n" with "nsl1" using the **-f** option.

Files:
(input) inname Must refer to a "inname.nss" file.
(output) outname.log.
outname.nso Pseudo object form of compiled net.
(temporary) outname.nsz.

Current Compiler Restrictions

1. While the compiler can handle all types used in data structures, it presently does not permit multidimensional arrays to be used for connections or initialised. To alleviate this restriction functions can be used to map between single and multidimensional. See standard functions.
2. In addition, it does not access or initialise unions. This is a bug to be fixed at some stage.

3. Only basic types are supported in list expressions, which must be cast accordingly. Note casts can only be applied to lists e.g. “(cast)[…]”, and are presently not supported in expressions.
4. At some stage these cast restrictions should be lifted. Internally, all expressions are evaluated using double precision. This is primarily due to the restrictive nature of “yacc” which is used to evaluate expressions.

The NeSeL Converter: nsltox

This program converts the pseudo object file produced by nsl2 into more standard formats. Presently these are either C or text.

Command line: **nsltox** *options* inname

Options:

1. **-d** Produce executable C data structures. Stored in “outname.nsd”.
2. **-i** Include array index numbers in text output. When necessary an array member’s true index number is stored so that virtual and physical connection addresses can be mapped correctly. Use this option if you want to check input and output addresses. Note the monitor net loading and saving routines compress addresses so that afterwards virtual and physical addresses are the same.
3. **-n netname** Give net this name. Used with **-d** option. Creates a variable for storing net information with given name and of type “NSLNSLnet”.
4. **-N auxname** Use symbol details specified by this name. When a net is finally linked into an executable file symbol details must be defined somewhere. If neither options **-n** or **-N** are specified the default net details produced by “nsl1” and stored in a “.nsc” file are used. If only **-n** is used then auxname is set to netname. Note, in this case the **-n** and other symbol options should have been specified in the “nsl1” command line. Setting the auxname to something else with **-N** implies the symbol details associated with the auxname net are to be used. Consequently such a net must be suitably compiled and linked. Specifying “0” for auxname sets symbol details to null and causes the default core monitor details to be accessed if required. This option may be useful for solving link errors**.**
5. **-o outname** Set output name. Chops off any extension, then adds set extensions.
6. **-t** Produce text output. Stored in “outname.nst”. This is a more readable form of the net data structure. Used primarily for checking net specifications.
7. **-w** Switch off warnings.
8. **-W** Switch on warnings.

Notes:

1. If neither options **-t** or **-d** are used, then a stub network variable is written to “outname.nsd”. This variable can be installed using the monitor function “nsl_install( )” as normal, but is unrunable. However, it defines the network functions to the monitor and enables instances of the net to be loaded and run via “nsl_get( )” etc.
2. The data structures include special variables for guarding outputs. At the start of a cycle all outputs are copied and saved in these special variables. Node accesses to outputs via the data structure access functions refer to these special variables. Therefore, any changes a node makes to its private data structure, such as changing an output value, will not have any affect on other nodes until the next cycle. So, when specifying a node function, there is no

need to worry about guarding inputs and outputs, the NeSeL simulator takes care of this.

Files:
(input) inname.nso, base.nss Base name determined from inname.nso.
(output) outname.nsd Data structures of net in standard C format,
outname.nst Text form of data structures for debugging.
(temporary)] outname.nsm Connection mapping table,
inname.nsy, inname.nsz.

The NeSeL Loader: nsl

This is a compiler driver program that will apply the C-pre-processor, nsl1, nsl2 and nsltox in turn. It enables network specifications to be compiled to ANSI C in one simple invocation.

Command line: **nsl** filename [*options*]
Options:

1. **-cpp [*args*]** Run C pre-processor with ***args***.
2. **-nsl1 [*args*]** Run nsl1 pre-processor with ***args***.
3. **-nsl2 [*args*]** Run nsl2 pre-processor with ***args***.
4. **-nsltox [*args*]** Run nsltox pre-processor with ***args***.
5. **-no-cpp** Skip this phase.
6. **-no-nsl1** Skip this phase.
7. **-no-nsl2** Skip this phase.
8. **-no-nsltox** Skip this phase.
9. **-config filename** Read options from configuration file. At start up **nsl** looks for the default config file “nsl.ini” in the executables directory. If the **-config** option is the first option it will suppress loading of the default config file. Syntax of config files is similar to windows “.ini” files. Note, config files can call other config files.
10. **-chdir path** Change working directory.
11. **-shell args** Issue **args** to DOS shell as a command
12. **-filename name** Change build name.

Notes:

1. **nsl** appends the correct file name extension and adds the name to a phase’s argument list automatically.
2. **%ENVVAR%** Environment variables are substituted when found.

The Stand-Alone Monitor: nslw & nslms

This program is a stand-alone monitor for examining nets. It simply calls the "nsl_monitor( )" routine which then allows nets to be loaded, run, examined, taught, saved etc. using the monitor commands and C interpreter. It works on the pseudo object files of nsl2, this avoids having to use "nsltox" or a C compiler. "nslw" is the windows version, and "nslms" the DOS version.

Command line: **nslw** [*commands*]

Upon start-up the optional monitor commands will be executed.

Notes:

1. The monitor assumes the small model is being used.
2. The stand-alone monitor must be created and linked with user dependent files defining the common routines and data structures used. This is normally a once only process carried out either when NeSeL is first installed or if the common routines are updated. See make files.

Network Specification File Formats

The following file templates apply to small models. However, the format is the same for large models, but use the appropriate large model files. A network specification should be divided into net and C specifics. The network details should be stored in files with ".n" extensions, and C details in ".c" files. A typical format for a network file is:

```
#include "math.h"
#include "nsltype.h"
#include "nslstd.h"
#include "nslios.h"
...
#include "logic.nh" /* and other include files */
...
/* net specifications */
...
net network( ) /* root net declaration */
{...}
```

A typical format for a ".c" file is:

```
#include "stdio.h" /* and other C include files */
#include "nsltype.h" /* NeSeL header files */
#include "nslstd.h"
#include "nslios.h"
#include "netname.nsh" /* type details generated from net */
#include "netname.nsc" /* C node function details from net */
#include "netname.nsd" /* the net data structure */
...
/* NSLmain makes calls to simulator, */
/* monitor & node access functions etc*/
NSLmain(int argc,char *argv[])
{...}
```

This file is compiled and then linked with the NeSeL library and "nslusrfn.c" as necessary.

The NeSeL Library Files

General Files

1. "nsltype.h" The various types used in NeSeL, such as NSLNSLid.
2. "nslstd.h" Macros for selectors and list operators, plus standard function prototypes such as "lcat( )".
3. "nslstd.c" The C source for the standard functions.
4. "nslusrfn.n" Simply invokes "nslstd.c" as a net to create "nslusrfn.c" from.
5. "nslmonfn.n" An example net file for producing "nslmonfn.c".

Small Model

The small model library files are for network specifications which once compiled will fit into the memory of a uniprocessor:

1. "nslios.h" Node and data structure access functions, simulator and monitor function prototypes, error return codes etc.
2. "nslios.c" The C source for the access functions and simulator code.
3. "nslmsio.h" DOS Window and graphics function prototypes etc.
4. "nslwsio.h" W95 Window and graphics function prototypes etc.
5. "nslmsio.c" Source code for windows and graphics routines.
6. "nslmslo.c", "nslmshi.c" Monitor source code for core routines.
7. "nslmsmon.c" Debugger source code.

Large Model

As for small model but file names end with an "l" instead of "s".

Standard functions

Input Access Functions (nslios.h)

These functions permit access to node inputs in a model independent way:

1. inc(c) fetch an input of type **char** from a node,
2. ini(c) **int** input,
3. inl(c) **long** input,
4. inf(c) **float** input,
5. ind(c) **double** input,
6. inp(c) fetch a pointer to input variable, user must cast accordingly.
7. inx(t,c) fetch an input of type "t".

In the above, parameter "c" is a variable of type **long** which contains a symbolic connection address. Normally "c" would be a data structure input variable. A similar set of functions are provided which return a pointer to a node input of the indicated type:

incp(c), inip(c), inlp(c), infp(c), indp(c),inxp(t,c).

Data Structure Access Functions (nslios.h)

To access the data structure of a node from outside its node function, e.g. access by other nodes or C routines used to drive, communicate or monitor the net, a set of functions have been defined. Again, these are designed to be model independent:

1. dsc(n,t,p) fetch value of **char** variable indicated by path,
2. dss(n,t,p) **short** value,
3. dsi(n,t,p) **int** value,
4. dsl(n,t,p) **long** value,
5. dsf(n,t,p) **float** value,
6. dsd(n,t,p) **double** value,
7. dsp(n,t,p) fetch a pointer value,
8. dsnp(n,t) fetch a pointer to node data structure,
9. dsvp(n,t,p) fetch a pointer to variable indicated by path.

In the above, parameter “n” is a node number of type **long**, “t” is the type name of the node being accessed, and “p” is the path of the variable to be accessed from the node. These functions should be used for reading only.

A similar set of routines have been provided for reading and writing to data structure values in a safe way when the network is being simulated on a multiprocessor. This is particularly important in large models. These routines guard any reads by semaphores. The variable cannot be read again until it is written to by the first guarded read. These guarded functions can be used in small models: the semaphores are simply ignored. They are:

rdsc(n,t,p),rdss(n,t,p),rdsi(n,t,p),rdsl(n,t,p),rdsf(n,t,p),
rdsd(n,t,p),rdsp(n,t,p),rdsnp(n,t),rdsvp(n,t,p),
wdsc(n,t,p,v),wdss(n,t,p,v),wdsi(n,t,p,v),wdsl(n,t,p,v),wdsf(n,t,p,v),
wdsd(n,t,p,v),wdsnp(n,t,v),wdsnp(n,t,v),wdsvp(n,t,p,v).

With the same meaning as for the “dsX( )” functions. Parameter “v” in the write functions is the value to be written. Note the “dsX( )” functions can be used for unguarded reads (if no following write is needed).

Note, if a node data structure belonging to another net beside the current, or a node needs to be accessed before calling the simulator, the “nsl_set( )” function must be called first, see below.

Simulator Functions (nslios.h)

int nsl_cycle(NSLNSLnet *npt, long cycles)

This function runs the specified net for the indicated number of cycles or until the net terminates by setting the “nsl_stop” or “nsl_abort” variables. The “nsl_stop” variable causes simulation to stop after the current cycle completes. The “nsl_abort” variable causes immediate termination. Only stopped simulations can be continued safely. It returns an NSL error code.

int nsl_driver(NSLNSLnet *npt, int cmd, va_list args)

Issues a command to a nets i/o drivers. “args” is a pointer to a list of optional arguments. A net i/o driver is a special sort of node that can be used to service the net. Any node that includes a structure member of type “NSLNSLio” and called “args” in its root node structure is treated as a driver. The structure “NSLNSLio” contains two members: “cmd” and “args” which are set to the driver call arguments. A driver should cater for the following commands:

1. NSLIO_NULL Issued when node is being called as a normal node during simulation.
2. NSLIO_SETUP Issued before simulation starts in order to set up environment, such as opening files, windows etc.

3. NSLIO_INIT Caller wants net to be reinitialised.
4. NSLIO_CLOSE Close any resources controlled by net, such as files, for eventual termination.
5. NSLIO_REDRAW Redraw screen output, such as images, status etc.
6. NSLIO_PATTERN Apply new input pattern to net.
7. NSLIO_PRE Perform pre-simulation cycle duties.
8. NSLIO_POST Perform post-simulation cycle duties.
9. NSLIO_USER User definable subfunctions.

Upon completion of a command the driver should set the “NSLNSLio” command argument to “NSLIO_NULL”. Note, it’s up to the user to call “nsl_driver( )” since the “nsl_cycle( )” function doesn’t.

Normal nodes can access the first net driver data structure during simulation via the macro “nsl_drv(driver struct type)”. This returns a pointer of the specified type to the driver data structure.

Monitor Functions (nslios.h)

    NSLNSLnet *nsl_get(char *name)

Reads the net indicated by “name” into memory. The net resides in a pseudo object format file (extension “.nso”). It returns a pointer to a NSLNSLnet element containing details on the loaded net, or NULL if unable to load. Variable “nsl_err” is set accordingly.

    int nsl_put(NSLNSLnet *npt, char *name)

Saves the net specified by the net pointer “npt” to disk in pseudo object format with the file name indicated. It adds a “.nso” extension to the file name. It sets variable “nsl_err” accordingly and also returns this value. Note the net still resides in memory afterwards and may be further simulated.

    void nsl_free(NSLNSLnet *npt)

Frees (most of) the memory used by a net for further use.

    void nsl_set(NSLNSLnet *npt)

Sets the current net being simulated to that pointed to by “npt”. This function is used in conjunction with the data access functions which operate on the current net. Note if a net needs to be accessed in some way before the simulator is called this function must be used.

    int nsl_install(NSLNSLnet *npt, char *name)

Installs a nets, or stub net, details into the symbol table. This enables it to be accessed via the “nsl_get( )”, “nsl_put( )” and monitor functions etc. “name” is any unique identifier for referring to the net.

    int nsl_uninstall(char *name)

Removes a net identified by “name” from the symbol table.

    long nsl_handle(NSLNSLnet *npt, char *path)

Returns the node number of the node specified by “path”. This function would be used with the data structure access functions when only the path of a node is known. Note, only absolute paths in either symbolic or “dip” format are allowed. Dip format is an ASCII string of object declaration numbers and object index numbers, each such pair being separated by a path separator, for example: “3:0 \ 2:1 \ 1:0”. Array indices must be simple ASCII integers, for example:

    hand_full = nsl_handle(npt,"/fulladder[4]/and:2")

The “:2” refers to the second “and” node declared in the “fulladder”. Additionally, a wild array is allowed in which case the size of the array is returned:

add_hd = nsl_handle(npt,"/fulladder[*]")

The function returns zero in the event of an error or path failure.

int nsl_write(NSLNSLnet *npt, long nodenum, char *path, double value)

After converting to the appropriate type this routine stores the “double” value in the variable referred to by the symbolic name string. When “nodenum” is zero “path” should be the name of a global variable, otherwise it is the path of a variable in the specified node’s data structure. Returns an NSL error code, e.g. zero if successful. The compiler option **-v** or the pragma variable must be set for global variables, and the **-m** option for node data structure variables. Note this is a monitor function which takes a path string parameter and should not be confused with the data access functions (“dsX( )”, “rdX( )”, “wdX( )”) which take a non string path parameter. This function and the related ones below override any guarded variable.

int nsl_write_p(NSLNSLnet *npt, long nodenum, char *path,*valp)

Similar to “nsl_write( )} except that “valp” points to the variable value which must be of the correct type.

double nsl_read(NSLNSLnet *npt, long nodenum, char *path)

Fetches the value of a variable and returns it cast as a “double”. Returns a zero value if access failed and sets “nsl_err” accordingly.

char *nsl_read_p(NSLNSLnet *npt, long nodenum, char *path)

Returns a pointer to the required variable, or null if access failed.

char *nsl_adjust(NSLNSLnet *npt, char *vpath)

Sets and returns the value of a node or global variable as a displayable string. The syntax of the string held in the “vpath” buffer on entry is:

vpath: [[nd ,]] path [[= value]]

Where “path” may be relative to the root net, or the node if preceded by a node number “nd”. The C “scanf( )” routine is used for parsing and the result is written back to the “vpath” buffer.

Debugger Functions (nslios.h)

int nsl_monitor(char *cmds)

This function invokes the NeSeL symbolic debugger. “cmds” is an optional list of debugger commands to execute upon start up. This should be set to null if not required. Commands to the debugger can be basic “C” statements or a debugger command. The following commands are available:

1. **:** ***expression***

   Go to line in view file.
2. **/** ***string***

   Search for pattern in view file.
3. **[**, **]**

   Page view file up or down, or by line.
4. **$** [NODEHANDLE\] VARPATH [= ***expression***]

   Access node variables, if NODEHANDLE is zero returns node handle for VARPATH.
5. **$**time.NAME

   Returns age of net NAME.
6. **break, continue, return** [(ARG)]

C flow control.

7. **call** NAME [ARGS]
   Execute commands in batch file "NAME.nat".
8. **cd** PATH, **dir** PATH, NAME**:**
   Change or display net directory, or select net.
9. **delete**, **disable**, **enable** NUM|NAME|all
   Change break-point,watch or net status.
10. **driver("**cmd**"**, ARGS**)**
    Issue command to net io driver.
11. **dump** [WID|WID FILENAME|WID close|auto|manual]
    Dumps windows to file.
12. **exit, quit**
    Leave monitor.
13. **for(**[***expression***]**;**[***expression***]**;**[***expression***]**), do, while(*expression*)**
    C flow control.
14. **go** [**(**ARGS**)**], **step** [**(**NUM [**,**RATE [**,**DELAY [**,**ARGS]]]**)**]
    Cycle networks NUM times at RATE epochs per cycle and with a millisecond DELAY between cycles. Note this command also issues the appropriate commands to any net i/o drivers.
15. **if(*expression*), else**
    C conditionals.
16. **load**, **save** NAME FILENAME
    Loads or saves network from/to "FILENAME.nso". Also calls net i/o drivers.
17. **log**, **logcmd** NAME
    Log i/o to ".nog" file, or commands to ".nat", NAME="off" turns off logging.
18. **new**
    Restarts monitor.
19. **open** FILEPATH, **close**
    View text file.
20. **print *expression*** ; **? *expression***
    Evaluate & display, or use "printf( )" like normal.
21. **shell** [ARGS]
    Issue command to shell.
22. **show** [all | functions | global | local | nets | stop | system | variables | watch | Windows]
    Display various information.
23. **stop** ARG**, watch** ARG
    Set break-points or watch. In the case of **stop,** ARG should be an expression which when evaluated terminates simulation if not equal to zero i.e. TRUE.

Notes:

1. Functions and variables are declared and used as normal except that arrays and structs are not supported.
2. Function parameters should be declared in the ANSI-C manner e.g.
   type func_name(type parm,...) {...}
3. Only basic types are allowed: **char**, **int**, **long**, **float**, **double**, *****.

4. All ambiguous commands should be terminated with a semi-colon.
Below is a file template illustrating how to debug a net with the debugger.

```
#include "stdio.h"/* normal C headers */
#include "nslstd.h" /* NeSeL headers */
#include "nslios.h"
#include "nslwsio.h"
#include "hop.nsh"  /* your net header file */

/* C specifics, such as net i/o driver service routines */

#include "hop.nsc" /* your compiled net files */
#include "hop.nsd"

void NSLmain(int argc,char *argv[])
{       nsl_mon_wnds( );
        nsl_install(&network,"aux_face");
        nsl_driver(&network,NSLIO_SETUP,(char *)0L);
        nsl_monitor("load myhop hop;myhop:;");
}
```

Note the main routine is called “NSLmain” not “main”. The function “nsl_mon_wnds( )” simply allocates windows for the debugger before the net is set up via the driver call. Finally the debugger is called with some start up commands.

Window & Graphics Functions (nslwsio.h)

The purpose of these routines is to provide a standard output mechanism across different computer platforms. The following routines have been tailored to suit neural net applications. In the following “w” is a window number.

    int w_x(long yx), int w_y(long yx), long w_xy(x,y)

Extract x or y number from (long) packed position value.

    BOOL w_active(int w)

Returns true if a key has been pressed in window.

    int w_arc(int w,x1,y1,x2,y2,x3,y3,x4,y4)

Draws an arc bounded by rectangle x1,y1,x2,y2 from vector x3,y3 to x4,y4.

    int w_bordercolor(int w,c)

Set border color for fills to “c”, see below.

    void w_blank(int w,BOOL frame)

Blank window. If frame is true blanks frame as well.

    int w_circle(int w,r)

Draw circle at current position with radius “r” pixels.

    void w_clear(int w)

Clear window and redraw frame plus any titles set.

    void w_clear_line(int w)

Clear current line.

    void w_close(int w)

Close window. If window is not shared by any other process it is removed from the screen.

    void w_dump(int w,NIO_DP_OPEN,char * filename)
    void w_dump(int w,NIO_DP_CLOSE)
    void w_dump(int w,NIO_DP_DUMP)

The contents of a window can be dumped to a file. The first option opens the dump file and connects it to the window. The second option turns dumping off and closes the dump file. The third option dumps the current window contents to the file.

    int w_ellipse(int w,rx,ry)

Draw ellipse at current position with radii rx, ry pixels.

    int w_fill(int w)

Fill window starting at current position using border colour as a boundary.

    int w_fillstate(int w,state)

Set fill state for circle, ellipse, rectangle. ‘state” is either zero for off, positive for on, or negative to return current state.

    void w_focus(int w)

Set focus to window.

    int w_getch(int w)

Waits for a key to be pressed and returns it. Does not echo.

    int w_getpixel(int w,x,y)

Returns colour of pixel at coordinates.

    char *w_gets(int w,char *bp)

Fetch a carriage-return terminated line of input into buffer “bp”.

    int w_move(int w,d)

Move in the current direction a distance “d” in current units. If pen is down and units is pixels a line is draw.

    int w_moveto(int w,x,y)

Move to position.

    int w_onscreen(int w)

Returns non-zero if the current location is inside the window.

    long w_origin(int w,x,y)

Set or get window origin on display to (x,y) measured from top, left. Returns position if x or y less than zero. Use w_x( ), w_y( ).

```
int w_open(char *name,
        BOOL share,frame,scale,ischr,iswrap,
        int lx,ly,fc,bc)
```
Opens a window and returns the window number. “name” is any string used for identification. If “share” is set the window is shared with any other window of the same name. If “scale” is set the “x” dimension is scaled to allow for the screen and pixel aspect ratios. If “ischr” is set the window dimensions are treated as char units otherwise as pixels. If “iswrap” is set text is wrapped at end of lines otherwise truncated. “lx” and “ly” are the window dimensions. “fc” and “bc” are the initial and frame foreground and background colours. Windows are positioned automatically on the screen. If a window cannot be opened zero is returned. Variable “wnd_std” can be used to specify the standard window.

```
int w_papercolor(int w)
```
Set window background color.

```
int w_penchar(int w,state)
```
Sets the pen units to chars if state is positive, pixels if zero, or returns current pen units if negative.

```
int w_pencolor(int w,c)
```
Sets the pen colour.

```
int w_pendown(int w,state)
```
Sets the pen down if state is positive, up if state is zero or returns current state if negative.

```
int w_penwidth(int w,width)
```
Sets the pen width to “width” pixels.

```
int w_poly(int w,num,side)
```
Draws a polygon at current position with “num” sides of length “side”.

```
long w_position(int w,x,y)
```
Sets the current position in units determined by pen char state. If “x” or “y” is negative returns current position.

```
int w_printf(int w,char *fmt,va_list args)
```
Writes formatted output to window.

```
void w_puts(int w,char *str)
```
Write string to window.

```
int w_rectangle(int w,x1,y1,x2,y2)
```
Draw rectangle. Fills depending on fill state.

```
void w_refresh(int w,BOOL alltrks)
```

Clears a window and redraws any tracks. If “alltrks” is set also draws any tracks from shared windows.

    long w_resize(int w,x,y)

Set or get window dimensions. Similar to w_origin( ).

    int w_setpixel(int w,x,y)

Set pixel.

    long w_size(int w,BOOL ischr)

Returns size of window in chars if “ischr” is true otherwise in pixels. Size is packed into a long. Use “w_x( )”, and “w_y( )” to extract a particular dimension.

    char *w_title(int w,pos,cmd,BOOL demal,char *sp)

Set a window frame title. “pos” indicates where to put title in window frame:
NIO_TL,NIO_TC,NIO_TR = top margin left, centre, right;
NIO_BL,NIO_BC,NIO_BR = bottom margin left, centre, right;
NIO_LT,NIO_LC,NIO_LB = left margin top, centre, right;
NIO_RT,NIO_RC,NIO_RB = right margin top, centre, right.
If “cmd=NIO_FREE”, frees current title string and redraws margin.
If “cmd=NIO_REDRAW”, draws current title.
If “cmd=NIO_DRAW”, if ‘sp” is set it becomes new title, draws title.
If “demal” is true the new string should be freed on w_close etc.
Returns current title.

    int w_track(int w,cmd, va_list args)

This functions provides a simple way of controlling moving objects. Three different types of objects and any number of each can be tracked. The object can be a pixel (NIO_PIXEL), character (NIO_CHR) or image (NIO_IMAGE). First of all an object to be tracked must be allocated a track number:

    int w_track(int w,NIO_PIXEL,fc,bc)
    int w_track(int w,NIO_CHR,fc,bc,int chr)
    int w_track(int w,NIO_IMAGE,fc,bc,char *bp,BOOL demal)

where “fc” and “bc” are the foreground and background colours of object, “chr” is the object character and “bp” a pointer to the image. If “demal” is true the object image memory is freed when the track is finally erased. The return value is a track handle unique for that window.

Now the objects new position is set by calling “w_track” again:

    int w_track(int w,NIO_FLOAT,track,float fx,fy)
    int w_track(int w,NIO_INT,track,int ix,iy)

The first method positions the object as though the window was a unit square and “fx”, “fy” the real co-ordinates (of type float). The second method treats “ix” and “iy” as char or pixel co-ordinates depending on the object type. Note “w_track” takes care of erasing the old object and updating any overlaid objects.

Finally, “w_track” can be called to free, redraw, hide or erase a track:

    int w_track(int w,NIO_FREE,track)
    int w_track(int w,NIO_HIDE,track)

int w_track(int w,NIO_DRAW,track)
int w_track(int w,NIO_ERASE,track)

int w_turn(int w,angle)
Turn relative to current angle.

int w_turnto(int w,angle)
Turn to absolute angle. Angles are measured in a clockwise direction in degrees. Thus, “twelve o’clock would be zero (or 360) degrees and “half-past four would be 135 degrees.

Notes:
1. To provide a standard way of specifying colours the following variables are defined:
   - int wnd_num_colors The number of colours available.
   - int wnd_white,wnd_black The colour values for white and black.
2. Co-ordinates are relative to each window. For character co-ordinates the top left window corner is (1,1) in (x=column,y=row) format. For pixel co-ordinates the top left corner is (0,0) in (x=horizontal,y=vertical) format.
3. Windows are updated after each call. To improve efficiency, updating can be turned off by passing the negative of the window handle. The window will then not be updated until the next postitive handle is passed.

List Functions (nslstd.h)

Throughout this section “vp” and “wp” refer to general lists which can be any of the permitted list types, whereas “mp” refers to lists which must be of type “mile *” (i.e. TYPE_SHORT *).
The following macros are for accessing lists in general:

LST_SYS /* number of reserved elements in a list */
LST_MEM /* mem overheads for storing lst */
LST_TYPE /* mask for list type */
LST_PBIT /* type bit set when list is in pair format */
mile lst_num(vp) /* number of elements in list vp */
mile lst_max(vp) /* maximum elements allowed */
mile lst_siz(vp) /* size of list element in **char** units */
mile lst_typ(vp) /* element type code */
mile lst_code(vp) /* check value for validating list */
type_name *lst_head(vp,type_name) /* pt to first list element */
type_name *lst_tail(vp,type_name) /* pt to last list element */
char *lst_base(vp) /* pt to list base */
char *lst_end(vp) /* pt to first byte after last element */

The following functions perform various operations on lists and work for all of the allowed list types:

void lfree(vp) /* free list space */
void *lnew(int num,typ) /* allocate space for a list of num elements */
(type *)lmore(vp,int need,typ,type) /* allocate more space */
void *lcopy(vp,wp) /* copy list into wp if not NULL or allocate space */
void *lcat(vp,wp) /* catenates list wp to list vp */
void *llcat(va_list args) /* catenates lists to single list by elements*/
void *lminus(vp,wp) /* removes elements in list mp from vp */
void *lintersect(vp,wp) /* returns intersection of lists */
void *lunion(vp,wp) /* returns union of lists */

The permissible list type codes “typ” for “lnew( )” are:

TYPE_CHAR, TYPE_DOUBLE, TYPE_FLOAT,
TYPE_INT, TYPE_LONG, TYPE_SHORT,
TYPE_USCHAR, TYPE_USINT, TYPE_USLONG, TYPE_USSHORT,
TYPE_PT.

“llcat( )” takes a variable number of list arguments and returns the list formed by taking the first element of each list in turn, then the next and so on.

```
BOOL lequal(vp,wp) /* returns true if lists vp & wp are equal */
mile lnum(vp) /* number of elements in list vp */
mile lpos(vp,double c) /* position of constant in list */
double lval(vp,mile n) /* value of n’th element in list */
double llo(vp) /* lowest element in list */
double lhi(vp) /* largest element in list */
double lsum(vp) /* sum elements of list */
void *lrev(vp) /* reverses list inline */
void *lnorm(vp) /* normalise list */
void *lmul(vp, double c) /* multiply each element by constant */
void *ladd(vp, double c) /* add constant to each element */
void *llmul(vp,wp) /* multiply corresponding elements of lists */
void *lladd(vp,wp) /* add corresponding elements */
```

Note, in general list returning functions return a new list. For example, “lnorm(vp)” returns a new list which is the normalised version of list “vp”. The following list returning functions take lists of type mile:

```
void *lmap(BOOL islist, void (*fnpt)( ), va_list args) /*map lists*/
mile *lsmap(mp) /* map stack indices */
mile *lrand(mile n,m,a,b) /*list of n to m random elements in range a..b*/
```

“lmap( )” applies the list returning function pointed to by “fnpt” to the argument list of lists. It is applied to the first set of elements, then the second, and so on. Each list it returns is catenated into one list. The list returning function may return a list of any of the list types.

“lsmap( )” maps a list of absolute stack offsets to relative stack offsets. Used when popping a number of connections from the “heap” or “pile”.

```
mile stodx(mile s,dx), stody(mile s,dx), dtos(mile x,y,dx)
```

Functions to map between single and double dimensions. Parameter “dx” is the width of the x-dimension, ‘s” the single dimension co-ordinate, and “x” and “y” the two dimensional co-ordinates.

The following are popular transfer functions:

```
float generic_bound(float x,top,bot)
```

Returns “x” unless it ranges outside limits “top” and “bot” in which case returns the appropriate limit.

```
float generic_limiter(float x,top,bot,mid)
```

Returns “top” if “x” is greater than “mid” otherwise returns “bot”.

```
float generic_sigmoid(float x,top,bot,gain,c)
```

Performs the function:

$$y = bot + \frac{top - bot}{1 + \exp^{-gain*(c+x)}}$$

Some other useful functions:

```
void srnd(int seed) /* reseed pseudo random number generator */
float rndf( ) /* return random number,  0.0 <= rndf( ) < 1.0 */
int rnd(int randmax) /* return random number, 1 <= rnd( ) <= randmax */
```

double lhopsum(mile i,j,noself,float g,k, va_list pats)

This computes a Hopfield weight ($\frac{1}{p}\sum_p w_{ij}$), where $w_{ij} = (gx_i + k)(gx_j + k)$. Parameter "i" is the node index and "j" the input index to node "i". Parameter "noself" should be set to a non-zero value if self-recurrent connections are not being used and the input index "j" includes the node index. "Pats" is a list of patterns. Patterns are lists of values, the x_n's.

Finally, a version of "printf( )" which returns its result as a string:

char *printfs(char *fmt, va_list args)

Standard List Macros (nslstd.h)

repeat(x,r) ≡ int r;for(r=0;r<(x);++r)

Vector Functions (nslios.h)

These functions perform various vector operations on connection inputs and data structure variables: The following macro definitions make calls to the generic vector functions "vec_sum( )", "vec_norm( )", "vec_prod( )" and "vec_liprod( )":

```
float sumX(void *a,*b,int n, NSLNSLnet *i) /* sum elements of vector */
float normX(void *a,*b,int n, NSLNSLnet *i) /* return norm of vector */
float prodXY(void *a,*b,*c,*d,int n,NSLNSLnet *i,*k) /* inner product of vectors */
float liprodXY(void *a,*b,*c,*d,int n,NSLNSLnet *i,*k,float g,k) /* linear ip */
```

In the above functions characters "X" and "Y" should be replaced with the vector type, such as "f" for float, "i" for int etc. "a" and "c" are pointers to the first element of the array. Array elements may be simple atomic types or structs. "b" and "d" are pointers to the first element, or variable within the element if a struct, which is to be operated on. For simple types "a" and "b" etc. will be the same. Parameter "n" is the vector dimension, "i" and "k" should be set to a net pointer or NULL depending on whether the vector is an input vector or basic vector. Function "liprodXY( )" computes a linear inner product: $\sum d_i(gb_i + k)$.

Variables (nslios.h)

NSLNSLnet *network

The default network to simulate or access.

int nsl_abort,nsl_stop

If a node function sets these to a non zero value during simulation the simulation will be terminated either immediately, or after the current cycle finishes, respectively.

int nsl_err

This is set to the appropriate error code whenever an error occurs.

node_type Nnnnnnn

For each node in a network a variable is defined with the node struct type and name equivalent to the node number prefixed by a capital "N" character. Normally these variables would be referred to via one of the standard functions in which case just the node number is required. This can be determined with the "nsl_handle( )" function. However, when symbolically debugging a net the variable name may prove useful.

NeSeL Error Codes

The various error codes returned by the monitor and simulator functions are:

```
NSLER_OK 0 /* no error */
NSLER_OO 1 /* intermediate value for ER_OK */
NSLER_AS 2 /* array size error */
NSLER_BF 3 /* bad file */
NSLER_FL 4 /* access to file went wrong */
NSLER_OS 5 /* out of memory space */
NSLER_DC 6 /* declaration error */
NSLER_NL 7 /* access to node object block went wrong */
NSLER_VR 8 /* version number error */
NSLER_IX 9 /* nodenum not in index */
NSLER_MO 10 /* monitor struct details need updating */
NSLER_RE 11 /* conx map reconstruct err */
NSLER_VA 12 /* variable access failed err */
NSLER_MP 13 /* monitor parsing err */
NSLER_NU 14 /* no network defined */
NSLER_NM 15 /* name in use already */
NSLER_ST 20 /* cycle was stopped */
NSLER_AB 21 /* cycle was aborted */
```

NeSeL Types

The various types used by NeSeL are as follows:

```
typedef short int mile  /* list values */
long /* node numbers & symbolic conx addresses */
typedef struct {long name; char *path;} NSLNSLid; /* node & group info */
typedef struct {int rate,wait;} NSLNSLtime;  /* time info */
typedef struct {int cmd; char *args;} NSLNSLio;  /* net i/o driver */
typedef struct {long nd; NSLNSLio *iop;}NSLNSLdrv; /* driver details */
```

The NSLNSLnet struct contains some net parameters that may prove useful:

```
typedef struct
{       int flags, /* internal flags */
                num_funcs, /* number of different node funcs used by net */
                num_conxs, /* number of unique output connections in net */
                num_nodes; /* number of nodes in net */
        long time; /* age of net in cycles */
        void (**fpt)( ); /* addresses of node functions */
        long sst_bs; /* position of root net in resource file */
        long *sstpt; /* file positions of node function resource details */
        NSLNSLconx *cpt; /* pointers to info on connections */
        NSLNSLnode *npt; /* pointers to info on node data structures */
        char *name,*sst_name; /* name of net, name of it's resource file */
        char *var_name,*ins_name;
        int *num_ufs, /* pt to num of user functions */
                *num_sts, /* pt to num of user structs */
                *num_six, /* pt to num in struct index */
                *num_vas, /* pt to num of user vars */
                *sixpt; /* pt to struct index */
        void (*ept)( ); /* func to eval user func */
        char **upt; /* user func names */
        NSLNSLfelm *apt; /* user func addresses & arg details */
        NSLNSLstruct *spt; /* pt to struct details if set */
        NSLNSLvar *vpt; /* pt to var details if set */
        NSLNSLdrv *dpt; /* pt to net io node list if set */
        int num_drv; /* num of io drivers */
        void **mpt; /* list of malloc'd blocks used by net */
        void *args; /* user defined parameter */
    } NSLNSLnet;
```

Installing NeSeL

The Source Disk

1. The NeSeL package is divided around a dozen directories on the source disk.
2. These are called “nsl”, “nsl1”, “nsl2”, “nsltox”, “nslw”, “nsllib”, “include”, “example”, “manual” “cpp”, and “yaccr”. A zipped version may be present.
3. Project make files are included for Microsoft Visual C/C++ version 4.2.
4. It is suggested that a directory be created on the system disk called “nesel”. The source directories and their contents should then be copied to this directory as subdirectories.
5. The appropriate system paths should be set depending upon where the final executable files go: “nsl”, “cpp”, “nsl1”, “nsl2”, “nsltox” and “nslw” etc.
6. A copy of the “nsl” config file “nsl.ini” should be modified to set the cpp include paths and then copied to the executables directory. There are normally two paths: the C compilers include directory & NeSeLs (nesel\include). A compiled version and the C source for a version of “cpp” that works with MSVC & NeSeL is provided in “cpp” & “yaccr”.
7. The phase one compiler “nsl1” should be built via the project file in “nsl1”. The binary can then be moved to the appropriate directory.
8. The batch file “makeusr.bat” in “nesel\nsl2” should be evoked in order to create the file “nslusrfn.c”.
9. The phase two compiler “nsl2” can be built from the “nsl2” directory.
10. The phase three converter “nsltox” can be built from the “nsltox” directory.
11. Next the batch file “makemon.bat” in “nsllib” should be evoked to create the “nslmonfn.c” file. Then build the NeSeL Library “nsllib.lib” and move it to the “nesel\include” directory.
12. Copy the executables “nsl2” and “nsltox” to the appropriate directory.
13. Then “nslw” can be built from the “nslw” directory.
14. MSVC Project Settings. To use NeSeL with MSVC, set the following:
    - Add “nsl” to MSVC as a Custom Build tool:

      Description: “Running NeSeL…”.

      Build Command: “c:\usr\bin\nsl.exe $(InputName) -chdir $(ProjDir)”.

      Output field: “$(ProjDir)\$(InputName).nsc”
    - Use the MFC as a dynamically linked DLL (NSLLIB uses threads).
    - Add the NeSeL\include path to the preprocessor include option.
    - Change a net file’s type to C/C++.
15. Build the demonstration “FACE”. Note the network file “face.n” can be compiled via the command “nsl face”. Run it!
16. If the yacc files are changed then the recursive version of yacc needs to be used. This has a modified parser which makes “yyparse( )” recursive. This is necessary because the NeSeL compiler recursively calls “yyparse( )”. Install the yacc executable supplied on the source disk. Run the yacc batch files in

"nesel\nsl1", "nesel\nsl2" and "nesel\nsllib". The "rmlines" utility removes hash lines from a file.

NeSeL Demonstration

A simple demonstration net is provided for testing if installation was successful. This net is called "rcn.n". It is based on the resonance correlation network of Ryan et al (1987). After compiling this net (via "nsl rcn") and running it, you should see a couple of boxes appear on the screen along with some characters moving around inside them.

Updating User Functions

User functions are auxiliary functions which provide new language features to NeSeL. For example, functions that produce a list which models a distribution of connections. Such functions can be integrated into the compiler or monitor. This is achieved as follows:

1. Add as necessary appropriate function declarations to the files "nslusrfn.n" (in nesel\nsl2) and "nslmonfn.n" (in nesel\nsllib).
2. Run the batch files "makeusr.bat" and "makemon.bat" as necessary.
3. Rebuild "nsl2", the library "nsllib.lib" and re-link any dependants.
4. The file "nslmonfn.c" is created from a net file, "common.n" say, which contains all the C routines, node functions and data structures likely to be used by a net. When compiling this net with "nsl1" the "-xmvasp" options should be used. The batch files should suffice.

Errors and Debugging Tips

1. First of all be careful not to use variables with the same name as one of the NeSeL functions or macros. This will cause weird C pre-processor errors.
2. When initialising arrays of type 'struct" don't forget two sets of curly-braces for the struct and array, even if all members are being initialised to the same value.
3. Some C pre-processors cannot handle macro parameters with embedded commas. Normally embedded commas will only occur as a NeSeL list separator. It is recommended that these be replaced by semi-colons, which are also valid list separators.
4. Certain other characters, such as "#", are treated in a special way by some pre-processors. These should be prefixed by a backslash if they cause problems.
5. Remember to use dummy lists in label definitions when necessary. For example, if a path is specified in a label definition and the label is consequently used with a list, then the dummy list "[*]" should be included in the label name and path:
   ```
   label[*]: A/B/C[*];
           …
   label[x,y]/E/F
   ```
6. Forgetting the label list may mean the calling list being mapped to the wrong path list.
7. If a network is being symbolically debugged it may be necessary to physically insert an include file into the main C file using a text editor so that the source and assembly line-numbers coincide.

8. Remember that if a node's data structure needs to be accessed before the simulator is called to use one of the "net_set( )" functions.
9. Don't forget to update "nslmonfn.c" after changing data structure definitions or node functions when using monitor functions. Alternatively, the "nsl1" compiler option **-m** can be used to add this information to the ".nsc" file, in which case "nslmonfn.c" should not be used.

Chapter 3

NeSeL Syntax

Syntax Notation

The syntax for the NeSeL extensions to the C programming language is given below in the YACC notation. The primary syntax roots are expressions (“exp”), net declarations (“net_def”), variable declarations (“var_decls”) and node parameters (“node_parms”).

```
exp: exp triop exp | exp “?” exp “:” exp | term
term: “-” term | “!” term | “~” term | “&” var_symbol | “(” exp “)”
        | DNUMBER | natnum | SQCHR
        | FIDENT “( ” “)” | FIDENT  “( ” arg_list “)”
        | VIDENT pmops | VIDENT eqops exp| pmops VIDENT | VIDENT
        | select app_path | select pmops | select eqops exp | pmops select | select
arg_list: arg | arg_list “,” arg
arg: exp | FIDENT | STRING | sslist | “( ” type_cast “)” sslist
triop: “*” | “/” | “%” | “+” | “-” | “<” “<” | “>” “>” | “<” | “>”
        | “<” “=” | “>” “=” | “=” “=” | “!” “=”
        | “&” | “^” | “|” | “&” “&” | “|” “|”
eqops: “=” | “*” “=” | “/” “=” | “-” “=” | “%” “=” | “+” “=”
        | “&” “=” | “^” “=” | “|” “=” | “<” “<” “=” | “>” “>” “=”
pmops: “+” “+” | “-” “-”
select: “$” selcode selnum | “$” “!” partnum | “$” “!” “!” | “$” “!” “@” | “$” “!” “#”
selcode: “&” | “|” | “^” | “_” | “%” | “:” | “@” | “#” | “*” | “+” | “-” | “.”
partnum:  exp | “|” exp
selnum: <nothing> | selrel term | selrel | term
selrel: “.” selcoor | “.” | selcoor
selcoor: “@” | “|” | “?” | “!” | “@” “|” | “@” “?” | “@” “!”
list: list_sel | “[” lelms lssep “]” | “[” “*” “]”
sslist: list_sel | “[” lelms lssep “]”
loop_statement: FOR “(” frexp “;” frexp “;” frexp “)” loop_list
        | WHILE “(” exp “)” loop_list
        | DO loop_list WHILE “(” exp “)” lssep
        | IF “(” exp “)” if_list
frexp: <nothing> | exp
if_list: loop_list | loop_list ELSE loop_list
loop_list: loop_apnd loop_ary | break_statement
loop_apnd: <nothing> | “+” | “+” “+”
loop_ary: sslist lssep | loop_statement | range lssep | “{” lelms lssep “}”
lelms: lelm | lelms lssep lelm lssep: <nothing> | “,” | “;”
lelm: list_sel | range | loop_statement | break_statement | id_decls
break_statement: BREAK | CONTINUE | RETURN | EXIT | ABORT
id_decls: id_type id_list “;”
id_type: DOUBLE | FLOAT | LONG | INT | CHAR
```

```
id_list: id_elm | id_list “,” id_elm
id_elm: “*” id_name | id_name
id_name: IDENT | IDENT “=” exp
range: exp | skips exp | exp “.” “.” exp
skips: “:” | skips “:”
list_sel: “[” list_opr selnum “]” | “[” “-” list_opr selnum “]”
list_opr: “&” | “|” | “^” | “_” | “%” | “:” | “=” | “.” | “!” | “+” | “?”
net_def: NET IDENT “(” macro_parms “)”
        comm_decls “{” net_decls “}” net_vars
comm_decls: <nothing> | comm_do
comm_do: comm_elm | comm_do comm_elm
comm_elm: comm_type comm_list “;”
comm_type: IN | OUT | INOUT
comm_list: wild_id | comm_list “,” wild_id
net_vars: <nothing> | “=” “{” var_decls “}”
net_decls: <nothing> | net_list
net_list: net_elm | net_list net_elm
net_elm: obj_decl “;” | add_decl “;” | “;”
obj_decl: net_or_node obj_ary obj_elms
add_decl: add_lab_path “=” conx_list | add_lab_path
net_or_node: NETID “(” macro_args “)” | NODEID “(” var_decls “)”
obj_ary: <nothing> | “[” fexp “]”
obj_elms: <nothing> | obj_list
obj_list: obj_elm | obj_list “,” obj_elm
obj_elm: obj_lab_path “=” conx_list | obj_lab_path
obj_lab_path: label obj_path | label | obj_path | vir_innet
obj_path: bas_path | tildes in_path
add_lab_path: label add_path | label | add_path
bas_path: gen_path | dol_path | dec_path | dodo_path
in_path: flo_path | ref_path abssep flo_path | ref_path | abssep flo_path
flo_path: wild_path | dol_path | dec_path
out_path: bas_path | tildes sel_path | tildes | dodo_path | vir_outnet list | vir_outnet
dodo_path: “$” “$” wild_path | “$” “$” IDENT “:” wild_path
vir_innet: BIN | PILE | HEAP
vir_outnet: PILE | HEAP
tildes: “~” | “~” “~”
sel_path: gen_path | dec_path
dec_path: list gensep wild_path | list
dol_path: dol_id gensep wild_path | dol_id
dol_id: “$” list | “$”
label: “$” “:” | IDENT “:” | IDENT list “:”
add_path: gen_path | tildes gen_path | vir_innet | dodo_path
gen_path: ref_path abssep abs_path | ref_path | wild_path | abssep wild_path
abs_path: par_path abssep wild_path | par_path | wild_path
app_path: dada_path | “$” app_abs | “$” tildes app_abs
        | “$” “!” app_stem | “$” “?” app_stem
app_stem: app_abs | tildes app_abs | dada_path
dada_path: “$” “$” IDENT “:” app_member | “$” “$” app_member
app_abs: ref_path abssep app_member | abssep app_member | app_member
app_member: app_member pathsep app_id | app_id
app_id: IDENT list | IDENT
ref_path: “.” | par_path
par_path: par_path abssep “.” “.” | “.” “.”
wild_path: wild_path gensep path_id | path_id
path_id: wild_id | “*” “*”
wild_id: wild_name list | wild_name
wild_name: “*” wild_elms | “*” | first_elm “*” | first_elm
first_elm: “?” wild_elms | “?” | IDENT wild_elms | IDENT
```

```
wild_elms: wild_elm wild_elms | wild_elm
wild_elm: “?” | IDENT | INUMBER
conx_list: conx_elm | conx_elm conx_list
conx_elm: outvar_decl | cop_list outvar_decl
outvar_decl: out_decl | out_decl “{” var_decls “}”
out_decl: “(” net_or_node “)” out_path | out_path
cop_list: cop cop_list | cop
cop: “+” | “-” | “+” “+” | “-” “-” | “<” | “>” | “^” | “!”
var_decls: var_decl | var_decls “,” var_decl
var_decl: var_name var_val
var_name: <nothing> | wild_path “=”
var_val: <nothing> | var_exp | “(“ type_cast “)” var_exp
var_exp: exp | STRING | IDENT | FIDENT | sslist | “{” var_decls “}”
node_parms: <nothing> | node_list
node_list: node_elm | node_list “,” node_elm
node_elm: wild_path
var_symbol: IDENT | var_symbol “.” IDENT
type_cast: type_list | type_list star_list
type_list: type_id | type_id type_id
type_id: IDENT
star_list: star_elm | star_list star_elm
star_elm: “*” | “[” “]” | “(” “)” | “(” star_list “)”
gensep: pathsep | “\” “\” | “/” “/” | “|” | “|” “|”
pathsep: “.” | abssep
abssep: “/” | “\”
```