libnix.info

COLLABORATORS			
	TITLE : libnix.info		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 7, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 libnix.info **1**

1.1 libnix.info . . . 1
1.2 libnix.info/Description . . . 1
1.3 libnix.info/Authors . . . 2
1.4 libnix.info/Disclaimer . . . 2
1.5 libnix.info/Naming . . . 2
1.6 libnix.info/Usage . . . 3
1.7 libnix.info/Features . . . 4
1.8 libnix.info/Startup codes . . . 4
1.9 libnix.info/Startup interface . . . 6
1.10 libnix.info/Startup usage . . . 7
1.11 libnix.info/Commandline parser . . . 7
1.12 libnix.info/libstack.a . . . 8
1.13 libnix.info/swapstack . . . 9
1.14 libnix.info/stackextend implementation . . . 9
1.15 libnix.info/stackextend usage . . . 10
1.16 libnix.info/Advanced . . . 10
1.17 libnix.info/Costs . . . 11
1.18 libnix.info/libnix.a . . . 11
1.19 libnix.info/Locale . . . 12
1.20 libnix.info/Formatted I-O . . . 13
1.21 libnix.info/atof strtod . . . 13
1.22 libnix.info/Memory management . . . 13
1.23 libnix.info/Standard I-O . . . 14
1.24 libnix.info/Signal handling . . . 14
1.25 libnix.info/setjmp longjmp . . . 15
1.26 libnix.info/ctype . . . 16
1.27 libnix.info/clock . . . 16
1.28 libnix.info/Multibyte character functions . . . 16
1.29 libnix.info/libstubs.a . . . 16

1.30 libnix.info/Auto-library-opening usage . . . 17
1.31 libnix.info/Auto-library-opening interface . . . 17
1.32 libnix.info/detach.o . . . 19
1.33 libnix.info/Special startups . . . 19
1.34 libnix.info/Calling convention . . . 20
1.35 libnix.info/libinit.o . . . 20
1.36 libnix.info/libinitr.o . . . 21
1.37 libnix.info/devinit.o . . . 21
1.38 libnix.info/Set elements . . . 21
1.39 libnix.info/geta4 . . . 22
1.40 libnix.info/Data models . . . 23
1.41 libnix.info/Code models . . . 25
1.42 libnix.info/Library bases . . . 25
1.43 libnix.info/libamiga.a . . . 26
1.44 libnix.info/Code size . . . 27
1.45 libnix.info/FAQs . . . 28

Chapter 1

libnix.info

1.1 libnix.info

```
   This is the documentation of libnix. The following is a list of
chapters. You need not read all of them - but reading the chapter
Features is recommended.


  Description        What is libnix?
  Authors        Who did it?
  Disclaimer         Copyright and other legal stuff.
  Naming         Naming conventions.
  Usage        How to use it ;-)
  Features         Details of the implementation (and more).
  Special startups   How to write shared libraries and devices with gcc.
  Set elements       A nice feature of gnu ld.
  geta4        Some words on code and data models
  Library bases      And how they work.
  Code size        How to write small programs with gcc
  FAQs         Frequently asked questions and answers.
```

1.2 libnix.info/Description

```
What is libnix?
***************

   libnix is a static (i.e. link) library for usage on amiga computers
together with gcc 2.3.3 or above. It is very amigalike and contains a
lot of features you probably don't want to miss:

   * auto-open-library feature

   * SAS compatible handling of WB startup message

   * auto-detach startup code

   * is very short
```

* does not require a shared library

* auto stack-extend

* and much more

So if you want to write amiga specific programs or if you only need ANSI support instead of unix compatibility - and if you don't want to redistribute ixemul.library - this can be your choice.

But be aware - libnix requires Amiga OS 2.0 or higher :(.

1.3 libnix.info/Authors

Authors
****

If you want to change anything in the library please contact one of us first - we know how to do it best since we wrote it. If you want to report bugs please contact us, too. If we don't know about them we cannot fix them.

Matthias Fleischer
fleischr@izfm.uni-stuttgart.de

Gunther Nikl
gnikl@informatik.uni-rostock.de

Special thanks go to Gerhard Müller for writing gerlib, Philippe Brand for finding a name for this beast and Kriton Kyrimis who contributed some of the non-ANSI functions and a lot of ideas.

1.4 libnix.info/Disclaimer

Disclaimer
****

1.5 libnix.info/Naming

Naming conventions
****

This library is not only for the end user but also for the library programmer (if you want to write your own startup, etc).

If you want to write code for it you should be aware of the normal naming conventions for ANSI libraries:

* Names with no underscore 'foo' are ANSI or POSIX compliant - there is absolutely no risk in using them. If you use only these you can write portable programs.

* Names with a single underscore '_foo' are ANSI extensions for the end user. Usually they are very common on certain systems but not used on others.

* Names with two underscores '__foo' are for the library programmers only. If they are not documented you cannot rely on them. And even if they are you should use them only for writing library code.

There is only one exception of these conventions ('__chkabort()') and this is for compatibility reasons.

1.6 libnix.info/Usage

Usage
*****

The usage of this library is like any other link library. The only important thing is the right linkage order:

1. The startup code has to be used first :-)

2. The stubs-library has to be used last since it contains the library base pointers.

3. The commandline parser should be used after your code but before most other things or you will run into problems.

Normally this is handled by the specs file of gcc.

'gcc (-fbaserel) (-resident) -noixemul YOUR_OBJECTS (-nix_main) (-lm)'

If you use '-lnix_main' you get a different commandline parser. '-lm' uses the math library.

Be aware that the formatted I/O-functions need the math library to work correctly for floating point numbers. Without the math library you get only floating point support for simple operators like '+', '*', casts and that like.

If you don't use the assembler inline functions of gcc you will have to use 'libamiga.a' if you want to use any Amiga OS function. gcc comes with a free version of libamiga.a which is a subset of the original one. You can build it yourself if you unpack the sources, but be prepared that it may take some time.

For compiling a4-relative programs you should choose the '-fbaserel' option. You get resident (pure) programs if you set the '-resident' option. Anything else necessary for these options is handled by the specs-file (choosing the right startups and libraries).

If you (for some reason) don't trust you specs-file you can call everything by hand:

'gcc -nostdlib ncrt0.o YOUR_OBJECTS libnixmain.a (libm.a) libnix.a libstubs.a'

But that's not the recommended way. Therefore I don't explain this in detail here - use the '-v' option of gcc for more details.

1.7 libnix.info/Features

Features - what you get
****

The following list contains the elements of this package in the right linkage order. This means if you follow the list from top to bottom and take one file of each menu entry you will get a working configuration.

Startup codes	Does all work necessary for startup.
Your objects	Need I say anything about that?
Commandline parser	Calculates argc and argv.
libm.a	The math library (optional).
libstack.a	Stack extension code (optional).
libnix.a	The library itself.
libamiga.a	If you have one.
detach.o	Auto detaching
libstubs.a	The library bases.

1.8 libnix.info/Startup codes

Startup codes
****

There is a lot of work to do before your 'main' function can be called - open shared libraries, open stdin, stdout, etc. Depending on the compiler options and the ANSI functions you use. This work is done by the startup code.

Startup interface
Startup usage

To get a short startup all the necessary modules are optional - they get only linked in if you use them. There are 2 exceptions from this (since the linker cannot check for it):

* The commandline parser. It can be deactivated by declaring

  '__nocommandline'

  somewhere in your program (the type doesn't matter).

* The shared-library-opening module. You should not disable it unless you know what you are doing since most library functions depend on it.

The startup codes itself are written in assembly to be as short as possible.

Here is a little program to get the point:

```
#include <inline/exec.h>
#include <dos/dos.h>
#include <inline/dos.h>
#include <workbench/workbench.h>

int __nocommandline=1; /* Disable commandline parsing  */
int __initlibraries=0; /* Disable auto-library-opening */

struct DosLibrary *DOSBase=NULL;

extern struct WBStartup *_WBenchMsg;

int main(void)
{ if(_WBenchMsg==NULL)
  { if((DOSBase=(struct DosLibrary *)OpenLibrary("dos.library",37))!=NULL)
    { Write(Output(),"Hello world\n",12);
      CloseLibrary((struct Library *)DOSBase); } }
  return 0;
}
```

compiled and linked with

'gcc -noixemul -s -O2 -fbaserel helloworld.c'

gives an executable of 492 bytes. And this with the normal 'main' function!

So you never need to try to write a program without a startup code.

1.9 libnix.info/Startup interface

```
Startup code interface
**********************

   The startup codes do the following:

   * They catch the workbench startup message and place it into the
     variable

     'extern struct WBStartup *_WBenchMsg'

     you can simply look into this place (and test for a 'NULL'
     pointer) to check if your program was started from WB. If this is
     a 'NULL' pointer

     'extern char *__commandline'

     contains the ('\n' terminated) parameters of the commandline.

   * They call all functions in the

     'long __INIT_LIST__[];'

     with ascending priority.

   * They call the function

     'int main(int argc,char *argv[])'

     You can exit by simply falling through the end of 'main' or by
     calling

     '__volatile void exit(int returncode)'

     which does the cleanup:

   * It calls all functions in the

     'long __EXIT_LIST__[];'

     with descending priority.

   * It replys the WB startup message if necessary, resets the
     stackpointer and returns to the shell.

   '__INIT_LIST__' and '__EXIT_LIST__' are two set elements which are a
speciality of the gnu ld. Since everything that needs initialization
works over these two lists the bare startups are very short. In fact
they are even shorter then some low-level-startups

   You can easily add your own functions to the startup procedure by
using the macros in the file 'headers/stabs.h' - but keep in mind that
this is non-portable. Priority values <=0 are reserved for library
implementors.
```

1.10 libnix.info/Startup usage

```
Startup code usage
******************

   There are currently 3 startup codes in this package (maybe there
will be more in the future). Depending on the code and data model you
use and some other things you should choose one of them:

'ncrt0.o'
     This is the normal (i.e. large code, large data model) startup.
     It contains a 'geta4()' entry point to enable you to use one source
     for two code models. There is no other need for this function.

'nbcrt0.o'
     This one is for compiling small data model (a4 relative) programs.
     There is a 'geta4()' entry that places the right information into
     a4.  Use this startup code if you compiled with '-fbaserel'.

'nrcrt0.o'
     This startup code allocates a new data area every time you call it.
     Even if you don't call it at all the data are is there once.  This
     gives you multientrant and reentrant code.  Therefore this startup
     code is for compiling resident (pure) programs.  Resident programs
     are always small data model if you let the compiler do the work.

     There is no 'geta4()' entry - I just don't know how this could be
     done.  (If you start your code 10 times and want to access global
     data out of a hook you cannot tell which one of the 10 data areas
     to use because you want to access the data from a different task!)
```

1.11 libnix.info/Commandline parser

```
Commandline parser
******************

   There are currently 2 commandline parser modules in the libnix
package.  You can easily write your own by looking into the examples

'libnixmain.a'
     This is the normal one, i.e. it does all the work necessary for
     ANSI compatibility and gives you the normal 'main' calling
     convention. You can shut down the commandline parsing (if you want
     to use the amiga OS commandline parser) by declaring

     '__nocommandline'

     somewhere in your code (the type of it actually doesn't matter).
     This spares some bytes and is compatible to every other compiler.

     And you can declare your own WB shell window by declaring a

     'char __stdiowin[]'
```

variable somewhere in your code (but only if you parse the commandline - without the commandline parser you get no window at all!).

'libnix_main.a'
This is a special version of a commandline parser - it doesn't call the normal 'main' but

'int main(char *commandline)'

'commandline' is the complete commandline - including the quoted filename of your program (it's only quoted, not escaped - this is for compatibility reasons :-( ). You might think the name of the game should be '_main' and not 'main' - and you are completely right. You can use '_main' for 'main' and '_exit' for 'exit' - there are symbol redirections for these and the linker does the work.

This commandline parser is useful for compatibility. You can use it as a second example or for recompiling PD programs that use the single argument. You cannot use it for compiling ANSI code.

1.12 libnix.info/libstack.a

special stack handling facilities
****

The current Amiga OS (V3.1) has a very limited stack handling compared to most other OSs: Every process has it's own fixed sized stack - and that's all about it. The usual default for this stack is 4k, but that's not enough for more complicated purposes (like for example compilers). Setting a higher default is no real solution because it costs a lot of (widely unused) memory and may be overrun, too :-(.

But fortunately you can get stack extension with a little help of the compiler ;-). Starting with V0.9 of libnix and V2.7.0 of gcc you get a fully featured stack extension facility. The old stack swap method is still provided (not only for compatibility but also because it's simpler) but please don't try to mix it with the newer check/extend methods.

swapstack	Old method.
stackextend implementation	How stack extend works.
stackextend usage	Usage. *read*
Advanced	Fine tuning.
Costs	Some damned lies (Benchmarks ;-)).

1.13 libnix.info/swapstack

```
Minimum stack setting
*********************
```

   Most large tools need more stack than the default 4096 bytes. If your tool is one of them you can either rely on the user being able to raise the current stack or you can let libnix raise the stack for you. At startup this module checks if the current stack is large enough for your needs and switches to a new one if not. All you have to do is to provide a variable

   'unsigned long __stack={required stacksize};'

   somewhere in your code and to link with the appropriate swapstack.o module.

1.14 libnix.info/stackextend implementation

```
Implementation of stack checking and extension
**********************************************
```

   The basic principle of stack checking is that the compiler emits special code to check if the stack is large enough whenever there's need for a bigger chunk of stackspace, i.e. at function entry when local arrays are allocated, at the start of blocks with local variable sized arrays and when calling 'alloca()'. If the needed stackchunk is bigger than the left stackspace the program ends.

   Since this special code costs memory and CPU time smaller stackneeds (e.g. when calling library functions) are handled by not really checking against the hard border of the stackframe but against one that leaves a certain amount of stackspace left (See Advanced.). If you like to call functions with a lot of arguments (more than 256 bytes) you should raise this value.

   Stack extension builds on the same basic principle but allocates a new stackframe whenever necessary. If this happens at the entry of a function with arguments they have to be copied to the new stackframe so that the function may use them. Since C allows for a variable number of arguments the compiler doesn't always know how many arguments there are. Therefore only a fixed number of bytes is copied. If your functions may have lots of arguments (again more than 256 bytes) you should raise this number.

   Since allocation and freeing of memory through OS functions costs a lot of time (while a stack tends to be very dynamic) libnix caches once used stackframes and utilizes them again if necessary. The memory needed for this doesn't accumulate or such but just sticks to a maximum value raised once. This may look like a memory leak (while in fact it isn't). Be prepared for it.

1.15 libnix.info/stackextend usage

```
Using stack checking or extension
*********************************

   To utilize the stack checking or extension feature you need at least
V2.7.0 of gcc. With this compiler you get 2 new amiga specific options
that emit special code whenever necessary:

   * '-mstackcheck' Emits code that checks if there is enough stack
     left.  The program exits if not.

   * '-mstackextend' Tries to extend the stack before exiting (this may
     happen due to low or fragmented memory).

   Always use those switches together with '-lstack' to link with the
stack extension code or you will get a lot of undefined references ;-).
You can mix functions compiled with or without stack checking and
extention without problems.

   *Caution:*

   Do not use stack checking and/or extension switches when compiling
hook or interrupt code. Both run in alien contexts with a different
stack and all stack magic must fail. Also don't try to do some other
stack magic on your own if you want to use stack extension.

   Also note that a program compiled with stack extension/checking may
'exit()' at *any* function entry or when using alloca or variable sized
arrays. Either prepare your cleanup function accordingly (use
'atexit()') or don't use this feature.

   If you like to write or call functions with more than 256 bytes of
arguments (64 ints, longs or pointers) you should adjust the behaviour
of the stack extension code (See Advanced.).
```

1.16 libnix.info/Advanced

```
Stack extension fine tuning
***************************

   To adjust the behaviour of the stack extension code to your personal
needs you may set some of the following variables (or functions)

   'unsigned long __stk_minframe' (default: 32768)

   Minimum amount of memory to allocate for a new stackframe. Setting a
higher value speeds the code up but costs more memory if it is unused.

   'unsigned long __stk_safezone' (default: 2048)

   Size of the safety zone. Set this to a higher value if you want to
*call* functions with lots of arguments.
```

```
   'unsigned long __stk_argbytes' (default: 256)

   Number of bytes copied as arguments. Set this to a higher value if
*your* functions may have lots of arguments.

   'void _CXOVF(void)'

   Is a user replaceable stack overflow handler. The default one just
pops up a requester, then exits. This function is not allowed to return.
```

1.17 libnix.info/Costs

```
Overhead of stack extension
***************************

   The additional code needed for stack extension (or checking) costs
memory and CPU power. Here are some numbers to give you a very rough
idea for it.  (Times are in 1/60s, sizes in bytes):

     Test     normal      checking     extending   extending
        (big stack)  (big stack)  (big stack)  (small stack)

     Simple recursive
     function runtime    152  221       225      226
     (function calling
     overhead)

     Variable sized      52   136       398      468
     array runtime

     alloca runtime      31   118       118      118

     Own code size     1040        1160      1140

     Library code size     0  184       788
```

1.18 libnix.info/libnix.a

```
Some ANSI (mis)features
***********************

   I suppose you are familiar with C and especially ANSI C - if not you
should read a good book about it (1).  This chapter only contains some
special features of the implementation - you should know these if you
want to use this library.


  Locale
  Formatted I-O
```

```
  atof strtod
  Memory management
  Standard I-O
  Signal handling
  setjmp longjmp
  ctype
  clock
  Multibyte character functions

   ---------- Footnotes ----------

   (1)  I recommend this one:

   Brian W. Kernighan, Dennis M.Ritchie:
The C Programming Language (Second Edition)
Prentice Hall, Englewood Cliffs, 1988
```

1.19 libnix.info/Locale

```
Locale
******

   One feature of a complete ANSI compatible library is locale support.
The ANSI standard only knows of two locales:

   * "C" locale (normal C behaviour).

   * Default locale.

   Every other locale depends very heavily on the implementation.

   To do locale support on the amiga I decided to use locale.library
(what else).  This means that you normally have only these two locales
- to have more than that you must make some extra preferences files
with the locale preferences editor and give the path of these to the
setlocale()-call. If you do not have locale.library you will get only
"C" locale. This is the default then :-(.

   Another important point is that the ANSI standard requires the
default locale to be loaded at program startup. i.e. if you use german
locale (for example) you will just get it - printf and scanf will not
work as expected but use the decimal comma ',' instead of the decimal
point '.' for their floating point numbers, ctype functions will behave
differently, too.

   This can be very annoying if you don't want to use ANSI locale but
rather locale.library (which is not portable but IMHO much better) or if
you don't need locale support. And even dangerous if you don't test
your program under different locales.

   To get around this problem I decided to do some nasty thing: To get
locale support you have to make up a reference to setlocale. You can do
this by just calling
```

```
   setlocale(LC_ALL,"C");
```

   immediately after program startup. (And get "C" locale then after program start which is a much better choice). Or by just using setlocale anywhere in your program - you will get default locale at program startup then.

1.20 libnix.info/Formatted I-O

```
Formatted I/O
*************
```

   The formatted I/O specifications are all there (remember: this library tries to be ANSI compliant). But there are two things you should know about them:

   * The formatted I/O is affected by the setlocale() call - this is no bug, just an ANSI feature.

   * Half of the code of a full blown printf handles floating point numbers - but not everybody needs them. So there are two functions for both 'vfprintf' and 'vfscanf' - one in 'libnix.a' not including floating point support and one in 'libm.a' including floating point support.

     So if you want to use one of the formatted I/O specifiers for floats you should link with the math library '-lm'.

1.21 libnix.info/atof strtod

```
atof strtod
***********
```

   The two functions 'atof' and 'strtod' require a working '%f' specifier in 'vfscanf' - therefore they require the formatted I/O functions in the math library. Since 'libm.a' is linked before 'libnix.a' these two functions have been gone into the math library.

1.22 libnix.info/Memory management

```
Memory management
*****************
```

   Most of the memory management of this library runs through malloc(). Only the commandline parser uses AllocVec() - so you can use it without having the malloc function somewere in your program.

The memory management uses a local (to this task) memory pool to reduce memory fragmentation. It uses the system functions to do so (not the new pooled memory functions but just the older Allocate(), Deallocate() pair which are the <3.0 fallback for libamiga.a's pooled memory functions, too) so there should be no problems with it - these functions are tested very good.

The default blocksize for memory allocations is 16384 bytes - equivalent to 4 MMU pages. Bigger allocations are blown up to a multiple of 4096 bytes. So don't be alarmed if your program uses more memory then expected.

If you don't like this value (if you use bigger portions frequently or only use very little memory) you can replace it by declaring

'unsigned long _MSTEP'

You should use a multiple of MMU pages. If you don't use a full MMU page you gain nothing - malloc rounds up anyway.

1.23 libnix.info/Standard I-O

Standard I/O - where stdin, stdout, stderr come from
****

2 of the 3 standard I/O streams are no real problem:

* 'stdin' is set to the value the 'Input()' function of 'dos.library' serves,

* 'stdout' is set to the 'Output()' value.

Both streams are managed by the OS and the library need not take much care about them. But 'stderr' is a different thing since there is no 'Errput()' ;-) function. So 'stderr' is handled as follows:

1. If 'process->pr_CES' is set, this value is taken. There are not much shells that set this value so most of the time this leads to NULL.

2. If this didn't work and your program was started from CLI the library opens 'Open("*",MODE_NEWFILE)'. This opens the last interactive terminal attached to stdout, i.e. if you use the normal Amiga shell and redirect your output to a file you get the terminal, if you redirect your output to 'NIL:' you get 'NIL:'.

3. If this didn't work too (you never know) or your program was started from WB you simply get the same stream as in 'stdout'.

1.24 libnix.info/Signal handling

Signal handling
****

There is only support for the two signals SIGABRT and SIGINT. The library knows of some other signals but cannot generate them. The support for SIGABRT is simple - but SIGINT is a completely different thing:

You cannot use exec signal handlers since they are called at any time - even in the middle of a library call. And if your library just blocked a private semaphore and you jump out of the library code you will get a nice deadlock :-(. (And for people who don't know: signal handlers are bogus upto OS 2.0 (even there you need a good setpatch)).

So SIGINT (CTRL-C) is just polled at the start of most I/O-functions by calling the function

'void __chkabort(void)'

Other signals are even more difficult to implement:

* SIGSEGV simply doesn't exist - and if it does it's due to a VM system and should not be generated.

* SIGFPE is not generated by the math libraries - so it would be a bad thing to generate it by the mathematical coprocessor.

* SIGILL should never happen - your program must be faulty if you get one. Most of the time this happens if you try to run a 68020+ compiled program on a plain 68000.

* SIGTERM couldn't be disabled - even if it was there ;-).

You can disable CTRL-C handling by replacing '__chkabort' with a do-nothing stub function - but there is a better way. Just call

'signal(SIGINT,SIG_IGN)'

Replacing __chkabort is used very often by amiga-programs and if your application does not need CTRL-C handling at all and is amiga specific you can use this. The second method is the ANSI standard method and works on all types of machines.

1.25 libnix.info/setjmp longjmp

setjmp, longjmp
****

This library is compatible to the header files that come with gcc - and the jmp_buf in there is not large enough for the FPU registers. So they are not restored! The ANSI standard doesn't even require to restore any of the other local variables (they are restored :-)), so this is NO incompatibility to the ANSI standard.

1.26 libnix.info/ctype

```
ctype.h functions
*****************
```

   If you look into ctype.h you will see that the functions in there are just macros - and that they are duplicate in the library as functions. This is NOT a mistake. The ANSI standard requires such macros to be duplicate as functions.

   And remember: These functions are affected by the setlocale() call.

1.27 libnix.info/clock

```
The clock function
******************
```

   The clock() function's work is to measure processor time for the specific task - but there is no information like this in the amiga OS :-(. So it just measures the time from program start on - and is compatible with this behaviour to all single tasking OSs around.

1.28 libnix.info/Multibyte character functions

```
Multibyte character functions
*****************************
```

   The multibyte character functions are all there - but since the Amiga OS uses no other character set than ECMA Latin I they simulate just "C" locale. This means they do nothing useful.

1.29 libnix.info/libstubs.a

```
libstubs - automatic library opening
************************************
```

   The Amiga OS shared libraries are a nice thing. All the tasks can use them in parallel, they eat up memory only if you use them and they are simple to use - and all this works even without a memory management unit (MMU).

   Another nice feature is the fact that you can open them under

program control, i.e. you can take some action if they do not exist - warn the user, disable some features, etc. This nice feature becomes a misfeature if you only need a certain list of functions that are there all the time - exec, dos, intuition - you still have to open the shared libraries.

So most Amiga compilers have a feature called automatic library opening feature. This means that all libraries you reference (by calling one of the functions) but don't open yourself get opened for you by the compiler.

Auto-library-opening usage How to use it.
Auto-library-opening interface How it works.

1.30 libnix.info/Auto-library-opening usage

Usage
*****

To use this feature you have to do nothing (therefore it's called automatic). But you can control the library version if you wish by declaring

'long __oslibversion;'

somewhere in your program. But don't set this lower than 37 - most functions of libnix (including the commandline parsers) need 37 or more.

1.31 libnix.info/Auto-library-opening interface

Interface
****

Implementing such a feature is no hard work if you know how - this implementation uses a (not so good known) feature of the gnu linker called set elements:

1. You write a library entry for every library base and link this library as the last one. This means that the linker uses this library for every library base pointer that is not defined but referenced somewhere.

2. You tell the linker to collect these library bases together into a set element.

3. You write a function that opens all libraries in the set element at program start and cleans them up later.

Some details:

There are two object files in the library for every library base pointer. The first one is a

```
struct lib
{ struct Library *base;
  char *name; };
```

containing the library base pointer (a 'NULL' pointer at program start) and a pointer to the name of the library. This name

'extern char name[]'

is the second object. All these structs are collected together into one single set element called

'extern struct lib *__LIB_LIST__[]'

To open and close the shared libraries there are two functions in libstubs:

'void __initlibraries(void)'

and

'void __exitlibraries(void)'

Since it is still possible to open the shared libraries by hand I had to take care about the library base pointers for libnix itself - they are used in the commandline parsers - even before anybody could open them. There exists a (library private) duplicate library base pointer for each of these. They have normal names with two underscores in front.

So don't be alarmed if some system monitor tells you that your program opened dos.library twice - this is normal behaviour, most libraries do this.

Opening libraries by hand works exactly the same way as on any other compiler:

* You declare the library base variable somewhere globally:

  'struct DosLibrary *DOSBase=NULL;'

  The initialization '=NULL' is necessary! Uninitialized variables get overwritten by initialized ones in other object files - and the library base pointers in 'libstubs.a' are initialized with 'NULL'. This is a feature of the GNU ld and I cannot do much about it :(.

* You open the library before using it:

  'DOSBase=(struct DosLibrary *)OpenLibrary("dos.library",37);'

  and do some action if it fails :).

1.32 libnix.info/detach.o

```
Detaching from the current CLI
******************************

   Some people like multitasking that much that they tend to start
everything in the background. Some tools are able to do this
automatically - and with libnix you can write such tools, too. (1)

   To be able to detach from the current CLI the detach module has to
know how much stack your program needs, how to call the new process,
etc.  Therefore you will have to provide some global variables that
contain this information. If you don't provide them you will get
default values (don't blame me for them - it's your own fault :-} ).

   Here is an example set of variables:

     char *__procname="My nifty tool";
     long __priority=-1;     /* We don't eat that much processor time */
     unsigned long stack=50000; /* but need a large stack       */

   ---------- Footnotes ----------

   (1)   Please be aware that you lose the ability to synchronize your
tool with the calling CLI - this can be very nasty if one needs to. Use
this feature very sparse: Most of the time it's better to just rely on
the user being able to type `Run >NIL: <NIL:'.
```

1.33 libnix.info/Special startups

```
Special startups
****************

   As a serious Amiga programmer you may sooner or later want to write
your own shared library or device. This can be a very difficult task if
you never did it before. To make your life easier we did some of the
work for you if you decide to use one of our startups. Be aware that
writing a shared library is a task for the experienced programmer (1).


  Calling convention   How to interface to the system.
  libinit.o          Shared library startup.
  libinitr.o           Shared library with a new data segment for each caller.
  devinit.o          Device startup.

   ---------- Footnotes ----------

   (1)   Most ANSI library functions don't work out of a shared
library. libnix makes no difference here - even simple operations like
multiplying two integers can fail - don't take anything for granted.
```

1.34 libnix.info/Calling convention

```
Calling convention
******************

   On the amiga almost all shared libraries are called with the library
base in a6 and other parameters in other registers. The result is
placed in d0 usually.

   If you want to write your own shared library you should stick to
this model to make it easier for others to interface with it - but
unfortunately gcc doesn't support registerized parameters. The solution
to this problem is to write an assembler wrapper for each function you
need. Since all those wrappers look the same it's easy to simplify
their notation by using preprocessor macros. You can find appropriate
macros in the stabs.h file of the libnix sources. Usage:

     /* Define some function that has 1 argument in d0 */
     ADDTABL_1(__UserFunc,d0);

   As a bonus these macros add your function to your library's jump
vector.  All you have to do is to care for the right linkage order. And
don't forget to add a `ADDTABL_END();' at the end of the vector.

   Attention: Other programmers may decide to patch into your library's
jump vector - therefore it's a good idea to call even own functions
over this vector. To achieve this you have to provide some inline
functions (like those in the gnu:os-include/inline directory) or glue
code and you have to privatize the function's name by adding some '__'s
or similar. You will also have to set up your OWN library base.
```

1.35 libnix.info/libinit.o

```
Shared library startup
**********************

   This startup gives you one data segment for all possible callers.
You will have to use semaphores to share special data between them.

   To write a shared library you will have to provide some global
variables

     const BYTE LibName[]="simple.library";
     const BYTE LibIdString[]="version 1.0";
     const UWORD LibVersion=1;
     const UWORD LibRevision=0;

   as well as some special functions (1)

     int __UserLibInit(struct Library *myLib);
     void __UserLibCleanUp();

   Please look into the examples directory for more details.
```

---------- Footnotes ----------

(1) It'll be possible to add an Open() and Close() function, too. But this would be incompatible to libinitr.o and wouldn't give any advantages over this method.

1.36 libnix.info/libinitr.o

Shared library with different data segments
****

If you don't like the hassle with semaphores you can use this startup. It provides a new data segment for each task that opens your library. There are two disadvantages over libinit.o:

* This method needs more memory.

* You cannot interact between your tasks.

The usage stays the same.

1.37 libnix.info/devinit.o

Device startup
****

The device startup uses different names over the library startup (though a device is always a shared library on the Amiga).

```
const BYTE DevName[]="simple.device";
const BYTE DevIdString[]="version 1.0";
const UWORD DevVersion=1;
const UWORD DevRevision=0;
int __UserDevInit(struct Device *myDev);
void __UserDevCleanUp();
int __UserDevOpen(struct IORequest *iorq,ULONG unit,ULONG flags);
void __UserDevClose(struct IORequest *iorq);
(And some begin and abort function as well)
```

1.38 libnix.info/Set elements

Set elements - a nice feature of the gnu ld
****

Set elements are used very often by this library. Since most people don't know them they are explained here a second time.

You can tell the linker to build up an array of pointers to every global symbol in your program (functions or variables) even if your symbols are scattered among some object files. These arrays are called set elements.

You can take 4 Library base pointers

'DOSBase', 'IntuitionBase', 'GfxBase', 'IconBase'

tell the linker to put them together into a set element called 'librarybases' by placing some assembler lines like

'asm(".stabs \"_librarybases\",24,0,0,_DOSBase")'

into your code (22 for text, 24 for data, 26 for bss - and don't forget the single underscore) and get an array of pointers like this:

```
  void *librarybases[]=
  { (void *)4,&DOSBase,&IntuitionBase,&GfxBase,&Iconbase,NULL };
```

The first element contains the number of symbols. The last element contains a NULL pointer. And remember: This are pointers to the pointer variables.

This is the basis of global constructors and destructors in C++ and is very useful on the amiga to implement an auto-library-opening feature :-). Set elements are used in this library for collecting together library bases, initialization routines and cleanup routines.

1.39 libnix.info/geta4

geta4 and other things - some words on code and data models
****

A program consists of two portions - code and data (with the exception of self modifying code - you cannot get this out of GCC and it is a bad thing to do - so forget about it).

A program usually accesses them in a unique style of addressing modes for the machine instructions - called a code (or data) model.

On the amiga OS there exist two addressing styles for both of them - code and data. With full 32 bit addresses - giving you access to 4 GB of address space. And with reduced 16 bit addresses - giving you access to only 32k of code and 64k of data. These styles are called large (or normal) and small code and data models. Usually small code comes together with small data - but that's not necessary.

(Don't mix code and data models with the memory models of MS-DOS machines: The memory in small data model is still flat 4 GB which means by using pointers you can still address the whole memory. Only the number of variables is limited.)

You may think that these limitations are a large disadvantage - where are the benefits?

The benefits are simple: Code size and performance.

Every time you access a 16 bit address instead of 32 bit you spare 2 bytes in code size meaning 2 bytes in program size. And the processor needs to load and process a smaller instruction that needs less processor cycles. And since these 16 bit addresses are relative the loader of the OS need not relocate them. Meaning that you spare even 8 bytes more in executable size and some loading time.

And there is another advantage: If all the data is addressed relative it is simple to relocate it at program start - which means that you can easily get multientrant and reentrant executables (the code section is constant and need not be relocated). You know these as pure=resident programs.

Some details:

Data models
Code models

1.40 libnix.info/Data models

Data models - large and small
****

Let's take a simple C program:

```
#include <stdio.h>

int max=100;
int count;
char string[]="Hello, world\n";

int main(void)
{ int i;
  for(i=0;i<max;i++)
  { count++;
    printf("%s",string); }
  return 0;
}
```

If we look at it carefully we see 4 different types of data in it:

1. The variable 'max' and the array 'string' - both are nonconstant initialized global data.

2. The variable 'count' - this is uninitialized (and therefore nonconstant) global data.

3. The string "%s" - this one is constant data.

4. The variable ‘i’ - this one is local data.

The compiler places these 4 types of data into 4 different places:

Data number 4 is local (and exists only in one function call). The compiler places such data into registers if possible. On the stack if this is not possible. If you do not want the compiler to place data into registers declare it volatile - it will be on the stack then all the time.

Data number 3 is constant - the compiler places it together with all the other constant data into the code section (code is also constant). Never change any constant data - the weirdest things can happen.

Data number 2 is not initialized - it would be a bad thing to put data without information into an executable. Therefore such data goes into a special section - the BSS section. BSS data does not increase executable size.

And the rest (number 1) goes into the data section :-).

To access the data section with machine instructions there are two possible methods:

You can take the whole 32 bit address and store it into your machine code. 32 bit means 4 byte every time you access a global variable. This is know as the large (normal) data model because you can access the whole bunch of 4 GB address space.

A lot of applications do not need such a large data section and it would be a waste of memory to do so. So there exists a second possibility:

You take one address register (a4) and use it as a pointer to your data section. You access your data relative to this pointer with 16 bit references. This is known as small data model.

Since there are only 16 bit references you can access a total of 64k of data (32k in each direction from a4 on). And since you use only one address register for this the data and BSS section get merged together (BSS data still need not increase executable size - there exist some tricks to prevent this - but not all linkers support such tricks).

Beware: you should never lose the contents of your address register (a4) or all the hell breaks lose.

If you ever lost them (this can only happen in certain cases when using interrupts or hooks) you can restore them by calling ‘geta4()’ or you can use no global data at all (and have no problems then).

The second method is recommended - and it is possible sometimes since the OS takes care of this and supports local data areas in these nasty cases. But don’t call any shared library - or you will access (hidden) a library base pointer.

It is not possible to have a ‘geta4()’ function with resident

programs = multiple data sections (which one would you choose? You
access the data from a different task!)

It's in general not possible to mix objects compiled for the two
data models. There are some exceptions, but people that know enough to
prevent collosions need no explanation of when it's possible ;-).

1.41 libnix.info/Code models

Code models
****

All your constant data and all (constant) code form the code section.
To access the code section there exist two code models:

You can take the whole 32 bit address to call a function and can
write programs 4GB large. This is the large (normal) code model.

But you can even call your functions relative to the program counter
(pc) with 16 bit offsets. This is the small code model. The advantages
are the same as in the small data model - only the disadvantages are
different:

* You can only have a total of 32k of code since you need to jump in
  both directions. But even this is enough for a lot of programs.

* There is no address register that can be lost - the program counter
  is valid the whole time.

* It is possible to mix large and small code model - but you can
  hardly get more than 32k of code out of it.

1.42 libnix.info/Library bases

Library base pointers - and how they work
****

The model of shared libraries on the amiga works as follows:

The library is not managed by the linker but by the application.
You open it through a system function. The advantage of this is clear:
You open libraries under program control, i.e. you can even check if
they are there and disable some features if not or take other action.

The result of this system function is a pointer to the upper end of a
jump table (a table of 'jmp' instructions to the different functions)
and the lower end of a library structure containing extra information
for every library you opened. These pointers are called library base
pointers. They are usually stored in normal global variables.

To call a system function you have to put the library base into address register a6 and the parameters into certain other registers. Then your program has to jump over the certain address of the jump table. The function returns with the result in register d0 (and sometimes some more).

A compiler can handle this behaviour by two different methods:

* It can just do the right thing and place everything into the desired registers - gcc does this by declaring special assembler inline functions that do the job.

* It can put the arguments on the stack (as in every normal function call) and call a glue function that does nothing else then taking the arguments from the stack and putting them into the right registers then calling the function. This glue code is contained in 'amiga.lib'.

Both methods require to access the library base pointer (and a valid value in it) so they make up a reference to this variable.

1.43 libnix.info/libamiga.a

Glue code and some other things
****

It's not possible for me to redistribute amiga.lib - but you should have one if you really want to use the possibilities of your amiga. (You don't need one if you only want to use ANSI features or if you use the inline headers of gcc) If you want to compile resident programs you will need a baserelative version too.

To solve this problem I decided to build a selfmade version of libamiga.a. The gluecode of this library is built out of the inline header files of gcc, some of the other functions are written from scratch. This does not give you a complete version of libamiga.a but a better that nothing version including sources. To rebuild it unpack the sources, then type a 'make libamiga'.

If you want to have a fully functional version of libamiga.a you can use the real one. To do this you will have to convert normal amiga objectfile format to a.out format (known by the linker). Type:

```
cd <some empty directory>
stack 300000
sh
Hunk2GCC <path>amiga.lib
ar -q libamiga.a obj*
rm *.o
ranlib libamiga.a
exit
```

Doing this on 'RAM:' will improve performance a lot.

This doesn't give you the baserelative version blib/libamiga.a - you will be unable to compile resident programs. To get a baserelative version of amiga.lib try to get the 'libtos' program of the 'DICE' compiler of M. Dillon (from fishdisk or somewhere else) - it converts libraries to baserelative ones:

```
cd <some empty directory>
lha x amigalibdisk491:dice/dice206_21.lzh #?/libtos
netdcc/bin/libtos <path>amiga.lib amigas.lib
```

Then do the same as above.

1.44 libnix.info/Code size

```
Writing small programs
**********************
```

Writing very small programs is a trivial thing - if you know how to do it. Here are some basic tips and tricks:

Do not use printf:
: 'printf' is the most high-level function of usual ANSI C libraries and builds on almost everything else. Often this function can be replaced by simpler routines like e.g. 'puts'. The same is true for the other formatted I/O functions.

  If you write amiga-only programs you can use the 'sprintf' routine of libamiga.a by linking with '-lamiga'.

Try to avoid using buffered I/O:
: A lot of programs can be written using the low-level I/O-functions 'read' and 'write'. This saves the code responsible for buffered I/O.

Avoid -O3:
: '-O3' activates all those optimizations that sacrifice code size for speed. An exception is '-fstrength-reduce' which costs compile time but not code size. '-O2 -fstrength-reduce' will give small programs.

Strip your executables:
: gcc doesn't strip executables by default. Setting '-s' will give worse debugable but small programs.

Use small code and data model:
: '-msmall-code -fbaserel' do this. Be aware that this won't work for larger programs.

Don't use libgcc.a:
: 'libgcc.a' which calls global C++ constructors and destructors builds on 'atexit' which builds on 'malloc', ... If your program doesn't use them you can save some memory by linking by hand ('-nostdlib').

```
Set __nocommandline:
     If you use 'ReadArgs' instead of 'argc, argv' you can save some
     memory by setting a global variable named '__nocommandline' to drop
     parsing of commandline arguments.
```

1.45 libnix.info/FAQs

```
FAQs
****

Q:
     I do not get a working executable out of it - my debugger tells me
     the library bases are broken.

A:
     The GNU ld that comes with GCC 2.5.8 (or lower) has some serious
     bugs in conjunction with set elements.  Use the fixed version of
     ld that comes with gcc 2.6.0 (or above).

Q:
     There are some prototypes missing in stdio.h.

A:
     This stdio.h is only for internal use - use the normal GCC stdio.h
     to compile your programs.

Q:
     While printing floats printf prints weird characters.

A:
     This problem should be fixed with the current release of libnix.
     It was caused by a bug in the system math libraries which happens
     if you open them in the wrong order. You can use the SetMathPatch
     program by Andreas Wolff to fix this and another more serious bug
     with mc68040 processors.
```