The Nst Loader library `libnstld.a`[1]

Helge Ritter
Faculty of Technology Bielefeld University

Synopsis: The library `libnstld.a` offers routines to instantiate units and circuits from `*.NST` description files as written by *Neo* and to save these circuits back to a *Neo* -loadable file. NST functions can then be called to execute the instantiated unit(s) and to exchange data with the accomodating `C` or `C++` program. In this way, a simple use of *Neo* circuits within existing programs becomes possible.

1 Example

The following code portion contains the essential operations to instantiate a `Neo` circuit that is stored in a file `pcFile`:[2]

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

After the necessary initialization line 5 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. The main step occurs in line 7: 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.

Depending on the format of the description in `pcFile`, the one of the following two cases will happen:

`pcFile` **specifies an "open" circuit:** the circuit will be instantiated as a subcircuit of `uHost` (here: `ROOT`). In this case, `uHost` must be an empty container unit (not checked, but the circuit will not be correctly instantiated otherwise). Return value *is the container handle* `uHost` (**!!**)or `NULL` in case of failure.

`pcFile` **specifies an "encapsulated" circuit:** in this case, the circuit will be instantiated as a subunit within `uHost`. `uHost` must again be a container unit. For adding an encapsulated unit, `uHost` need not be empty, but existing subunits should all be from previous invocations of `nst_load_circuit_from_file()` only. The ordering of the newly instantiated unit among existing units within `uHost` will be as if it had been loaded by `Neo`. Return value is the handle of the newly created subunit in this case (or `NULL` in case of failure).

In both cases, access to the inputs and outputs of the instantiated unit(s) and their execution will be via the usual NST routines.

Initialization. Initialization with data can be either from a separate `*.nst` file (which then must be given as the second argument instead of `NULL` in line 7 [if the basename is the same as for the

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

[2] this example can be found as the file `nstloa.c` in the NST `srvsrc` directory. To make it, type `make nstloa` in the NST directory.

circuitfile, the shorthand "*" can be used]), or – if the second argument is `NULL` as here – the load routine expects the initialization data as a second part of `pcFile` and takes them from there; if a data initialization is undesired, the second argument must be the empty string "" instead of `NULL`.

2 Description of Routines

`nst_init_folders(void)` initialization routine; must be called to initialize folder stuff.

`nst_load_folders_from_file(pcFile,pcPathList)` Call this routine when the to-be-loaded file requires the presence of particular *Neo* folders[3] The routine expects the name `pcFile` of a file specifying all to-be-loaded folders (typically, this is the file `.neofolders`) and a colon-separated path list where to look for the file. The file format is the usual `.neofolders` file format and is more fully described in the *Neo* documentation.

`void nst_set_circuits_load_path(pcHomeDir,pcPath)` specifies a home directory and a colon-separated list of search paths relative to this directory (i.e., when `pcHomeDir` is non-NULL, it is prepended to each relative (i.e., not starting with a '/') path name in `pcPath`) that is used to find the circuit file and or data file passed to `nst_load_circuit_from_file()` (Note that the loading of folders uses an independently defined search path, given as the last argument to routine `nst_load_folders_from_file()`). Defaults are `pcHomeDir=NULL` and `pcPath="./"` and search only therein.

`unitptr nst_load_circuit_from_file(pcCircuit,pcData,uHost)` This is the main routine. It creates a circuit from a *Neo* description file `pcCircuit` and inserts the instantiated unit(s) as subunit(s) in `uHost`. If `pcData` specifies a file name, it will be taken as the data initialization file for the circuit; otherwise, if `pcData="*"`, the data file name will be derived from `pcCircuit` by replacing the `.NST` suffix by `.nst`; otherwise, if `pcData=NULL`, the data specification will be assumed to be appended to file `pcCircuit`; finally if `pcData=""`, this will suppress any loading of data. If `pcCircuit` described an encapsulated circuit, the routine will return a `unitptr` to the instantiated encapsulated circuit; otherwise, the routine will return the `unitptr uHost`. `pcCircuit` can have an appended list (separated with + signs) of folder names (same format as `-i` option for `Neo`. If specified, these will be loaded before the circuit.

`nst_free_inob_t(uNew->inob)` remove "interface object" associated with a NST unit `uNew` NST units that have been constructed from a *Neo* circuit file contain an interface object with additional wrapper information. The interface object is not required to execute (or do any other operation with) the unit; however, its presence is vital if the unit shall be saved back as a circuit file that is readable by *Neo* or that can be re-instantiated by the above routines. The interface object is also not present in NST units created directly from NST's constructor routines (and such "naked" NST units cannot be saved into a *Neo* circuit file). The routine is provided to complete the standard NST `remove_unit()` method, which does not include an automatic freeing of the interface object. The next routine has the necessary `nst_free_inob_t` calls built in.

`unitptr nst_remove_circuit_inside(unitptr uHost)` Convenience routine to remove an open circuit that was instantiated with `nst_load_circuit_from_file(...uHost)` before. (This routine removes all subunits of `uHost`, as well as their and `uHost`'s *inob* interface objects).

[3] Usually, the saved circuits contain the necessary information about to-be-loaded folders in their header part. However, it may happen that this information is out-of-data (e.g., if shared libraries of folders have become moved into different directories etc.). Therefore, it may be preferable not to rely on the specification in a circuit header but instead load the required folders before. This is done by the present routine.

`unitptr nst_fscanf_circuit(FILE *fp, unitptr uHost)` instantiates in `uHost` the circuit as described in file `*fp`. `Fp` is expected to point to the beginning of a properly formatted `*.NST` circuit description, as written by *Neo* . Return value is a pointer to the container of the instantiated circuit, if the file describes an open circuit, or a pointer to the created unit, if the file describes an encapsulated circuit.

`nst_fprintf_circuit(FILE *fp, unitptr uCircuit)` write circuit description for unit `uCircuit` to file `*fp`. Will only work, if `uCircuit` contains an "interface object", i.e., was loaded with one of the above routines. Deals only with circuit, any data must be saved separately (using `save_unit(fp,uCircuit)`).

`char *nst_load_folders_from_string(pcString)` The to-be-loaded folders can also be concatenated to the name of the to-be-loaded circuit, in the same manner as given with the `-i` option on the *Neo* command line (e.g., `pcString="circuit+vector.so+string.so"`). The present routine will then load the necessary folders and return a pointer to a newly allocated string in which only the circuit name has "survived" (i.e., `"circuit"` in the present example).

`char *nst_remake_circuit(iArg,apcValue,uCircuit,uHost)` Remake circuit `uCircuit` with parameters set according to command line arguments in `apcValue[i], i=0..iArg-1`. Setting new parameters requires that the circuit has an associated "modify window" (the modify window is the dialog window that is associated with the circuit's container to define the circuit's parameters). Parameters are identified by the serial position $k = 0, 1, \ldots$ of their entry slot (or button or slider, any widget with a value contributes to the counting) in the modify window. For each parameter to be changed, `apcValue` must provide a pointer `apcValue[i]` to a string of the form `@k=newvalue`, where k is the parameter's entry slot position, as explained above. Each `@k=newvalue` specification replaces the k-th parameter in `uCircuit's` top level definition window (if it has one) by `newvalue` (limit: currently, any part of `newvalue` beyond 999 chars is ignored). Then a new instance of the circuit is made and placed in `uHost` (which initially should be empty). The old circuit `uCircuit` is not discarded by this routine. Therefore, `uHost` must be different from the host unit of `uCircuit` (otherwise the new circuit gets copied into the container of the old circuit while copying still is in progress, resulting in an infinite recursion or other undesireable events). If the remake was successful, the return value will be the handle of the new circuit, `NULL` otherwise.

3 How to Reparametrize a Loaded Circuit

Assuming that `pcFile` is the name of a circuit file with a circuit that has a modify window with five entries, changing the second and the last entry (assuming them of types *numeric* and *string*) would require the following additions to the previous code (lines have been renumbered):

```
...
char *apcValue[2];
unitptr uHost;
...
uHost = void_unit(0,0,ROOT);                          /* 1 */
uNew = nst_load_circuit_from_file(pcFile,NULL,uHost); /* 2 */
apcValue[0] = "@1= 47.11";                            /* 3 */
apcValue[1] = "@4=hello world!\\nAnother line";       /* 4 */
uNew = nst_remake_circuit(2,apcValues,uNew,ROOT);     /* 5 */
if (uNew) ...                                         /* 6 */
```

Note the following points: (i), the loading of the (open) circuit can now no longer be into `ROOT` (since then we would obtain `uNew = ROOT` in line 2, with no "free spot" left over to accomodate the reparametrized copy), therefore we make an auxiliary container in line 1. (ii) Lines 3 and 4 set the two values for entry slots 1 (the second) and 4 (the last). Note that (iii) special characters, such as newlines, must be quoted with a backslash. Note also that (iv) there may be no space between the initial `'@'`, the following slot number, and the `'='` that introduces each assignment. Finally, line 5 performs the remake operation, and the returned handle will be a pointer to a reparametrized copy of the previous `uNew`, placed as a subunit into `ROOT`.

To make this run, compile and link it with `libnstld.a` and `libnst.a` (and, of course, `-lX11 -lXext -lm`).

4 Remarks

The present library represents the core set of *Neo* functions that must be added to NST to load folders and circuits and to save circuits back to circuit files. The usual NST functions can be used to execute, adapt or control the instantiated unit(s) or to exchange data between them and the accommodating `C` or `C++` program (the NST routines can also be used to restructure the loaded circuits; however, this will not be properly reflected when the restructured circuit is saved back: this would require special editing functions which also update the *Neo* wrapper information. Such routines are not very difficult to write, but presently not provided.

`libnstld.a` itself contains no graphics calls; therefore, if the instantiated NST units and the loaded folders also do not use any graphics, it suffices to link against a NST library version in which the graphics parts have been removed. This should make it fairly easy to port the then remaining functionality to other systems.

5 Compilation

If a subset of folders shall be compiled into `libnstld.a` (analogous to the compiled folders in *Neo*), they must be listed in `Nst_defs.h` (in similar format as in `neo_defs.h` for *Neo*).

A few definitions are no available in `Nst.h` and require occasional minor changes: to set the label of the current icon, routine `neo_set_iconlabel(pcLabel)` must be used.

The sources for `libnstld.a` are prepared to become extended to a server exchanging messages with a separate `''neo client''` containing the GUI.

6 Predefined Constants

Locations for the manual directories and suffixes are compiled in and are currently defined at the beginning of `Nst_file.c`. The settings are:

```
#define MANPATH          "~/nst5/"
#define MANDIR_NROFF     "man/cat3"
#define MANSUFFIX_NROFF ".3"
#define MANDIR_HTML      "man/html3"
#define MANSUFFIX_HTML   ".html"
#define EXAMPLE_DIR      "man/examples"
```