UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED
HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

SNNS

Stuttgart Neural Network Simulator

Implementation Manual, Version 4.0

Contents

Chapter 1

The Purpose of this Document

Most of the users of SNNS do not want to alter any of the source code that is part of the distribution. They therefore do not need to know about any implementation details. In order to keep the user manual down in size we decided to take out all the parts dealing with the implementation.

Since on the other hand there are some people who do need this information we decided to supply the information in this seperate file.

When reading this file, bear in mind that some implementation detail might be outdated, since our main focus is to keep the user manual current (and thereby the majority of users happy).

Chapter 2

Design of the Simulator Kernel

2.1 Network Model of the Simulator

The network model, on which the simulator kernel is based, was designed to accommodate many network types. An important feature of the kernel is the complete encapsulation of all internal data structures and the efficient memory management.

The network model of the simulator kernel has the following properties:

- each unit has an arbitrary, user definable
  - *activation function* which takes the network input, the present activation, and the threshold to compute the new activation.
  - *output function* which computes the output value from the activation of the unit.
- each unit has an arbitrary number (possibly zero) of inputs which can be defined by the user. These so called *sites* have separate user definable input functions.
- sites may have an arbitrary number of connections from other units. A recurrent connection to the unit itself is possible.

2.2 Design Factors

The goal in designing the simulator kernel was to meet the following specifications:

- representation of a universal network model for small to medium sized networks (10^4 units, 10^6 connections)
- interactive manipulation of the net by the user
- encapsulation of all internal data structures with proper interfaces
- extendability

- very high efficiency
- modularity
- portability

The size of the networks that can be handled by the kernel is limited only by the size of (virtual) memory and the address space of the computer used. The simulator memory management relieves the operating system (UNIX), especially with larger nets. In interactive mode, the user has some powerful commands at hand to create and manipulate networks. These interface functions reduce the complex internal representation of the data to a representation at the logical network level.

Naturally, the demands of encapsulation and efficiency contradict each other. Nevertheless, a good compromise has been found: the functions, that can be defined by the user (activation and site functions)may be used with a macro library to access the kernel structures. This principle makes the combination of tight encapsulation and high execution speed possible. Measurements on several computer systems yielded the following propagation rates:

Machine type	**Operating System**	**Propagations** connections/sec. (CPS)	**Weight Updates** conn. updates/sec. (CUPS)
Intel 80486-33	Linux	453.000	194.000
DECserver 5400	Ultrix V4.2	398.000	175.000
DECstation 2100	Ultrix V4.2	494.000	211.000
DECstation 3100	Ultrix V4.2	689.000	287.000
DECstation 5000/200	Ultrix V4.2	733.000	248.000
SUN 3/60	SUN-OS 4.1.1	120.000	42.000
SUN SPARCstation ELC	SUN-OS 4.1.1	791.000	334.000
SUN SPARCstation 2	SUN-OS 4.1.1	950.000	415.000
SUN SPARCstation 10	SUN-OS 4.1.1	950.000	415.000
IBM RS 6000/320	AIX V.3.1	1.998.000	773.000
IBM RS 6000/320H	AIX V.3.1	2.207.000	814.000
HP 9000/720	HP-UX 8.07	1.707.000	764.000
HP 9000/730	HP-UX 8.07	2.147.000	1.093.000

The propagation rate measures the speed of the simulator in recall mode, i.e. the forward propagation rate. In this mode no weight updates take place. The usual measurement unit is connections/sec (CPS).

The weight update rate measures the speed of the training of a fully connected feedforward network trained with 'vanilla' (on line) backpropagation. Because there is a forward propagation, a backward propagation and a weight update phase for every pattern in each cycle, the weight update rates, measured in connection updates/sec. (CUPS) are usually lower by a factor of between 2 and 3 than the propagation rates.

These performance numbers have been obtained on machines running in our lab during normal use, with other users on the machines, with different main memory sizes and with the SNNS home directory mounted via NFS over the ethernet. Therefore, these numbers should only be regarded as performance indicators of SNNS but may not be quoted as machine architecture benchmarks.

The use of simulators for neural nets almost demands the use of parallel computers like the Connection Machine CM-2 [Hil85, HS86a, HS86b] or the MasPar MP-1, because of the inherent parallelism of the algorithm. The use of parallel computers will therefore surely increase the above values.

2.3 Layer Model of the Simulator Kernel

The simulator kernel is internally structured in three layers. Each higher layer represents a higher level of abstraction and is based on the layers below.

- The bottom layer is the SNNS memory management. It sits on top of the Unix memory management and offers functions to allocate and free data structures for the layer above.
- The next layer contains all functions to operate on, and to modify the network as well as propagation functions.
- The interface functions for the graphical user interface are located in the topmost layer.
- In the same layer as the function interface, there exists a file interface to the Nessus compiler.

In chapter 3 the data structures are described in detail. The description of the compiler interface can be found in appendix A of the SNNS user manual.

Chapter 3

Internal Data Structures

The kernel needs numerous data structures to administer and display the network. The kernel has to display the units, links and sites (the *static* component), and to save the functions and connection weights of the units (the *dynamic* component). A symbol table is used to handle all the names and types associated with the units which are defined by the user. The allocation of data structures is done in large blocks of several hundred or even thousands of single structure components. This chapter describes all the relevant data types in detail.

3.1 Unit Array

The units and all their components are stored in the unit array. The data type 'array' was selected for storing the units, because of its short access time. When the array is filled up by requests from the user interface, the memory management automatically requests a new, bigger array from the operating system. The pointers to the structure components are reallocated, and the old empty array is returned to the operating system.

Each element of the unit array has the following components:

- *Out* stores the output value of the unit
- *flags* contains information for the memory management and the topological type of the unit (see chapter3.6)
- *lun* the logical unit number
- *lln* the logical number of the layer the unit belongs to
- **ftype_entry* points to the internal representation of the unit type, if a prototype exists for this unit
- *Aux* saves temporary values needed by internal kernel functions (e.g. backpropagation and topological sort)
- *TD* saves values needed for time delay networks

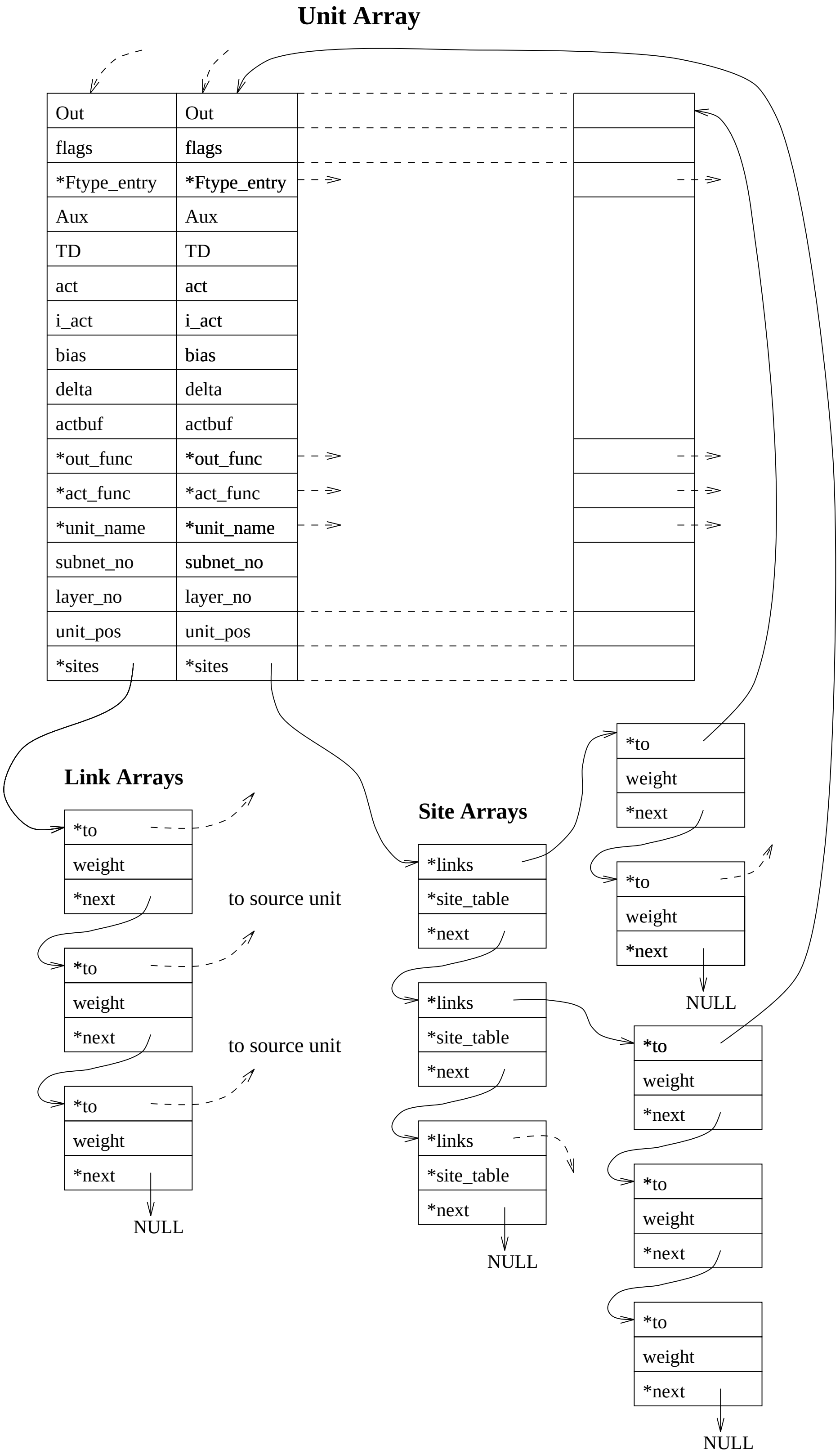

Figure 3.1: Internal network data structures (simplified)

- *act* contains the activation of the unit
- *i_act* saves the initial activation, used to reset the net
- *bias* contains the threshold of the unit
- *value_a, value_b, value_c* general purpose elements for the learning functions
- *olddelta, newdelta* error values in recurrent networks
- *actbuf* an array for storage of 10 miscellaneous values
- **out_func* points to the output function
- **act_func* points to the activation function
- **act_deriv_func* points to the derivation function of the activation function
- **act_2_deriv_func* points to the second derivation function of the activation function
- **unit_name* points to the unit name, a string in the symbol table
- *subnet_no* contains subnet membership information
- *layer_no* contains the location of the unit in a multilayer network
- *unit_pos* stores the position of the unit in the network
- **sites* points to a linked structure of input functions or connections

The internal representation of the network is displayed in figure 3.1.

3.2 Sites

The site data structure was designed to handle the input functions of the unit. The sites are constructed as a linear linked list. Several sites are grouped in an array (site array), each containing several hundred sites. The site arrays are also connected by a linked list. Thereby, empty, no longer needed, arrays can be freed.

The internal representation of a site has three components:

- **links* is a pointer to the input connections (links) of the site.
- **site_table* is a pointer to the so called *site table*. The site table contains information about the input function and the name of the site.
- **next* points to the next site of the unit. The list is terminated by NIL.

3.3 Links

The links make up the connections of the units. The direction of the internal pointers of the link data structure is from the target unit (destination unit) to the source unit (origin unit). Note that this is reversed from the direction of the links in the neural network

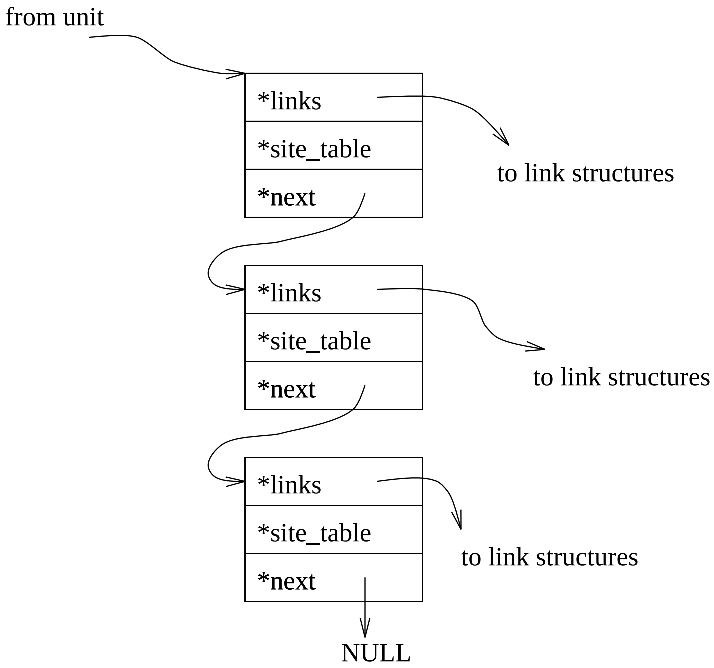

Figure 3.2: Site list

(which run from source to target unit). This approach allows the efficient storage of the network structure with a minimum of memory. However, the memory needed to store big nets is still the limiting factor. For a net with 10.000 units with a connectivity of 20%, 20 million connections have to be stored, requiring 230 megabyte of memory. In comparison, the fraction of the memory used to store all other components is negligible.

The structure of a link is quite similar to that of a site:

- **to* points to the source unit of the link.
- *weight* contains the connection weight.
- *value_a, value_b, value_c* general purpose elements for the learning functions
- **next* points to the next connection of the same target unit (terminated by NIL).

Like sites, the links are grouped in linked arrays which are handled by the SNNS memory management routines.

3.4 Network Memory Management

When a network is loaded or constructed interactively, the memory management always has to supply enough units. As already mentioned, the user doesn't have to care about memory allocation and deallocation. During the interactive manipulation of the net, the user can change or delete whole parts of the net. The empty components are grouped in a linked free storage list for later reuse. This is also true for tables like the symbol table. There, identical names (identifiers) are stored only once. A reference counter decides

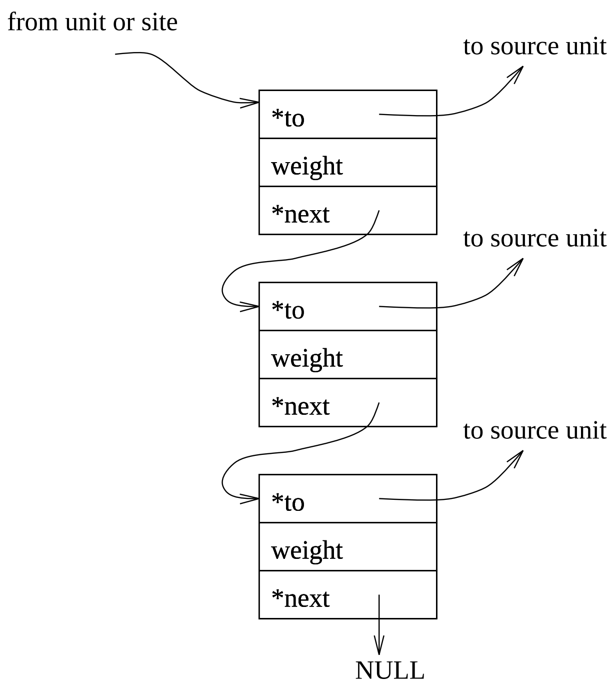

Figure 3.3: Link list

whether an entry can be freed on a delete operation or not. An exhaustive garbage collection is performed when the whole net is deleted.

3.4.1 Link/Site Arrays

The memory management is presented in figure 3.4 with the example of link arrays. Each first entry is used for backward links. The data structure of site arrays is similar.

3.4.2 Symbol Table

The symbol table administers the identifiers of the units, sites and unit prototypes. An entry in the table is deleted when the reference counter is zero. Figure 3.5 shows the symbol table with linked empty entries.

3.5 Unit Flags

The simulator computes the topological position (topological type) and the current state of the unit with the help of the unit flags. The topological type is important for the teaching and propagation algorithms. With certain status bits, the state of the unit in the network is checked. The flags are stored as bit patterns in the unit array. Figure 3.6 shows the correspondence between bits and flags of the unit entry.

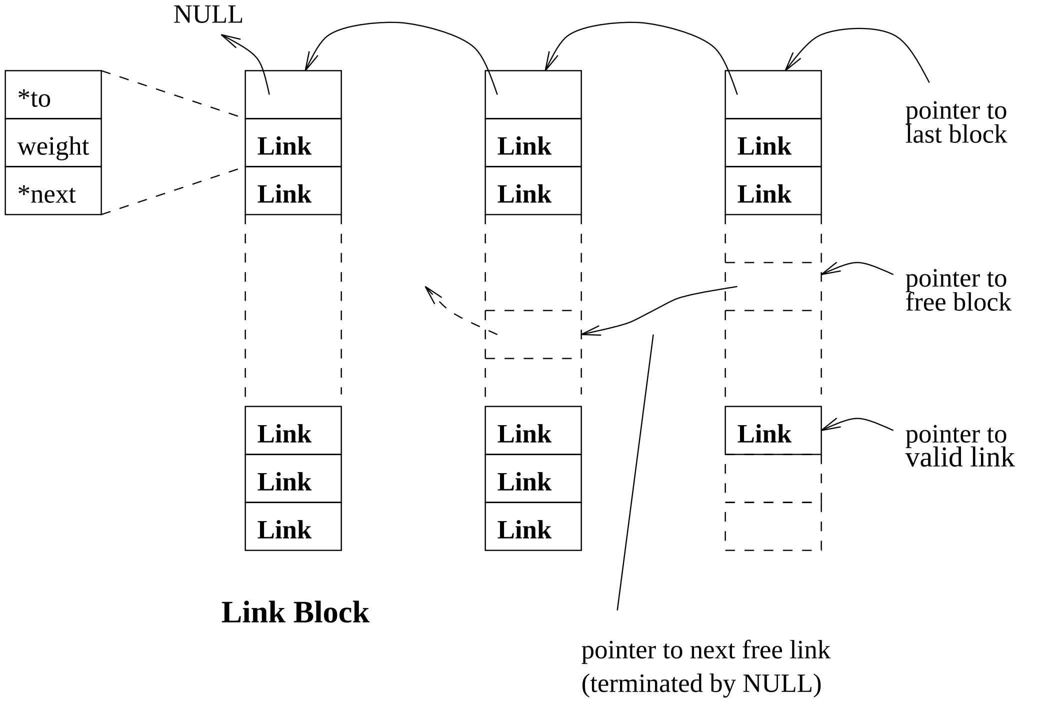

Figure 3.4: Link arrays

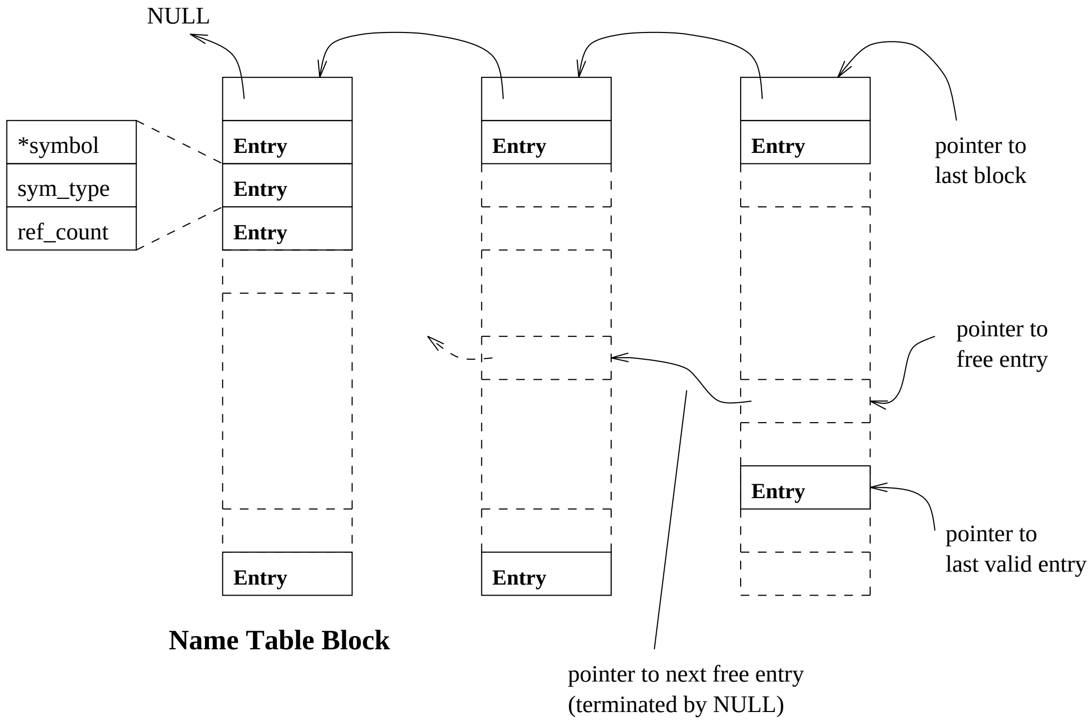

Figure 3.5: Symbol table

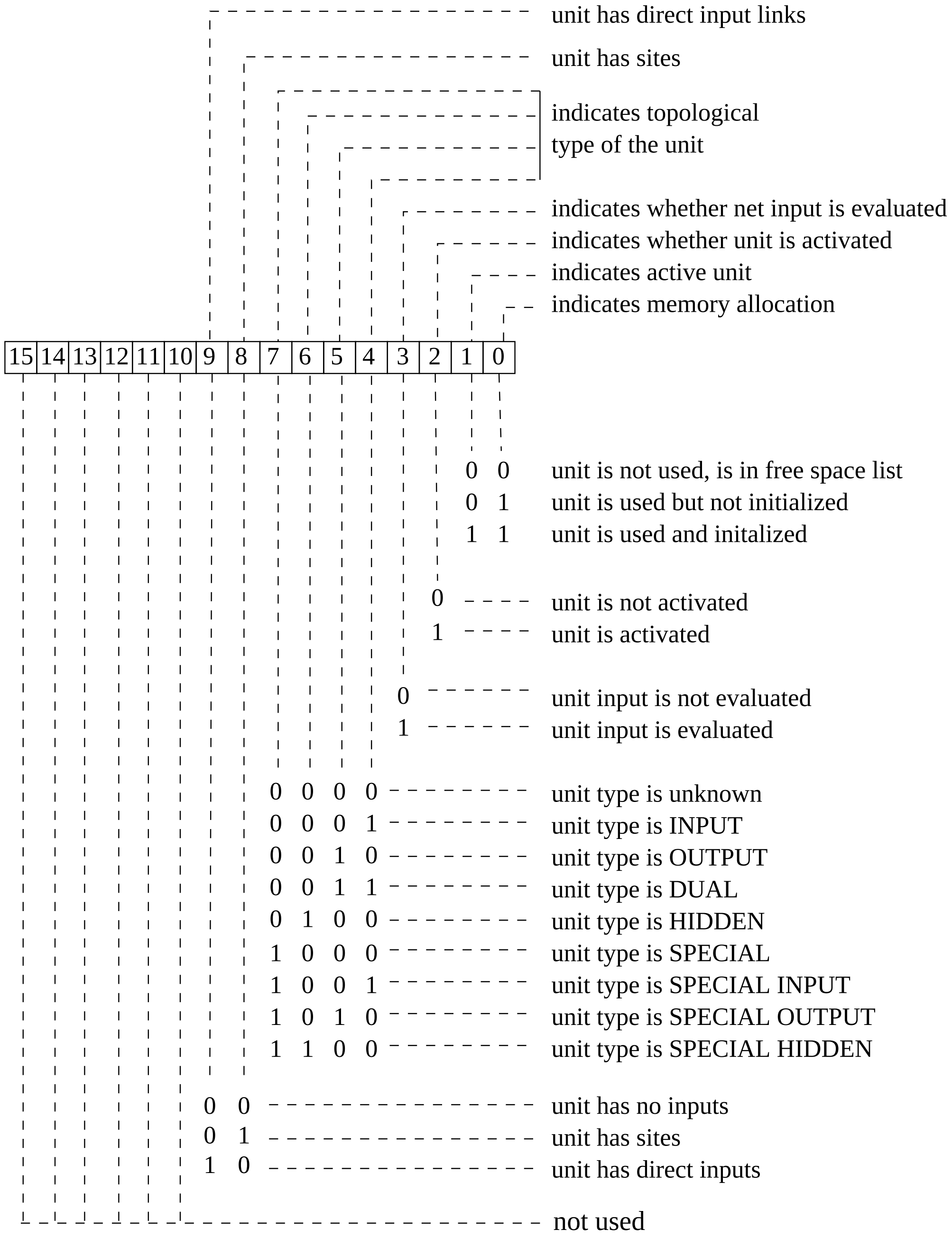

Figure 3.6: Unit flags

3.6 Function Table

The various site and unit functions are stored in the function table. The user can define his own functions in form of a C program. Information about these functions (name, type) is stored in a table, to give the kernel access to and use of these functions.

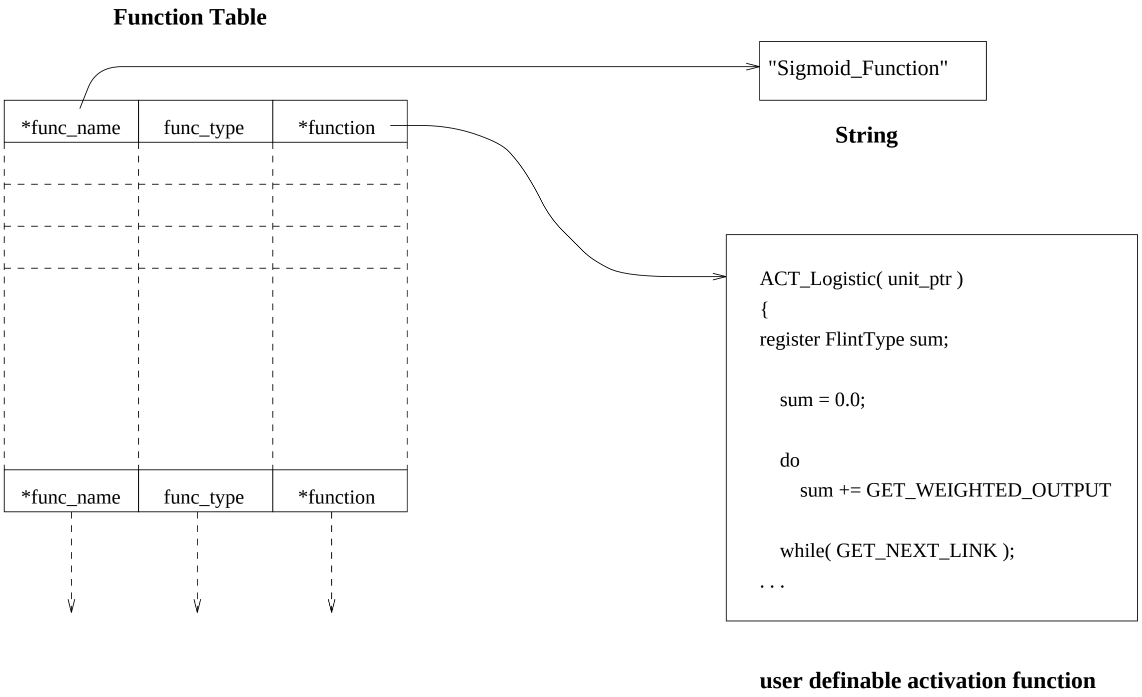

Figure 3.7: Function table

Chapter 4

Kernel Function Interface

4.1 Overview

The simulator kernel offers a variety of functions for the creation and manipulation of networks. These can roughly be grouped into the following categories:

- functions to manipulate the network
- functions to determine the structure of the network
- functions to define and manipulate cell prototypes
- functions to propagate the network
- learning functions
- functions to manipulate patterns
- functions to load and save the network and pattern files
- functions for error treatment, search functions for names, functions to change default values etc.

The following paragraphs explains the interface functions in detail. All functions of this interface between the kernel and the user interface carry the prefix `krui_`...(kernel user interface functions).

Additionally there are some interface functions which are useful to build applications for ART networks. These functions carry the prefix `artui_`...(ART user interface functions).

4.2 Unit Functions

The following functions are available for manipulation of the cells and their components:

```
krui_getNoOfUnits()
krui_getNoOfSpecialUnits()
```

```
krui_getFirstUnit()
krui_getNextUnit()
krui_setCurrentUnit( int  UnitNo )
krui_getCurrentUnit()
krui_getUnitName( int  UnitNo )
krui_setUnitName( int  UnitNo, char *unit_name )
krui_searchUnitName( char  *unit_name )
krui_searchNextUnitName( void )
krui_getNoOfTTypeUnits()
krui_getUnitOutFuncName( int  UnitNo )
krui_setUnitOutFunc( int  UnitNo, char *unitOutFuncName )
krui_getUnitActFuncName( int  UnitNo )
krui_setUnitActFunc( int  UnitNo, char *unitActFuncName )
krui_getUnitFTypeName( int UnitNo )
krui_getUnitActivation( int  UnitNo )
krui_setUnitActivation( int  UnitNo, FlintType  unit_activation )
krui_getUnitInitialActivation( int  UnitNo )
krui_setUnitInitialActivation( int  UnitNo, FlintType unit_i_activation)
krui_getUnitOutput( int  UnitNo )
krui_setUnitOutput( int  UnitNo, FlintType  unit_output )
krui_getUnitBias( int  UnitNo )
krui_setUnitBias( int  UnitNo, FlintType  unit_bias )
krui_getUnitSubnetNo( int UnitNo )
krui_setUnitSubnetNo( int UnitNo, int  subnet_no)
krui_getUnitLayerNo( int UnitNo )
krui_setUnitLayerNo( int UnitNo, unsigned short layer_bitField )
krui_getUnitPosition( int  UnitNo, struct PosType  *position )
krui_setUnitPosition( int  UnitNo, struct PosType  *position )
krui_getUnitNoAtPosition( struct PosType  *position, int subnet_no )
krui_getUnitNoNearPosition( struct PosType  *position, int subnet_no,
                            int range, int gridWidth )
krui_getXYTransTable( struct TransTable  * *xy_trans_tbl_ptr )
krui_getUnitCenters( int unit_no, int center_no,
                     struct PositionVector  * *unit_center )
krui_setUnitCenters( int unit_no, int center_no,
                     struct PositionVector  *unit_center )
krui_getUnitTType( int  UnitNo )
krui_setUnitTType( int  UnitNo, int  UnitTType )
krui_freezeUnit( int  UnitNo )
krui_unfreezeUnit( int  UnitNo )
krui_isUnitFrozen( int  UnitNo )
krui_getUnitInputType( UnitNo )
krui_getUnitValueA(int UnitNo)
krui_setUnitValueA(int UnitNo, FlintTypeParam unit_valueA)
krui_createDefaultUnit()
krui_createUnit( char  *unit_name, char  *out_func_name,
                 char  *act_func_name, FlintType  act,
```

```
                 FlintType  i_act, FlintType  out,
                 FlintType  bias)
krui_createFTypeUnit( char  *FType_name)
krui_setUnitFType( int  UnitNo, char *FTypeName )
krui_copyUnit( int  UnitNo, int copy_mode )
krui_deleteUnitList( int no_of_units, int unit_list[] )
```

4.2.1 Unit Enquiry and Manipulation Functions

`int  krui_getNoOfUnits()`
determines the number of units in the neural net.

`int  krui_getNoOfSpecialUnits()`
determines the number of special units in the neural net.

`int  krui_getFirstUnit()`
Many interface functions refer to a current unit or site. `krui_getFirstUnit()` selects the (chronological) first unit of the network and makes it current. If this unit has sites, the chronological first site becomes current. The function returns 0 if no units are defined.

`int  krui_getNextUnit()`
selects the next unit in the net, as well as its first site (if present); returns 0 if no more units exist.

`krui_err  krui_setCurrentUnit( int  UnitNo )`
makes the unit with number *UnitNo* current unit; returns an error code if no unit with the specified number exists.

`int  krui_getCurrentUnit()`
determines the number of the current unit (0 if not defined)

`char     *krui_getUnitName( int  UnitNo )`
`krui_err  krui_setUnitName( int  UnitNo, char *unit_name )`
determines/sets the name of the unit. `krui_setUnitName` returns NULL if no unit with the specified number exists.

`int  krui_searchUnitName( char  *unit_name )`
searches for a unit with the given name. Returns the first unit number if a unit with the given name was found, 0 otherwise.

`int  krui_searchNextUnitName( void )`
searches for the next unit with the given name. Returns the next unit number if a unit with the given name was found, 0 otherwise. `krui_searchUnitName( unit_name )` has to be called before at least once, to confirm the unit name. Returns error code if no units are defined.

```
char  *krui_getUnitOutFuncName( int  UnitNo )
char  *krui_getUnitActFuncName( int  UnitNo )
```
determines the output function resp. activation function of the unit.

```
krui_err  krui_setUnitOutFunc( int  UnitNo, char *unitOutFuncName )
krui_err  krui_setUnitActFunc( int  UnitNo, char *unitActFuncName )
```
sets the output function resp. activation function of the unit. Returns an error code if the function name is unknown, i.e. if the name does not appear in the function table as output or activation function. The f-type of the unit is deleted.

```
char  *krui_getUnitFTypeName( int UnitNo )
```
yields the f-type of the unit; returns NULL if the unit has no prototype.

```
FlintType  krui_getUnitActivation( int UnitNo )
krui_err   krui_setUnitActivation( int UnitNo,
                                   FlintTypeParam unit_activation )
```
returns/sets the activation of the unit.

```
FlintType krui_getUnitInitialActivation( int  UnitNo )
void      krui_setUnitInitialActivation( int  UnitNo,
                                         FlintType unit_i_activation )
```
returns/sets the initial activation of the unit, i.e. the activation after loading the net. See also `krui_resetNet()`.

```
FlintType  krui_getUnitOutput( int  UnitNo )
krui_err   krui_setUnitOutput( int  unit_no, FlintTypeParam  unit_output )
```
returns/sets the output value of the unit.

```
FlintType  krui_getUnitBias( int  UnitNo )
void  krui_setUnitBias( int  UnitNo, FlintType  unit_bias )
```
returns/sets the bias (threshold) of the unit.

```
int   krui_getUnitSubnetNo( int UnitNo )
void  krui_setUnitSubnetNo( int UnitNo, int  subnet_no)
```
returns/sets the subnet number of the unit (the range of subnet numbers is -32736 to +32735).

```
unsigned short  krui_getUnitLayerNo( int UnitNo )
void            krui_setUnitLayerNo( int UnitNo,int layer_no)
```
returns/sets the layer number (16 Bit integer).

```
void  krui_getUnitPosition( int  UnitNo, struct PosType  *position )
void  krui_setUnitPosition( int  UnitNo, struct PosType  *position )
```
determines/sets the (graphical) position of the unit. See also include file `glob_typ.h` for the definition of PosType.

```
int  krui_getUnitNoAtPosition( struct PosType  *position, int subnet_no )
```
yields the unit number of a unit with the given position and subnet number; returns 0 if no such unit exists.

```
int  krui_getUnitNoNearPosition( struct PosType  *position,
                                 int subnet_no, int range,
                                 int gridWidth )
```
yields a unit in the surrounding (defined by *range*) of the given position with the given graphic resolution *gridWidth*; otherwise like `krui_getUnitNoAtPosition(...)`.

```
krui_err krui_getUnitCenters( int unit_no, int center_no,
                              struct PositionVector  * *unit_center )
```
returns the 3D-transformation center of the specified unit and center number. Function has no effect on the current unit. Returns error number if unit or center number is invalid or if the SNNS-kernel isn't a 3D-kernel.

```
krui_err krui_setUnitCenters( int unit_no, int center_no,
                              struct PositionVector  *unit_center )
```
sets the 3D-transformation center and center number of the specified unit. Function has no effect on the current unit. Returns error number if unit or center number is invalid or if the SNNS-kernel isn't a 3D-kernel.

```
krui_err krui_getXYTransTable( dummy )
```
returns the base address of the XY-translation table. Returns error code if the SNNS-kernel isn't a 3D-kernel.

```
int       krui_getUnitTType( int  UnitNo )
krui_err  krui_setUnitTType( int  UnitNo, int  UnitTType )
```
gets/sets the IO-type[1] (i.e. input, output, hidden) of the unit. See include file `glob_typ.h` for IO-type constants. `Set` yields an error code if the IO-type is invalid.

```
krui_err  krui_freezeUnit( int  unit_no )
```
freezes the output and the activation value of the unit, i.e. these values are not updated anymore.

```
krui_err  krui_unfreezeUnit( int  unit_no )
```
switches the computation of output and activation values on again.

```
bool  krui_isUnitFrozen( int  unit_no )
```
yields TRUE if the unit is frozen, else FALSE.

```
int  krui_getUnitInputType( UnitNo )
```
yields the input type. There are three kinds of input types:

`NO_INPUTS`: the unit doesn't have inputs (yet).
`SITES`: the unit has one or more sites (and therefore no direct inputs).
`DIRECT_LINKS`: the unit has direct inputs (and no sites).

See also file `glob_typ.h`.

```
FlintType krui_getUnitValueA(int UnitNo)
void      krui_setUnitValueA(int UnitNo, FlintTypeParam unit_valueA)
```
returns and sets the Value A field of the unit structure.

[1]The term T-type was changed to IO-type after completion of the kernel

4.2.2 Unit Definition Functions

`int krui_createDefaultUnit()`
creates a unit with the properties of the (definable) default values of the kernel. The default unit has the following properties:

- standard activation and output function
- standard activation and bias
- standard position-, subnet-, and layer number
- default IO type
- no unit prototype
- no sites
- no inputs or outputs
- no unit name

Returns the number of the new unit or a (negative) error code. See also include file `kr_def.h`.

```
int  krui_createUnit( char  *unit_name, char  *out_func_name,
                      char  *act_func_name, FlintTypeParam i_act,
                      FlintTypeParam  bias)
```
creates a unit with selectable properties; otherwise like `krui_createDefaultUnit()`. There are the following defaults:

- standard position-, subnet-, and layer number
- default IO type
- no unit prototype
- no sites
- no inputs or outputs

Returns the number of the new unit or a (negative) error code. See also include file `kr_def.h`.

`int krui_createFTypeUnit( char *FType_name)`
creates a unit with the properties of the (previously defined) prototype. It has the following default properties:

- standard position number, subnet number and layer number
- no inputs or outputs

The function returns the number of the new unit or a (negative) error code.

`krui_err krui_setUnitFType( int UnitNo, char *FTypeName )`
changes the structure of the unit to the intersection of the current type of the unit with the prototype; returns an error code if this operation has been failed.

`int krui_copyUnit( int UnitNo, int copy_mode)`
copies a unit according to the copy mode. Four different copy modes are available:

- copy unit with all input and output connections
- copy only input connections

- copy only output connections
- copy only the unit, no connections

Returns the number of the new unit or a (negative) error code. See `glob_typ.h` for reference of the definition of constants for the copy modes.

`krui_err krui_deleteUnitList( int no_of_units, int unit_list[] )`
deletes 'no_of_units' from the network. The numbers of the units that have to be deleted are listed up in an array of integers beginning with index 0. This array is passed to parameter 'unit_list'. Removes all links to and from these units.

4.3 Site Functions

Before input functions (sites) can be set for units, they first have to be defined. To define it, each site is assigned a name by the user. Sites can be selected by using this name. For the definition of sites, the following functions are available:

```
krui_createSiteTableEntry( char  *site_name, char  *site_func )
krui_changeSiteTableEntry( char  *old_site_name, char  *new_site_name,
                           char  *new_site_func )
krui_deleteSiteTableEntry( char *site_name )
krui_getFirstSiteTableEntry( char * *site_name, char * *site_func )
krui_getNextSiteTableEntry( char * *site_name, char * *site_func )
krui_getSiteTableFuncName( char  *site_name )
krui_setFirstSite( void )
krui_setNextSite( void )
krui_setSite( char  *site_name )
krui_getSiteValue()
krui_getSiteName()
krui_setSiteName( char  *site_name )
krui_getSiteFuncName()
krui_addSite( char *site_name )
krui_deleteSite()
```

4.3.1 Functions for the Definition of Sites

`krui_err  krui_createSiteTableEntry( char  *site_name, char  *site_func )`
defines the correspondence between site function and name of the site. Error codes are generated for site names already used, invalid site functions, or problems with the memory allocation.

```
krui_err  krui_changeSiteTableEntry( char  *old_site_name,
                                     char  *new_site_name,
                                     char  *new_site_func )
```

changes the correspondence between site function and name of the site. All sites in the network with the name `old_site_name` change their name and function. Error codes are generated for already defined site names, invalid new site function, or problems with the memory allocation.

`krui_err  krui_deleteSiteTableEntry( char *site_name )`
deletes a site in the site table. This is possible only if there exist no sites in the network with that name. Returns an error code if there are still sites with this name in the net.

`bool  krui_getFirstSiteTableEntry( char * *site_name, char * *site_func )`
`bool  krui_getNextSiteTableEntry ( char * *site_name, char * *site_func )`
returns the first/next pair of site name and site function. The return code is TRUE if there is (still) an entry in the site table, else FALSE.

`char  *krui_getSiteTableFuncName( char  *site_name )`
returns the name of the site function assigned to the site. If no site with this name exists, a pointer to NULL is returned.

4.3.2 Functions for the Manipulation of Sites

`bool  krui_setFirstSite( void )`
initializes the first site of the current unit, i.e. the first site of the current unit becomes current site. If the current unit doesn't have sites, FALSE is returned, else TRUE.

`bool  krui_setNextSite( void )`
initializes the next site of the current unit. If the unit doesn't have more sites, FALSE is returned.

`krui_err  krui_setSite( char  *site_name )`
initializes the given site of the current unit. An error code is generated if the unit doesn't have sites, the site name is invalid, or the unit doesn't have a site with that name.

`FlintType  krui_getSiteValue()`
`char       *krui_getSiteFuncName()`
returns the name/value of the site function of the current site.

`char  *krui_getSiteName()`
returns the name of the current site.

`krui_err  krui_setSiteName( char  *site_name )`
changes the name (and thereby also the site function) of the current site. An error code is returned if the site name is unknown. The f-type of the unit is erased.

`krui_err krui_addSite( char *site_name )`
adds a new site to the current unit. The new site is inserted in front, i.e. it becomes the first site of the unit. Therefore it is possible to make the new site current by a call to `krui_setFirstSite()`. `krui_addSite(...)` has no effect on the current site! Error codes are generated if the unit has direct input connections, the site name is invalid, or problems with the memory allocation occurred. The functionality type of the unit will be cleared.

`bool krui_deleteSite()`
deletes the current site of the current unit and all input connections to that site. The functionality type of the unit is also erased. `krui_setFirstSite()` or `krui_setNextSite()` has to be called before at least once, to confirm the current site/unit. After the deletion the next available site becomes current. The return code is TRUE if further sites exist, else FALSE. The following program is sufficient to delete all sites of a unit:

```
if ( krui_setFirstSite() )
  while ( krui_deleteSite() ) { }
```

4.4 Link Functions

The following functions are available to define or determine the topology of the network:

```
krui_getFirstPredUnit( FlintType  *strength )
krui_getNextPredUnit( FlintType   *strength )
krui_getCurrentPredUnit( FlintType  *strength )
krui_getFirstSuccUnit( int  UnitNo, FlintType  *strength )
krui_getNextSuccUnit( FlintType  *strength )
krui_isConnected( int  source_unit_no )
krui_areConnected( int source_unit_no, int target_unit_no,
                   FlintType  *weight )
krui_getLinkWeight()
krui_setLinkWeight( FlintTypeParam  weight )
krui_createLink( int source_unit_no, FlintTypeParam  weight )
krui_deleteLink()
krui_deleteAllInputLinks()
krui_deleteAllOutputLinks()
krui_jogWeights( FlintTypeParam  minus, FlintTypeParam  plus)
```

`int krui_getFirstPredUnit( FlintType *strength )`
determines the unit number of the predecessor unit of the current unit and site; returns 0 if no such unit exists, i.e. if the current unit has no inputs. If a predecessor unit exists, the connection between the two units becomes current and its strength is returned.

`int krui_getNextPredUnit( FlintType *strength )`
gets another predecessor unit of the current unit/site (returns 0 if no more exist). Otherwise like `krui_getFirstPredUnit(...)`.

`int krui_getCurrentPredUnit( FlintType *strength )`
yields the current predecessor unit.

`int krui_getFirstSuccUnit( int UnitNo, FlintType *strength )`
yields unit number and connection strength of the first successor unit to the current unit. The return code is 0 if no such unit exists, i.e. the current unit has no outputs. If a successor unit exists, the connection between the two units becomes current. If the successor unit has sites, the site connected with this link becomes current site. The function is slow, because

the units are connected only backwards (lookup time is proportional to the number of connections in the net).

`int krui_getNextSuccUnit( FlintType *strength )`
gets another successor unit of the current unit (returns 0 if no other successors exist). Otherwise like `krui_getFirstSuccUnit(...)`. The function is slow, because the units are only connected backwards.

`bool krui_isConnected( int source_unit_no )`
checks, whether there is a connection between the current unit and the source unit. If this is true, this link becomes current and TRUE is returned.

```
bool  krui_areConnected( int source_unit_no, int target_unit_no,
                         FlintType  *weight )
```
checks, whether there is a connection between the source unit and the target unit. In contrast to `krui_isConnected(...)` this function traverses sites during the search. If there is such a connection, this link becomes current and TRUE is returned.

```
FlintType  krui_getLinkWeight()
void       krui_setLinkWeight( FlintTypeParam  weight )
```
determines/sets the connection weight of the current link.

`krui_err krui_createLink( int source_unit_no, FlintTypeParam weight )`
creates a new link between the current unit/site and the source unit. An error code is generated if a link between these two units already exists, or if the source unit does not exist.

`krui_err krui_deleteLink()`
deletes the current link. To delete a connection between the current unit/site and the source unit a sequence of `krui_isConnected( source_unit_no )` and `krui_deleteLink()` is ideal.

```
krui_err  krui_deleteAllInputLinks()
krui_err  krui_deleteAllOutputLinks()
```
deletes all inputs/outputs at the current unit/site.

`void krui_jogWeights( FlintTypeParam minus, FlintTypeParam plus)`
adds uniform distributed random values to the connection weights of the network. Minus must be less then plus. See also `krui_setSeedNo(...)`.

4.5 Functions for the Manipulation of Prototypes

By describing the characteristic properties of units, like activation/output function and sites, the user can define unit prototypes (called f-types in SNNS). Thereby the user can create a library of units. It is a big advantage that each change to the prototypes in the library affects all units of this f-type in the whole network. This means, that all units of a certain type are updated with a change in the library. With the following functions prototypes can be defined and manipulated:

```
krui_setFirstFTypeEntry()
krui_setNextFTypeEntry()
krui_setFTypeEntry( char  *Ftype_symbol )
krui_getFTypeName()
krui_setFTypeName( char  *unitFType_name )
krui_getFTypeActFuncName()
krui_setFTypeActFunc( char  *act_func_name )
krui_getFTypeOutFuncName()
krui_setFTypeOutFunc( char  *out_func_name )
krui_setFirstFTypeSite()
krui_setNextFTypeSite()
krui_getFTypeSiteName()
krui_setFTypeSiteName( char  *FType_site_name )
krui_createFTypeEntry( char *FType_symbol,char *act_func,
                       char *out_func, int no_of_sites,
                       char * *array_of_site_names )
krui_deleteFTypeEntry( char *  FType_symbol )
```

```
bool  krui_setFirstFTypeEntry()
bool  krui_setNextFTypeEntry()
```
initializes the first/next prototype and makes it current. The return code is FALSE if no unit types are defined, otherwise TRUE.

```
bool  krui_setFTypeEntry( char *Ftype_symbol )
```
selects a prototype by a name and returns TRUE if the name exists.

```
char  *krui_getFTypeName()
```
determines the name of the current prototype.

```
krui_err  krui_setFTypeName( char  *unitFType_name )
```
changes the name of the current prototype. The name has to be unambiguous, i.e. all names have to be different. If the name is ambiguous, or if memory allocation failed, an error code is returned.

```
char  *krui_getFTypeActFuncName()
```
determines the name of the activation function of the current prototype.

```
krui_err  krui_setFTypeActFunc( char * act_func_name )
```
changes the activation function of the current prototype; returns an error code if the given function is not a valid activation function. All units of the net that are derived from this prototype change their activation function.

```
char  *krui_getFTypeOutFuncName()
```
determines the name of the output function of the current prototype.

`krui_err krui_setFTypeOutFunc( char *out_func_name )`
changes the output function of the current prototype; returns an error code if the given function is not a valid output function. All units of the net that are derived from this prototype change their output function.

`bool krui_setFirstFTypeSite()`
selects the first site of the prototype (this site becomes current prototype site); returns TRUE if the prototype has sites.

`bool krui_setNextFTypeSite()`
selects the next site of the prototype (this site becomes current prototype site); returns TRUE if the prototype has more sites.

`char *krui_getFTypeSiteName()`
determines the name of the current prototype site.

`krui_err krui_setFTypeSiteName( char *FType_site_name )`
changes the name (and also the site function) of the current prototype site. All units of the net that are derived from this prototype change their site names or site functions. An error code is generated if the new site name is not yet defined.

```
krui_err krui_createFTypeEntry( char *FType_symbol,char *act_func,
                                char *out_func, int no_of_sites,
                                char * *array_of_site_names )
```
defines the prototype of a unit. If the prototype is supposed to have sites, i.e. `no_of_sites > 0`, an array of pointers to strings is needed to define the sites. These pointers must point to strings containing the names of the sites. The number of pointers in the arrays must match `no_of_sites`. An error code is generated if the site names are ambiguous, one or several site names are unknown, the prototype name is ambiguous, or a memory allocation error occurred.

`void krui_deleteFTypeEntry( char *FType_symbol )`
deletes the specified prototype. If there are still units in the net which are derived from this prototype, they loose their unit type.

4.6 Functions to Read the Function Table

`int krui_getNoOfFunctions()`
determines the number of available functions (unit, site, and learning functions).

```
void krui_getFuncInfo( int func_no,
                       char * *func_name,
                       int *func_type )
```
determines the name and type (activation, output, site, or learning function) from the function table. See include file `glob_typ.h` for the definition of the function types.

`bool krui_isFunction( char *func_name, int func_type )`
returns TRUE if the specified function is a valid function.

```
bool krui_getFuncParamInfo( char *func_name, int  func_type,
                            int *no_of_input_params,
                            int *no_of_output_params )
```
returns the number of input and output parameters of the given learning, update or initialization function. Returns TRUE if the given function exists, FALSE otherwise.

4.7 Network Initialization Functions

`krui_err  krui_setInitialisationFunc( char  *init_func )`
changes the initialization function; returns an error code if the initialization function is unknown.

`char  *krui_getInitialisationFunc( void )`
returns the current initialization function. The default initialization function is *'Randomize_Weights'* (see also `kr_def.h`).

`krui_err  krui_initializeNet( float *parameterArray, int  NoOfParams )`
initializes the network with the current initialization function.

4.8 Functions for Activation Propagation in the Network

`krui_err  krui_updateSingleUnit( int  UnitNo )`
evaluates the net input, the activation and the output value of the specified unit; returns an error code if the unit doesn't exist. `krui_updateSingleUnit(...)` also evaluates 'frozen' units.

`char  *krui_getUpdateFunc( void )`
returns the current update function. The default update function is *'Serial_Order'* (see also `kr_def.h`).

`krui_err  krui_setUpdateFunc( char  *update_func )`
Changes the current update function; returns an error code if the update function is invalid.

`krui_err  krui_updateNet( float *parameterArray, int  NoOfParams )`
updates the network according to the update function. The network should be a feed-forward type if one wants to update the network with the topological update function, otherwise the function returns a warning message. To propagate a pattern through the network the use of the following function calls is recommended:

```
krui_setPatternNo( pat_no );
krui_showPattern( OUTPUT_NOTHING );
krui_updateNet( parameterArray, NoOfParams );
```

See also `krui_setSeedNo` for initializing the pseudo random number generator. The function returns an error code if an error occurred. The following update functions are available:

- synchronous firing: the units of the network all change their activation at the same time.
- chronological order: the user defines the order in which the units change their activations.
- random order: the activations are computed in random order. It may happen that some units are updated several times while others are not updated at all.
- random permutation: the activations of all units are computed exactly once, but in a random order.
- topological order: the units change their activations according to their topological order. This mode should be selected only with nets that are free of cycles (feedforward nets).

The topological order propagation method computes the stable activation pattern of the net in just one cycle. It is therefore the method of choice in cycle free nets. In other modes, depending upon the number of layers in the network, several cycles are required to reach a stable activation pattern (if this is possible at all).

4.9 Learning and Pruning Functions

```
krui_err  krui_setLearnFunc( char *learning_function)
krui_err  krui_setFFLearnFunc( char *FF_learning_function)
krui_err  krui_setPrunnFunc( char *pruning_function)
```
selects the learning / subordinate learning / pruning function; returns an error code if the given function is unknown.

```
char  *krui_getLearnFunc( void )
char  *krui_getFFLearnFunc( void )
char  *krui_getPrunFunc( void )
```
returns the name of the current learning / subordinate learning / pruning function. The default learning / subordinate learning function is *'Std_Backpropagation'*, the default pruning function is *'MagPruning'* (see also `kr_def.h`).

```
krui_err  krui_learnAllPatterns( float *parameterInArray,
                                 int  NoOfInParams,
                                 float  * *parameterOutArray,
                                 int *NoOfOutParams )
krui_err  krui_learnAllPatternsFF( float *parameterInArray,
                                   int  NoOfInParams,
                                   float  * *parameterOutArray,
                                   int *NoOfOutParams )
```
learns all patterns (each consisting of an input/output pair) using the current learning /

subordinate learning function. `parameterInArray` contains the learning parameter(s). `NoOfInParams` stores the number of learning parameters. `parameterOutArray` returns the results of the learning function. This array is a static array defined in the learning function. `*NoOfOutParams` points to an integer value that contains the number of output parameters from the current learning function. The function returns an error code if memory allocation has failed or if the learning has failed. Patterns must be loaded before calling this function.

```
krui_err  krui_learnSinglePattern( int  pattern_no,
                                   float   *parameterInArray,
                                   int NoOfInParams,
                                   float  * *parameterOutArray,
                                   int *NoOfOutParams )
krui_err  krui_learnSinglePatternFF( int  pattern_no,
                                     float   *parameterInArray,
                                     int NoOfInParams,
                                     float  * *parameterOutArray,
                                     int *NoOfOutParams )
```

same as `krui_learnAllPatterns( ... )` or `krui_learnAllPatternsFF( ... )` respectively, but teaches only the current pattern.

4.10 Functions for the Manipulation of Patterns

`krui_err  krui_setPatternNo( int  patter_no )`
sets the current pattern; returns an error code if the pattern number is invalid.

`krui_err  krui_deletePattern( void )`
deletes the current pattern.

`krui_err  krui_modifyPattern( void )`
modifies the current pattern. Sets the pattern to the current activation of the units.

`krui_err  krui_showPattern( int  mode )`
outputs a pattern on the activation or output values of the input/output units. The following modes are possible:

- `OUTPUT_NOTHING`: stores the input pattern in the activation of the input units.
- `OUTPUT_ACT`: like `OUTPUT_NOTHING`, but stores also the output pattern in the activation of the output units.
- `OUTPUT_OUT`: like `OUTPUT_ACT`, additionally a new output value of the output units is computed.

`krui_showPattern(...)`
draws pattern on the display. Generates an error code if the number of input and output units does not correspond with the previously loaded pattern. The constants of the various modes are defined in `glob_typ.h`.

`krui_err  krui_newPattern( void )`
creates a new pattern (an input/output pair). A pattern can be created by modifying the activation value of the input/output units. The function returns an error code if there is insufficient memory or the number of input/output units is incompatible with the pattern.

Note: `krui_newPattern()` switches pattern shuffling off. For shuffling the new patterns call:

```
krui_newPattern(...)
krui_shufflePatterns( TRUE )
```

`void  krui_deleteAllPatterns()`
deletes all previously defined patterns in main memory.

`krui_err krui_shufflePatterns( bool on_or_off )`
shuffles the order of the patterns if `on_or_off` is true. If `on_or_off` is false the original order can be restored. See also `krui_setSeedNo(...)`.

`krui_err  krui_shuffleSubPatterns(bool on_or_off)`
shuffles sub pattern pairs by using pseudo random generator.
`krui_shuffleSubPatterns(TRUE)` switches shuffeling of sub patterns on,
`krui_shuffleSubPatterns(FALSE)` switches shuffeling of sub patterns off.
The default presetting is `krui_shuffleSubPatterns (FALSE)`.

`int  krui_getNoOfPatterns( void )`
returns the actual number of patterns (0 if no patterns have been loaded).

`krui_err  krui_allocNewPatternSet(int *set_no)`
returns the number of the allocated pattern set. In case of an error the error code will be returned.

`krui_err krui_setCurrPatSet(int number)`
chooses the number of the current pattern set. The number ranges from 0 to n-1.

`krui_err  krui_deletePatSet(int number)`
deletes all patterns of the pattern set with the given number.

```
krui_err  krui_GetPatInfo(pattern_set_info *set_info,
                          pattern_descriptor *pat_info)
```
gets all available information concerning the current pattern set and the current pattern.

```
krui_err  kruiDefShowSubPat(int *insize,int *outsize,int,
                            int *inpos,int *outpos)
```
defines the sub pattern that will be shown with the next call of `krui_showPattern`.

```
krui_err  krui_DefTrainSubPat(int *insize,int *outsize,int *instep,
                              int *outstep, int *max_n_pos)
```
defines how sub pattern should be generated during training. `krui_DefTrainSubPat()` has to be called before any training can take place.

```
krui_err   krui_AlignSubPat(int *inpos,int *outpos,int *no)
```
aligns the given sub pattern position to a valid position which fits the defined sub pattern training scheme (`krui_DefTrainSubPat`).

```
krui_err  krui_GetShapeOfSubPattern(int *insize,int *outsize,int *inpos,
                                    int *outpos,int *n_pos)
```
gets the shape of the sub pattern by using the current set,current pattern and the current train scheme (defined with `krui_DefTrainSubPat`).

4.11 File I/O Functions

```
krui_err  krui_loadNet( char  *filename, char  * *netname )
krui_err  krui_saveNet( char  *filename, char  *netname)
```
loads/saves a network from/to disk and generates an internal network structure.

```
krui_err krui_loadNewPatterns(char *filename,int *set_no)
krui_err  krui_saveNewPatterns(char *filename,int set_no)
```
saves and loads a pattern file with new style conventions.
The pattern set with the given number will be saved and loaded.

```
krui_err krui_saveResultParam(char *filename,bool create,int startpattern,
                      int endpattern,bool includeinput,bool includeoutput,
                      float *Update_param_array, int NoOfUpdateParam)
```
Saves the network result which depends on the loaded patterns. If create is false,the new file will be appended to an existing file. Startpattern and endpattern determines the range of patterns to use. The input patterns and the teaching output patterns can be included.

An error code is returned if an error occurred during the execution of the I/O functions.

4.12 Functions to Search the Symbol Table

```
bool  krui_getFirstSymbolTableEntry( char * *symbol_name,
                                    int *symbol_type )
bool  krui_getNextSymbolTableEntry( char * *symbol_name,
                                    int *symbol_type )
```
determines the name and type of the first/next entry in the symbol table and returns TRUE if another entry exists.

```
bool  krui_symbolSearch( char *symbol, int  symbol_type)
```
returns TRUE if the given symbol exists in the symbol table.

4.13 Miscelaneous other Interface Functions

```
char  *krui_getVersion()
```
determines the version number of the SNNS kernel.

```
void  krui_getNetInfo( int  *no_of_sites, int  *no_of_links,
                       int  *no_of_STable_entries,
                       int  *no_of_FTable_entries )
```
gathers various information about the network.

```
void  krui_getUnitDefaults( FlintType  *act, FlintType  *bias,
                            int *io_type, int *subnet_no, int *layer_no,
                            char * *act_func, char * *out_func)
```
determines the default values for generating units. See also `krui_createDefaultUnit()` and `krui_createFTypeUnit(...)`.

```
krui_err  krui_setUnitDefaults( FlintTypeParam act, FlintTypeParam bias,
                                int io_type, int subnet_no, int layer_no,
                                char *act_func, char *out_func)
```
changes the default values; returns an error code if the IO-type or the activation/output function is unknown.

```
void  krui_setSeedNo( long seed )
```
initializes the random number generator. If `seed = 0`, the random number generator is re-initialized (this time really at random!).

```
int  krui_getNoOfInputUnits()
int  krui_getNoOfOutputUnits()
```
return the number of input/output units

```
int  krui_getNoOfSpecialInputUnits()
int  krui_getNoOfSpecialOutputUnits()
```
return the number of special input/output units

```
void  krui_resetNet()
```
sets the activation values of all units to their respective defaults.

4.14 Memory Management Functions

```
krui_err  krui_allocateUnits( int  number )
```
reserves <number> units in memory. Additional units can be requested by multiple calls to that function. This function doesn't have to be called, since the SNNS kernel always reserves enough memory for units, sites and links. If a large amount of units is needed, however, a call to `krui_allocateUnits(...)` eases the administration of system resources. If `krui_allocateUnits(...)` is never called, units are always requested in blocks of size <UNIT_BLOCK>. See also `kr_def.h`.

```
void krui_getMemoryManagerInfo(int *unit_bytes, int *site_bytes,
                               int *link_bytes, int *NTable_bytes,
                               int *STable_bytes, int *FTable_bytes)
```
determines the number of allocated (not the number of used) bytes for the various components.

`void  krui_deleteNet()`
deletes the network and frees the whole memory used for the representation of the data structures.

4.15 ART Interface Functions

The functions described in this paragraph are only useful if you use one of the ART models ART1, ART2 or ARTMAP. They are additional functions, not replacing any of the kernel interface functions described above. To use them, you have to include the files `art_ui.h` and `art_typ.h` in your application.

`krui_err artui_getClassifiedStatus( art_cl_status *status )`
returns the actual classification state of an ART network in parameter `status`. Type `art_cl_status` is described above. An SNNS error code is returned as function value. The function can be used for ART1, ART2 and ARTMAP models.

`krui_err artui_getClassNo( int *class_no )`
returns a number between 1 and M in parameter `class_no`, which is either the index of the actual winner unit in the F_2 layer of an ART1 or ART2 network or the one of the activated MAP unit in an ARTMAP network. It will return -1 as `class_no` if no actual class is active. An SNNS error code is returned as function value.

`krui_err  artui_getN (int *N )`
determines the number of F1-units in an ART1 or ART2 network.

`krui_err  artui_getM (int *M )`
determines the number of F2-units in an ART1 or ART2 network.

`krui_err  artui_getNa (int *Na )`
determines the number of F1a-units in an ARTMAP network.

`krui_err  artui_getMa (int *Ma )`
determines the number of F2a-units in an ARTMAP network

`krui_err  artui_getNb (int *Nb)`
determines the number of F1b-units in an ARTMAP network.

`krui_err  artui_getMb (int *Mb)`
determines the number of F2b-units in an ARTMAP network.

4.16 Error Messages of the Simulator Kernel

Most interface functions return an error code if the parameters contradict each other, or if an error occurred during execution. If no error occurred during execution of a kernel interface function, the function returns the code: `KRERR_NO_ERROR`. `KRERR_NO_ERROR` is equal to 0. The simulator kernel can generate 54 different error messages. The error code constants are defined in `glob_typ.h`. There is also a function to translate an error code into text:

`char *krui_error( int error_code )`
converts an error code to a string. The following error messages are used:

```
KRERR_NO_ERROR:                 No Error
KRERR_INSUFFICIENT_MEM:         Insufficient memory
KRERR_UNIT_NO:                  Invalid unit number
KRERR_OUTFUNC:                  Invalid unit output function
KRERR_ACTFUNC:                  Invalid unit activation function
KRERR_SITEFUNC:                 Invalid site function
KRERR_CREATE_SITE:              Creation of sites is not permitted
                                because unit has direct input links
KRERR_ALREADY_CONNECTED:        Creation of a link is not permitted
                                because there exists already a link
                                between these units
KRERR_CRITICAL_MALLOC:          allocation failed in critical operation.
KRERR_FTYPE_NAME:               Ftype name is not definite
KRERR_FTYPE_ENTRY:              Current Ftype entry is not defined
KRERR_COPYMODE:                 Invalid copy mode
KRERR_NO_SITES:                 Current unit does not have sites
KRERR_FROZEN:                   Can't update unit because unit is frozen
KRERR_REDEF_SITE_NAME:          Redefinition of site name not permitted
                                (site name already exists)
KRERR_UNDEF_SITE_NAME:          Site name is not defined
KRERR_NOT_3D:                   Invalid Function: Not a 3D-Kernel
KRERR_DUPLICATED_SITE:          Unit has already a site with this name
KRERR_INUSE_SITE:               Can not delete site table entry because
                                site is in use
KRERR_FTYPE_SITE:               Current Ftype site is not defined
KRERR_FTYPE_SYMBOL:             Given symbol is not defined in the symbol
                                table
KRERR_IO:                       Physical I/O error
KRERR_SAVE_LINE_LEN:            Creation of output file failed
                                (line length, limit exceeded)
KRERR_NET_DEPTH:                The depth of the network does not fit the
                                learning function
KRERR_NO_UNITS:                 No Units defined
KRERR_EOF:                      Unexpected EOF
KRERR_LINE_LENGTH:              Line length exceeded
```

KRERR_FILE_FORMAT:	Incompatible file format
KRERR_FILE_OPEN:	Can not open file
KRERR_FILE_SYNTAX:	Syntax error at line
KRERR_MALLOC1:	Memory allocation error 1
KRERR_TTYPE:	Topologic type invalid
KRERR_SYMBOL:	Symbol pattern invalid (must match [A-Za-z]^[|,]*)
KRERR_NO_SUCH_SITE:	Current unit does not have a site with this name
KRERR_NO_HIDDEN_UNITS:	No hidden units defined
KRERR_CYCLES:	cycle(s):
KRERR_DEAD_UNITS:	dead unit(s):
KRERR_INPUT_PATTERNS:	Pattern file contains not the same number of input units as the network
KRERR_OUTPUT_PATTERNS:	Pattern file contains not the same number of output units as the network
KRERR_CHANGED_I_UNITS:	Number of input units has changed
KRERR_CHANGED_O_UNITS:	Number of output units has changed
KRERR_NO_INPUT_UNITS:	No input units defined
KRERR_NO_OUTPUT_UNITS:	No output units defined
KRERR_NO_PATTERNS:	No patterns defined
KRERR_INCORE_PATTERNS:	Incore patterns incompatible with current network (remove loaded patterns before loading network)
KRERR_PATTERN_NO:	Invalid pattern number
KRERR_LEARNING_FUNC:	Invalid learning function
KRERR_PARAMETERS:	Invalid parameters
KRERR_UPDATE_FUNC:	Invalid update function
KRERR_INIT_FUNC:	Invalid initialization function
KRERR_DERIV_FUNC:	Derivation function of the activation function does not exist
KRERR_I_UNITS_CONNECT:	input unit(s) with input connections to other units
KRERR_O_UNITS_CONNECT:	output unit(s) with output connections to other units
KRERR_TOPOMODE:	Invalid topological sorting mode
KRERR_LEARNING_SITES:	Learning function doesn't support sites
KRERR_SITES_NO_SUPPORT:	Sites are not supported
KRERR_NO_MASPAR_KERNEL:	This isn't a MasPar Kernel
KRERR_NOT_NEIGHBOUR_LAYER:	Connection(s) between unit(s) in non-neighbour layers are not supported
KRERR_MUCH_LAYERS:	The number of layers is too high
KRERR_NOT_FULLY_CONNECTED:	The network layers aren't fully connected
KRERR_MODE_FF1_INVALID_OP:	This operation is not allowed in parallel kernel mode
KRERR_NET_TRANSFORM:	Change of network type isn't possible in parallel kernel mode

KRERR_NO_CURRENT_LINK:	No current link defined
KRERR_NO_CURRENT_UNIT:	No current unit defined
KRERR_UNIT_NO_INPUT:	Current unit does't have any inputs
KRERR_TOPO_DEFINITION:	Invalid parameter in topologic definition section
KRERR_BAD_CONNECTION:	Creation of link between these units is not permitted
KRERR_MASPAR_NO_RESPOND:	MasPar doesn't respond
KRERR_NOT_IMPLEMENTED_YET:	This function isn't implemented yet
KRERR_NOT_PARRALLEL_MODE:	Kernel isn't in parallel mode
KRERR_MASPAR_OUT_OF_PE_MEM:	MasPar ran out of memory
KRERR_MASPAR_COMMUNICATE:	MasPar communication error
KRERR_MASPAR_OUT_OF_PE:	MasPar ran out of processors
KRERR_MISSING_DEFAULT_FUNC:	Missing default function (check function table)
KRERR_MASPAR_MULTI_OUT_FUNC:	MasPar kernel does not support multiple unit output functions
KRERR_MASPAR_MULTI_ACT_FUNC:	MasPar kernel doesn't support multiple unit activation functions
KRERR_NET_DEPTH:	The depth of the network does not fit to the learning function
KRERR_NO_OF_UNITS_IN_LAYER:	Wrong no. of units in layer
KRERR_UNIT_MISSING:	Unit missing or not correctly connected
KRERR_UNDETERMINED_UNIT:	Unit does not belong to a defined layer in the network
KRERR_ACT_FUNC:	Unit has wrong activation function
KRERR_OUT_FUNC:	Unit has wrong output function
KRERR_SITE_FUNC:	Unexpected site function at unit
KRERR_UNEXPECTED_SITES:	Unit is not expected to have sites
KRERR_UNEXPECTED_DIRECT_INPUTS:	Unit is expected to have sites
KRERR_SITE_MISSING:	Site missing at unit
KRERR_UNEXPECTED_LINK:	Unexpected link
KRERR_LINK_MISSING:	Missing link(s) to unit
KRERR_LINK_TO_WRONG_SITE:	Link ends at wrong site of target unit
KRERR_TOPOLOGY:	The network is not fitting the required topology
KRERR_PARAM_BETA:	Wrong beta parameter in unit bias value
KRERR_CC_ERROR1:	Topo_ptr_array is sorted in the wrong way
KRERR_CC_ERROR2:	There is no memory allocated
KRERR_CC_ERROR3:	Not enough memory to run Casscade
KRERR_CC_ERROR4:	Hidden unit with more than one recurent link detected
KRERR_CC_ERROR5:	Hidden unit without recurrent link found
KRERR_CC_ERROR6:	Hidden layer does not fit the required topology
KRERR_CC_ERROR7:	The first link of a hidden unit is not a recurent link

KRERR_CC_ERROR8:	Special unit with more than one recurent link detected
KRERR_CC_ERROR9:	First link of a special unit is not a recurent link
KRERR_CC_ERROR10:	Wrong update function
KRERR_CC_ERROR11:	Wrong init function
DLVQ_ERROR1:	There are empty classes
DLVQ_ERROR2:	There is a class lower than zero
DLVQ_ERROR3:	Wrong number of output units.
DLVQ_ERROR4:	This network is not fitting the required topology
DLVQ_ERROR5:	There isn't a unit for every class
KRERR_NP_NO_MORE_ENTRIES:	No more free pattern sets available
KRERR_NP_NO_SUCH_PATTERN_SET:	No such Pattern set defined
KRERR_NP_CURRENT_PATTERN:	No current pattern defined
KRERR_NP_DIMENSION:	Specified sub pattern does not fit into pattern
KRERR_NP_NO_SUCH_PATTERN:	No such pattern available
KRERR_NP_NO_CURRENT_PATTERN_SET:	No current pattern set defined
KRERR_NP_DOES_NOT_FIT:	Pattern (sub pattern) does not fit the network
KRERR_NP_NO_TRAIN_SCHEME:	No sub pattern shifting scheme defined
KRERR_NP_NO_OUTPUT_PATTERN:	Pattern contains no output information
KRERR_NP_INCOMPATIBLE_NEW:	New Pattern does not fit to existing set
KRERR_NP_PARAM_ZERO:	Illegal parameter value <= 0 specified
KRERR_IP_ISNOTINITED:	Paragon kernel is not initialized
KRERR_IP_ISINITED:	Paragon kernel is already initialized
KRERR_IP_SENDMSGFAILED:	Sending a message fails
KRERR_IP_MSGSYNTAX:	Syntax error in received message
KRERR_IP_UNKNOWENCMD:	Receive unknown command
KRERR_IP_LESSPAT:	There are less patterns than allocated nodes
KRERR_IP_WEIGHTUPDATE:	Weight update with global sum fails
KRERR_IP_LEARNFNOTPAR:	Learning function is not parallelized

Chapter 5

Transfer Functions

5.1 Predefined Transfer Functions

The following site, activation, and output functions are already predefined. Future releases of the kernel will have additional transfer functions.

Site Functions:

Function	**Formula**
`Linear`	$net_j(t) = \sum_i w_{ij} o_i$
`Produkt`	$net_j(t) = \prod_i w_{ij} o_i$
`PI`	$net_j(t) = \prod_i o_i$
`Max`	$net_j(t) = \max_i w_{ij} o_i$
`Min`	$net_j(t) = \min_i w_{ij} o_i$
`At_least_2`	$net_j(t) = \begin{cases} 0 & \text{if } \sum_i w_{ij} o_i < 2 \\ 1 & \text{else.} \end{cases}$

Several other site functions have been implemented for the ART models in SNNS: `At_least_2`, `At_least_1`, `At_most_0`, `Reciprocal`. These functions normally are not useful for other networks. So they are mentioned here, but not described in detail. For more information please refer to the section about the corresponding ART model in SNNS.

Activation functions:

Function	Formula
`BAM`	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ a_j(t-1) & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) < 0 \end{cases}$
`BSB`	$a_j(t) = net_j(t) * \theta_j$
`Elliott`	$a_j(t) = \frac{net_j(t)+\theta_j}{1+\lvert net_j(t)+\theta_j \rvert}$
`Identity`	$a_j(t) = net_j(t)$
`IdentityPlusBias`	$a_j(t) = net_j(t) + \theta_j$
`Logistic`	$a_j(t) = \frac{1}{1+e^{-(net_j(t)+\theta_j)}}$
`Logistic_notInhibit`	like Logistic, but skip input from units named "Inhibit"
`Logistic_Tbl`	like Logistic, but with table lookup instead of computation
`MinOutPlusWeight`	$a_j(t) = \min_i(w_{ij} + o_i)$
`Perceptron`	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) \geq \theta_j \\ 0 & \text{if } net_j(t) < \theta_j \end{cases}$
`Product`	$a_j(t) = \prod_i w_{ij} o_i$
`RBF_Gaussian`	see the chapter about RBFs in the user manual
`RBF_MultiQuadratic`	see the chapter about RBFs in the user manual
`RBF_ThinPlateSpline`	see the chapter about RBFs in the user manual
`RM`	$a_j(t) = \begin{cases} 0.85 * a_j(t-1)+ \\ 0.15 * net_j(t)* \\ (1 - a_j(t-1)) & \text{if } net_j(t) > 0 \\ 0.85 * a_j(t-1)+ \\ 0.15 * net_j(t)* \\ (1 + a_j(t-1)) & \text{if } net_j(t) \leq 0 \end{cases}$
`Signum`	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ -1 & \text{if } net_j(t) \leq 0 \end{cases}$
`Signum0`	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) < 0 \end{cases}$
`StepFunc`	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) \leq 0 \end{cases}$
`TanH`	$a_j(t) = \tanh(net_j(t) + \theta_j)$
`TanH_Xdiv2`	$a_j(t) = \tanh(net_j(t)/2 + \theta_j)$

Several other activation functions have been implemented for the ART models in SNNS: `Less_than_0`, `At_most_0`, `At_least_2`, `At_least_1`, `Exactly_1`, `ART1_NC`, `ART2_Identity`, `ART2_Rec`, `ART2_NormP`, `ART2_NormV`, `ART2_NormW`, `ART2_NormIP`, `ART2_Rst`, `ARTMAP_NCa`, `ARTMAP_NCb`, `ARTMAP_DRho`.

These functions normally are not useful for other networks. So they are mentioned here, but not described in detail. For cascade correlation and time delay networks the following modified versions of regular activation functions have been implemented: `RCC_logistic`, `RCC_LogisticSym`, `RCC_Tanh`, `RCC_Linear`, `TD_Logistic`, `TD_Elliott`. They behave like the ordinary functions with the same name body.

Output Functions:

Function	Formula
`Identity`	$o_j(t) = a_j(t)$
`Clip_0_1`	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0 \\ a_j(t) & \text{if } a_j(t) < 1 \\ 1 & \text{if } a_j(t) \geq 1 \end{cases}$
`Clip_1_1`	$o_j(t) = \begin{cases} -1 & \text{if } a_j(t) \leq -1 \\ a_j(t) & \text{if } -1 < a_j(t) < 1 \\ 1 & \text{if } a_j(t) \geq 1 \end{cases}$
`Threshold_0.5`	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0.5 \\ 1 & \text{if } a_j(t) > 0.5 \end{cases}$

Two other output functions have been implemented for ART2 in SNNS: `ART2_Noise_PLin` and `ART2_Noise_ContDiff` These functions are only useful for the ART2 implementation So they are mentioned here, but not described in detail.

5.2 User Defined Transfer Functions

The group of transfer functions can be exented arbitrarily by the user. In order to make them available inside SNNS the following steps have to be performed:

1. In file "`.../SNNSv4.0/kernel/func_tbl.c` the name of the function has to be inserted in the function table. An example entry for an activation function would be

   "Act_MyFunction",ACT_FUNC, 0, 0, (FunctionPtr) MyFancyFunction,

   Notice that the second entry defines the type (activation, initialization, etc.) of the new function!

   If the new function is an activation function, the corresponding derivation function has also to be inserted in the function table. E.g.:
   "Act_MyFunction",ACT_DERIV_FUNC, 0, 0, (FunctionPtr)ACT_DERIV_MFF,

   This entry has to be given, even if no such derivation function exists in the mathematical sense. In that case `''ACT_DERIV_Dummy''` has to be specified as name of the derivation function.

   If the function exists, it has to be declared and implemented just as the activation function.

Please note, that activation and derivation function have to have the same name suffix (here: "MyFunction")!

2. The functions must be implemented as C programs in following files:

   activation functions in "`.../SNNSv4.0/kernel/trans_f.c`"
   output functions in "`.../SNNSv4.0/kernel/trans_f.c`"
   site functions in "`.../SNNSv4.0/kernel/trans_f.c`"
   initialization functions in "`.../SNNSv4.0/kernel/init_f.c`"
   learning functions in "`.../SNNSv4.0/kernel/learn_f.c`"
   update functions in "`.../SNNSv4.0/kernel/update_f.c`"

   The name of the implemented function has to match the name specified in the function table!

3. Function prototypes have to be defined in the corresponding header (.h) and private header files (.ph).

4. Go to the SNNSv4.0 root directory and build (using the command 'build') new executables. Remember that if you don't recompile, the modification to the kernel won't take any effect.

The new function should be available now in the user interface together with all predefined functions.

Chapter 6

Simulator Kernel Implementation

The simulator kernel alone, including comments, has a volume of about 72000 lines of source code with about 2.0 Mbyte. It is implemented in ANSI C. The simulator kernel consists of the following files:

Header files:

All source code files have corresponding public (.h) and private (.ph) header files associated with them. The public header files are used to make the exported functions of the modul accessible to the other moduls, while the private ones define function prototypes and variables exclusively global to the functions of the modul. Additionally there are the following header files without corresponding c-moduls.

Source file	**Meaning**
art_typ.h	global types for all ART moduls
cc_mac.h	macros for cc and rcc moduls
cc_typ.h	global types for all cc moduls
func_mac.h	macros for transfer functions
glob_typ.h	user interface data types
kr_const.h	constants of the kernel
kr_def.h	defaults for the kernel
kr_mac.h	macros of the kernel
kr_typ.h	data types of the kernel
krart_df.h	macros for the ART kernel functions
krui_typ.h	user interface functions
random.h	randomize functions for System V
version.h	version and patchlevel of the kernel and the kernel interfaces

Sourcecode files:

Source file	**Meaning**
art_ui.c	ART user interface functions
arttr_f.c	special transfer functions for ART models
calc_err.c	Calculates the error of a table-lookup function
cc_learn.c	learning functions for cascade correlation
cc_rcc.c	Common functions of CC and RCC
cc_rcc_topo.c	CC and RCC topology check
dlvq_learn.c	dynamic LVQ learning function routines
func_tbl.c	contains the function table
init_f.c	user definable initialisation functions
kernel.c	kernel low-level functions
kr_JordElm.c	functions for Jordan and Elman type networks
kr_amap.c	kernel low-level functions for ARTMAP model
kr_art.c	kernel low-level functions for ART models
kr_art1.c	kernel low-level functions for ART1 model
kr_art2.c	kernel low-level functions for ART2 models
kr_funcs.c	routines to handle the function table
kr_inversion.c	network inversion functions
kr_io.c	compiler-kernel interface (file I/O)
kr_mem.c	functions of the memory management
kr_newpattern.c	functions for the pattern handling
kr_pat_parse.c	the pattern parser
kr_pat_scan.c	the pattern scanner
kr_rand.c	random functions for MS-DOS implementation
kr_td.c	time delay learning functions
kr_ui.c	high level user interface functions
learn_f.c	user definable teaching functions
make_tbl.c	create a lookup table for transfer functions
matrix.c	matrix handling functions
prun_f.c	pruning functions
rcc_learn.c	learning functions for recurrent cascade correlation
strdup.c	contains the strdup(...) function (not in ULTRIX-32)
tbl_func.c	sigmoid activation function with table lookup
trans_f.c	user definable transfer functions (unit, site, and output functions)
update_f.c	user definable update functions

Example Networks:

Source file	Meaning
art1_letters.nett	ART-1 network
art2_tetra.net	ART-2 network
artmap.net	ARTMAP network
bdg_TDNN.net	Time Delay Network for bdg-sound recognition
bdg_TDNN_untrained.net	untrained version of bdg_TDNN.net
dlvq_ziff.net	digit recognition with DLVQ
dlvq_ziff_trained.net	trained version of dlvq_ziff.net
dlvq_ziff_untrained.net	untrained version of dlvq_ziff.net
eight_elman.net	Elman network to recognize a lying 8
eight_elman_untrained.net	untrained version of eight_elman.net
eight_jordan.net	Jordan network to recognize a lying 8
eight_jordan_untrained.net	untrained version of eight_jordan.net
encoder.net	8-3-8 encoder network
encoder_untrained.net	untrained version of encoder.net
font.net	16x16 character recognition network
font_untrained.net	untrained version of font.net
laser.net	Prediction of NH3 laser with Elman network
laser_untrained.net	untrained version of laser.net
letseq_elman.net	Letter sequence prediciton with Elman network
letseq_elman_untrained.net	untrained version of letseq_elman.net
letseq_h_elm.net	letseq_elman.net with an hierarchical net
letters.net	character recognition network
letters3D.net	character recognition network in 3D
letters_untrained.net	untrained version of letters.net
nettalk.net	nettalk network
patmat.net	toy pattern matching auto-associator network
patmat_untrained.net	untrained version of patmat.net
som_cube.net	SOM cube example
watch.net	image processing example
watch_2out.net	advanced image processing example
xor.net	XOR Network
xor_rec.net	recurrent XOR Network
xor_untrained.net	untrained version of xor.net

Probably there will be more example networks in the new SNNS release. See the files in the examples directory.

Chapter 7

Implementation of the User Interface

This chapter explaines important data types, functions, and the program structure of the graphical user interface. Definitions and data structures used by the program, are located not only in the file `glob_typ.h` of the SNNS kernel, but also in the header files of the X-window system and the Athena Toolkit. For reference of those definitions see the X11 documentation release 4.0 or 5.0.

How X, XGUI and the kernel cooperate is sketched in figure 7.1. The MIT Athena Toolkit was selected because of its widespread use. In this toolkit, so called *widgets* are defined (window elements like boxes, menu buttons, toggles, scrollbars and so forth). XGUI should work with all window mangers, but the following descriptions refer only to the *twm* window manager.

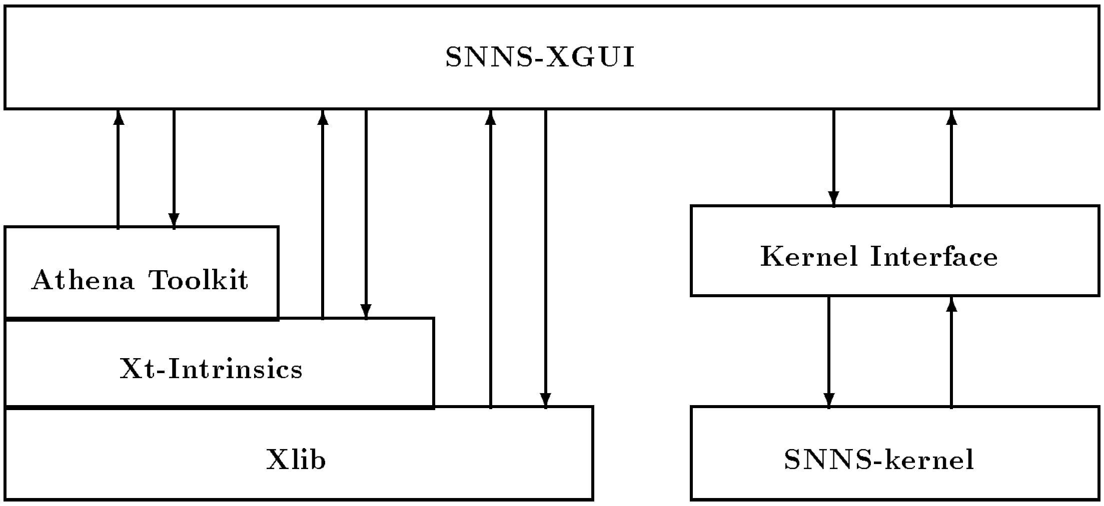

Figure 7.1: How X, XGUI and the simulator kernel cooperate

After opening of the first window, SNNS-XGUI transfers control to X by entering the main-event-dispatch-loop[1].To react to different events (e.g. mouse movement), X now calls predefined subroutines like callbacks and event handlers, which return control to X after completion. All outputs (e.g. line drawing) are buffered by the X server until it has resumed control. This asynchronous behaviour complicates debugging, since changes in the user interface are invisible at the time they are processed. The compile switch `DEBUG` forces the X server with calls to `XFLUSH()` to produce output at crucial points. The same result is obtained by specifying `-synchronous` as an argument in the call to XGUI. In both cases output speed decreases dramatically.

For windows which return a value, like the Layer window or the Confirmer, X must react directly to events. There, the Event-Dispatch-Loop is re-programmed where the exit condition is true after the call to a routine for a specific button (e.g. `DONE`). The next program segment can immediately use the result. The call to the confirmer demonstrates this very clearly:

```
if (ui_confirmYes(''Load will erase current network. Load?''))
    ui_file_loadNet(filename);
else
    ui_printMessage(''Loading aborted.'');
...
```

`ui_confirmYes()` builds the confirmer and dispatches all events until one of the buttons `YES` or `NO` is pressed. Then it returns this value as a boolean. Without this loop, the confirmer would become visible only after the processing of the code, and a default value would always be returned!

On processing the teaching cycles or the update steps, it is necessary to check for new events after each step. This is the only way to recognize the clicking of the `STOP` button in time. The *WorkProc* construct is a means for solving this problem. The routine for performing one step is declared as WorkProc with the statement `XtAppAddWorkProc(...)`. This routine is called as long as no events are waiting. When all cycles/steps are processed it returns `True`, thereby being removed. A soon as the button `STOP` is pressed, the corresponding callback routine `ui_rem_stopProc` removes the WorkProc with the command `XtRemoveWorkProc(ui_workProcId)`. A WorkProc, already existing when a new one is defined, gets deleted (see also [You89], p. 136ff). This also explains, why the starting of update steps cancels a running teaching process and vice versa. A running WorkProc always terminates correctly, i.e. it is not interrupted by incomming events. Therefore XGUI processes all events as usual, but slows down with increasing running time of the WorkProcs.

In general there is one callback function for each button. However, some buttons call the same function, and use just different values for the argument `client_data`. This argument, the calling widget and an aditional argument `call_data` are passed over to the callback procedure. `client_data` is defined by `AtAddCallback()`, `call_data` by the kind of widget that is calling. The scrollbar widget, for example, passes the position of the slider.

[1]SNNS-XGUI interpretes this as an application.

With the use of window systems like X, a structuring of the source code according to the windows used is almost automatic (see also table 7.1 and 7.2).

Callback procedures are kept in a separate file. Since SNNS-XGUI consists mainly of those functions and calling depth remains fairly shallow, the source code is easily readable.

7.1 Administration of the Windows

There are three kinds of windows. They differ in the shell selected, and in the mode of the event handling. Transient shell windows (`transientShellWidgetClass`) have only a thin frame and block all other windows. This means that no more events are processed until this window is closed and the shell widget is destroyed. This is done by generating the window with the call `Xt_popup(widget, XtGrabExklusive)` with the toplevel widget `ui_toplevel` as the parent. Some windows, like the file panel, do not block XGUI, because they are called with `Xt_popup(widget, XtGrabNone)`. The third kind of windows are created as a top level shell (`topLevelShellWidgetClass`) like the manager panel.

Most of the windows (AsciiTextWidget, ToggleWidget, ScrollbarWidget) have panel items that can be changed by the user. These values are commonly stored in global variables, to enable other parts of the program to access them, even after the widget is destoyed. Some resources are predefined as so called *fallback resources* in the file `ui_main.h`. The bitmaps for the buttons are located in the directory `iconsXgui` and are linked to the program at compile time.

A callback routine is called every time a window is destroyed (for example by clicking `DONE` in popups). This routine takes care of all necessary actions, before vital information is lost, like copying the values of AsciiTextWidgets into global variables. In addition, there are flags which indicate whether or not a popup is existing.

Many panels contain form widgets, where other widgets can be positioned freely. In XGUI the location of these new widgets is always relative to the position of already existing widgets. The info panel, for example, takes the first line as horizontal reference. The change of widget positions has to be made very carefully, since some widgets have the same variable[2] and may not be addressable anymore.

To ease the handling of the widgets some routines are given in the file `ui_xWidgets.c`. They allow the creation of Label-, AsciiText-, Button-, MenuButton-, Toggle- and Scrollbar-Widgets in a form widget. Three functions return the value of an ascii text widget as a string (`ui_xStringFromAsciiWidget`), float (`ui_xFloatFromAsciiWidget`) or integer (`ui_xIntFromAsciiWidget`). To set the contents of a widget to a specific value use `ui_xSetString`, or `ui_xSetLabel`. To set/read the state of a toggle use `ui_xSetToggleState` and `ui_xGetToggleState`.

[2]Otherways every widget would need its own local variables

bn_TD_bignet.c	the BIGNET time delay panel & functions
bn_art1.c	the BIGNET ART1 panel & functions
bn_art2.c	the BIGNET ART2 panel & functions
bn_artmap.c	the BIGNET ARTMAP panel & functions
bn_basics.c	subroutines common to all BIGNET tools
bn_bignet.c	the BIGNET feed forward panel & functions
bn_menu.c	the BIGNET tools selection menu
cc_main.c	cascade correlation control panel
d3_anageo.c	3–D matrix operations
d3_disp.c	event loop for 3–D display
d3_dither.c	color rastering
d3_draw.c	high level drawing functions
d3_font5x7.c	5x7 font array
d3_font5x8.c	5x8 font array
d3_font8x14.c	8x14 font array
d3_fonts.c	text output control
d3_global.c	global variables for 3–D
d3_graph.c	low level drawing functions
d3_light.c	illumination panel
d3_links.c	3–D link display panel
d3_lists.c	3–D list managenemt
d3_main.c	3–D interface handling
d3_model.c	solid/wire display setting
d3_move.c	rotation,translation and scaling
d3_pannels.c	3–D control panel
d3_point.c	3–D pixel output
d3_project.c	3–D projection panel
d3_setup.c	3–D setup panel
d3_shade.c	shading algorithm
d3_units.c	3–D unit setup panel
d3_xUtils.c	X window routine interface
d3_zValue.c	z–value input panel
d3_zgraph.c	z–buffer functions
o_graph.c	plotting of the error graph
pr_main.c	pruning panel
ui_action.c	editor actions
ui_colEdit.c	changing of the display colors

Table 7.1: The source files of SNNS-XGUI, Part I

ui_color.c	handles the color maps
ui_config.c	load & save of configurations
ui_confirmer.c	handling of the confirmer
ui_display.c	administration of displays
ui_displwght.c	implements the Hinton– and WV–diagrams
ui_edit.c	editing of f-types and sites
ui_event.c	event handler for mouse or window events
ui_file.c	creation of the file panel
ui_fileP.c	load, save (high level)
ui_funcdispl.c	displays the activation and output functions
ui_info.c	creation of the info panel
ui_infoP.c	callbacks and handling of the info panel
ui_inversion.c	handels the inversion display & algorithm
ui_key.c	event handler for keyboard
ui_layer.c	creation of layer popups
ui_layerP.c	handling of the layer popup
ui_lists.c	administration of the list popup
ui_main.c	initializing
ui_mainP.c	windows and popups, help facility
ui_maspar.c	maspar user interface
ui_netGraph.c	drawing of network elements (mid level)
ui_netUpdate.c	drawing of the network, whole or parts (high level)
ui_print.c	creation of the print panel
ui_printP.c	callbacks and handling of the print panel
ui_printps.c	handling of the Postscript generation
ui_control.c	creation of the control panel
ui_controlP.c	callbacks and handling of the control panel
ui_result.c	handling of the result file generation
ui_selection.c	administration of the selection
ui_setup.c	creation of the setup panel
ui_setupP.c	administration of the setup panel
ui_status.c	status information in the manager panel
ui_textP.c	text output (stdout, Log file)
ui_utilP.c	utility functions
ui_xGraphic.c	X graphic interface (low level)
ui_xWidgets.c	handling of widgets
ui.h	datatypes, definitions

Table 7.2: The source files of SNNS-XGUI, Part II

7.2 Main Program

The function `main()` in the file `ui_main_.c` performs the initialisation and creates the manager panel. It then passes the control to X with XtMainLoop(). Here is the pseudo code:

```
Initialize the application;
Initialize the selection list;
Initialize the display list;
Initialize the editor;
Initialize all global variables;
Load default configuration from the file default.cfg;
Create manager panel;
Pass control to X;
```

Loading data from the configuration file changes some global variables, therefore they must be initialized already. The X event dispatcher calls the SNNS-XGUI callback routines and event handler routines. Actions that affect the network require communication with the kernel (see also chapter 7.12). The following sections describe the reactions of the windows and widgets to different events.

7.3 Manager Panel

The manager panel consists of the info panel, a message line and the XGUI menu. The panel is created at initialization time. The callback functions in the file `ui_mainProcs.c` create all the windows and the procedure `ui_printMessage()` creates messages in the panel. SNNS-XGUI is left by pressing the `QUIT` button with the function `ui_quit()`.

If multiple displays are used, all status information is best kept in one window. To display the information in different windows would waste space on the displays. Therefore this information is collected in the manager panel.

Functions, which change a value displayed in the manager panel have to use the routine `ui_stat_displayStatus` to updated the panel. The function `ui_stat_displayStatus` displays the other status information with the help of the variables `ui_sel_numberOf-SelectedItems`, `ui_key_flags`, `currentLayer` and `ui_currentDisplayPtr`. The scanning position is pased on as a parameter. To avoid flickering, this function is called only when there is a change in information (i.e. it is not called if the mouse is dragged over the display without changing the scanning position).

Sometimes the displayed attributes have to be checked for validity. The function `ui_info_-anyUnitSelected()` checks whether the displayed source and target really exist. The function `ui_info_makeUnitInfoPanelConsistent()` updates the panel, if units or sites have been erased. It is also called by several functions of the action module.

The Info panel is the most complex of SNNS-XGUI. To ease access to the widgets of the source and target units, they are accumulated in structures (`ui_targetWidgets, ui_sourceWidgets, ui_linkWidgets`). This was used in programming the routines in ui_infoProcs.c. Each attribute is also stored in a corresponding attribute structure (ui_targetUnit, ui_sourceUnit, ui_link).

7.4 Layer Panel

The toggle button in the layer panel is initialized with the bits of the variable ui_layerStartValue. Bit 0 corresponds to layer 1, bit 1 to layer 2, etc. Because the layer panel yields a value, X has to display the panel and allow user input before the program is continued (just like it was the case with the confirmer). The result is available in the variable ui_layerReturnValue.

7.5 Graphic Windows

All data for the graphic windows is stored in a linear linked list. Valid for all these windows are the following event types: GraphicExpose, ButtonPress, ButtonRelease, KeyPress, MotionNotify, EnterNotify, LeaveNotify, MapNotify, UnmapNotify, and StructureNotify.

There is no background storage for the graphic output, because it would be too memory intensive. The drawback is a longer screen setup time which depends mainly on the number and kind of link options that are displayed.

For simplicity reasons, all displays use the same graphic context ui_gc (see [You89], p. 184 for reference).

The events for all graphic windows are processed with the three event handlers ui_KEvent() for KeyPress events, ui_can_MWEvent() for mouse and window events and ui_can_MapEvent() for window mapping events. These functions are installed for each window with XtAddEventHandler. They determine in which window the event has occurred. This information is needed to determine subnet number and grid position because of the different origins in the various displays.

Input to the windows are all keyboard events, mouse events, and window events. A window is always active, if it has last sent an EnterNotify event (the mouse has been moved into the window) and is not closed. Thereby, the editor becomes independent from the windows.

The processing of graphical output is more complicated, since some operations affect only one (e.g. a refresh; UI_LOCAL), others all open windows (e.g. a change in the network; UI_GLOBAL). A simple approach would just draw a new network in every display. This, however, would take much more time than the solution found in SNNS-XGUI, where the editor determines whether it is better to re-draw the whole network or only the updates. This approach leaves some pixels white on each update, which slowly spoils the picture.

X refreshes the displays with a call to ui_refresh after each GraphicExpose event, if the count component of the event is 0 (see also [You89], p. 118). This routine is also defined as a callback function for expose events on the generation of a display.

7.5.1 Event Handler for Mouse and Window Events

This function, located in the file ui_can_MWEvent.c, does the following:

```
SWITCH (event type)
  ButtonPress:
    get event data;
    SWITCH (mouse button):
      left button:
        remember current first position for selection of area;
        draw selection rectangle;
      middle button:
        remember current position as source unit, if there is
        a unit;
  ButtonRelease:
    get event data;
    SWITCH (mouse button):
      left button:
        delete selection rectangle;
        remember second position for area and normalize both
        coordinates;
        select or unselect area;
      middle button:
        IF info panel created THEN
          show this unit as target and remembered
          unit as source;
      right button:
        IF empty position THEN
           unselect all (in this subnet)
        ELSE
           unselect this single unit
  EnterNotify (mouse is moved into a graphic window):
    set this window as actual and show new status;
  MouseMoves (mouse is moved):
    show new status, if new grid position reached;
  MouseDragg (mouse is moved with at least one button down):
    if the left button is pressed, delete old selection box,
    draw a new one.
save position of the event in the global variables
ui_pixPosOld and ui_gridPosOld;
```

The actions to these events are defined in this function. The results affect the info panel and the status panel.

7.5.2 Event Handler for Keyboard Events

This function, in the file `ui_KEvent.c`, is implemented as a big SWITCH statement. It represents an automaton for the recognition of editor commands. The transition from one state to the following state (stored in the global variable `ui_key_currentState`) is performed at the next call to the function. Depending upon the selected mode (normal, `Mode Units`, `Mode Links`), the following state is stored in one of the global variables `ui_key_returnUnitState` or `ui_key_returnLinkState`. This means that the automata changes its behaviour according to these two variables.

On `Units Move/Copy` the user is expected to click on a target position. In the meantime the automata is on hold (`UI_STATE_GETDEST`), and signals this to the event handler for mouse and window events in the global varaible `ui_outlineActiv`. When the target is defined, this event handler defines the next state of the automata.

Some actions ask the user for a value in a popup window. Since this blocks SNNS-XGUI as a whole (with `XtPopup(widget, XtGrabExclusive)`), no additional steps have to be taken.

7.5.3 Editor Actions

The file `ui_action.c` contains all editor operations that are called exclusively by the event handlers for keyboard and pointer events. The basic structures of the functions that process selected units look very much alike. The concept is the following:

```
IF operation can not be performed THEN
  error message;
  return;

delete marks of all selected units;
WHILE there is a selected unit DO
 IF desired operation for this unit can be  performed THEN
   execute operation;
   change graphic if necessary;
 ELSE
   report (if desired)
END WHILE
keep graphic and values in info panel consistent;
Redraw marks for selected units;
```

To perform the actions `Units Copy` and `Units Move`, it first has to be checked, whether all target positions in the grid are available for the operation. Therefore, the function `ui_action_checkNewPositions()` checks, whether the target position is empty for `UI_ACTION_COPY`, or occupied by unselected units for `UI_ACTION_MOVE`. The whole operation is skipped if an error condition results from the check.

After links or units have been deleted, it is possible that information in the info panel has become invalid. A call to `ui_info_makeInfoPanelConsistent()` assures consistency in that panel.

Often it is necessary to reiterate the list of selected units within the while-loop (e.g. the function `ui_action_linksMakeClique()` generates connections between all selected units). In that case, a second while-loop is used that traverses the selection list, either from the start, or from the current element.

7.5.4 Setup Panel

All setup data is contained in a structure (`struct SetupDataType`) which is itself contained in a display structure.

```
typedef struct SetupDataType {
    Bool          showValueFlg;
    Bool          showValue;
    Bool          showTitleFlg;
    Bool          showTitle;
    Bool          showLinkFlg;
    Bool          showDirectionFlg;
    Bool          showWeightFlg;
    FlintType     linkPosTrigger, linkNegTrigger;
    FlintType     unitScaleFactor;
};
```

When the setup panel is opened, the corresponding display is current. Therefore all changes are performed in the current display `ui_currentDisplayPtr`, and all inital data is read there. If a scrollbar is manipulated, the callback function `ui_thumbed()` is called to save the value in the corresponding global variable. To set the values of `showValue` and `showTitle`, there are two callback functions that assign the value to the parameter `client_data` in each menu option. The layer value is determined by the common method of the layer popup.

The toggles, that switch the various display options on and off call the function `ui_set_-toggleValues()`. This function sets the corresponding component of the setup variables in the current graphic window.

- `showValueFlg`: If this flag is set, the value that determines the size of the unit is numerically displayed below it.
- `showValue`: This number gives the displayed value and can be one of the following: `UI_ACTIVATION`, `UI_OUTPUT`, `UI_INITIAL_ACTIVATION` oder `UI_BIAS`.
- `showTitleFlg`: States whether or not the unit title should be given.
- `showTitle`: States whether the unit number (`UI_NUMBER`), or the unit name (`UI_-NAME`) will be the title.
- `showLinkFlg`: Links are displayed only if this flag is set.
- `showDirectionFlg`: If this flag is set, the connections will be displayed as arrows.

- `showWeightFlg`: States, whether the link weights are displayed.
- `linkPosTrigger` and `linkNegTrigger`: For positive and negative weights, these flags hold the threshold for drawing links.
- `unitScaleFactor`: If the activation of a unit is above this limit it is displayed with maximum extension.

When the `DONE` button is pressed, the setup panel becomes invisible again. The new grid size and origin are read from the panel, and the graphic window is redrawn. There the new parameters are used, and the desired picture is created.

7.5.5 Freezing Displays

The `FREEZE` button flips a switch in the display structure which prevents the high level display routines from affecting this window. The call `ui_netCompleteRefresh(UI_LOCAL)`, necessary after a GraphicExpose event, is the only one that changes a frozen display.

7.6 List Module

The construction of a panel with a list widget is a separate module. These panels are used to select functions, IO-types and F-types. There are only three functions which must react to the various lists in a specific way: `ui_list_getFirstItem()` yields the first element of the list, `ui_list_getNextItem()` the rest, and `ui_list_setValue` handles the result. Depending upon the type of the list, the result is either written to the appropriate place in the panel (source unit or target unit), or put into the global variable `ui_list_returnName`. In any case, the variable `ui_list_returnIndex` contains the list index used last. The list is constructed with `ui_list_buildList(parentWidget, type)`, where `type` can have one of the following values:

Type	Meaning
`UI_LIST_IOTYPE`	all IO-types
`UI_LIST_FTYPE`	all defined f-types
`UI_LIST_FTYPE_NAME`	ditto (edit f-type)
`UI_LIST_ACT_FUNC`	all activation functions
`UI_LIST_FTYPE_ACT_FUNC`	ditto (edit F-Typ)
`UI_LIST_OUT_FUNC`	all output functions
`UI_LIST_FTYPE_OUT_FUNC`	ditto (edit F-Typ)
`UI_LIST_SITE_FUNC`	all site functions
`UI_LIST_SITE`	all defined sites
`UI_LIST_FTYPE_SITE`	all sites on the current F-Typ
`UI_LIST_UNIT_SITE`	all sites at the current unit
`UI_LIST_LEARN_FUNC`	all teaching functions

7.7 File Panel

After clicking `DONE`, the directory and the four file names are copied from the AsciiTextWidgets (`ui_path`, `ui_fileNET`, `ui_filePAT`, `ui_fileCFG`, `ui_fileTXT`) into the corresponding global variables (`ui_pathname`, `ui_filenameNET`, `ui_filenamePAT`, `ui_filenameCFG`, `ui_filenameTXT`).

The file `ui_fileProcs.c` contains the functions for loading and saving of nets, patterns, configurations and texts. The structure of these functions is:

```
PROCEDURE save;
IF a file with this name already exists THEN
  IF User allows overwrite THEN
   save;
  ELSE
   abort and message;
END;

PROCEDURE load;
IF a file with this name already exists THEN
  load;
  IF error THEN message;
END;
```

7.8 Help Window

The callback function `ui_help_searchText` is called by clicking one of the buttons `MORE`, `LOOK` or `TOPICS` with a flag as `client_data`. On clicking `LOOK` or `MORE` the string is read from the X primary section, and the help text is searched for this string from the beginning. On clicking `TOPICS` the string `*␣TOPICS` is searched.

7.9 Confirmer

The file `ui_confirmer.c` consists mainly of the functions listed in [SUN86], p. 348 ff (prefix `ui_cf_`) for the construction of a confirmer package adapted to X windows. This package was enhanced only for the functions `ui_confirmOk()` and `ui_confirmYes()` to highlight the calls to the confirmer in the source code. Additionally an icon (`STOP` or exclamation mark) is displayed in the confirmer.

The intrinsics of X don't allow an interuption of XtAppMainLoop(). Therefore, a temporary event dispatch loop has to be programmed (see below). This loop is exited when a button in the confirmer is pressed, whose callback routine changes the variable `ui_cf_exit` to `TRUE`. Since XtGrabExklusive is set and the XGUI toplevel widget `ui_toplevel` is the parent of the confirmer, it blocks all events for SNNS-XGUI until it is destroyed.

```
ui_cf_exit = FALSE;
while (NOT ui_cf_exit) {
    XtAppNextEvent(ui_appContext, &event);
    (void) XtDispatchEvent(&event);
}
```

7.10 Graphic

The graphic module consists of three levels: The first level is responsible for drawing the network as a whole. The middle level draws or delets single network elements. This level uses graphic primitives of the lowest level, which make up the X graphic interface, as well as basic Xlib output functions for graphics and text.

The file `ui_netUpdate.c` contains the functions for drawing the whole net (prefix `ui_net_`). All routines in this module determine in which of the open displays to draw. Iconified or frozen displays are ignored. Depending upon parameters, the function `ui_net_updateWholeNet()` draws either all links or all units. For a complete update of the net use the function `ui_net_completeRefresh()`. This function first clears the window, then draws the links and finally the units.

The file `ui_netGraph.c` contains the functions for drawing network elements (`ui_drawUnit()`, `ui_drawLink()`). Two switches are important, one to determine whether to draw or delete the element (`UI_ERASE` or `UI_DRAW`), and one to determine whether to do the output globally (`UI_GLOBAL`) or only in the current window (`UI_LOCAL`). Units are drawn if the subnet number is the same as in the window, at least one layer of the unit is activated in the window, and the display is neither frozen nor iconified. Links are drawn if the corresponding units are visible, have the same subnet number, and have a weight that exceeds the threshold set by the trigger sliders in the setup panel.

Functions to compute grid coordinates from pixel coordinates and vice versa are located in the file `ui_utilProcs.c`. Normalizing functions are also located here. Normalization of edges of a rectangle here means exchanging the coordinates, if necessary, to place the first edge to the left and above the second (`ui_normalize_rect()`). A normalization with `ui_normalize_coord()` only puts the first coordinate left of the second.

All functions which call X graphic routines are kept in the file `ui_xGraphic.c`. Among them is a function for drawing an arrow (`ui_xDrawArrow()`), a function for drawing a box (`ui_xDrawBox()`),as well as a function to delete a black or white rectangle (`ui_xDeleteRect()`). To generate an arrow head, a triangle is rotated appropriately (see also the algorithm in [Har83]).

7.11 Selection Mechanism

All functions and global variables dealing with the selection mechanism have the prefix `ui_sel_`. All selected units are held in a linear linked list starting at the pointer `ui_sel_listPtr`. It is set to point to an element on initialization. The number of selected

bit 0	selected, if set
bit 1	failure of last action on this unit; not used
bit 3	action UNITS MOVE: Unit was already moved
bit 4	action LINKS INVERSE: Links were already turned

Table 7.3: Meaning of the Bits in the flag item of each list element for the selection.

units is stored in the globale variable `ui_sel_numberOfSelectedItems`. The total number of allocated items is kept in `ui_sel_numberOfItems`. This list is always increasing, i.e. memory is not freed by `dispose()`, but all unused elements are kept in the list, marked as such (Bit 0 in the component `flags`), and are used again when needed. New elements are attached at the beginning of the list.

```
typedef struct SelectionType {
    struct SelectionType     *nextPtr;
    struct PosType           gridPos;
    int                      subNetNo;
    int                      unitNo;
    int                      copyNo;
    int                      flags;
};
```

For an unambiguous identification of the units, their subnet number is stored in addition to their grid position. This is necessary, since units in different subnets may have the same grid positions. On processing the action `Units Copy Structure` the unit number is saved, in order to keep track of the units during the copying process. After completion of the action, the component is superfluous. Table 7.3 gives the meaning of the bits in the component `flags`.

When several windows are used, units can be selected in multiple windows. The case of one unit being displayed in more than one window has to be observed. SNNS-XGUI therefore allows selection only of units belonging to an active layer, and at least one window with the same subnet number must be open. If a unit is selected in one window, the selection mark becomes visible in all displays.

When a unit is marked selected, a cross is drawn over the unit. It has to be assured that prior to drawing or deleting, all marks are reset, and get updated afterwards with the function `ui_sel_reshowItems`. All editor functions dealing with moving or deleting units (prefix "'`ui_action_`"'), have to assure consistency of the selection list. This can be done by changing the flag, since these functions anyway have access to this element, when they process the selected unit.

7.12 Interface to the Simulator Kernel

The simulator kernel of SNNS offers a set of interface functions (see also chapter 4) which allow a one way communication with the graphic routines. There the graphical user interface acts as the master, the kernel as the slave. The functions can be assigned to the following tasks:

- read and set unit, site, and link attributes and functions
- administer f-types and sites
- change the network topology
- load and save networks and patterns
- simulator functions: update, teaching

The keys to all operations of the user interface (like adding or deleting units or links, changing attributes etc.) are always the unit number and the unit position. Therefore these two attributes always have to be unambiguous. On dealing directly with the unit numbers great caution is needed, since the kernel compresses its data structures during a save, thereby changing the unit numbers. The access via unit number, however, is much faster than via position, because a sequential search can be avoided that way.

The search is necessary, however, when the user interface has to process an event (ButtonPress, ButtonRelease, KeyPress, ...) which has occured in a window. A possible improvement would be an additional data structure which sorts all units by their grid positions. A linear linked list could, for example, be assigned to each row of the grid. The search time is then limited by the maximum number of units in a row. Unfortunately, for small nets the longer time to build up the display might outweigh the faster access time. In very large nets this mechanism could be a real improvement.

To increase the stability of the simulator, the current version always requests its data from the kernel (and not from global variables) in all situations where consistency is important. There are also data consistency checks contained in the code, before any change on the net is performed. This does not decrease system performance, since these requests are answered very fast by the kernel.

The type `bool`, used by all boolean functions of the kernel, is identical with the type `Bool`, used by all SNNS-XGUI functions. The type `Bool` is used by X windows and therefore is also used in the user interface.

```
typedef int Bool; \* User Interface *\
typedef int bool; \* Kernel *\
```

All functions in this interface have the prefix "'`krui_`"', to be able to identify the functions in this module. They are a part of the kernel functions.

More detailed information about the interface between XGUI and the kernel can be found in chapter 4 or in the source files `krui_typ.h` and `glob_typ.h` of the kernel. Modules of the user interface which use kernel functions must include the file `kr_ui.h`.

Chapter 8

3D-Display Implementation

This chapter gives a short overview of the implementation of the 3D visualization component. The modules for 3D-viewing can be divided into four groups:

- Modules for the data structures
- Modules for the panels
- Modules for visualisation
- Modules for drawing

Each program file has a corresponding header file declaring the exported functions. Exceptions are the files `d3_global.c` and `d3_global.h`, where the global variables are defined. All modules carry the prefix `d3_` for better distinction from the 2D XGUI functions (prefix `ui_`) and the kernel functions (prefix `krui_`).

The functions that are visible from outside a module also carry the prefix `3d_` of the 3D user interface.

8.1 Contents of the Modules

The following files contain data structures and variables of the 3D user interface:

`d3_global.h`	Global data structures and constants
`d3_global.c`	Global variables
`d3_font5x7.c`	Array for the 5×7 character set
`d3_font5x8.c`	Array for the 5×8 character set
`d3_font8x14.c`	Array for the 8×14 character set

The following files contain panels of the 3D user interface:

`d3_pannels.c`	Control panel
`d3_move.c`	Panel for stepwise rotation, scaling and translation
`d3_zvalue.c`	Panel for input of the z-value
`d3_setup.c`	Panel for the step sizes and initial values

`d3_model.c`	Panel for the kind of display
`d3_project.c`	Panel for the projection
`d3_light.c`	Panel for the illumination
`d3_units.c`	Panel for displaying values in the units
`d3_links.c`	Panel for displaying values in the links
`d3_disp.c`	Handling of the display window and the event loop
`d3_xUtils.c`	X Window assistance routine

The following files contain 3D visualization functions:

`d3_main.c`	Interface and controls of the visualisation
`d3_shade.c`	Illumination algorithm
`d3_draw.c`	High level drawing routines for units and links
`d3_anageo.c`	Matrix transformations and basic operations
`d3_lists.c`	Administration of lists

The following files contain low level drawing routines:

`d3_graph.c`	Low level drawing routines and color control
`d3_zgraph.c`	z-buffer functions
`d3_point.c`	Pixel oriented output
`d3_fonts.c`	Control of text output
`d3_dither.c`	dithering of the color values

8.2 Global Data Types and Variables

All global data types and constants are declared in the module `d3_global.h`. The basic data types are

`typedef float vector[4]`	the definition of a vector
`typedef float matrix[4][4]`	the definition of a matrix
`typedef vector cube[9]`	the corner points of a cube (unit), as well as the center

The following data type is used for the z-buffer algorithm of the unit surfaces:

```
typedef struct {
    int n;
    int mask;
    vector vert[4];
} d3_polygon_type;
```

`vert`	contains the vectors of the polygon edges
`mask`	contains the interpolation mask for the x-, y- and z-coordinates
`n`	used as a counter of polygon edges

The following data type is used for the values that can be represented by a unit:

```
typedef struct {
     int size;
     int color;
     int top_label;
     int bottom_label;
} d3_unit_mode_type;
```

`size`	a value is displayed by the size of the unit
`color`	a value is displayed by the color of the unit
`top_label`	a value is written to the upper right corner of the unit
`bottom_label`	a value is written to the lower right corner of the unit

The fields may each contain one of the constants: `activation_on`, `init_act_on`, `output_on`, `bias_on`, `name_on`, `number_on`, `zvalue_on` or `nothing_on`.

The light source is described by the following declaration:

```
typedef struct {
    int    shade_mode;
    vector position;
    float  Ia, Ka, Ip, Kd;
} d3_light_type;
```

`shade_mode`	specifies, whether the units are to be illuminated
`position`	is the position of the light source
`Ia, Ka, Ip, Kd`	are the illumination constants

The whole status of the network display is contained in:

```
typedef struct {
    vector trans_vec, rot_vec, scale_vec;
    vector trans_step, rot_step, scale_step;
    vector viewpoint;
    float unit_aspect;
    float link_scale;
    float pos_link_trigger;
    float neg_link_trigger;
    int font;
    int projection_mode;
    int color_mode;
    int link_mode;
    d3_unit_mode_type unit_mode;
    d3_light_type light;
} d3_state_type;
```

`trans_vec`	current translation
`rot_vec`	current rotation
`scale_vec`	current scaling
`trans_step`	step size for the translation in the transformation panel
`rot_step`	step size for the rotation in the transformation panel
`scale_step`	step size for the scaling in the transformation panel
`viewpoint`	viewpoint for the central projection
`unit_aspect`	ratio between the edge length of the units and the distance between them
`link_scale`	specifies a factor for a region in which the links are to be drawn
`font`	specifies the character set for labeling the units and links
`projection_mode`	contains the constant `parallel` or `central`
`model_mode`	specifies the display mode for the network. The field contains the constant `wire_frame` or `solid`
`color_mode`	specifies whether a monochrome or color terminal is used
`link_mode`	contains the display mode of the links. Values may be `links_on, links_off, links_color` or `links_label`
`unit_mode`	see `d3_unit_mode_type`
`light`	see `d3_light_type`

The global variables are located in the module `d3_global.c`. Most of them are initialized with default values.

The variables

```
Display *d3_display;
Window  d3_window;
GC      d3_gc;
int     d3_screen;
```

address the display window of Xlib (the data types `Display, Window` and `GC` are declared in Xlib).

```
int d3_displayXsize;
int d3_displayYsize;
```

contain the current size of the display window.

```
bool d3_displayIsReady;
bool d3_controlIsReady;
```

specify, whether the display window and the control panel are created.

```
bool d3_freeze;
```

specifies, whether updates are possible in the display window.

```
int d3_fontXsize;
int d3_fontYsize;
```

specify the size of the current character set used to label the units and links.

```
int d3_numberWidth;
int d3_shortNumberWidth;
```

specify the width of float and short int in the X Window dialogue widgets.

```
struct TransTable *d3_xyTransTable;
```

is a pointer to the 2D $\rightarrow$ 3D translation table. The table itself is declared in the kernel.

```
int d3_transOffset;
```

specifies the offset for zero in the table. This is necessary, since C does not allow negative indices.

```
d3_state_type d3_state;
```

contains the status of the network display (see `d3_state_type`)

```
cube d3_e_cube;
```

contains the corner points of the standard cube. The cube is moved by the vector (-0.5, -0.5, -0.5) to be centered in the coordinate system.

```
int d3_cube_lines[12][2];
```

contains the twelve edges of a cube. The numbers are used as indices to `d3_e_cube`.

```
int d3_vertex_index[6][4];
```

contains the six planes of a cube. The numbers are also used as indices to `d3_e_cube`

8.3 Drawing the Network in 3D

This section describes the functions that are called to draw the network. Most of the routines are located in the module `main.c`. In other cases, the name of the module is given in parentheses without the prefix d3_ . The functions are explained "top-down". Figure 8.1 shows the hierarchy of the calls.

d3_drawNet:The routine is called by the function `ui_net_updateWhole` in the module `ui_netUpdate.c` as well as by all routines which make a redraw of the network necessary.(e.g. upon changing from wire to solid model). When the display is frozen or not opened, the routine is left immediately. In the case of solid representation, the z-buffer is cleared. Then the units are drawn by `draw_units`. If links are to be displayed, `draw_links` calls the appropriate functions.

draw_units: This function draws all units in a network. First, the maximal dimensions of the net are calculated by `get_net_extrema`, then the center of rotation is computed from there. Afterwards the matrices for the global translation, scaling, and rotation are determined. The corners for labeling are determined next by `get_label_vert_indices`. Now

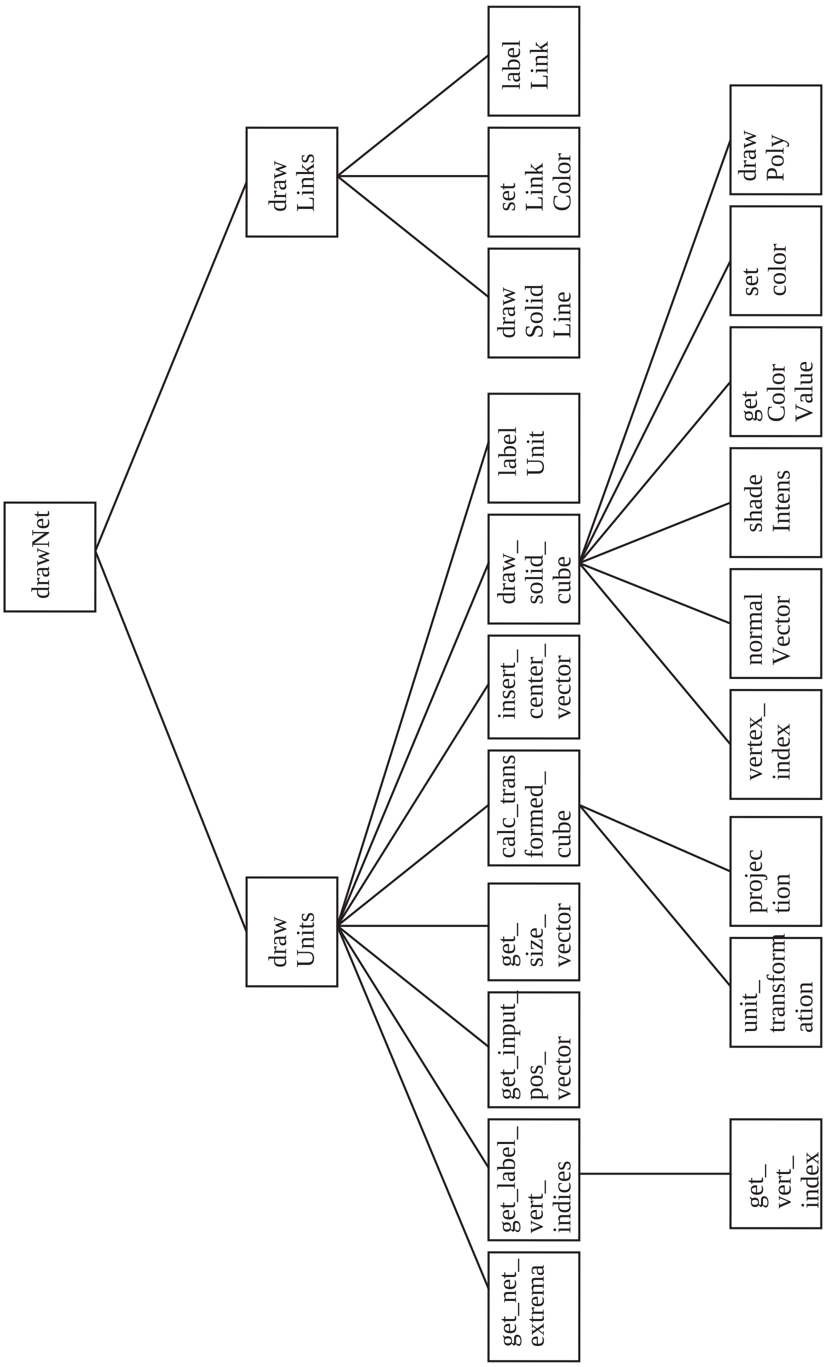

Figure 8.1: Function calls of drawNet

the net is traversed unit by unit. For each unit its location in space (in `get_unit_pos_vector`) and the corresponding matrix is computed. If a value is to be displayed by the size of the unit, the value is first changed to a vector in `get_size_vector` and then to a matrix. The matrices are multiplied with each other by `calc_transformed_cube`. With this new matrix, the vectors of the standard cube are transformed to picture space. The center of the transformed unit is stored by `insert_center_vector`, to speed up the subsequent drawing of links. Depending upon the representation mode the unit is drawn with `d3_drawWireframeCube` or `d3_drawSolidCube` in the module `draw.c`. If the units are to be labeled, `d3_labelUnit` puts the value at the appropriate place.

get_net_extrema: This routine determines the extension of the network in x-, y-, and z-direction. For each unit position it is checked whether one of its components is a maximum. The results are two vectors enclosing the net.

get_label_vert_indices: The indices to the unit vectors for labeling are computed in this function. This can be done at a single unit, since all units are rotated by the same angle and the alignment doesn't change. The routine calls the actual computing function `get_vert_index` once for the upper and once for the lower label.

get_unit_pos_vector: This function contains the 2D $\rightarrow$ 3D transformation. Input parameter is the unit number, output value the 3D–vector. All functions which require 3D–coordinates have to call this routine.

The position of the unit is read by `krui_getUnitPosition`. This value contains the x and y coordinate in the 2D–display as well as the z-value assigned to the unit. The displacement of the unit against the 2d–display in x and y direction is computed by the 2D $\rightarrow$ 3D translation table `d3_xyTransTable`. The z-coordinate is used to index the table.

get_size_vector: In this routine the activation, the initial activation, the output, or the threshold is translated into a vector for the unit size. All components of the vector contain the same value. Values bigger than 1 are clipped. This guarantees that the unit can not grow arbitrarily and cover other units. For negative values and values around zero, a flag is returned that keeps the unit from being drawn. This is necessary, since the unit would otherwise be drawn as a point.

calc_transformed_cube: This subprogram has to calculate the transformed unit coordinates. Therefore the following matrices have to be multiplied: matrix for 3D–position, for centering the unit in space, for the base values of the unit, for the size of the unit representing a value, for the scaling of the net, for a displacement of the net, and the matrix for rotation of the net. The resulting matrix transforms the vectors of the standard cube by `unit_transformation`. Finally a central projection is performed by `d3_projection` if that option was selected.

unit_transformation: In this routine, the vectors of the standard cube are multiplied by the transformation matrix.

insert_center_vector: The transformed center of the unit is written to the unit structure of the kernel by `krui_setUnitCenters`.

d3_drawWireframeCube: This function draws a "transparent" unit. The eight transformed vectors of the unit corners are connected with each other according to the entries in the d3_cube_lines array. The drawing is performed by the low level routine d3_drawLine (graph.c)

d3_drawSolidCube: This subprogram draws a “massive“ unit. The corner points of the planes are defined by the array d3_vertex_index in global.c. If no value is to be displayed by the color of the unit, the brightness is computed for all six planes of the cube. For this purpose, d3_normalVector (anageo.c) computes a vector perpendicular to the plane. d3_shadeIntens in module shade.c takes the position of the light source, the normal vector of the plane, and a vector on the plane and computes the intensity value for the plane. This is converted to a palette index by d3_intens_to_grayval and activated by d3_setColor (draw.c). If a value is to be displayed by the color, d3_getColorValue determines the appropriate color value and d3_value_to_color the palette index. Then d3_drawPoly (graph.c) draws the plane.

d3_shadeIntens: Here the intensity of the light source is computed. The value varies between -1 and 1.

d3_getColorValue: This function returns a value in the range $[-1, 1]$. The value may represent the activation, initial activation, output, or threshold of the unit.

d3_labelUnit: To label a unit, the appropriate value is converted to a string. Then, the position for the string is computed. The function get_label_vert_indices did already compute the index of the corner. If the label is to be put at the upper corner, the y-coordinate has to be adjusted by the font height. The string is drawn by a call to d3_draw_string in module fonts.c.

draw_links: This function draws the links in a network. For each unit the numbers of all units connected with the current one are obtained by krui_getFirstPredUnit and krui_getNextPredUnit. Additionally the weight of the connection is stored. The coordinates of the units to be connected are determined by krui_getUnitCenters. A link is either black, or gets a color assigned by d3_setLinkColor which corresponds to its weight. The drawing is performed by the functions d3_drawWireframeLine or d3_drawSolidLine in module draw.c. An optional labeling with the weight is performed by a call to d3_labelLink.

d3_setLinkColor: The color of a link is determined by its weight. For this purpose the weight is converted to the interval $[-1, 1]$ with the scale factor d3_state.link_scale. Bigger weights get the value 1, smaller ones -1. The implementation is analogous to the 2D–display.

d3_drawWireframeLine: To draw a link in the wire frame model, the vector is rounded and drawn by the function d3_drawLine (graph.c).

d3_drawSolidLine: The points of a link in solid model display are computed from the line equation and drawn with d3_putPixel.

d3_labelLink: This routine writes the weight of a link at the center of the link. The center between head and tail is given by the line equation with $\mu = 0.5$. The value is put at this position with d3_draw_string.

8.4 Low Level Drawing Routines

The low level drawing routines are located in the modules `d3_graph.c`, `d3_fonts.c` `d3_point.c`, and `d3_dither.c`.

The following functions only call the corresponding Xlib routines. The justification for these additional routines is, that they simplify porting to other graphic systems, since they are the lowest level of drawing calls.

function	Xlib Call	Task
`d3_setColor`	`XsetForegroand`	sets the foreground color
`d3_clearDisplay`	`XClearWindow`	deletes the display window
`d3_drawLine`	`XDrawLine`	draws a line
`d3_putPixel`	`XDrawPoint`	draws a pixel

The z-buffer is controlled by the routines in module `d3_zGraph.c`. The z-buffer is organized as a one dimensional array of float. Thereby, clearing the buffer is very fast. The necessary memory is dynamically allocated (and freed) by the operating system. The position (x, y) is computed explicitly from the size of the buffer.

`d3_initZbuffer`	allocate the z-buffer on the heap
`d3_clearZbuffer`	reset the z-buffer to the maximum distance
`d3_readZbuffer`	get distance z at position (x, y)
`d3_writeZbuffer`	set distance z at position (x, y)
`d3_freeZbuffer`	free the z-buffer on the heap

A polygon is drawn by the function `d3_drawPoly`. The corresponding Xlib call cannot be used, since the visibility of each polygon point has to be checked. The implementation is derived from the article "Generic Convex Polygon Scan Conversion and Clipping" by Paul Heckbert.

Several routines for labeling the links and units had to be created. This was necessary because the characters must be addressed point by point because of the z-buffering. The routines are listed in the module `d3_fonts.c`.

`d3_select_font`	selects a font
`d3_get_font_sizes`	returns the width and height of the current font
`d3_draw_string`	writes the string to the position (x, y, z)

The local functions `draw_char` and `draw_zbuffered_char` are used by `d3_draw_string` to generate a pixel sequence from the ASCII code of a character and the font array.

The creation of grey values is performed with the function `dither` in module `d3_dither.c`.

8.5 Matrix Calculations

The module `d3_anageo.c` contains the routines for vector and matrix calculations.

`e_matrix`	creates an identity matrix
`d3_transMatrix`	creates the translation matrix
`d3_scaleMatrix`	creates the scaling matrix
`d3_rotateXmatrix`	creates matrix for rotation around the x-axis
`d3_rotateYmatrix`	creates matrix for rotation around the y-axis
`d3_rotateZmatrix`	creates matrix for rotation around the z-axis
`d3_rotateMatrix`	creates matrix for rotation around all axes
`d3_multMatrix`	multiplies two matrices
`d3_multMatrixVector`	multiplies a vector and a matrix
`d3_normalVector`	calculates the normal vector for a plane
`d3_projection`	performs a central projection

8.6 The 3D Display Window

The 3D display window is opened by the callback function `d3_createDisplayWindow`, linked to the `DISPLAY` button in the control panel.

The display window has its own event handler. It reacts to ConfigureNotify and Expose events. When an Expose event occurs, the net is redrawn. Since a series of Expose events is created in some cases, all but the last one are ignored. If the size of the display window is changed, a ConfigureNotify event causes the net to be centered in the new window and redrawn. All Expose events are blocked in this case.

The display window is created by `XtCreatePopupShell`. It contains an inner frame (see 8.7). After the window becomes visible the Xlib variables `d3_display, d3_window` and `d3_screen` are created as well as the graphic context `d3_gc`. Each subsequent reference to the display window is performed by Xlib functions (see 8.4).

8.7 Panels

The panels were implemented with the Intrinsics – and Athena Toolkit for X11 Release 4. The structure of the panels is similar to those of the 2D–interface, to make the display as homogeneous as possible. Therefore, the routines in module `ui_xWigets.c` have been used.

The module `d3_xUtils.c` contains the two functions `d3_xCreateButtonItem` and `d3_xCreateToggleItem`. They create the buttons and import the "xbm" bitmaps for labeling the buttons.

Most of the panels are programmed in a similar fashion. Therefore, the implementation of the panel for the projection is described as an example. The panel looks like this:

In module `d3_panels.c` five functions handle the project panel.

d3_createProjectpanel: This routine is a callback function of the button $\boxed{\text{PROJECT}}$ in the control panel. As a parameter, the button itself is passed. From there, `XtTranslateCoords` computes the position at which the project panel is to be opened with `Xt-`

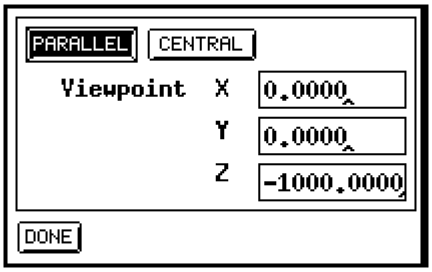

Figure 8.2: Project panel

CreatePopupShell. Since the panel is supposed to have an inner frame, a window of boxWidgetClass is created with XtCreateManagedWidget. There the actual window of class formWidgetClass is located. Now the two radio buttons PARALLEL and CENTRAL are created with d3_xCreateToggleItem. Then the labeling **Viewpoint X** with ui_xCreateLabelItem and the dialogue window follow. The next two lines are created similarly. Then the DONE button is created with d3_xCreateButtonItem. In the next step the three callbacks for the three buttons are created by XtAddCallback. setParallelProjection and setCentralProjection are linked to the two radio buttons, d3_closeProjectpanel to the DONE button. Now a temporary variable temp_state of type d3_state_type for the status of the display is generated. All changes to the status are stored there as long as the panel exists. Now one of the two radio buttons gets inverted by setProjectToggleState, corresponding with the current status. At last, the panel is made visible by XtPopup with XtGrabExclusive.

setParallelProjection: If the PARALLEL button is clicked, this function sets the field temp_state.projection_mode to parallel. A call to setProjectToggleState updates the panel.

setCentralProjection: like setParallelProjection

setProjectToggleState: With this function, the status of the two radio buttons is updated. The variable temp_state.projection_mode is read, one of the buttons is set, the other reset.

d3_closeProjectpanel: When the panel is closed, the values of the dialogue windows have to be written to the variable temp_state.viewpoint. Now temp_state is compared with the initial status variable d3_state. If they differ, a parameter has changed and the net has to be redrawn. This is done by copying temp_state to d3_state and calling d3_drawNet afterwards. The indirection with the temporary variable is necessary, since the panel may have been opened by mistake or for control reasons. In this case the net doesn't have to be redrawn.

Bibliography

[Har83] S. Harrington. *Computer Graphics - A Programming Approach.* McGraw-Hill, 1983.

[Hil85] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[HS86a] W.D. Hillis and G.L. Steele. Data parallel algorithms. *ACM*, 29(12):1170–1183, 1986.

[HS86b] W.D. Hillis and G.L. Steele. Massively parallel computers: The Connection Machine and NONVON. *Science*, 231(4741):975–978, 1986.

[SUN86] SUN. Sunview user reference manual. Technical report, SUN microsystems, 1986.

[You89] D.A. Young. *X-Window-System - Programming and Applications with Xt.* Prentice Hall, 1989.