Neo/Nst Technical Synopsis

Helge Ritter, May 99 [1]

Department of Information Science
Bielefeld University email: helge@techfak.uni-bielefeld.de

– PRELIMINARY VERSION –

Abstract

This manual provides a brief overview of the architecture of the *Neo* /*NST* system (V7.0), sketches the various ways how it can be extended with new modules and briefly describes its various configuration variables. The emphasis is on a quick overview, with pointers into more detailed documentation.[2]

Contents

[1] Compiled: March 28, 2002, Id: techdoc.tex,v 1.2 2001/02/21 14:13:20 walter Exp

[2] the latex- and ps-files `.tex` and `.ps` for this document are currently located in `/homes/helge/nst5/doc/tex/techdoc.tex` and `/homes/helge/nst5/doc/techdoc.ps`.

1 Overview

Neo /NST provides a framework to construct programs from an extensible collection of objects ("units") sharing a uniform interface.

The system is implemented at two different levels: the (lower) *NST* level provides a `C`-API for the construction and combination of all *NST* objects, together with auxiliary functions to program new units or to wrap existing code as a *NST* unit. At this level, *NST* API-calls can be fully intermixed with normal `C` and `C++` code.

The (upper) *Neo* level extends each *NST* object by graphical wrapper, allowing to create and manipulate *NST* objects with a graphical editor. A *Neo* program is built by graphically connecting the ports of a suitable set of "icons" (one for each *NST* unit) into a data flow diagram that expresses the desired data processing operation. Data flow programs can be executed, inspected and modified interactively. When a program has been completed, it can be saved as a new building block ("loadable unit") for later use.

Unlike many other graphical systems of a similar flavor, a prime concern with the realization of *Neo /NST* was *run-time efficiency*. Execution of a *NST* object adds only a function call plus a few `C`-statements overhead to the wrapped code. Data exchange between ports through a "wire" is by a fast by-reference mechanism (unless data conversions are required, which require a copy). Once a *NST* circuit has been set up, its execution is just the execution of a preformed chain of fast, compiled operations.

This efficiency is fully preserved at the *Neo* level. *Neo* provides only a GUI on top of *NST* for more convenient editing and inspection of *NST* circuits. When it comes to execution, it just issues a fast call into the *NST* API, triggering the same operations as if on the *NST* level itself.

2 System Architecture

Figure 2 provides an overview of the *Neo /NST* architecture. The vertical dimension is the axis along which the system is extended by new modules: the most basic functions are at the bottom (below line A), standard extensions which by now have become a fixed part of *NST* are on top of them (between lines A and B). Dynamically loadable extension modules that are not a fixed part of *Neo /NST* are on top of line C.

The horizontal dimension separates different implementation levels: each extension is composed of a collection of algorithms (implemented in `C` or `C++` (leftmost column), its *NST* wrapper (next-to-leftmost column) and its *Neo* wrapper (next-to-right column).

Conceptually, the leftmost column is the level where the actual functions of a module are implemented. The *NST* column adds a uniform object wrapper to this, and the *Neo* column adds the further convenience of graphical manipulation to the *NST* object wrapper.

Keeping these levels strictly separate for the implementation of each module allows to use every created circuit also without *Neo* , and it allows to use the algorithms in the circuit also without *NST* (leftmost column)[3]. Generally, wrapping an existing algorithm for *NST* should not introduce any changes in the existing implementation and should be entirely based only on the header files for the algorithm's implementation. Likewise, adding the *Neo* wrapper should only rely on the header file of the the *NST* wrapper. The manual `mkudoc` provides a detailed guide of how to proceed.

The core of the *NST* functionality is contained in the files below the horizontal line A (at the *NST* level, this includes files `nst0..10.c`, auxiliary routines in `nst_aux0.c .. nst_dbug.c` as well as a number of others (have a look into `Makefile.all` for a current listing). Note that this part of *NST* does not yet include any graphics calls.

The core of the *Neo* functionality is contained in the files `neo0..15.c` (these routines, of course, contain graphics calls).

A first tier of extensions sit on top of this basic stuff. At the *Neo* level, the wrappers for

[3]While a strict separation of the *NST* and *Neo* levels is obeyed for almost all *NST /Neo* objects (with the exception of a very small number of "meta"-operations, such as remakeing another unit or an array thereof), many *NST* objects from the early development phases of *NST* , as well as objects whose operation makes no sense outside of *NST* contain their algorithmic code directly within the *NST* wrapper, limiting the use of their operations to the presence of `libnst`.

these units are grouped into three compiled modules ("folders") neo_stdr.o neo_scrn.o and neo_misc.o. At the *NST* level, a subset of these units is implemented in the correspondingly named modules files nst_stdr.c, nst_scrn.c and nst_misc.c. These extensions also introduce the dependence on a graphics interface. All of *NST* 's and *Neo* 's graphics calls are via calls to routines in nst_X11.c. As the name indicates, this sits on top of libX11 (and also needs libXext). To port *Neo* /Nst to a different environment would require to implement the routines in nst_X11.c for this environment, everything else then will follow.

As a good practice, there should be a simple naming correspondence between the different implementation levels of a module: when file xxxx.c (or xxxx.C) implements the algorithms, nst_xxxx.[cC] should contain the *NST* wrapper and neo_xxxx.c the *Neo* wrapper. However, in the present case each of the three modules contains a number of rather complex units (among them units for creating graphical GUIs, various plots or for interpreting a C-style script) whose code is so substantial that it had to be subdivided over a larger number of different nst_xxxx.c and further, auxiliary nst_aux_xxxx.c files (some (but not all!) of which are indicated in the shown boxes). Therefore, in this case the recommended simple naming correspondence is broken and each *Neo* wrapper file relies on several *NST* wrapper files (also containing large parts of the implementations) and further auxiliary files.

The first tier of extensions ends at the line B, and libnst contains all modules of the implementation and *NST* wrapper levels, i.e., everything below line B that is in the leftmost two columns (as a consequence, compilation of libnst requires the X11 headers).

The next two rows delimited by lines B-C add two important *Neo* files: neo_main.c contains the main interactive loop of the *Neo* editor; from here, all other *Neo* actions, including start-up and termination, are invoked. [4]

File neo_defs.c contains the central routine to make all compiled extension modules known to *Neo* at start-up ("registration"). The set of extension modules that shall be registered at startup is determined in the configuration header file neo_defs.h (the rightmost column in Fig.2 shows configuration files). In the standard configuration, neo_defs.h contains a ADD_FOLDER() call for each of the three modules ("folders") misc, stdr and screen[5]. just described. Additional HIDE_ICON() statements may make individual members in a folder "invisible", but in such a way that these members are still available when circuits are loaded into *Neo* (this is useful for desupporting the use of obsolete units in new work). ADD_ICON() statements allow to make members appear in a folder different from the folder where they have been defined. Statements for registering further compiled folders might be added here, but see below.

The files in the third (*Neo* -) column underneath line C (plus their associated headers) provide the implementation of the *Neo* editor and are compiled into libneo.

The small file neo.c adds to this the main() function in order to make a runnable version of *Neo* (this needs to link neo.o with libneo and libnst plus standard system libraries).

A runnable *Neo* created from these files allows already to create many useful circuits (among other things, it already contains the prog_unit and most graphics units).

However, most of the functionality of *Neo /NST* is implemented in a large collection of extension modules ("dynamic folders") shown above line D in Fig. 2. Each dynamic folder is entirely separate from *Neo* and is is dynamically loadable (and unloadable) during start-up or runtime. Usually, each dynamic folder is independent from the other dynamic folders above line D in the sense that any combination of them can be loaded in any order (an exception are the folders solid and solid2: the latter needs the former to be loaded first; if such dependencies exist, they should be indicated by a clear naming).

The structure of each dynamic folder is the same as for the already described compiled folders: an implementation part (left column), a *NST* wrapper (middle column, optionally containing also the implementation), and the *Neo* wrapper (third column). Fig. 2 indicates some of the existing folders,

[4] neo_main.c includes a file neo_opt.c which contains the command line argument processing for *Neo* and which also defines corresponding variables (whose declarations are thus not seen in the source code of neo_main.c). Currently, neo_opt.c is generated from a structured comment in neo_main.c that specifies the command line parameters and their default values as well as help texts. The generator script mkopt is invoked from the makefile as needed.

[5] the module names must match the module name specified within the module's neo_xxxx.c wrapper. This name may but need not coincide with xxxx.

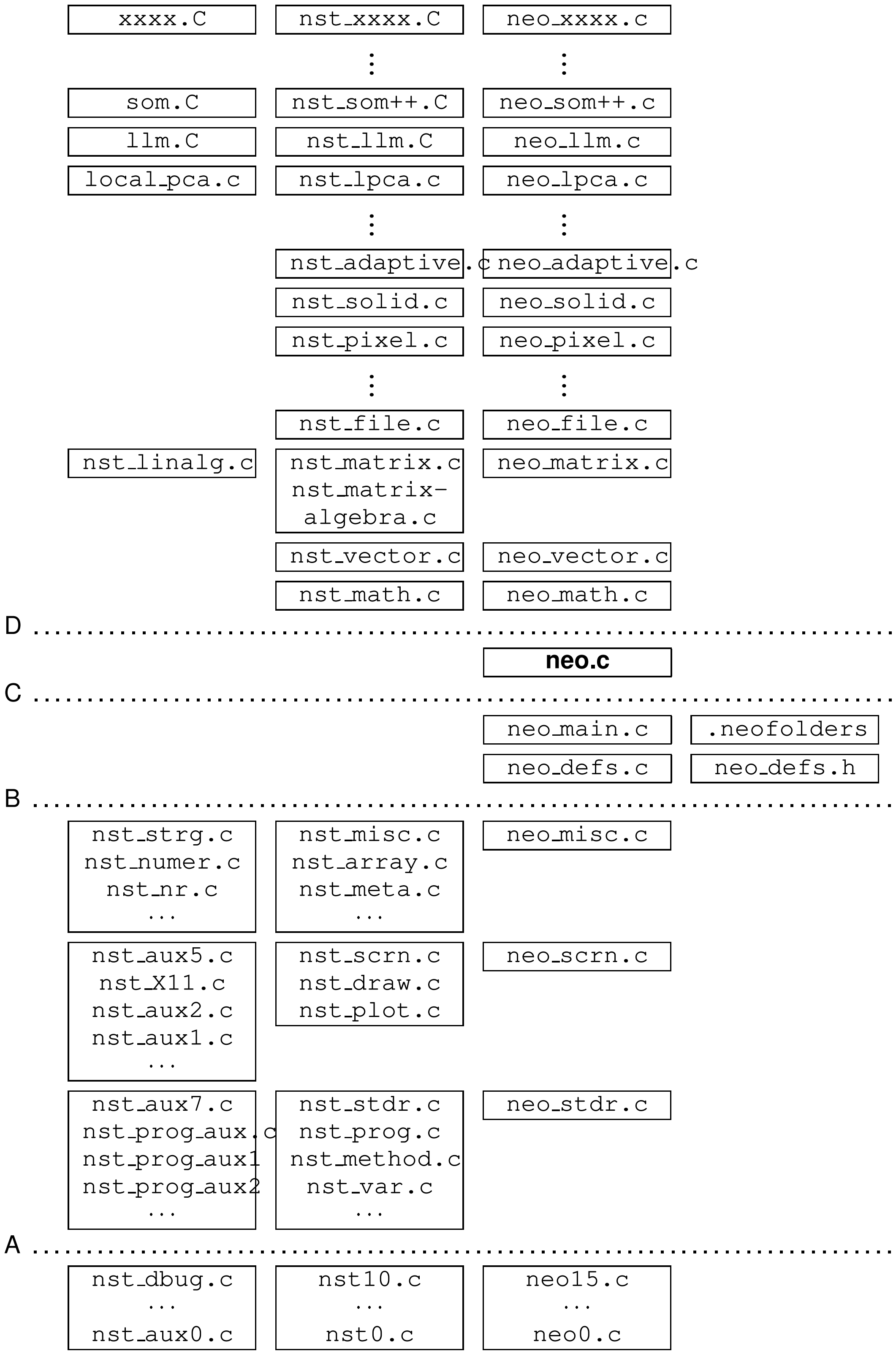

Figure 1: *Neo* /*NST* Architecture

as well as the prototypical structure of a general extension folder for a `C++` class implemented in `xxxx.C` in the topmost line.

Usually, it is convenient to load some dynamic folders already at start-up into *Neo* . The set of dynamic folders to be loaded in this way can be specified in the startup file `.neofolders` in the home- or current directory (searched first).

As regards implementation, there is no difference between a dynamic and a compiled folder, and a dynamic folder named `xxxx` can be rigidly compiled into *Neo* by simply add the statement `ADD_FOLDER(xxxx)` to *Neo* 's `neo_def.h` file and adding the names of the required objects to `PREDEFINED_FOLDERS` in `Makefile.all`.

Neo is compiled with the flag `-rdynamic`, allowing each dynamic folder to reference globally defined symbols in *Neo* and *NST* . Many of the existing dynamic folders do this and rely in particular on symbols defined in the code of the three compiled folders `stdr`, `scrn` and `misc`. This is one of the reasons why these folders must always be present as compiled folders in *Neo* .

3 Environment Variables

The following environment variables affect the operation of *Neo* and *NST* :

`NST_LD_LIBRARY_PATH` determines the search path used by *NST* when loading a dynamic library (such as a folder or a `loadlib`). If this environment variable is undefined, `LD_LIBRARY_PATH` will be used instead (all dynamic library loads are made through the *NST* function `nst_dlopen()` which is a wrapper to `dlopen()` using the above search path).

`NSTPATH` search path used for accessing data files when filename is relative and file is not found in current directory (strictly speaking, this affects only file read operations attempted through `nst_fopen()` in `nst0.c`, which is used by most Neo/NST file access operations.

`NEOHOME` directory relative to which important other directories are looked up by *Neo* (e.g. manual pages are expected in `NEOHOME/man` etc.). Can be overridden by specifying a command line value after `-NeoHome` when starting `neo`.

`MYEDITOR` external editor invoked when issueing ctrl-J command in NST text input windows.

`EDITOR` dito, used when `MYEDITOR` is not defined.

`NEOEDITBUFNAME` If defined, it will be used as the name of the temporary file created when invoking external editor with ctrl-J command (otherwise, the temp file name will be randomly created). `NEOEDITBUFNAME` can contain two '%d' tokens, which the will be filled by pid and counter. A definition of `NEOEDITBUFNAME` will also prevent the temporary file from automatic deletion after the edit session.

`ARCH` expected to hold architecture name (default `linux` will be used, if not defined). Is used by *Neo* 's source generation when generating a Makefile.

4 Configuration Files

`.neofolders` looked up by *Neo* in the local directory, then in the home directory, to find names of folder to load at start-up. Each entry is a line of the form

```
vector          pathname/vector.so
```

for a dynamic folder, or

```
myfolder        pathname/myfolderdir
```

for a loadable folder. `pathname` may contain tokens `$NEOHOME`, and `$ARCH`. A `#` at the start of a line denotes the begin of a comment line.

`.neo` looked up by *Neo* in the local directory, then in the home directory, to find values to override defaults of *Neo* 's command line parameters. Each line is of the form

```
-parametername value
```

`neo -H` prints a listing of recognized command line parameter names and their default values. Values specified on the command line override those in `.neo`.

`.neo_exitstatus` left behind after each *Neo* session. Stores some settings to later continue a new session with the same values.

5 Extending *Neo* /*NST*

To make existing code callable within *Neo* /*NST* requires to write wrappers for the *NST* and the *Neo* level. Templates for a pair `nst_xxxx.c` and `neo_xxxx.c` of NST/Neo wrapper files to wrap an object by a wrapper function named `xxxx` are generated by the utility call

```
mku xxxx
```

The generated skeleton programs are already annotated where to fill in missing code. As a brief background, below we sketch the basic structure of a NST and *Neo* wrapper.

5.1 NST wrapper

The basic structure of a *NST* wrapper function is (for clarity, all error checking has been omitted):

```
unitptr my_wrapped_object(..arguments.., uHost) {

   unitptr uNew;

   uNew = primitive_unit(...interfacedesc..., uHost);

   nst_var_of(uNew) = instance_of_object_to_wrap(...);

   /*** set method(s): */
   uNew->exec   = my_exec;
   uNew->load   = my_load;
   uNew->save   = my_save;
   uNew->remove = my_remove;
   label_unit(uNew,"myobject");
   return uNew;
}
```

Here, `instance_of_object_to_wrap` is assumed to be a constructor call for an object that implements the to-be-wrapped algorithm(s). Function pointers `my_xxxx` are wrapper functions to execute the object's main method, to load/save the state of the object from/to a file and to remove the object. `..interfacedesc..` must pass suitable parameters (usually computed from `..arguments..` to the `primitive_unit` constructor to make a NST unit with a suitable set of input/output ports of the required type(s) (cf. man page).

In many cases, the object will just be a data structure, with a fixed set of functions that operate on the data. As a simple illustrative case, assume an object that adds an input vector of dimension `n` to an internally held "vector sum" of the same dimension and outputs the result. The following header file of an implementation of that "object" might be available:

```
typedef my_data_{  int iDim; float *pfSum; } my_data_t;
void addtosum(float *pfVec, my_data_t *pVar);
```

Then the above code might become specialized in the following way:

```
int my_exec(unitptr u) {
   my_data_t *pVar = nst_var_of(u);
   float *pfInp = pfNstInp(u,0,0); /* adr of inp 0 */
```

```
   float *pfOut = pfNstOut(u,0,0); /* adr of out 0 */
   addtosum(pfInp,pVar);  /* invoke operation */
   /* copy result to output: */
   memcpy(pfOut,pVar->pfSum,pVar->iDim*sizeof(float));
   return OK;  /* OK = 0 is NST default return value */
}

int my_remove(unitptr u) {
   my_data_t *pVar = nst_var_of(u);
   free(pVar->pfSum);
   return nst_remove0(u); /* includes a free on pVar */
}

unitptr my_wrapped_object(int iDim, unitptr uHost) {

   unitptr uNew;  my_data_t *pVar;

   uNew = primitive_unit(NST_INP, NST_FLOAT, iDim,
                         NST_OUT, NST_FLOAT, iDim,
                         NST_HOST, uHost);

   /*** construct & initialize to-be-wrapped instance: */
   pVar = calloc(1,sizeof(my_data_t));
   pVar->iDim   = iDim;
   pVar->pfSum  = calloc(iDim,sizeof(float));

   nst_var_of(uNew) = pVar;

   /*** set method(s): */
   uNew->exec   = my_exec;
   uNew->remove = my_remove;
   label_unit(uNew,"myobject");
   return uNew;
}
```

This just illustrates the basic structure of a wrapper. We have not bothered to customize the default settings of the other NST methods associated with each *NST* unit (these are `int adapt(unitptr)`,`int ctrl(int,unitptr)` `int load(FILE*,unitptr)` and `int save(FILE,unitptr)`

Just a few finishing remarks:

error checking: the above code would have to be augmented by various error checks. To issue warnings and error messages, use the two *NST* system functions

```
nst\_warning(pcFmt,...);
nst\_error(pcFmt,...);
```

which use a `printf-style` parameter list to print their arguments together with parameter values.

optimization: in the present case, instead of allocating the result vector `pfSum`, we might have directly used the memory storage of the output pins:

```
 ... instead of calloc(iDim,sizeof(float)):
pVar->pfSum  = pfNstOut(uNew,0,0);
```

We then would save the `memcpy()` operation and must (!) omit the `free(pVar->fSum)` in `my_remove`.

remove method: The final call is always a `nst_remove0(u)` to remove the current NST unit. This will always include a `free()` operation on the private variable struct `nst_var_of(u)`. If the private variable struct needs a more complicated destructor, this destructor must be called before calling `nst_remove0(u)`. To then disable the automatic `free()` in `nst_remove0(u)`, *the private variable pointer must the be explicitly set* `NULL` *(*`nst_var_of(u)=NULL;`*) since* *otherwise the second, built-in free() will be fatal!*. The present example is a mix of both ways: instead of a package-specific destructor, we free the internal data `pfSum` by hand, but leave the freeing of the enclosing struct instance to NST.

more than one method: If the additional methods are a small set of simple operations, they can be wrapped into the *NST* `ctrl` method, which allows to pass an additional selection integer as the first argument. In more complicated cases, wrapping should use the technique of class containers as described in manual *mkudoc*.

C++: Due to different conventions, `C++` function names cannot be passed as function pointers to `C`. In this case, the wrapping occurs via a derived auxiliary class; details and examples are described in the manual `mkudoc`.

5.2 *Neo* wrapper

To illustrate the basic structure of a *Neo* wrapper, we here give a simple wrapper for the *NST* unit of the previous section:

```
ICONFORM1(myobject) /* format of create dialog */
   " create instance of myobject: %n"
   " vector dimension: %2i %n"
ICONFORM2(myobject) /* format of config dialog, "" if none */
   ""
ICONLABEL("my|object")   /* default label */

ICONMAKE(myobject) {    /* specify constructor wrapper */
   unitptr uNew;
   SRCLINE("$ = #($,@);\n"); /* informs source line generator */
   uNew = myobject(IVAL(0), HOST);  /* NST constructor call  */
   return uNew;
}
ICONPARS(myobject) {
   /* customize here */
   return 1;
}
```

The format for each of the two dialog windows is given as text string constant, with input items indicated by embedded `%` directives (here a single `%2i` for a single `int` input item with default value 2), as explained in the manual page for the `input_window` NST unit.

The values of the specified input items will then be used in the ICONMAKE routine to issue a suitably parametrized constructor call of the object's *NST* wrapper (which here is function `unitptr myobject(int,unitptr)`).

The `SRCLINE` macro uses a `printf` like parameter list to specify a piece of `C` code to be emitted when *Neo* is requested to create source code for a circuit containing an instance of the present unit. Its specification is optional and only needed when the source code option shall be used later. Tokens `$i`, `#i`, `@i` and occurring in the first argument parameter of `SRCLINE` will be replaced by the values of `IVAL(i)`, `FVAL(i)`, `TVAL(i)` and `HOST`; the token `$` indicates the returned unit handle (cf. below)). Thus, the above `SRCLINE` specification tells *Neo* to emit the source line

```
u = myobject(IVAL(0),HOST);
```

for each occurrence of the present unit. For more details, see the *Neo Tutorial*.

The following three function calls provide access to the values in the dialog windows (inside the body of `ICONMAKE`, they refer to the create dialog window, inside the body of `ICONPARS` they refer to the config dialog window):

IVAL(i) value of i-th numeric dialog entry item, converted to an `int`.

FVAL(i) value of i-th numeric dialog entry item, converted to a `float`.

TVAL(i) ptr to string of i-th textual dialog entry item.

Note that depending on the type of widget an entry (such as a written number) may both qualify as a numeric item and as a textual item at the same time, i.e., the values of such items can then be obtained as strings or as numbers, and they then will contribute to the index counting of all three functions above.

Each *Neo* folder file contains the above parts once for each wrapped type of unit. Additionally, the file must begin with a declaration of the following form:

```
extern NstIconDef_t ICONDEF_xxxx;
extern NstIconDef_t ICONDEF_yyyy;
...
extern NstIconDef_t ICONDEF_zzzz;

BEGINFOLDER(NameOfFolder)
   ICONENTRY(xxxx), ICONENTRY(yyyy), ... ICONENTRY(zzzz)
ENDFOLDER;
...
... the wrappers for xxxx, yyyy ... zzzz as described above
...
```

assuming that the folder contains wrappers for unit types `xxxx, yyyy, ... zzzz`.

5.3 Makefile Entry

To automatically make the new module, place its sources in a subdirectory of the *Neo* home directory and choose a name ending with `src` for the subdirectory name (any subdirectory `*src` will be automatically detected by the toplevel Makefile in the *Neo* root directory) and place a (possibly empty) `Makefile.custom` in `*src`. If the *Neo* and *NST* wrapper files are already named `neo_xxxx.c` and `nst_xxxx.c` and no further objects are needed, nothing more needs to be specified.

If the folder needs to be linked with further objects, add a dependency line of the form

```
xxxx.so: neo\_xxxx.o nst\_xxxx.o furtherobj1.o furtherobj2.o
```

to `Makefile.custom`.

Then, issue from the *Neo* home directory the commands

```
make depend_folders
make folders
```

to produce a `xxxx.so` in `o.$ARCH`). If the expected actions do not take place, check the object directory `o.$ARCH` for the presence of `*xxxx.d` files that may be in the way.

6 Importing C-functions via the `prog_unit`

If the above wrapping scheme appears too complex, a simpler (but somewhat more limited) way to import an existing function, e.g., `float myfun(float x1, .. float x5)` is via the `prog_unit`. If the function(s) is implemented in a shared library `mylib.so`, a line

```
#loadlib "mylib.so"
```

in the `prog_unit` will make the function callable from within the `prog_unit`, provided `mylib.c` contains a simple declaration

```
char *FUNCTIONS[] = {"myfun<5>",NULL};
```

telling the `prog_unit` the name and the number of arguments of the function(s). Similar declarations are possible for more complex argument lists, see the man page of the `prog_unit`.

There are also functions for package initialization and for registering and initializing for each calling `prog_unit()` a private data object instance and a destructor for it (again, see `prog_unit()` man page).

7 Using *Neo* circuits from C-Programs with `libnstld`

`Libnstld.so` is a shared library, providing basically those routines of *Neo* that are not connected with GUI operations. It can be thought of consisting those parts of *Neo* below line C in Fig. 2, but with `neo0..15.c` and neo_main.c replaced by routines that offer the *Neo* operations without invoking any GUI graphics (actually, it is largely a more modular re-implementation of many *Neo* routines).

When linked together with `libnst` to an executable, `libnstld.so` offers a set of simple function calls to instantiate *NST* circuits from descriptions saved previously to a file from *Neo* . The following is a simple example:

```
#include "nst.h"                                     /* 1 */
#include "nstld.h"                                   /* 2 */
extern int giNstWidgetFont;                          /* 3 */
unitptr uNew, uHost;                                 /* 4 */
...
init_nst();  uHost = ROOT;                           /* 5 */
nst_init_folders();                                  /* 6 */
nst_load_folders_from_file(".neofolders",".:~");     /* 7 */
nst_font(giNstWidgetFont);                           /* 8 */
uNew = nst_load_circuit_from_file(pcFile,NULL,uHost); /* 9 */
if (uNew) ...                                       /* 10 */
```

After the necessary initialization line 6 reads those folders that must be available for the to-be-loaded circuit. A listing of these folders must be given in a startup file (here `.neofolders`). The second argument is a list of directories where the startup file is searched. Line 7 chooses a font size (`giNstWidgetFont` is a global NST font size parameter), if the circuit uses graphical widgets and can be omitted, if no graphics is used. The main step occurs in line 9: if `pcFile` contains a valid `Neo` circuit description and all necessary folders are available, it will instantiate the circuit or NST unit described in `pcFile` and returns a `unitptr` handle. A return value of `NULL` indicates that the instantiation was unsuccessful).

If the loaded circuit has a definition window with global parameters, a re-parametrized version of the loaded circuit can be made into container `uDest` with the call

```
uNew = nst_remake_circuit(iArg,apcArg,uNew,uDest);
```

where `apcArg` must hold a token `@i=newvalue` for each slot `i` of the circuit's definition window whose parameter is to be changed (`iArg` is the number of els in `apcArg`). Access to input and output values of the created circuit is via the usual *NST* pin access functions/macros. For more details, cf. the document *nstloader.tex*.

8 Special *Neo* Dialog Window Variables

These refer to variable names recognized within the entry slots of *Neo* dialog windows.

`$i` where `i` is nonnegative integer denotes the value held by the i-th data entry widget in the circuit's top-level variable dialog window. These values can be specified on the command line, when the circuit is loaded when *Neo* is started.

`@i` where `i` is nonnegative integer denotes the value held by the i-th data entry widget of the definition window associated with the circuit.

`$EXAMPLE_DIR` is the full path name of the directory where example circuits are located (useful for writing examples that need to access data files).

`$LOAD_DIR` is the full path name of the directory where the circuit was loaded from.

$# ("array index token"). The value of this token depends on the location of the unit, to which the dialog window in which the token occurs belongs. If the unit is the immediate successor of an array_of unit, $# will be the (multi-)index of the currently selected array element (the multi-index is given as a comma-separated list of integers, e.g., $# = 2,3 if the 2,3-array element is selected). If the unit is not the immediate successor of an array_of unit, the token will just be the unexpanded string $#. (See example circuit array_of.NST for an example of the use of this token).

~ (tilde char) in file names, this is substituted by the path specified for the *Neo* command line parameter -HOME. If this is empty, the environment variable $HOME will be used. Otherwise, no substitution occurs.

$HOME Same as tilde char.

9 Special Variables in *Neo* Annotations

@i, $i Same meaning as for a dialog window entry.

MANPAGE: (only if first token in annotation title) Makes annotation define manpage text for loadable circuit. The colon is required!

NAME: (only if first token in annotation title) makes annotation act as a naming command when loading the current circuit as a loadable unit from file. The unit will then get as its name the next token that follows after NAME:. Again, the colon is required!

10 Running *Neo* without Graphics

Neo can be made to run a circuit without opening any own windows (i.e., when the circuit itself contain no window opening commands, everything can then be executed from a ascii terminal without graphics console). To suppress *Neo* 's windows, run the circuit with the command

```
neo -hide -i NameOfCircuit -x nr
```

The -hide flag will suppress *Neo* 's main windows, and the -x will cause the circuit to be run for nr iterations, after which *Neo* exits.

11 Special Identifiers

_init

12 Compiler Flags

OGL when compiled with this flag defined (-DOGL), nst_X11.c and some modules (e.g., nst_iso.c, nst_pixel.c, nst/neo_solid.c) will get OpenGL support compiled in. This will enable the use of 3D OpenGL rendering operations but also requires then to link with the following MesaGL libraries (for Linux) -lMesaGL -lMesaGLU.

NODL ("no dynamic linking") indicates that the operating system has no capability to dynamically load shared libraries. Omits corresponding function calls (and many Neo/Nst features with that). If not specified, linking with -rdynamic -ldl is required (the loaded modules expect to resolve external references into NST, thus -rdynamic is required).

architecture flag one of linux, sun4, sol2, alpha, irix5 to indicate architecture to compile for. If no flag is given, default is linux.

13 Important Header Files

`nst_types.h` defines *NST* data types

`nst.h` basic *NST* routines

`neo.h` definitions for writing *Neo* wrappers

`neo_types.h` *Neo* data types

14 Useful Global Variables and Constants of `libnst`

The following global variables may be useful for programming/debugging work:

`char *pgcNstVersion` version string; set in `nst0.c`

`char *FFMT` float output format of *NST* output functions (default `" % 15.8e"`)

`char *NFMT, *LFMT` dito, for `int` and `long int`

`TRACE_ON` if true, most NST operations become very verbose.

`void (*NST_ERROR_PRINTER)(char*)` addr of user-defined output function to redirect *NST* error messages (NULL to use *NST* default)

`ERRORS_FOUND`

`WARNINGS_FOUND` global error/warning counters; evaluated at first `exec_unit` call.

`NST_STRBLKSIZ` allocation block size for *NST* 's dynamic string functions

`MAXFIELDS` max. nr of input or output fields a unit can have (current limit is 251)

`ROOT` the global container unit that contains all other NST units as subunits

`SECRETROOT` a “parallel universe” that is useful for some operations

`OK` int return value of *NST* methods `exec,adapt,...,remove` to indicate proper operation (value = 0).

`NST_RETURN` if *NST* `exec` or `adapt` returns this value, control passes to the next higher container level.

`NST_BREAK` if *NST* `exec` or `adapt` returns this value while called within a *NST* loop operator, control continues with the first unit after (or before, in case of `adapt`) the loop.

type specifiers: there are two kinds with similar names: `NST_SCALAR`, `NST_FLOAT`, `NST_BYTE` etc. (see `nst0.h`) are `int` and used to specify field types in `primitive_unit` calls. `NST_SCALAR_T`, `NST_FLOAT_T`, `NST_BYTE_T` etc. (see `nst0.h`) are `char` and are used to specify pin types in most other *NST* functions.

`giNstWidgetFont int` number of currently used widget font (*NST* loads a fixed number of fonts in different sizes and selects one by number).

`gfNstCurrentFontDx,Dy` current font width and height in pixels.

`NST_BLANK_ENTRY` numeric value used to indicate that a numeric input field of a *NST* widget is blank.

15 Useful Global Variables and Constants of `libneo`

`giTraceLoading` if true, *Neo* load operations become verbose.