GNU Nana

Improved support for assertions and logging in C and C++
last updated 27 February 1998 for version 1.14

P.J.Maker (pjm@cs.ntu.edu.au)

1 Introduction

Nana is a library that provides support for assertion checking and logging in a space and time efficient manner. The aim is to put common good practise[1] into a library that can be reused rather than writing this stuff every time you begin a new project.

In addition assertion checking and logging code can be implemented using a debugger rather than as inline code with a large saving in code space.

Nana aims to solve the following problems:

1. Avoid the two executables problem (one with asserts in and another without any).

   The code space and time costs of having assertion checking and detailed logging code in a program can be high. Normally people construct two versions of the program, one with checking code for testing and one without checking code for production use.

   With nana one version of the executable can be built for both testing and release since debugger based checking has negligible space and time impact.
2. Configurable: the nana library is designed to be reconfigured by the user according to their needs. For example we can:
   - Modify the behaviour on assertion failure, e.g. to attempt a system restart rather than just shutting down.
   - Selectively enable and disable assertion checking and logging both at compile and run time.
   - Send the logging information off to various locations, e.g.
     - Users terminal
     - A file for later checking.
     - Another process, e.g. a plotting program or a program that verifies that the system is behaving itself.
     - A circular buffer in memory.

       This is an old embedded systems trick and is very useful for production systems. The time cost of logging into memory is not large and when your production system in the field has problems you can then see what was happening in the minutes before its unfortunate demise rather than asking some user what was happening before it died.
3. Time and space efficient.

   For example the GNU '`assert.h`' implementation uses 53 bytes for '`assert(i>=0)`' on a i386. The nana version using the i386 '`stp`' instruction on assert fail uses 10 bytes. If your willing to accept the time penalty this can be reduced to 0 or 1 byte by using debugger based assertions.
4. Support for formal methods.
   - Before and after state (e.g. x,x' in the Z notation).

     Specifications are often written in terms of the state of variables before and after an operation. For example the '`isempty`' operation on a stack should leave the stack unchanged. To verify this in nana we could use:

     ```
     bool isempty(){ /* true iff stack is empty */
       DS($s = s); /* copy s into $s in the debugger */
       ...; /* code to do the operation */
       DI($s == s); /* verify that s hasn't been changed */
     ```

[1] Which is unfortunately quite uncommon in the authors experience.

```
}
```

These '`$..`' variables are called convenience variables and are implemented by gdb. They have a global scope and are dynamically typed and initialised automatically to 0.

In addition a C only version of before and after state is provided. For example:

```
bool isempty() { /* true iff stack is empty */
  ID(int olds); /* declare variable to hold old value */
  IS(olds = s); /* copy s into $s in the debugger */
  ...; /* code to do the operation */
  I(olds == s); /* verify that s hasn't been changed */
}
```

- Support for Predicate Calculus.

  Nana provides some support for universal (forall) and existential (exists one or more) quantification. For example to specify that the string v contains only lower case letters we could use:

  ```
  I(A(char *p = v, *p != '\0', p++, islower(*p)));
  ```

  These macros can be nested and used as normal boolean values in control constructs as well as assertions. Unfortunately they depend on the GNU CC statement value extensions and so are not portable. The following macros are defined in '`Q.h`':

  `A` For all values the expression must be true.

  `E` There exists one or more values for which the expression is true.

  `E1` There exists a single value for which the expression is true.

  `C` Returns the number of times the expression is true.

  `S` Returns the sum of the expressions.

  `P` Returns the product of the expressions.

- Verifying timing.

  As well as using nana to verify timings with assertions using a hardware supported timer you can also a simulator (e.g. the PSIM power pc simulator by Cagney) with gdb. These simulators can model time and provide a register called '`$cycles`' which represents the current cycle count of the program. This can be used to check that timing constraints are being meet.

  ```
  void process_events() {
    for(;;){
      DS($start = $cycles);
      switch(get_event()){
        case TOO_HOT:
          ...;
          DI($start - $cycles <= 120);
          break;
        case TOO_COLD:
          ...;
          DI($start - $cycles <= 240);
          break;
      }
    }
  }
  ```

The intended audience for Nana includes:

- Software Engineers.
- Formal methods community.
- Real time programmers.
- System testers.
- People teaching programming.

1.1 Related work

The Nana project was inspired by some other projects, in particular:

- Anna - Anna stands for "Annotated Ada" where the programmer inserts various assertions into the code which can be automatically validated. To quote from the WWW Virtual Library entry on Anna:

  > Anna is a language for formally specifying the intended behaviour of Ada programs. It extends Ada with various different kinds of specification constructs from ones as simple as assertions, to as complex as algebraic specifications. A tool set has been implemented at Stanford for Anna, including:
  >
  > 1. standard DIANA extension packages, parsers, pretty-printers;
  > 2. a semantic checker;
  > 3. a specification analyser;
  > 4. an annotation transformer; and
  > 5. a special debugger that allows program debugging based on formal specifications
  >
  > All tools have been developed in Ada and are therefore extremely portable. Anna has thus been ported to many platforms. For more information send e-mail to "anna-request@anna.stanford.edu". Before down loading the huge Anna release, you may wish to copy and read some Anna LaTeX reports.

  Anna is available from: `ftp://anna.stanford.edu/pub/anna`
- Eiffel - the Eiffel programming language provides support in the language flexible assertion checking. To quote from the Eiffel page in WWW Virtual library:

  > Eiffel is a pure object-oriented language featuring multiple inheritance, polymorphism, static typing and dynamic binding, genericity (constrained and unconstrained), a disciplined exception mechanism, systematic use of assertions to promote programming by contract, and deferred classes for high-level design and analysis.
- APP - Annotation PreProcessor. The APP was written by David S. Rosenblum and provides assertion checking functions for C and C++. It is implemented using a preprocessor wrapper around the C preprocessor and supports quantifiers and before/after state.

  See "A Practical Approach to Programming with Assertions" in Vol 21, No. 1, January 1995 of IEEE Transactions on Software Engineering for an interesting paper describing APP. Unfortunately the APP tool doesn't seem to be freely available (I'm willing to be corrected on this). Note that any similarity between my examples and David's are due to morphic resonance.

Nana is essentially a poor mans implementation of some of these ideas which works for C and C++. Ideally in the best of all possible worlds you might want to look at Eiffel or in the military world Ada and Anna. If you use TCL/TK you might also be interested in Jon Cook's '`AsserTCL`' package.

1.2 Assert.h considered harmful

Most C programmers become familiar with assertions from the the `assert.h` header. As such its a very good thing and has a nice simple implementation. However it is also inefficient and leads some people to the conclusion that assertion checking is an expensive luxury.

The implementation of `assert.h` as distributed with `gcc` looks like the following (after a bit of editing):

```
# ifndef NDEBUG
# define _assert(ex) {if (!(ex)) \
                         {(void)fprintf(stderr, \
                           "Assertion failed: file \"%s\", line %d\n", \
                           __FILE__, __LINE__);exit(1);}}
# define assert(ex) _assert(ex)
# else
# define _assert(ex)
# define assert(ex)
# endif
```

There are are two main problems with this:

1. Code space overhead: each call to '`assert`' generates 2 function calls with 4 and 1 arguments plus strings for error messages. If `assert.h` had library code support we could make the implementation much more space efficient, e.g. by calling a single function on error detection.
2. The default behaviour simply prints a message and dies, ideally you like to be able to use a debugger to determine why the assertion failed. Even if you run this under the debugger you can't observe the failures of variables are an assert failure because the process exits rather than aborting back to the debugger.

Of course everyone merely rewrites their own '`assert`' macro so these are not significant objections. The only problem is if the author uses the libraries without modification.

1.3 Scope of this document

This document aims to both describe the library and provide a tutorial in its use. Further work is required, particularly on the tutorial sections. If anyone has any suggestions please send them to me.

2 Installing the Nana library

Nana uses the normal GNU install method in the same way as 'gcc' and 'gdb'. To install nana in the default location '/usr/local/{bin,lib,include}' you would use:

```
% gzcat nana-1.10.tar.gz | tar xvf -
% cd nana-1.10
% ./configure
% make
% make install
% make check
% make check-mail
% make subscribe
```

The *check-mail* and *subscribe* targets both send e-mail. If you need to change the mailer used try something like:

```
% make MAIL=elm subscribe
```

Note: we need to install nana before running the 'make check' target. The 'check-mail' target sends the test report via e-mail to the 'nana-bug@cs.ntu.edu.au'.

Of course things are never that simple. If you want to install Nana in a different location or change the behaviour on error detection see Section 2.3 [Configure], page 6.

Each of the sub-directories nana can be compiled and installed separately, e.g. if you don't need the documentation you can just compile and install from the 'src' sub-directory after doing the configure statement.

2.1 Required Software

The following software is possibly required to run nana.

gcc-2.7.2
: Nana makes use of two GNU extensions in its library so you really should be using 'gcc'. Some of the code can be used with any C compiler, though it may not be worth the bother. The dependencies on gcc are in 'Q.h' which uses the statement value extension and in 'L.h' which uses the variable number of arguments extension to 'cpp'.

gdb-4.16
: A recent version of 'gdb' is worthwhile, some early 4.?? versions had problems setting a large number of breakpoints.

gmake
: The 'configure' script and 'Makefiles' are generated using the 'automake' and 'autoconf' programs. They should be reasonably portable but if you have problems try using GNU make. For example on some old DEC boxes we have had strange behaviour using the system make.

For a listing of porting results including software versions see:
'http://www.cs.ntu.edu.au/homepages/pjm/nana-bug/'

2.2 Optional Software

In addition to the required software you might also be interested in:

- '`http://www.cs.tu-bs.de/softech/ddd/`' – a smart frontend for gdb which can display dynamic data structures such as linked lists, etc.
- '`ftp://ftp.ci.com.au/pub/psim/`' – a cycle level simulator for the PowerPC. A fine piece of work.

2.3 Configure

Nana uses a standard GNU `autoconf` generated `configure` script. The `configure` script checks the setup on your machine and then generates the appropriate Makefiles. Some of the things checked by configure include:

1. Which compiler, compiler flags and libraries to use, e.g. you might need to include a `-lposix` flag to the linker to build programs on your machine.
2. Which header (.h) files are available on this machine, e.g. is `unistd.h` available on this machine.
3. Where to install programs, header file, man pages, etc.

In addition '`configure`' uses the host architecture and operating system to generate the '`nana-config.h`' file. This file contains some macro definitions which define how nana works on particular operating systems and hardware architectures.

For example on '`i386`' machines we would use the '`asm("hlt")`' instruction whenever an assertion fails, on a '`sparc`' we would use '`asm("stp")`'. Otherwise we would default to a plain C call to '`abort()`' If '`configure`' does not recognise your machine it uses plain C code.

You may wish to change these defaults on installation, one method is to edit a local copy of the '`nana-config.h`' file. Alternately you can define the code yourself in the call to '`configure`'. For example to redefine the action we take when an error is detected by the `I` macro we can use:

```
I_DEFAULT_HANDLER="restart_system()" ./configure
```

As well as simple calls to routines various other bits of information are passed off to the '`I_DEFAULT_HANDLER`' such as the expression that failure and a failure code. For example:

```
% I_DEFAULT_HANDLER="restart(line,file,param)" ./configure
```

The default for '`I_DEFAULT_HANDLER`' calls a function which prints a message and then dumps core. Different behaviour on failure can be organised by setting the '`I_DEFAULT`' to '`fast`', i.e. plain core dump or '`verbose`' which prints an error messsage and then does the core dump.

```
% I_DEFAULT=fast ./configure
```

For nana the following examples may be useful:

1. `./configure`

   Accept the default values for everything. In particular the files will be installed in:

   '`/usr/local/{bin,include,lib,man,info}`'
2. `./configure --prefix=~project/tools`

   Install the files into:

   '`~project/tools/{bin,include,lib,man,info}`'
3. `./configure --bindir=~project/bin --libdir=~/project/lib \`
   `--includedir=~/project/headers --infodir=/usr/local/info \`
   `--mandir=~/project/doc`

   The install directory for program ('`bin`'), etc can all be set with command line arguments to'`configure`'.
4. `CC=xacc LIBS=-lposix ./configure sun3`

   If the defaults chosen by '`configure`' are not correct you can override them by setting variables such as `CC` before calling '`configure`'. The '`sun3`' argument is used to identify the machine we are running on and may be necessary on some machines.
5. `./configure --help`

   And of course when in doubt ask for help.

For even more details see the file '`INSTALL.con`' which contains the generic instructions for use with '`autoconf`' generated '`configure`' scripts.

2.4 Variables for ./configure

The configure program uses the following shell variables to change various defaults. Another method is simply to edit the '`nana-config.h`' file. Most of these values should be auto detected, so you can ignore this section until your need to save a few bytes of store by using '`asm("hlt")`' instead of a call to '`abort()`'.

`DI_MAKE_VALID_BREAKPOINT`

This text is inserted when the '`DI.h`' library needs to set a breakpoint in the generated code. It should ideally update all variables which being kept in registers etc so that gdb gets the correct values for each variable.

Possible values include:

1. '`asm("nop")`' – a single '`nop`' instruction to set the breakpoint at.

   This is the default.
2. '`_vi = 0`' – where '`_vi`' is a global volatile int.
3. '`_vi = (exprn)`' – where *exprn* is the expression we are checking for this assertion.
4. '`/* nothing */`' – nothing at all, this means the breakpoint will be set at the start of the next statement which works most of the time. However for some examples this will do the wrong thing.

`DL_MAKE_VALID_BREAKPOINT`

Used for the same purpose as '`DI_MAKE_VALID_BREAKPOINT`' for '`DL.h`'. It also defaults to '`asm("nop")`'.

`I_DEFAULT_HANDLER`

The code called when '`I.h`' detects an error.

`asm("hlt")`

Some machines use a '`hlt`' instruction.

`asm("stp")`
: And other machines use a '`stp`' instruction.

`abort()`
: Or we could use a call to '`abort`' which is at least standard C. On some machines this is significantly larger than a single '`hlt`' instruction.

`restart()`
: Or a call to a function which attempts to restart the system.

`ALWAYS_INCLUDE_MALLOC`
: This is a dodgey for some versions of Linux which don't seem to include '`malloc`' when you include '`stdio.h`' and use '`print`'. This causes problems for '`gdb`' since it uses '`malloc`' in the executable to implement parts of its functionality.

  This kludge should be removed!

2.5 Supported Platforms

Nana has been tested on the following platforms:

1. i386-unknown-linux, gcc-2.7.0, gdb-4.16
2. sparc-sun-sunos4.1.4, gcc-2.7.2.f.1, gdb-4.16
3. sparc-sun-solaris2.3, gcc-2.7.2, gdb-4.16
4. alpha-dec-osf3.2, gcc-2.7.2, gdb-4.16
5. mips-sgi-irix5.3, gcc-2.7.0, gdb-4.16
6. powerpc-ibm-aix3.2.5, gcc-2.6.3, gdb-4.16

The '`alpha-dec-osf3.2`', '`mips-sgi-irix5.3`' and '`powerpc-ibm-aix3.2.5`' implementations have problems when you compile with '`-O2`' or '`-O3`' optimisation. This causes some errors in the the debugger based assertion and logging code since variables can be removed or changed by optimisation. At '`-O`' everything passes. Regardless of optimisation the C based checking code passes all tests on these platforms.

If you use nana on a new platform please send the '`check.log`' file to '`nana-bug@cs.ntu.edu.au`' via the '`make check-mail`' command. A machine generated list of this information is available at:

'`http://www.cs.ntu.edu.au/homepages/pjm/nana-bug/`'

2.6 Supported Debuggers

Currently Nana works with the GNU GDB debugger which is available on a wide range of platforms including embedded systems and even provides support for remote debugging. Porting to any reasonable debugger with conditional breakpoints and commands is not very difficult.

As an example of an unreasonable debugger, Nana has been ported to work with the MicroSoft Codeview debugger. The port is small (60 lines of code) but suffers from a problem with variable scoping in Codeview. If a breakpoint is set at a point in the code the expressions are not evaluated from that particular scope. For example setting a breakpoint in the function `f` cannot access a variable local to `f` directly. Codeview has a unique (expletive deleted) scope operator which you must use to set the scope '`{...}`'. This makes the interface somewhat less than beautiful.

Another good thing about CodeView is to try a debug command which prints a message which contains a single open '{'. This of course causes it to hang and was the main problem during the porting to CodeView which took a whole day[1].

If anyone is interested I may release the CodeView implementation, please contact me if you are interested. Of course a better bet is probably to move to the '`gdbserver`' system. I think '`gdb`' has been released as a native even for some MicroSoft operating systems.

Other debuggers like DBX don't seem to be worth the trouble since gdb works on those machines. A redesign of the nana internals may also be useful if we decide portability between debuggers is actually useful.

2.7 Known Problems

Nana has the following known features (or perhaps problems):

1. Nana macros which use the debugger such as `DI` or `DL` should be on lines by themselves. If you mix code and nana macros on the same line you will get errors, e.g:
   ```
   main(){
      int x;
      x = 5; x--; DI(x == 4);
   }
   ```
   This doesn't work since breakpoints are set at line boundaries rather than statement ones. Of course anyone who writes code like this deserves whatever happens to them.
2. Optimisation can remove variables so that debugger based assertions ('`DI.h`') do not work correctly. As usual the interaction between the debugger and the compiler is rather complicated. This may not be a problem if the appropriate compile-time flags are selected, e.g. '`-O0 and -O1`' work on most platforms.
3. The '`Q.h`' macros depend on the statement value extension to GNU CC so if you wish to use them you must use GCC. This can be fixed for C++ in a possibly useful manner, I can't see any solution for C.
4. The logging macros depend on the Var Args extension provided by the GNU C Preprocessor.[2] We could (probably will) implement a fix for this based on the tricks in the C FAQ. Unfortunately these tricks are not pretty. For now interested users could simply replace their CPP with the GNU CPP if they wished to stay with non-standard compilers.
5. The '`Q.h`' macros do not work in the debugger since '`gdb`' does support the statement expression extension.
6. Multiline expressions do not work as expected in the debugger since you need to use a blackslash as an escape at the end of the line. For example:
   ```
   DI(x +
               10 > 30);
   ```
   A few backslashes may solve this particular problem.
7. Problems with the '`configure`' script.

   The '`configure`' script automatically detects the target operating system and architecture and then generates '`nana-config.h`'. If the options selected in '`nana-config.h`' are incorrect

[1] And about 60 reset cycles where the machine went off into hyperspace

[2] This allows variable number of arguments to C preprocessor macros.

they can be edited by hand and installed in the usual include directory. The easiest method is simply to delete all macros in '`nana-config.h`' since the system defaults to more portable (and less efficient) implementations. If you wish to do this from the configure script you can try giving a unsupported machine type, e.g.

```
% ./configure pdp11-dec-ultrix
```

8. Some users have reported problems with the `configure` script detecting `vsnprintf`. If `configure` doesn't find it and it does exist then simply define it in '`nana-config.h`' as per the previous question.

   If `vsnprintf` really doesn't exist then get a new C library, possibly the GNU libc.
9. The use of `vsprintf` opens a security hole since no bounds checking is done by it. Nana attempts to use `vsnprintf` which is safe when it exists but it will resort to `vsprintf` if it can't find `vsnprintf`. All careful people should make sure that they have a library with `vsnprintf`.
10. `STL` header file errors due to nana.

    The C++ `STL` header files for version 3.0 at least must be included before the `Q.h` file. The problem is caused by the STL files using `S` as a template argument. Of course `Q.h` uses `S` for summing a series. Namespace pollution strikes again.

    (Thanks to Han Holl for this particular problem).

2.8 Bug Reports

If you think you have found a bug in the Nana library, please investigate it and report it.

- Please make sure that the bug is really in the Nana library.
- You have to send us a test case that makes it possible for us to reproduce the bug.
- You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way. Make sure that the bug report includes all information you would need to fix this kind of bug for someone else.

If your bug report is good, we will do our best to help you to get a corrected version of the library; if the bug report is poor, we won't do anything about it (apart from asking you to send better bug reports).

Send your bug report to:

'`nana-bug@cs.ntu.edu.au`'

Copies of bug reports will be kept at:

'`http://www.cs.ntu.edu.au/homepages/pjm/nana-bug/`'

2.9 New Versions

New versions of nana will be made available at:

'`ftp://ftp.cs.ntu.edu.au/pub/nana/`'

If you wish to be informed about new releases of nana then subscribe to the nana mailing list. Send a message containing '`subscribe`' <your e-mail address> to:

'`mailto:nana-request@it.ntu.edu.au`'.

A hypermail archive of this list is kept at:

‘`http://www.cs.ntu.edu.au/hypermail/nana-archive`’

If you wish to send a message to the list send it to ‘`mailto:nana@it.ntu.edu.au`’.

3 Invoking Nana

The functions defined by Nana are implemented either as pure C code or as a set of commands which are generated for the debugger. To use the C based support for assertion checking you would use something like:

```
#include <nana.h> /* this file includes the other nana .h files */

int floor_sqrt(int i) { /* returns floor(sqrt(i) */
  int answer;
  I(i >= 0); /* assert(i >= 0) if i -ve then exit */
  ...; /* code to calculate sqrt(i) */
  L("floor_sqrt(%d) == %d\n",
        i, answer);  /* logs a printf style message */
}
```

To compile and link the previous code you may need to use the '`-Ipath`' or '`-lnana`' flags with the compiler. For example:

```
% gcc toy.c -lnana
```

If the nana headers have been installed in a strange location you may need to do something like:

```
% gcc -I<strange location> toy.c -lnana
```

The next example uses the debugger versions of '`L`' and '`I`'. If the code is run under the debugger these checks will occur, otherwise they take up a negligible amount of space and time.

```
#include <nana.h> /* this includes the other nana .h files */

int floor_sqrt(int i){
  int answer;
  DI(i >= 0); /* assert(i >= 0) if i -ve then exit */
  ...; /* code to calculate sqrt(i) */
  DL("floor_sqrt(%d) == %d\n", i, answer);  /* logs a printf style message */
}
```

To generate the debugger commands from the C source we just run the '`nana`' filter over the program and then execute the commands under gdb using the '`source`' command.

```
% nana sqrt.c >sqrt.gdb
% gdb a.out
(gdb) source sqrt.gdb
breakpoint insert: ...
(gdb) run
...
(gdb) quit
```

Note that any C preprocessor flags which you use must be passed off to the '`nana`' command. The best way to do this of course is in a Makefile. Something like the following works for GNU Make:

```
%.nana: %.c
        nana $(CFLAGS) $< >$@
```

The '`nana`' filter can also be run over multiple source files in a single run if thats more convenient.

For convenience a number of other simple scripts are provided, in particular to:

`nana-run` Run a program under the debugger without prompting, etc. For example:

```
% nana-run a.out -x main.gdb
output from program
```

`nana-clg` Compiles the program, generates the debugger commands and the runs the program using '`nana-run`'. For example:

```
% nana-clg -O3 main.c
output from program
```

You can change the compiler invoked by '`nana-clg`' by redefining the '`NANACC`' environment variable. For example:

```
% NANACC=g++ nana-clg -O3 main.cc
```

The installation also '`nana-c++lg`' which compiles your code using a GNU C++ compiler.

`nana-trace`

Generates a line by line trace of the execution of a program using GDB. For example:

```
% nana-trace a.out
54          printf("main()\n");
55          x = distance(5,-5);
distance (i=5, j=-5) at test.c:47
47          i = -i;
48          j = -j;
...
```

The arguments to '`nana-trace`' are passed directly to GDB. If you wish display variables or call procedures on each line then could use something like:

```
% nana-trace -x mycommands.gdb a.out
```

Where the '`mycommands.gdb`' contains the GDB commands such as '`display x`' which causes '`x`' to be printed every time the debugger gets control of the program.

4 Interface

This section describes the details of the interface to nana library.

All of the files can be included multiple times without ill–effect since they use the C preprocessor to make sure the header declarations are only seen the first by the compiler. Each of the files can also be included individually.

If any of the following routines have an internal problem (e.g. malloc fails due to lack of memory) they will call the '`nana_error`' function defined in '`nana_error.c`'. By default this function prints a message and dumps core using '`abort`'. If you wish to override this behaviour you should define your own handler before linking in the nana library.

4.1 nana.h: the main header file

The '`nana.h`' file includes most of the other files in the library. In particular it '`#include's`' the following files:

`I.h`

`DI.h`

`L.h`

`DL.h`

`Q.h`

`GDB.h`

4.2 WITHOUT_NANA: disabling all nana code for portability.

If you wish to disable all nana code you can '`#define`' the '`WITHOUT_NANA`' macro. This selects versions of the macros defined in '`I.h`','`L.h`', etc which map to '`/* empty */`'.

So if you are using nana for your development but don't wish to force your customers to use it you can add an option to your '`configure`' script to define/undefine '`WITHOUT_NANA`'. In addition you will need to distribute copies of the nana header files with your package to get the stubs.

Note that the '`L.h`' and '`DL.h`' macros use the macro variable number of arguments extension provided by GNU C. If you wish your code to be portable you should use the macros '`VL((..))`', etc rather than '`L(..)`' to avoid problems with non GNU C preprocessors which only take a fixed number of arguments.

4.3 I.h: C based invariant checking

This implements the C based invariant checking code and is a replacement for '`assert.h`'. The first two macros are the normal user interface; the remainder are used for configuring the behaviour on failure, etc.

void **I** (bool *exprn*) Macro

The *exprn* should always be true if the program is correct. If the *exprn* is false a message will be printed, followed by core dump.[1]

Checking can be enabled and disabled by using the *I_LEVEL* and *I_DEFAULT_GUARD* macros. See the definitions below for these macros for further details.

Note that *exprn* should have no side-effects[2] since disabling checking shouldn't change your programs behaviour.

```
I(z != 0);
x = y / z;
```

void **N** (bool *exprn*) Macro

The opposite of 'I', i.e. the expression must never ever be true if the program is working properly. It is equivelant to `I(!(e))` and exists as a piece of syntactic sugar which may be helpful for complicated boolean expressions.

```
char* strdup(char *s) {
  N(s == NULL);
  ...;
}
```

int **I_LEVEL** Macro

The '`I_LEVEL`' macro is used to globally enable and disable checking by the macros in this file. It can take on one of three values:

`0` Disable all checking. Regardless of anything else no code will be generated for `I`, `N`, etc.

`1` Enable checking only if the corresponding guard condition is true. The guard condition can be used to enable and disable checking at compile and run time.

`2` Enable all checking regardless of guard conditions.

`I_LEVEL` defaults to `1`.

bool **I_DEFAULT_GUARD** Macro

The `I_DEFAULT_GUARD` is used to selectively enable or disable checking at compile or run time.

`I_DEFAULT_GUARD` defaults to `TRUE`, i.e. always enabled.

A user would typically define `I_DEFAULT_GUARD` to be global or local variable which is used to turn checking on or off at run–time. For example:

---

[1] If you don't want a core dump then look at stopping the core dumps with `ulimit` rather than changing the handler.

[2] Side-effects include such operations as input/output or assignments, e.g. '`x++`'

```
#define I_DEFAULT_GUARD i_guard > 0

extern int i_guard;
```

text **I_DEFAULT_PARAMS** Macro

This is passed off to the `I_DEFAULT_HANDLER` and defaults to nothing, it is just some text and is intended to pass failure codes (e.g. `IEH303`) or requests (e.g. `HW_DEAD`) information off to the handler.

`I_DEFAULT_PARAMS` defaults to nothing.

void **I_DEFAULT_HANDLER** (char **exprn*, char **file*, int *line*, *param*) Macro

When an error is detected the `I_DEFAULT_HANDLER` will be called to handle the error. The arguments are:

`exprn` A string representation of the expression that has failed, e.g. `"I(i>=0)"`.

`file` The file that this error occurred in, i.e. `__FILE__`.

`line` The line number for the error, i.e. `__LINE__`.

`param` An optional parameter which can be passed across which defaults to `I_DEFAULT_PARAMS`. This can be used to pass failure codes or other information from the checking code to the handler.

All of the remaining macros are used to individually override the default values defined above. Normally these macros would be used in a system wide header file to define macros appropriate for the application. For example you might use '`IH`' to define different checking macros for hardware and software faults.

void **I** (bool *e*) Macro
void **IG** (bool *e*, bool *g*) Macro
void **IH** (bool *e*, Macro *h*) Macro
void **IP** (bool *e*, Text *p*) Macro
void **IGH** (bool *e*, bool *g*, Macro *h*) Macro
void **IGP** (bool *e*, bool *g*, Text *p*) Macro
void **IHP** (bool *e*, Macro *h*, Text *p*) Macro
void **IGHP** (bool *e*, bool *g*, Macro *h*, Text *p*) Macro
void **N** (bool *e*) Macro
void **NG** (bool *e*, bool *g*) Macro
void **NH** (bool *e*, Macro *h*) Macro
void **NP** (bool *e*, Text *p*) Macro
void **NGH** (bool *e*, bool *g*, Macro *h*) Macro
void **NGP** (bool *e*, bool *g*, Text *p*) Macro
void **NHP** (bool *e*, Macro *h*, Text *p*) Macro
void **NGHP** (bool *e*, bool *g*, Macro *h*, Text *p*) Macro

We also provide support for referring to previous values of variables in postconditions. The `ID` macro is used to create variables to save the old state in. The `IS` and `ISG` macros are to set these values.

`void` **ID** (`Text` *decln*) Macro
`void` **IS** (`Text` *assignment*) Macro
`void` **ISG** (`Text` *decln*, `bool` *g*) Macro

For example:

```
void ex(int &r) {
  ID(int oldr = r); /* save parameter */
  g(r);
  I(oldr == r); /* check r is unchanged */
  while(more()) {
    IS(oldr = r); /* assign r to oldr */
    h(r);
    I(oldr == r * r);
  }
}
```

4.4 DI.h: debugger based invariant checking

This implements the debugger based invariant checking code. The first two macros are the normal user interface; the remainder are used for configuring the behaviour on failure, etc. Note that these macros have no effect unless you run your program under the debugger and read in the commands generated by the '`nana`' command.

`void` **DI** (`bool` *exprn*) Macro

The *exprn* should always be true if the program is working. If it is true then nothing happens otherwise the code given by '`DI_DEFAULT_HANDLER`' will be called which by default prints a message and dies just like '`assert.h`'.

The checking using *DI* can be enabled and disabled by using the *DI_LEVEL* and *DI_DEFAULT_GUARD* macros. See the definitions below for these macros for further details.

Note that *exprn* should have no side-effects[3] since disabling the checking shouldn't change your programs behaviour.

`void` **DN** (`bool` *exprn*) Macro

The opposite of '`DI`', i.e. the expression must never ever be true if the program is working properly. It is equivelant to `I(!(e))` and exists as piece of syntactic sugar which is helpful for complicated boolean expressions.

`int` **DI_LEVEL** Macro

The '`DI_LEVEL`' macro is used to globally enable and disable checking, in particular it can take on one of three values:

`0` Disable all checking. Regardless of anything else no code will be generated for `DI`, `DN`, etc.

[3] Side-effects include operations like input/output or assignments.

`1` Enable checking only if the corresponding guard condition is true. The guard condition can be used to enable and disable checking at compile and run time.

`2` Enable all checking regardless of guard conditions, etc.

`DI_LEVEL` defaults to `1`.

`bool` **DI_DEFAULT_GUARD** Macro

The `DI_DEFAULT_GUARD` is used to selectively enable or disable checking at compile or run time.

`DI_DEFAULT_GUARD` defaults to `TRUE`, i.e. always enabled.

A user would typically define `DI_DEFAULT_GUARD` to be global or local variable which is used to turn checking on or off at run–time. For example:

```
#define DI_DEFAULT_GUARD (i_guard)

extern int i_guard;
```

`text` **DI_DEFAULT_PARAMS** Macro

This is passed off to the `DI_DEFAULT_HANDLER` and defaults to nothing, it is just some text and is intended to pass failure codes (e.g. `IEH303`) or requests (e.g. `HW_DEAD`) information off to the handler.

`DI_DEFAULT_PARAMS` defaults to nothing.

`void` **DI_DEFAULT_HANDLER** (`char *`*exprn*, `char *`*file*, `int` *line*, *param*) Macro

When an error is detected the `DI_DEFAULT_HANDLER` will be called to handle the error. The arguments are:

`exprn` A string representation of the expression that has failed, e.g. `"I(i>=0)"`.

`file` The file that this error occurred in, i.e. `__FILE__`.

`line` The line number for the error, i.e. `__LINE__`.

`param` An optional parameter which can be passed across which defaults to `DI_DEFAULT_PARAMS`. This can be used to pass failure codes or other information from the checking code to the handler.

`void` **DI_MAKE_VALID_BREAKPOINT** (`exprn` *e*) Macro

This macro is used to ensure that a breakpoint can be set at the location we are checking using `DI`, etc. It defaults to `asm("nop")` and can be redefined by the user.

void **DI** (bool *e*) Macro
void **DIG** (bool *e*, bool *g*) Macro
void **DIH** (bool *e*, Macro *h*) Macro
void **DIP** (bool *e*, Text *p*) Macro
void **DIGH** (bool *e*, bool *g*, Macro *h*) Macro
void **DIGP** (bool *e*, bool *g*, Text *p*) Macro
void **DIHP** (bool *e*, Macro *h*, Text *p*) Macro
void **DIGHP** (bool *e*, bool *g*, Macro *h*, Text *p*) Macro
void **DN** (bool *e*) Macro
void **DNG** (bool *e*, bool *g*) Macro
void **DNH** (bool *e*, Macro *h*) Macro
void **DNP** (bool *e*, Text *p*) Macro
void **DNGH** (bool *e*, bool *g*, Macro *h*) Macro
void **DNGP** (bool *e*, bool *g*, Text *p*) Macro
void **DNHP** (bool *e*, Macro *h*, Text *p*) Macro
void **DNGHP** (bool *e*, bool *g*, Macro *h*, Text *p*) Macro

All of these macros are used to individually override the default values defined above. Normally these macros would be used in a system wide header file to define macros appropriate for the application.

void **DS** (*e*) Macro
void **DSG** (*e*, *g*) Macro

These macros are used to assign values to convenience variables in the debugger. Convenience variables are dynamically typed, global in scope and initialised to 0. They start with a single `$` and can be used be used for saving the state of program or for counting events. The '`DS`' macro executes *e* under the same rules as `DI`. The '`DSG`' macro executes *e* only if the the expression *g* is true.

Note that '`DS`' and '`DSG`' can also be used for modifying C variables and calling functions.

4.5 L.h: support for printf style logging

These routines are used to provide logging functions. Messages can be divided into classes and separately enabled and disabled.

void **L** (*args...*) Macro

Used to log a message in a similar way to printf.

Defaults to a using `fprintf` on `stderr`.

void **LG** (bool *guard*, *args...*) Macro
void **LH** (function *handler*, *args...*) Macro
void **LP** (text *param*, *args...*) Macro
void **LGP** (bool *guard*, text *param*, *args...*) Macro
void **LHP** (function *handler*, text *param*, *args...*) Macro
void **LGHP** (bool *guard*, function *handler*, text *param*, *args...*) Macro

And all of the special functions.

The macros such as 'L' depend on the GNU CC variable number of arguments to macros extension. If you wish to compile your code on other systems you might wish to use the following variations on 'L', etc.

void **VL** ((*args*...)) Macro
void **VLG** ((bool *guard*, *args*...)) Macro
void **VLH** ((function *handler*, *args*...)) Macro
void **VLP** ((text *param*, *args*...)) Macro
void **VLGP** ((bool *guard*, *text param*, *args*...)) Macro
void **VLHP** ((function *handler*, text *param*, *args*...)) Macro
void **VLGHP** ((bool *guard*, function *handler*, text *param*, *args*...)) Macro

Each of these macros calls the corresponding function from the previous group, i.e. by default 'VLG' is the same as a call to 'LG'. If you define 'WITHOUT_NANA' all these macros are translated to '/* empty */'.

Thus you can have nana under GCC whilst the code is still portable to other compilers. However debugging information will not be available on other platforms.

Note: the argument list is surrounded by **two** sets of brackets. For example:

```
VL(("haze in darwin = %d\n", 3.4));
```

void **L_LEVEL** Macro

Used to enable and disable logging independently of guard expressions.

2 Always print message

1 Print message only if the guard expression is true.

0 Never print any messages.

Defaults to 1.

text **L_DEFAULT_HANDLER** Macro

The default handler for printing which is simply the name of the logging function or macro.

Defaults to fprintf

bool **L_DEFAULT_GUARD** Macro

The default guard condition for logging.

Defaults to TRUE.

text **L_DEFAULT_PARAMS** Macro

The default parameter passed off to the logging function or macro.

Defaults to stderr

void **L_SHOW_TIME** Macro

If defined then display the time in front of each message.

char* **L_SHOW_TIME_FORMAT** Macro

A format string for the time stamp in the log. By default it prints the time out in seconds.

value **L_SHOW_TIME_NOW** Macro

The name of a function that returns the time for the time stamp. This defaults to the ‘now’ function from ‘now.h’.

4.6 L_buffer.h: a circular buffer for logging.

A traditional embedded systems trick is to log messages to a circular buffer in core. This has the following benefits:

1. Speed – writing to a in core buffer is much faster than spitting out messages to a file on disk. It is often fast enough to leave at least most of the messages in the final product.
2. Field debugging – what the ... was the user doing before the system crashed. Oh lets ask them, I'm sure they'll give us a good problem report.

struct **L_BUFFER** Type

Used to define buffer variables, it is similar to ‘FILE*’ type in ‘stdio.h’. To create an instance use ‘L_buffer_create’.

L_BUFFER* **L_buffer_create** (size_t *size*) Function
L_BUFFER* **L_buffer_delete** (L_BUFFER **b*) Function

These are used to create or delete a buffer which can contain *size* characters.

```
L_BUFFER *lbuffer;

lbuffer = L_buffer_create(32*1024); /* create a 32K buffer */
...;
L_buffer_delete(lbuffer); /* and delete it after use */
```

void **L_buffer_wraparound** (L_BUFFER **b*, int *w*) Function

A buffer created by ‘L_buffer_create’ is set up so that the new messages will overwrite the older messages in the buffer. If you wish to disable this overwriting, e.g. to keep the first 32K bytes of your system startup messages you should use ‘L_buffer_wraparound’. For example:

```
L_BUFFER *lb = L_buffer_create(32*1024);
L_buffer_wraparound(lb, 0); /* disable wraparound */
```

void **L_buffer_printf** (L_BUFFER **b*, const char **fmt*, ...) Function
void **L_buffer_puts** (L_BUFFER **b*, const char **str*) Function
void **L_buffer_putchar** (L_BUFFER **b*, char *ch*) Function

These are the routines which do that actual printing to the buffer.

```
L_buffer_printf(lbuffer, "U: user input %c\n", c);
L_buffer_puts(lbuffer, "warning: its too hot");
L_buffer_putchar(lbuffer, '*');
```

Note: a null pointer passed to the '`L_buffer_puts`' function prints as '`(null)`'.[4]

void **L_buffer_clear** (L_BUFFER **b*) Function

Clear the log, i.e. remove all messages and start again.

void **L_buffer_dump** (L_BUFFER **b*, FILE **fp*) Function

Dump the contents of the log **b* to the file descriptor **fp*. Typically **fp* would be '`stderr`'.

Note that this does not change the contents of the buffer. This is important since we may have a hardware or software problem part of the way through the dump operation and you don't want to loose anything.

To reset the buffer after a successful dump use '`L_buffer_clear`'.

The output of '`L_buffer_dump`' consists of a starting message followed by the contents of the log. If a character in the log is not printable we print it out in hex on a line by itself.

```
* L_buffer_dump =
log message
and another
* non-printable character 0x1
more log messages
* end of dump
```

You also need to be able to integrate these functions into your design. See '`examples/ott.c`' for a complicated example. Here we will provide a simplified version which implements a new logging macro called '`LFAST`' which does a '`printf`' to the '`log_buffer`'. If you want to have all messages going to a '`L_BUFFER`' then you can redefine '`L_DEFAULT_HANDLER`'.

```
/* project.h - the project wide include file */

#include <nana.h>
#include <L_buffer.h>

/* LFAST(char *, ...) - log a message to the log_buffer */
/*     ##f translates as the rest of the arguments to LFAST */
```

[4] This was suggested by Phil Blecker.

```
#define LFAST(f...) LHP(L_buffer_printf,log_buffer,##f)

extern L_BUFFER *log_buffer; /* the log buffer */
```

The main program merely creates the *log_buffer* and eventually calls '`L_buffer_dump`' to print out the buffer when the system dies.

```
/* main.c - initialise the system and start things */

#include <project.h>

L_BUFFER *log_buffer;

main() {
  log_buffer = L_buffer_create(16000);
  if(log_buffer == NULL) { /* not enough store */
    ...
  }
  LFAST("system starting at %f\n", now());
  ...;
}

void fatal_error() { /* called on fatal errors */
  FILE *f = fopen("project.errors","w");
  L_buffer_dump(b, stderr); /* print log to stderr */
  L_buffer_dump(b, f); /* print log to file */
}
```

4.7 L_times.h: recording events and times.

This component is used to record events and times with a lower time and space overhead than the '`L_buffer.h`' component. Instead of using a '`printf`' style string we simply record the time and a pointer to a string identifying the event in a circular buffer.

Note 0: the string arguments should not be modified after a call since we record pointers to the strings rather than the strings themselves.

Note 1: there is no *printf* style formatting, e.g. '`%d`' in this package.

`struct` **L_TIMES** — Type

Used to define buffers, it is similar to '`FILE*`' type in '`stdio.h`'. To create an instance use '`L_times_create`'.

`L_TIMES*` **L_times_create** (`int` *size*) — Function
`L_TIMES*` **L_times_delete** (`L_BUFFER` **b*) — Function

These are used to create or delete a buffer which can contain *size* messages.

`void` **L_times_wraparound** (`L_TIMES` **b*, `int` *w*) Function

A buffer created by '`L_times_create`' is set up so that the new messages will overwrite the oldest messages in the buffer. If you wish to disable this overwriting, e.g. to keep the first few messages messages you could use '`L_times_wraparound(b,0)`'.

`void` **L_times_add** (`L_BUFFER` **b*, `char` **m*, `NANA_TIME` *t*) Function

Add an event identified by message *m* at time *t* to *b*. The type *NANA_TIME* defaults to '`double`'.

`void` **L_times_dump** (`L_TIMES` **b*, `FILE` **fd*) Function

Dump the contents of the buffer out.

`void` **L_times_clear** (`L_TIMES` **b*) Function

Clear all the messages from *b*.

4.8 DL.h: support for printf style logging

These routines are used to provide logging functions. Messages can be divided into classes and separately enabled and disabled.

`void` **DL** (*args...*) Macro

Used to log a message.

Defaults to a using `fprintf` on `stderr`.

`void` **DLG** (`bool` *guard*, *args...*) Macro
`void` **DLH** (`function` *handler*, *args...*) Macro
`void` **DLP** (`text` *param*, *args...*) Macro
`void` **DLGP** (`bool` *guard*, `text` *param*, *args...*) Macro
`void` **DLHP** (`function` *handler*, *args...*) Macro
`void` **DLGHP** (`bool` *guard*, `function` *handler*, *args...*) Macro

And all of the special functions.

The macros such as '`DL`' depend on the GNU CC variable number of arguments to macros extension. If you wish to compile your code on other systems you might wish to use the following variations on '`DL`', etc.

`void` **VDL** ((*args...*)) Macro
`void` **VDLG** ((`bool` *guard*, *args...*)) Macro
`void` **VDLH** ((`function` *handler*, *args...*)) Macro
`void` **VDLP** ((`text` *param*, *args...*)) Macro
`void` **VDLGP** ((`bool` *guard*, *text param*, *args...*)) Macro
`void` **VDLHP** ((`function` *handler*, *args...*)) Macro
`void` **VDLGHP** ((`bool` *guard*, `function` *handler*, *args...*)) Macro

Each of these macros calls the corresponding function from the previous group, i.e. by default '`VDL`' is equivelant to a call to '`DL`'. If '`WITHOUT_NANA`' is defined then the call too '`VDL`' is equivelant to '`/* empty */`'.

Thus you can have debugging under GCC whilst the code is still portable to other compilers. However debugging information will not be available on other platforms.

Note: the argument list is surrounded by **two** sets of brackets. For example:

```
VDL(("haze in darwin = %d\n", 3.4));
```

`int` **DL_LEVEL** Macro

Used to enable and disable logging independently of guard expressions.

`2` Always print message

`1` Print message only if the guard expression is true.

`0` Never print any messages.

Defaults to `1`.

`text` **DL_DEFAULT_HANDLER** Macro

The default handler for printing which is simply the name of the printing function.

Defaults to `fprintf`

`bool` **DL_DEFAULT_GUARD** Macro

Defaults to `TRUE`.

`text` **DL_DEFAULT_PARAMS** Macro

Defaults to `stderr`

`flag` **DL_SHOW_TIME** Macro

Each message can be given an individual time stamp by defining `DL_SHOW_TIME`. This causes the `_L_gettime` routine to be called before each message which generates the timestamp. A default version is provided by the nana library.

4.9 GDB.h: sending plain gdb commands to the debugger

'`GDB.h`' provides macros for generating user specified commands in the output of the '`nana`' command. They are not included by default in the '`nana.h`' file.

`void` **GDB** (*command*) Macro

Emit a single line command when running this file through '`nana`'. Note that each line must be passed off separately to the '`GDB`' macro.

This could be used to set debugger options or to define procedures inside '`gdb`', e.g.

```
GDB(define checkstack);
```

```
GDB(  if 0 <= n && n <= 10);
GDB(    print "stack ok");
GDB(  else);
GDB(    print "stack corrupted");
GDB(  end);
GDB(end);
```

void **GDBCALL** (*command*) Macro

Causes a single gdb *command* to be executed whenever control passes through this line of code. After the users command is executed control automatically returns to the program.

```
GDBCALL(set memory_check = 1)
```

These macros could used for instrumenting code or setting up test harnesses, e.g.

```
GDB(set $siocall = 0);
GDB(set $sioerr = 0);

void sio_driver() {
  GDBCALL(set $siocall++)
  if(SIO_REQ & 0x010) {
    GDBCALL(set $sioerr++);
    ...
  }
}
```

4.10 Q.h: support for quantifiers

'`Q.h`' provides support for the quantifiers of predicate logic. For example to check that all elements in a data structure have some property we would use universal (forall, upside down A) quantification. To check that one or more values in a data structure have some property we would use existential (exists, back the front E) quantification. For example:

```
/* all values in a[] must be between 0 and 10 */
I(A(int i = 0, i < n_array, i++, 0 <= a[i] && a[i] <= 10));

/* there exists a value in linked list l which is smaller than 10 */
I(E(node *p = l, p != NULL, p = p->next, p->data <= 10));
```

The first three arguments to '`A`' and '`E`' are similar to a C '`for`' loop which iterates over the values we wish to check. The final argument is the expression that must be true.

The only minor difference from the C '`for`' loop is that variables may be declared at the start of the loop, even if you are using C rather than C++ which already supports this.[5]

[5] ANSI C does not allow variable declarations at the beginning of loops unlike C++. The '`Q.h`' macros get around this by starting each loop with a new scope.

The 'Q.h' macros can also be nested and used anywhere a boolean value is required. For example:

```
if(A(int i = 0, i < MAXX, i++,
     A(int j = 0, j < MAXY, j++,
       m[i][j] == (i == j ? 1 : 0)))) {
      /* identity matrix, i.e. all 0's except for 1's on */
      /* the diagonal */
      ...
} else {
      /* not an identity matrix */
      ...
}
```

The results from these macros can also be combined using boolean operations, e.g.

```
/* the values in a[i]  are either ALL positive or ALL negative */
I(A(int i = 0, i < MAX, i++, a[i] >= 0)
  ||
  A(int i = 0, i < MAX, i++, a[i] < 0));
```

Portability: note the macros in this file require the GNU CC/C++ statement expression extension of GCC to work. If your not using GNU CC then for now you are out of luck. At some time in the future we may implement a method which will work for standard C++, standard C is a bit of a challenge.

Portability: unfortunately these macros do not for the 'DI' and 'DL' macros since the statement expression extension has not been implemented in GDB.

`bool` **A** (*init*, *condition*, *next*, *exprn*) Macro

For all values generated by '`for(`*int*;*condition*;*next*`)`' the *exprn* must be true.

```
I(A(int i = 0, i < MAX, i++, a[i] >= 0)); /* all a[i] are +ve */
```

`bool` **E** (*init*, *condition*, *next*, *exprn*) Macro

There exists at least one value for *exprn* generated by '`for (`*int*;*condition*;*next*`)`' which is true.

```
/* one or more a[i] >= 0 */
I(E(int i = 0, i < MAX, i++, a[i] >= 0));
```

`long` **C** (*init*, *condition*, *next*, *exprn*) Macro

Returns the number of times the *exprn* is true over the values generated by '`for(`*int*;*condition*;*next*`)`'.

```
/* 3 elements of a[] are +ve */
I(C(int i = 0, i < MAX, i++, a[i] >= 0) == 3);
```

bool **E1** (*init*, *condition*, *next*, *exprn*) Macro

There exists only one value generated by '`for(`*int*`;`*condition*`;`*next*`)`' for which the *exprn* is true.

```
/* a single elements of a[] is +ve */
I(E1(int i = 0, i < MAX, i++, a[i] >= 0));
```

`typeof(`*exprn*`)` **S** (*init*, *condition*, *next*, *exprn*) Macro

Sum the values generated by *exprn* for all values given by '`for(`*int*`;`*condition*`;`*next*`)`'. The type of the value returned is given by the type of the *exprn*.[6]

```
/* sum of a[] is 10 */
I(S(int i = 0, i < MAX, i++, a[i]) == 10);

/* sum of all +ve numbers in a[] is 10 */
I(S(int i = 0, i < MAX, i++, a[i] >= 0 ? a[i] : 0) == 10);
```

`typeof(`*exprn*`)` **P** (*init*, *condition*, *next*, *exprn*) Macro

Returns the product of the values generated by *exprn* for all values given by '`for(`*int*`;`*condition*`;`*next*`)`'. The type returned is the type of the expression.

```
/* product of all the values in a[] is 10 */
I(P(int i = 0, i < MAX, i++, a[i]) == 10);

/* a = x^y i.e. x*x..*x y times */
I(P(int i = 0, i < y, i++, x) == a);
```

4.11 Qstl.h: quantifiers for STL containers.

The Standard Template Library (STL) is a library for C++ that makes extensive use of templates to implement the standard container classes and much more. Each of the container classes provides an interface to iterate over all the objects in the container, e.g.

```
// MAP is an associate array from location(lat,long) onto the name.
typedef map<location,string,locationlt> MAP;

void print_map_names(MAP& m) { // print out all the names in the map
  for(MAP::iterator i = m.begin(); i != m.end(); ++i) {
    cout << (*i).second << "\n";
  }
}
```

'`Qstl.h`' provides the same facilities as '`Q.h`' but uses the standard STL iterator protocol shown above. The names in '`Qstl.h`' are generated by appending a '`O`' (O not zero!) to the names in '`Q.h`'. In particular:

[6] This uses yet another GNU CC extension, however since we are already using statement expressions we might as well use '`typeof`' as well.

`bool` **AO** (*name*,*container*,*predicate*) Macro

For all values in the *container* class the *predicate* must be true. The *predicate* refers to individual values using *name*. See the STL documentation for more details. Another way of putting this is forall *name* in *container* the *predicate* must be true.

```
map<int,char *,ltint> m;
// all keys (or indexes) into m are positive
I(AO(i, m, (*i).first >= 0));
```

`bool` **EO** (*name*,*container*,*predicate*) Macro

There exists one or more values in the *container* class for which the *predicate* is true.

```
map<int,char,ltint> m;

// one or more characters in m are '$'
I(EO(i, m, (*i).second == '$'));
```

`bool` **E1O** (*name*,*container*,*predicate*) Macro

There exists one value in the *container* for which the *predicate* is true.

```
map<int,char,ltint> m;

// one characters in m is a '$'
I(E1O(i, m, (*i).second == '$'));
```

`int` **CO** (*name*,*container*,*predicate*) Macro

Returns the number of times the *predicate* was true for all values in the *container*.

```
map<int,char,ltint> m;
int nalpha;
// count the number of alphabetic chars in the map
nalpha = CO(i, m, isalpha((*i).second));
```

`typeof(`*exprn*`)` **SO** (*name*,*container*,*exprn*) Macro

Sum the *exprn* for all values in the *container*.

```
map<int,float,ltint> m;
float sum;
// sum all the values in m
sum = SO(i, m, (*i).second);
```

`typeof(`*exprn*`)` **PO** (*name*,*container*,*exprn*) Macro

Take the product of the *exprn* for all values in the *container*.

```
map<int,float,ltint> m;
float product;
// multiply all the values in m
product = PO(i, m, (*i).second);
```

4.12 now.h: measuring time

The 'now.h' file provides some simple time measurement routines. It is not included in 'nana.h' so you must include this file separately.

It uses the 'gettimeofday' system call and has an accuracy of between 1us and 10ms depending on the operating system and hardware configuration.

See the IPM package if you require better measurement tools.[7]

double **now** () Function

Returns the time in seconds since the beginning of time as defined by your system. If you call 'now_reset' the time will start again at 0.

double **now_reset** () Function

Reset the times returned by 'now' to 0.

double **now_delta** (double **dp*) Function

Returns the elapsed time between **dp* and *now()*. It then sets **dp* to *now*, thus giving a delta time between particular events.

```
t = now();
for(;;) {
  ...; /* code that must finish in 50ms */
  I(now_delta(&t) <= 0.050);
}
```

4.13 eiffel.h: eiffel type assertions

Eiffel is a very nice language which provides the assertion checking facilities of nana inside the language itself. The 'eiffel.h' library is intended to provide a similar setup to Eiffel in the C++ language. It is a pretty poor emulation, but it is hopefully better than nothing.

Assertion checking is controlled by the *EIFFEL_CHECK* variable which can take on any of the following values:

CHECK_NO Disable all checking.

CHECK_REQUIRE
: Check the preconditions for each method.

CHECK_ENSURE
: And also check the postconditions.

CHECK_INVARIANT
: And also check the class invariant before and after each method is called. The programmer should provide a class method called 'invariant' which returns 'true' if the object is consistent, 'false' otherwise.

---
[7] In the fullness of time, we may integrate these routines in here.

`CHECK_LOOP`
: And also check the loop invariants.

`CHECK_ALL`
: And also check any assertions using the '`CHECK`' instruction.

A typical compile flag to the compile might be:

```
% g++ -c -DEIFFEL_CHECK=CHECK_ALL play.cc
```

And then we have the actual assertion macros.

`void` **REQUIRE** (*exprn*) Macro

Called at the beginning of each method. This checks the precondition to a method and the class invariant.

`void` **ENSURE** (*exprn*) Macro

Called at the end of each method. This checks the postcondition to a method and the class invariant.

`void` **INVARIANT** (*exprn*) Macro

Used to check a loop invariant.

`void` **CHECK** (*exprn*) Macro

Used for any other inline assertions.

And finally a small example:

```
#include <eiffel.h>

class example {
  int nobjects;
  map<location,string,locationlt> layer;
public:
  bool invariant();
  void changeit(location l);
};

bool example::invariant() {
  return AO(i,layer,valid_location((*i).first)) &&
         nobjects >= 0;
}

void example::changeit(string n, location l) {
  REQUIRE(E1O(i,layer,(*i).second == n));
  ...;
  while(..) {
    INVARIANT(...);
```

```
    ...
    INVARIANT(...);
  }
  ...
  CHECK(x == 5);
  ...
  ENSURE(layer[l] == n);
}
```

Note that the invariant checking macro '`example::invariant`' is called automatically on function entry/exit using the '`REQUIRE`' and '`ENSURE`' macros.

4.14 assert.h: a drop in replacement for assert.h

A drop in replacement for '`assert.h`' is provided in the '`src`' directory. It is **not** installed by default. If you wish to use it then you need to copy it to your include directory by hand.

This might be of use if you are already using '`assert.h`' and wish to save some code space since the nana implementation is more space efficient.

Calls to '`assert`' are translated to calls to '`I`' and can be disabled by defining '`NDEBUG`'.

4.15 calls.h: checking/printing many objects/facts.

The '`calls`' module implements a simple list of functions which can be modified and executed at run-time. It is similar in spirit to the ANSI C '`atexit`' function. It is intended to be used for:

- Checking the consistency of the components in your system.

  For example each module could register a self checking function which uses the rest of the nana library. All of these functions would then be called using '`calls.h`' to check that the entire system is consistent.
- Printing out the state of your program in a readable format.

typedef FUNC Type

A pointer to a '`void`' function which takes a single '`void*`' argument. The '`void *`' argument is intended to be used to pass information such as arguments or pointers to objects (e.g. '`this`' in C++). All of the checking/printing functions must be of this type, e.g.

```
void print_object(void *f) {
  ...;
}
```

struct CALL Type

This structure represents a single call to a function, i.e. a function pointer ('`FUNC`') and the '`void*`' argument.

```
CALL *head = 0;
```

void **calls_add** (CALL **head, FUNC fp, *arg) Function

Adds a call to function '`fp`' with argument '`arg`' to the list pointed to by '`head`'.

```
CALL *global_checks = 0;

calls_add(&global_checks,complex_ok,(void *)x);
```

void **calls_exec** (CALL **head, FUNC fp, void *arg) Function

Execute all/some of the calls in the list given by '`head`'. The arguments '`fp`' and '`arg`' must both match for each individual call. The null pointer ('`0`') matches anything whilst any other value requires an exact match between the '`CALL`' and the arguments to '`calls_exec`'. For example:

```
calls_exec(&l,0,0); /* execute all functions in l  */
calls_exec(&l,complex_print,0); /* calls complex_print(*) in l */
calls_exec(&l,0,(void*) &b); /* calls *(&b) in l */
calls_exec(&l,f,(void*) &b); /* calls f(&b) in l */
```

void **calls_delete** (CALL **head, FUNC fp, void *arg) Function

Delete all/some of the calls in the list given by '`head`'. The arguments '`fp`' and '`arg`' must both match for each individual call. The null pointer ('`0`') matches anything whilst any other value requires an exact match between the '`CALL`' and the arguments to '`calls_delete`'. For example:

```
calls_delete(&l,0,0); /* delete all functions in l  */
calls_delete(&l,complex_print,0); /* delete complex_print(*) in l */
calls_delete(&l,0,(void*) &b); /* delete *(&b) in l */
calls_delete(&l,f,(void*) &b); /* delete f(&b) in l */
```

Note: that calls are added to the head of the list rather than the tail. This means that the most recently added call will be executed first (as in a stack).

5 Using Nana

This chapter is intended to provide some hopefully useful examples of Nana. If any of the users of this library would be so kind as to contribute a section on their usage I would be obliged.

This section is under development

5.1 Simplest example

As a nice simple, indeed trivial example consider the following:

1. Include '`nana.h`' in your project wide include file.
2. Use '`I`' to check invariants in your code. In particular all functions or methods should be required to check their pre and post conditions.
3. Use '`DL`' to print debugging messages in your code. This means that debugging messages only occur when you run the program under the debugger.[1]

5.2 Syslog

'`syslog`' is a comprehensive logging system written by Eric Allman which is available on most UNIX systems. It provides facilities for sorting messages by source or severity and mechanisms for sending messages off to files, terminals or other machines. See chapter 12 of the "UNIX System Administration Handbook (2nd Edition)" by Nemeth, Snyder, Seebass and Hein for an excellent tutorial on '`syslog`'.

The rules used by '`syslog`' are normally defined in '`/etc/syslog.conf`'. Each line specifies the destination for messages with particular sources and priorities. For example:

```
# log mail messages at priority info to mailinfo
mail.info /var/log/mailinfo
# log all ftp information to syslog on a remote machine
ftp.* @remote-machine.cs.ntu.edu.au
# all critical errors to console
*.crit   /dev/console
```

To use '`syslog`' we merely redefine the default handlers and parameter values for '`nana`'. For example:

```
#include <syslog.h>
#define L_DEFAULT_HANDLER syslog /* replaces fprintf(3) by syslog(3) */
#define L_DEFAULT_PARAMS LOG_USER /* default priority for syslog info */
#include <nana.h>
```

[1] Of course you also need to use the gdb commands generated by the '`nana`' command, perhaps using '`nana-clg`'

```
int main() {
     openlog("nana", /* identifier for these log messages */
        LOG_PID, /* also log the process id */
        LOG_DAEMON /* facility */
     );

     L("temperature is falling to %d", 35); /* logged at LOG_USER priority */
     LP(LOG_CRIT, "the snow is falling in Darwin"); /* LOG_CRIT message */

     closelog();
     return 0;
}
```

This program results in the following addition to '/var/adm/messages':

```
Jun 12 16:04:46 chingiz nana[2671]: the temperature is falling to 35
Jun 12 16:04:46 chingiz nana[2671]: the snow is falling in Darwin
```

In addition the 'LOG_CRIT' message about snow falling in Darwin is sent to the console for immediate action.

Note: 'syslog' normally uses 'UDP' for its network communications and that 'UDP' does not guarantee delivery.

5.3 GNU Programs: how to avoid nana sometimes

Imagine you[2] are building a GNU program. Ideally it should run on systems without any other software including GCC, GDB and even Nana.

To achieve this noble goal you can provide your configure script with a '--without-nana'[3] flag which then '#define's' 'WITHOUT_NANA'. You should also use the 'VL' macro rather than 'L' macro since 'L' takes a variable number of arguments and will break non GNU C preprocessors.

```
int half(int n) {
  /* Don't use L(...) since it takes a variable number of args! */
  VL(("hello world = %d\n", 10)); /* Note the doubled (( */
  ...;
}
```

5.4 Embedded Systems: testing non-deterministic systems.

One of the uses (in fact the original use) for nana is in the testing of non-deterministic systems. These systems behave in a different way each time they are run because of timing or measurement noise. Thus you can't just 'diff' the results against a known good run since they run differently each time.

[2] Gordon Matzigkeit contributed some of the ideas presented here and raised this problem.
[3] Or add a '--with-nana' flag to configure that does the opposite.

Instead you can use a series of programs which execute over a the results of the program and check that the behaviour meets the specification automatically.

5.5 Realtime Systems

Assertions for the timing behaviour of you system can be done in nana nicely. You might also consider using an instruction level simulator such as PSIM and use the '`cycles`' variable to test realtime constraints.

5.6 A database

Ah databases, business boffins using assertions. It would nice to get a real example for this one!

5.7 Program Visualisation: pretty pictures

One nice way to use '`L`', '`DL`', etc is to punt off the messages off to a program which displays the information. You would use the '`L_DEFAULT_PARAM`' argument to send commands off to a pipe with gnuplot or TCL/TK interpreter running at the end.

For a small example of this, see '`tcl/status.c`'.

6 FAQ

This chapter is intended to answer some general questions about the usage of Nana. Most of the material (perhaps) has been presented elsewhere, my apologies for repeating myself.

1. Can I use GNU Nana in a commercial product?

   See the license details in the file 'COPYING'. In summary GNU Nana is Free software which has been released under a license that allows commercial use provided the copyright is maintained. It is not under the GPL.
2. How do you completely disable assertion checking?

   Set the 'I_LEVEL' command to '0' using '-D' or '#define' before including 'nana.h'.
3. How do I find out about future releases?

   Subscribe to the nana mailing list by using the 'make subscribe' target in the nana root directory.
4. Where do I send bug reports, suggestions, etc?

   'nana-bug@it.ntu.edu.au' – bug archive

   'nana@it.ntu.edu.au' – mailing list

   'pjm@cs.ntu.edu.au' – the author

7 Future work

As usual, there is more work to be done than time to do it in. Workers or test pilots are invited to apply for the following tasks which may get down in the fullness of time minister. I'm particularly interested in which projects people think are worthwhile (or rubbish).

1. Ada support - pre 1.00 versions of nana provided support for Ada. Adding a full version of nana into the GNATS compiler would probably be useful, particularly the real time part.
2. FORTRAN support.
3. Message browsing (`emacs/fess.el`) - a very prototypical mode for browsing message logs is distributed with nana. Features include hiding/showing lines by regular expression or predicate. If someone who knows what they are doing rewrites this that would be a good thing. Or perhaps a modified version of `less` would be useful.
4. Program Visualisation (`tcl/status.c`) - another prototype which uses a small TCL/TK library and nana logging to generate some nice pictures showing the behaviour of the program. For example the history of variables over time can be recorded, graphs drawn and long message logs kept. A new version of this has been created and may be released shortly.
5. Eiffel shortform program - the shortform of an Eiffel program contains the function signatures (arguments, etc) together with the assertions (pre/post conditions). An extension could perhaps be put in a program such as `DOC++` or `cxref` which extracts the nana assertions.
6. Automated Testing - combine nana logs with a workbench that lets you verify properties of the logs using programs `gawk` or the `PRECC` (LL infinity version of YACC). This technique has been used on industrial strength products quite successfully.
7. GNU standards - perhaps it would be worthwhile to set up a prototype for checking and logging in GNU tools.
8. Extend GDB(1) so that we can support `Q.h` style quantifiers in `DI.h` expressions. Basically we need to add a restricted form of the statement value expression to GDB.
9. Support for other (particularly ANSI only) compilers.

Appendix A Index

L

N

P

Q

R

S

T

V

Short Contents

Table of Contents