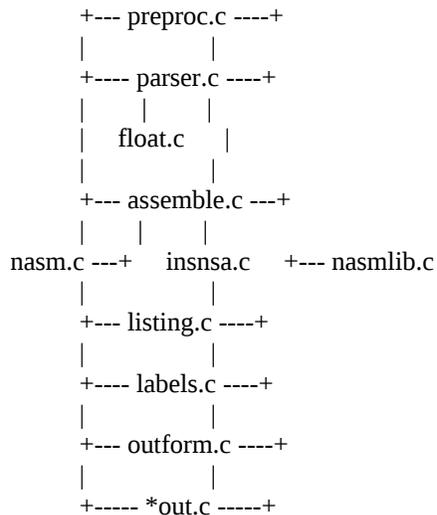


## Internals of the Netwide Assembler

The Netwide Assembler is intended to be a modular, re-usable x86 assembler, which can be embedded in other programs, for example as the back end to a compiler.

The assembler is composed of modules. The interfaces between them look like:



In other words, each of `preproc.c', `parser.c', `assemble.c', `labels.c', `listing.c', `outform.c' and each of the output format modules `\*out.c' are independent modules, which do not directly inter-communicate except through the main program.

The Netwide *\*Disassembler\** is not intended to be particularly portable or reusable or anything, however. So I won't bother documenting it here. :-)

### nasmlib.c

-----

This is a library module; it contains simple library routines which may be referenced by all other modules. Among these are a set of wrappers around the standard `malloc' routines, which will report a fatal error if they run out of memory, rather than returning NULL.

### preproc.c

-----

This contains a macro preprocessor, which takes a file name as input and returns a sequence of preprocessed source lines. The only symbol exported from the module is `nasmpp', which is a data structure of type `Preproc', declared in nasm.h. This structure contains pointers to all the functions designed to be callable from outside the module.

### parser.c

-----

This contains a source-line parser. It parses 'canonical' assembly source lines, containing some combination of the 'label', 'opcode', 'operand' and 'comment' fields: it does not process directives or macros. It exports two functions: 'parse\_line' and 'cleanup\_insn'.

'parse\_line' is the main parser function: you pass it a source line in ASCII text form, and it returns you an 'insn' structure containing all the details of the instruction on that line. The parameters it requires are:

- The location (segment, offset) where the instruction on this line will eventually be placed. This is necessary in order to evaluate expressions containing the Here token, '\$'.
- A function which can be called to retrieve the value of any symbols the source line references.
- Which pass the assembler is on: an undefined symbol only causes an error condition on pass two.
- The source line to be parsed.
- A structure to fill with the results of the parse.
- A function which can be called to report errors.

Some instructions (DB, DW, DD for example) can require an arbitrary amount of storage, and so some of the members of the resulting 'insn' structure will be dynamically allocated. The other function exported by 'parser.c' is 'cleanup\_insn', which can be called to deallocate any dynamic storage associated with the results of a parse.

names.c  
-----

This doesn't count as a module - it defines a few arrays which are shared between NASM and NDISASM, so it's a separate file which is #included by both parser.c and disasm.c.

float.c  
-----

This is essentially a library module: it exports one function, 'float\_const', which converts an ASCII representation of a floating-point number into an x86-compatible binary representation, without using any built-in floating-point arithmetic (so it will run on any platform, portably). It calls nothing, and is called only by 'parser.c'. Note that the function 'float\_const' must be passed an error reporting routine.

assemble.c  
-----

This module contains the code generator: it translates 'insn' structures as returned from the parser module into actual generated code which can be placed in an output file. It exports two functions, 'assemble' and 'insn\_size'.

``insn_size'` is designed to be called on pass one of assembly: it takes an ``insn'` structure as input, and returns the amount of space that would be taken up if the instruction described in the structure were to be converted to real machine code. ``insn_size'` also requires to be told the location (as a segment/offset pair) where the instruction would be assembled, the mode of assembly (16/32 bit default), and a function it can call to report errors.

``assemble'` is designed to be called on pass two: it takes all the parameters that ``insn_size'` does, but has an extra parameter which is an output driver. ``assemble'` actually converts the input instruction into machine code, and outputs the machine code by means of calling the ``output'` function of the driver.

insnsa.c

-----

This is another library module: it exports one very big array of instruction translations. It is generated automatically from the `insns.dat` file by the `insns.pl` script.

labels.c

-----

This module contains a label manager. It exports six functions:

``init_labels'` should be called before any other function in the module. ``cleanup_labels'` may be called after all other use of the module has finished, to deallocate storage.

``define_label'` is called to define new labels: you pass it the name of the label to be defined, and the (segment,offset) pair giving the value of the label. It is also passed an error-reporting function, and an output driver structure (so that it can call the output driver's label-definition function). ``define_label'` mentally prepends the name of the most recently defined non-local label to any label beginning with a period.

``define_label_stub'` is designed to be called in pass two, once all the labels have already been defined: it does nothing except to update the "most-recently-defined-non-local-label" status, so that references to local labels in pass two will work correctly.

``declare_as_global'` is used to declare that a label should be global. It must be called `_before_` the label in question is defined.

Finally, ``lookup_label'` attempts to translate a label name into a (segment,offset) pair. It returns non-zero on success.

The label manager module is (theoretically :) restartable: after calling ``cleanup_labels'`, you can call ``init_labels'` again, and start a new assembly with a new set of symbols.

listing.c

-----

This file contains the listing file generator. The interface to the

module is through the one symbol it exports, `nasmlist`, which is a structure containing six function pointers. The calling semantics of these functions isn't terribly well thought out, as yet, but it works (just about) so it's going to get left alone for now...

outform.c

-----

This small module contains a set of routines to manage a list of output formats, and select one given a keyword. It contains three small routines: `ofmt\_register` which registers an output driver as part of the managed list, `ofmt\_list` which lists the available drivers on stdout, and `ofmt\_find` which tries to find the driver corresponding to a given name.

The output modules

-----

Each of the output modules, `outbin.o`, `outelf.o` and so on, exports only one symbol, which is an output driver data structure containing pointers to all the functions needed to produce output files of the appropriate type.

The exception to this is `outcoff.o`, which exports `_two_` output driver structures, since COFF and Win32 object file formats are very similar and most of the code is shared between them.

nasm.c

-----

This is the main program: it calls all the functions in the above modules, and puts them together to form a working assembler. We hope. :-)

Segment Mechanism

-----

In NASM, the term `segment` is used to separate the different sections/segments/groups of which an object file is composed. Essentially, every address NASM is capable of understanding is expressed as an offset from the beginning of some segment.

The defining property of a segment is that if two symbols are declared in the same segment, then the distance between them is fixed at assembly time. Hence every externally-declared variable must be declared in its own segment, since none of the locations of these are known, and so no distances may be computed at assembly time.

The special segment value `NO_SEG` (-1) is used to denote an absolute value, e.g. a constant whose value does not depend on relocation, such as the `_size_` of a data object.

Apart from `NO_SEG`, segment indices all have their least significant bit clear, if they refer to actual in-memory segments. For each segment of this type, there is an auxiliary segment value, defined to be the same number but with the LSB set, which denotes the segment-base value of that segment, for object formats which support

it (Microsoft .OBJ, for example).

Hence, if ``textsymb'` is declared in a code segment with index 2, then referencing ``SEG textsymb'` would return zero offset from segment-index 3. Or, in object formats which don't understand such references, it would return an error instead.

The next twist is `SEG_ABS`. Some symbols may be declared with a segment value of `SEG_ABS` plus a 16-bit constant: this indicates that they are far-absolute symbols, such as the BIOS keyboard buffer under MS-DOS, which always resides at 0040h:001Eh. Far-absolutes are handled with care in the parser, since they are supposed to evaluate simply to their offset part within expressions, but applying `SEG` to one should yield its segment part. A far-absolute should never find its way `_out_` of the parser, unless it is enclosed in a `WRT` clause, in which case Microsoft 16-bit object formats will want to know about it.

#### Porting Issues

-----

We have tried to write NASM in portable ANSI C: we do not assume little-endianness or any hardware characteristics (in order that NASM should work as a cross-assembler for x86 platforms, even when run on other, stranger machines).

Assumptions we `_have_` made are:

- We assume that ``short'` is at least 16 bits, and ``long'` at least 32. This really `_shouldn't_` be a problem, since Kernighan and Ritchie tell us we are entitled to do so.
- We rely on having more than 6 characters of significance on externally linked symbols in the NASM sources. This may get fixed at some point. We haven't yet come across a linker brain-dead enough to get it wrong anyway.
- We assume that ``fopen'` using the mode "wb" can be used to write binary data files. This may be wrong on systems like VMS, with a strange file system. Though why you'd want to run NASM on VMS is beyond me anyway.

That's it. Subject to those caveats, NASM should be completely portable. If not, we `_really_` want to know about it.

#### Porting Non-Issues

-----

The following is `_not_` a portability problem, although it looks like one.

- When compiling with some versions of DJGPP, you may get errors such as ``warning: ANSI C forbids braced-groups within expressions'`. This isn't NASM's fault - the problem seems to be that DJGPP's definitions of the `<ctype.h>` macros include a GNU-specific C extension. So when compiling using `-ansi` and `-pedantic`, DJGPP complains about its own header files. It isn't a problem anyway, since it still generates correct code.