# snma.hyper

# Contents

# Chapter 1

# snma.hyper

## 1.1 Samu Nuojua's Macro Assembler, SNMA v1.95

```
                *******************************************************
 *              *
 *       S   N   M   A    V1.95   *
 *       ~~~~~~~~~~~~~~~~~~~~~~   *
 *******************************************************

   SNMA is 680x0 conditional macro assembler.
```

## 1.2 1. Introduction

```
                   Introduction to Samu Nuojua's Macro Assembler, SNMA

SNMA is a 680x0/6888x macro assembler. SNMA requires OS 2.0+.
If you have used another assembler, snma should not throw big
```

surprises in your face (I hope). (Look at things to note )

## 1.3 1.1 Copyright ©

SNMA stands for Samu Nuojua's Macro Assembler.

SNMA is © copyright 1993-1994 by Samu Nuojua.

All SNMA documents are © copyright 1993-1994 by Samu Nuojua.

SNMA is FREEWARE. I reserve all rights to SNMA. You can copy it as long you don't ask payment (a small fee is allowed to cover the expenses of a possible disk/postage fee). Permission is granted to upload SNMA to bulletin boards and FTP sites. However, you must include all the files present in the original archive when distributing SNMA to anywhere or anybody. This includes all documents.

DISCLAIMER:
~~~~~~~~~~~
SNMA software and documents are provided 'as is'. No guarantee of any kind is given as to what SNMA does or that the information in files is correct in any way. You are using this software at your own risk. The author of SNMA is in NO WAY responsible for any loss or damage caused by SNMA.

## 1.4 1.2 What you need

What you need to use SNMA to produce stand alone programs.

- AmigaOS 2.04 or higher (V37)
- Text editor (to write/edit programs)
~~~~~~~~~~~

```
-   The following libaries:
            V.    Where  note
            ~~    ~~~~~  ~~~~
  - dos.library       37    in rom
  - intuition.library     36    in rom
  - utility.library     36    in rom
  - icon.library        any     * in rom   WB support
  - mathieeedoubbas.library   any     * libs:  fp support
  - mathieeedoubtrans.library any     * libs:  fp support
  - rexxsyslib.library      36      * libs:  ARexx support

    Libraries marked with a * are not neccessarily required.  The math
    math  libraries  are  needed  for single and double floating point
    conversions.  A 6888x or 680x0 with FPU is needed for the extended
    floating point  conversions.  Rexxsyslib is required for the ARexx
    support  and  icon.library for  the WB support (if snma is started
    with the icon).


  Although snma  can now produce executables, you may still need a linker to
  to link object modules together (i.e. when your sources consist of several
  modules).  There are many choices for the linker and snma should work with
  the ones which can deal with standard hunks.  I have  used DLink (from the
  freely distributable  DICE, not the registered  or commercial versions - I
  don't know if they differ anyway) which works just fine and is free.

  Recommended:

-   Hard Disk
-   Manuals, manuals...
-   Debugger
-   Time (8')
-   Development tools (Includes and so on)

  *********************************************************************
  *  Remember, this isn't pascal, this is REAL programming.           *
  *  - 68000 Assembly language, techniques for building programs  *
  *********************************************************************

  (Good book (a bit old, however) by D.Krantz and J.Stanley).
```

## 1.5 1.3 Installing SNMA

```
  There are couple of files to be copied.  Installer? Well, as soon as I
  have some time to spare.

  SNMA  The main file.  Copy it somewhere  in your search  path if
  you are using snma from the shell.   If you are using snma
  in its ARexx mode, it needs to be started only once so  it
  is not necessary that  this file be  in the search path in
  this case.   Also, snma now has  an icon so that it can be
  started directly from the WB.

  SNMA.guide  Documents in AmigaGuide format.  Copy this one to anywhere
  you like.
```

```
    examples/ Very simple example files.

    arexx/  ARexx macros.  Copy to  your  ARexx:  directory, whichever
    ones you are going to use.


    See also examples/alias.txt file.


    (User friendlies at the best 8'| )
```

## 1.6 1.4 Good & Bad

```
                    My personal view on this assembler.
    (Things are not listed in any particular order)

    Good:

  - It's free.
  - Most  common
                directives
                are supported.
  - Macros are supported.
  - All 680x0, 6888x, 68851 and 68030 PMMU instructions are supported
  - Does normal optimizations, including forward branches.
  - It's coded in assembler. (See below, Bad things).
  - Enforcer was in duty all the time I coded, checked, debugged...
  - Supports all data types of 680x0 6888x family (I think).
    (FFP conversions are not supported).
  -
                ARexx
                interface.
  - Global symbol table

    Bad:

  - 68040/68060 instructions (those few) are not yet supported.
  - All source files must fit into memory at the same time.
  - It's coded in assembler. Messy code sometimes - my fault, my
    problem. (Moral: Assembly language is a two edged sword).
  - Doesn't support small data models.
  - Some sort of beginner's help would be good (sources...).
  - Only output format is Amiga object code.
  - No LINE DEBUG HUNK (or whatever it is)  nor any new hunks.
  - This document is a translator's nightmare!!!!
    From the one (TK) who tried to fix english...
    (Author's note: Oh, life is so hard. 8^)
  - No GUI, but I'm not so sure this would even be useful.
  - It's driving me crazy, sometimes.
```

## 1.7 1.5 History of SNMA

History is now contained in a separate text file entitled "History".

## 1.8 1.6 Bug reports

```
                Bug reports are WELCOME.

Please, state the following facts:

1) Your system configuration (Model, CPU, MEM, OS, ...) and the
   version of SNMA.

2) What you did -  Source code which caused the bug - If I can't make
   the bug reappear,  it is an awful task to find out what went wrong.
   If at all possible try to isolate the bug. Usually, only  a tiny
   section of source code is  required to show it.  Or, if the bug is
   not directly  related to the source code, describe  it clearly (in
   any case).

If you find something is implemented badly, missing or could use a
little polishing, along with other similar things, suggestions are
welcome.

I'm also intrested if snma works on all (OS2.04+) Amiga models
with enough memory.

Where to report, see
               author
              .

     ----------------------------
     --* fixed bug better bug *--
     ----------------------------
```

## 1.9 1.7 Misc., general things

```
Some words from the inner workings of SNMA
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SNMA is not a traditional  two pass assembler although its operations
can be divided into two stages. Source files are read and parsed only
once. SNMA creates its own internal structures in this first pass. In
the  second pass,  snma solves all undefined symbols, optimizes code,
recalculates  the changed values  (like pc-relative stuff) and writes
object code (and other requested files).

Listing  file creation is sloooow.  (I suspect the buffered IO, but I
cannot  make sure because  I don't have 3.1  where SetVBuf() actually
does something).

Memory usage by snma isn't the most economical since all source files
```

files must fit into memory. (Not to mention the way I implemented all the other stuff). Strip the comments from the include files you use because that will reduce memory usage.

As a memory usage example:

```
One source code module:     size 25625 bytes, 1061 lines
          (7432 lines with includes)
```

SNMA uses 99403 bytes to store source files and 225356 bytes for all the other stuff making a total of 324759 bytes.

The way I have implemented macros costs memory since snma needs the produced macro lines to be in memory during the second pass where expressions with relative and undefined symbols are re-calculated and possible errors shown. (SNMA won't read the source twice).

I know the actual snma file is quite large also. Well, I can't help that a lot, without a complete re-design and re-write which is not in my top ten list of how to spend the next years of my life.

The development of SNMA
~~~~~~~~~~~~~~~~~~~~~~~

I started the development of snma somewhere in the first half of 1993. I have coded SNMA entirely in assembler - most of it with a68k and minor parts with snma itself. It operates pretty stable on my system (A2000, GVP A3001 28Mhz 68030/882, 4M fast 1Mchip, OS2.04). I'm developing it in my free time and have spent long nights staring at my old 1081 and wondering what the **** is wrong with everything.

## 1.10 1.8 Thanks

Thanks
~~~~~~

First, to the following people:

Eric Augustine for helping me with this guide.
Laura Mahoney for sending me source code.

and all the other people who have sent in bug reports.

I really must thank all you who have made so much wonderful 'Freely Distributable' software for the Amiga.

Also, thanks to the following people for the software I mainly used while developing SNMA:

- Charlie Gibbs, A68k
- Matt Dillon, DME and DLink
- Aaron Digulla, XDME
- Steffan Becker, ToolManager

```
  - Jorrit Tyberghein, PowerVisor.
```

## 1.11 2. How to use

```
                    SNMA can be started from the Shell or from the WorkBench.

    2.1
                 Shell startup
                    2.2
                 Workbench startup
                    2.3
                 Starting arexx host
```

## 1.12 2.1 Startup from shell

```
                    SNMA can be started  from the shell like a 'traditional'  ↩
                       assembler.  It
    parses its arguments using AmigaDOS 2.0 templates.  Some of the options
    can be specified also in the
                 'old way'
                , i.e. with '-'. These 'old way'
    arguments override the template arguments.


    Command Line template:

    SNMA
      SOURCEFILE/A O=OBJ/K O=OBJ/K I=INCLUDE/K H=HEADER/K E=EQUATE/K
      L=LISTING/K Q=QUICKOPT P=PCOPT A=ADDRESSOPT B=BASEFORCE S=SYMBOL
      L=LONGBRA AREXX/S PORTNAME/K QUIET/S PAGELEN/K/N LNM=LSTNOMAC/S
      LOD=LISTONLYDATA/S EXEOBJ/S SR=SHORTRELOC32 KEEPOBJ/S MOVEM/T


    Where:

    SOURCEFILE      is the name of the source file.  It is a first argument
        and must always be there.

    OBJ       defines the name of the object file.
        alias: -o<name>

    INCLUDE     defines a list of directories where the
                 INCLUDE
                        directive searches for include files.
        alias: -i<namelist>

    HEADER      defines one  file to be  included before any lines from
        the source file are assembled.  Only one is allowed.
        alias: -h[<name>]
```

```
EQUATE      defines the equate  file name to be created.   An empty
    string is  allowed in  which case SNMA creates the name
    from  the source file name.  Currently  the equate file
    is generated at the end of assembly.  Symbols are taken
    from the hash  table, so they are in mixed order. You
    can use the AmigaDOS SORT command if you like some sort
    of order.
    alias: -e[<name>]

LISTING     Listing  file to be created.  Minimal formatting can be
    done by using the PAGELEN option.
    alias: -l[<name>]

QUICKOPT      Quick optimizing flag,   default: on
PCOPT     pc-relative optimizing,  default: off
ADDRESSOPT      effective address optimizing default: on
BASEFORCE     Auto-force (Bd,An)->(disp16,An) default: on
SYMBOL      Write symbol data hunk   default: off
LONGBRA     Long branches (wo/size field) default: off

AREXX     flag to start
             AREXX
             command host SNMA. Overrides
    other directives.

QUIET     Disables informational output.

PAGELEN     Defines  pagelen used in  listing  file.  If null, snma
    does  not  create  pages (useful if you want to do your
    own formatting).
    Default: NULL.

LSTNOMAC      Don't include macro expansions in the listing file.
    Default: off.

LSTONLYDATA     Only list the lines which actually define some data.
    Default: off.

EXEOBJ      Produces  an  executable,  instead of an object module.
    SNMAOPT e+ causes same effect.
    Default: off.

SHORTRELOC32    write HUNK_RELOC32SHORT  instead of HUNK_RELOC32 when-
    ever  possible.  Has effect  only  when EXEOBJ is also
    specified.   Note that  the actual hunk written is NOT
    HUNK_RELOC32SHORT (1020)  due to a  2.0 bug, but it is
    HUNK_DRELOC32 (1015).  Since that hunk cannot be in an
    executable it  doesn't matter.  Executables which have
    this hunk, work only in OS 2.0 or later.
    Default: off.

KEEPOBJ     Don't delete object file if there are errors in source
    file.
    Default: off.

MOVEM     Optimize movem (to move or remove) if possible.
    Default: on.
```

See
Options
.

SNMA does not check the stack. I haven't had any problems with a stack size of 4000 bytes. You can overflow the stack with very deep expressions or very nested includes. One level on both cases takes about 100 bytes, so "very" means something like 40 levels. Actually, a little less. If you worry about that, use a bigger stack.

Examples

## 1.13 2.1.1 template: old flags

Many applications use the '-' as an option start character.
(like: a68k file.asm -l -iinclude:).

If you wanted to just flag that you needed some file to be generated and wanted snma to create the file name from the source name, you had to pass an empty string (like LISTING ""). I have to admit that this isn't a very elegant method and when trying to pass options to ARexx macros I had many problems with the quotes. So, to allow easier use, a couple of '-' flags are also allowed on the commandline. In template they show as OLDFLAGS/M template. In the template explanation they are referred to as 'alias:'.

## 1.14 2.1.2 Command Line Examples

```
            prompt> SNMA  mycode.asm obj mycode.o include myinc:
```

Mycode.asm is a source file, mycode.o is an object file and include files are searched for in the current directory and then from myinc: directory.

```
prompt> SNMA mycode.asm  Q on A off B off S on   I work:,work2:inc
```

mycode.asm is a source file. Flags are set on and off. Include files are searched for in the current directory, work: and work2:inc.

```
prompt> SNMA mycode.asm EQUATE myequ
```

produces the equate file named myequ.equ and mycode.o object code.

```
prompt> SNMA mycode.asm E ""
```

produces the equate file named mycode.equ and mycode.o object code.

```
prompt> SNMA arexx
```

```
    Starts SNMA
                 ARexx
                 command host.

prompt> SNMA mycode.s -e -l -iinclude:

    mycode.s is source code, -e and -l flags  snma to produce equate and
    listing file, and -iinclude: tells snma to search include files from
    the include: directory.
```

## 1.15 2.2 Workbench support

```
                    SNMA may  be started  from the  Workbench,  too.  Its   ←
                        behaviour is
    controlled with ToolTypes. SNMA can assemble file(s) or start ARexx
    SNMA.  You  can  disable a ToolType by removing it or setting it to
    parenthesis "()".

    Tooltypes:
    ~~~~~~~~~

    AREXX       flag to start snma in ARexx mode.

    PORTNAME=<name>     AREXX port name.  If omitted, snma uses the
            default name (SNMA).

    WINDOW=<file>     Specify output file.  If omitted, snma will
            use its  default  output (CON:...).  If the
            AREXX  flag  is set  too, no default output
            is created, if omitted.  See
                 ARexx/SET
                    The  default  tooltypes are set  so that AREXX snma will be  ←
                        started.
    You can have several SNMAs running at the same time (although it is
    not very useful).  If you click several times on the snma icon many
    SNMAs will be launched, each with a different portname. See
                 ARexx
                    To stop SNMA you have to send a QUIT command to it.  Here's  ←
                        how you
    do that from the shell:

    ->rx "address SNMA QUIT"

    where "SNMA" is the name of the ARexx port.

    How  do you  stop  snma from the  WB?  For example, use a tool like
    ToolManager and create  the  "SNMA OFF"  command which is just like
    the above shell command.

    If  you want  to  assemble files using the icons, I suggest you use
    something like ToolManager, which makes your life a lot easier.
```

```
I added the  WB support  mainly because  it may be helpful to start
SNMA in ARexx mode from an icon.

If you start SNMA  from  the WB and pass it arguments (ie: you have
selected other icons as well),  SNMA will try to open WINDOW=<file>,
if omitted it will open its default output window.  Then the passed
file(s) is(are) assembled  just  like  in  shell  mode.  SNMA won't
check any of the  tooltypes or arguments  and it does  not check if
there's already an snma ARexx  port  where  the  assembly  could be
directed.  (I highly doubt that anyone will use this method, but it
was rather easy to implement so there it is).
```

## 1.16 2.3 Starting the ARexx SNMA host

```
                SNMA can be started as an ARexx  host and I  suggest it be used
that way.

Startup from:

      - Shell use AREXX template

      - WB  use AREXX tooltype

See
               ARexx
               section (4.).
```

## 1.17 3. Features of SNMA

```
                  This  section  covers  all features of SNMA, relating to the ↩
                     actual
  assembly process.
```

3.8
Errors

## 1.18 3.1 Source code format

```
                 The format of the source code is 'standard'.

One line can be 256 bytes long (after macro expansion, too).

One source code line may have the following components:

<Label>   <opcode>  <operands>  <comment>


<Label> Labels must start from the first column.
It may end with a colon (':').

Legal  label  characters  are  'A-Z', 'a-z', '0-9', '_',
'.' or codes 127-255 (like äöåÞÐ).  I decided to handle
all the  characters  in the range 127 to 255  as symbol
characters.

First  character  must  be: 'A-Z', 'a-z', codes 127-255,
'_', '.' After that digits (0-9) are legal too.

Local labels are supported. You have three alternatives
to define a local label:

1)  add a '.' in front of it.   For example: .local
2)  add a '\' in front of it.   For example: \local
3)  add a '$' to the end of it. For example: local$

Local labels  may also start with a digit (actual label
portion). (1$, .1, ...)

<opcode>  Opcode  field  is separated from  the label field by at
least one space.  An Opcode can be:

  1)  MC680x0 operation code (instruction).
  2)  Assembler
             directive
                  3)  Macro invocation

<operands>  Operand field is  separated from the opcode field by at
least one space.  The Operand  field may contain 0 to 9
operands  depending  on  what  is in the  opcode  field.
Operands are separated by a comma (,). There can now be
99 macro operands.

<comment> Anything after the operand field is ignored and treated
as a comment.  Those MC680x0  instructions  which don't
have operands ignore  anything after  the opcode  field.
Anything after a ";"  character is treated as a comment.
If  the  character  in the  first column  is a "*", the
```

entire line is considered a comment or if the opcode is
a * it is ignored.

## 1.19 3.2 Symbols in SNMA

Symbols in SNMA have different meaning, depending on where they are
used.

Absolute symbols
~~~~~~~~~~~~~~~~

Absolute symbols are defined with the
EQU
or
SET
directive.
Symbols may be local symbols in the same way as labels. See the

label
definition. Absolute symbols refer to numerical values.

Example
SNMA
pre-defines
some symbols with the SET directive.

Relative Symbols
~~~~~~~~~~~~~~~~

Relative symbols are labels, or equates which have relative symbols
in the expression which defines it. The only exception to this is
expression Relative-Relative which results in an absolute type. See

expressions
for restrictions on relative symbols.

"*" is a special symbol and is the value of the program counter(PC).

For example:
```
 data    ds.b    100      ; define space 100 bytes
 size    equ     *-data     ; size gets value of 100 (abs type)
```

Register Equates
~~~~~~~~~~~~~~~~

Register equates are defined with the
equr
directive.
~~~~~~~~~~~~~~~~

```
    "Register symbol" refers to the register (Dn or An).


  Register lists
  ~~~~~~~~~~~~~~

    Register lists are only  allowed in movem  and fmovem instructions.
    They are defined with
                 reg
                 directive.
    "Register list" refers to the list of registers.


  Macro symbols
  ~~~~~~~~~~~~~

    Macros are defined with the
                 macro
                 and
                 endm
                 directives.
    "Macro symbol" refers to the defined macro.
```

## 1.20 Pre-defined symbols

```
                SNMA pre-defines the following symbols with SET.

    symbol name       value
    ~~~~~~~~~~~       ~~~~~
      SNMA      0
      snma      0
      NARG      0   (actually number of args in macro call)
      M68000      1
      M68010      2
      M68020      4
      M68030      8
      M68040      16
      M68881      512
      M68882      512
      M68851      2048
      F040      1024


    These symbols are case-sensitive. 'SnMa' is not same as 'SNMA'.
    M68xxx symbols are meant to be used with the
                 cpu
                 directive.

    See also
                 register names
                .
```

## 1.21 Register names

```
SNMA uses the following register names:

D0-D7  A0-A7  SP CCR SR SFC  DFC CACR USP VBR CAAR MSP ISP FP0-FP7
FPCR FPSR  FPIAR TT0 TT1 TC DRP  SRP CRP CAL VAL  SCC AC MMUSR PSR
PCSR BAD0-BAD7 BAC0-BAC7

Registers from the CCR in the above list are special registers. If
the name of special register is  the only component of  an address
mode (like 'lea CAL,a0') snma considers it to be a special register.
If the  name of a special  register is one of the components of an
address more  (like 'lea (CAL,pc),a0')  it is treated  as a normal
symbol.  However, to  avoid any confusion, I strongly suggest that
you use these  names only when  referring to the special registers.
```

## 1.22 3.3 Expressions in SNMA

```
                Expressions can be used almost anywhere where numerical  ←
                   components
are needed.  The only exception being floating point numbers which
don't allow expressions.  Expressions use 32 bit integer math.

Expressions may have: 1) symbols, 2) constants, 3) operators,
    4) parenthesis.

1)
             Symbols
             must be absolute or relative symbols.

2) Constants are numbers. They can be decimal, hexadecimal ($), or
   in binary (%) form.  Sorry, I'm too lazy to do octal conversion,
   but if somebody really needs it, I may add it (never, ever used
   it myself).

3) operators

-   Unary minus
~   bitwise NOT (one's complement)
<< or <   left shift
>> or >   right shift
&   bitwise AND
! or |    bitwise OR
*   multiply
/   divide
+   add
-   subtract


4) Parenthesis are (). They can be nested.
   For example: 3*((12-6)/(2+2))
```

Expressions are either absolute or relative, depending on the types of symbols and operators. Relative symbols are allowed only to add and sub(tract) operands. When an expression is evaluated, it is divided into the sub-expressions to the stage

<number operand number>

where number can be a symbol. This sub-expression gets its own sub-type. Confused? See below.

For example: (Rel2-Rel1)/4 is a legal expression, because the type of Rel2-Rel1 is absolute although they are both relative symbols. See the table below.

Following table shows types of expressions.
A = Absolute , R = Relative, - = not allowed

| Operator | A op A | A op R | R op A | R op R |
|---|---|---|---|---|
| ~~~~~~~ | ~~~~~~~~ operands | | | |
| + | A | R | R | - |
| - | A | - | R | A |
| *, /, &, !, <<, >> | A | - | - | - |

## 1.23 Local symbol example

The following example demonstrates local and global symbols.

```
; start-----------------------------------------
num equ 123   ; define global symbol
start:
.num  equ 10    ; define local symbol
  move.l  #.num,d0  ; move 10 to d0
  move.l  #num,d1   ; move 123 to d1
new:
.num  equ 23    ; define local symbol
  move.l  #.num,d0  ; move 23 to d0
  move.l  #num,d1   ; move 123 to d1
  rts
  end
; end-------------------------------------------
```

## 1.24 3.4 Address modes

SNMA supports all the addressing modes of the 680x0.

SNMA supports Motorola's new addressing mode format as well as the old format. I don't have any of the Motorala's manuals, but I have seen enough 'new format' sources. The change is more cosmetic, I

```
think.  ( Aku(a0) in the new format is (Aku,a0)).


Forcing the size
~~~~~~~~~~~~~~~~
In some  places, you can force  a value to  be either word or long
word.  This is generally used to force something to word size, but
it can be  used to force  something to  long as  well.  Forcing is
implemented by adding the .w (for word) suffix or the .l (for long
word) suffix to the symbol.  For example, 'move.l  (4.w),a0'.

Note1: The  above  addressing  mode is optimized when using simple
 'move.l (4),a0'. 'move.l (4.l),a0' will not allow optimizing
 due to the forcing suffix.

Note2: SNMA won't complain about the mode (4).w, but it does ignore
 the forcing suffix.  The above mode is discouraged anyway if
 you take a look at the new Motorola addressing mode syntax.
 At least in SNMA.


Base displacement modes (including Memory Indirect)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

You can force Base  Displacement  and  Outer  Displacement to be a
long word or word with the .l (long) or .w (word) suffix.

For example:    ([BD.w,A0,d1.w],OD.l)  where BD and OD are symbols.


SNMA will  optimize  addressing  modes to  the  best possible.  If
certain  components are omitted a change to a  quicker mode can be
made. When you force some value with a .l or .w, the value must be
within range and even if the actual value could be optimized it is
not because it is forced.

Example:

jsr (BD.w,a6)
  BD  equ 0

  ; Although BD is null, generated addressing mode will be disp16(An)
  ; because we forced BD to be word.

tst.l (BD,a0)
  BD  equ   0

  ; This will generate addressing mode (An)
  ; end of example

The same applies to Outer Displacement, except that addressing mode
is always Memory Indirect if there is OD.

Don't worry if the  above is  confusing.  You don't usually have to
force anything  since SNMA  optimizes to the best mode for you.  If
you want some value to be an exactly specified size (when importing
a value, for example) forcing can be a handy feature.
```

```
No Address Register in addressing mode
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
When an address register is not part of address generation you have
two  ways to  exclude it.  First, just  don't set it (like (d2.l) ).
Then you can always use a Zero Suppressed address register using the
name ZAn (n does not matter) instead of An.  For example,

(10,d3.l)       =       (10,za0,d3.l)
(d1.l)          =       (za0,d1.l)
(10,a2.w)       =       (10,za0,a2.w)   index register is An


You may wonder what the <---> is this Zero suppress stuff.  The book
I used as a reference manual mentioned it and I being a fool, didn't
realize until  now (a long time passed) that the  index register has
suffix information which says, "HEY ! I'm the index register".

Oh well...


Immediate values
~~~~~~~~~~~~~~~~

SNMA checks immediate values so that they are in range, depending on
the size of the  instruction.  Immediate  values can  also be longer
than  32  bits if the instruction  is some fpu/mmu instruction.  FPU
instructions allow real mode definition of a number like -11.13232e-2.
The sizes of FP values are not checked.

Optimizing Address modes
~~~~~~~~~~~~~~~~~~~~~~~~
Address modes are optimized very well in SNMA. Forward optimizing is
also  included.   See  the
             SNMAOPT
              directive  and above for more
information.  I suggest that  you keep addressing mode optimizing on,
especially when  the 68020+ mode is on because of the way SNMA works.
You will definately get the worst addressing mode if the symbols are
not defined when the line is assembled first time.
```

## 1.25 3.5 Directives

```
                The following directives are supported:



             CLRFO
                 Reset _FO

             CNOP
                 Conditional NOP
```

```
CNUL
    Conditional NULL

CPU
    Define CPU ID type

DC
    Define Constant

DCB
    Define Constant Block

DS
    Define Space

ELSEIF
    Conditional assembly toggle

END
    End of source file

ENDC
    End conditional assembly

ENDIF
    Alias for endc

EDNM
    End macro definition

ENDR
    End repeat block

EQU
    Define symbol value

EQUR
    Define register equate

EVEN
    Ensure PC is even

FAIL
    User error

FO
    Frame offset

IDNT
    Set program unit name

IFC
    Assemble if strings equal

IFcc
    Assemble if condition true
```

```
IFD
    Assemble if symbol defined

IFNC
    Assemble if strings not equal

IFND
    Assemble if symbol undefined

INCBIN
    Include Binary

INCDIR
    Include directory list

INCLUDE
    Include source file

LIST
    Turn on listing file generation

MACRO
    Start macro definition

MC680x0
    CPU mode

MEXIT
    Exit from macro

NOLIST
    Turn off listing file generation

OPT
    Changed to SNMAOPT

REG
    Define register list

REPT
    Start repeat block

RS
    Define structure offset

RSRESET
    Reset _RS

RSSET
    Set _RS

SECTION
    Start new section

SET
    Define SET value
```

```
                SETFO
                    Set _FO

                SNMAOPT
                    Define options

                XDEF
                    Export symbol

                XREF
                    Import symbol
```

## 1.26 CLRFO directive

```
                 CLRFO

  Resets the _FO variable to 0.
  See also
                FO
                and
                SETFO
               .
```

## 1.27 CNOP directive

```
    Conditional NOP.  This directive is used to align data arbitrarily.

    CNOP    offset,alignment


    Offset is a value which is added to the alignment.
    Alignment is an alignment boundary value.

    cnop 0,4  aligns pc to the long word boundary.


    cnop 2,8  align pc to the 8 byte boundary plus two bytes.
```

## 1.28 CNUL directive

```
                    CNUL    offset,alignment

    Conditional NULL.  Same as
                CNOP
                but, unlike the NOP directive,
    it pads with null word.
```

## 1.29 CPU directive

```
                CPU     <expression>

Defines what kind of instructions are legal.  This is also used for FPU
and MMU instructions.

Expressions  are simple  numerical values which are set to the internal
SNMA variable  (CPU command is only way  to set it, it is not a symbol).
See
             Pre-defined symbols
            .

M68000
M68010
M68020
M68040
M68851  MMU
M68881  FPU
M68882  FPU   same as M68881
F040  MC68040 floating point only
M030  68030 MMU


Examples:

CPU M68000      enables 68000 instructions
CPU M68000!M68010!M68020  enables instructions of those cpu's.

CPU M68020!M68030!M68881  enables FPU instruction also.


This  directive  can be  used so that  if code is  desired to be run on
M68000 machines you can check that there are no other instructions than
those which the MC68000 recognizes.  All combinations are possible. CPU
-1 enables all instructions to be assembled.  SNMA performs "CPU -1" in
the beginning of the assembly.

See also
             MC680x0
             directives.
```

## 1.30 DC directive

```
                Define Constant

DC.n    <expression> or <single value>

n is size of data.


See
             Data types
```

.

```
dc.b    2+1    reserves one byte and sets 3 to its value.
dc.l    12     reserves long word and sets 12 to its value
dc.s    $FEEBD00D  reserves long word and sets it to $FEEBD00D
dc.d    +12.9292e-2  reserves two long words and sets its value what
   that number is in double format (binary).
```

Expressions are only valid as integers. Expressions that are floating point numbers are not allowed.

If n is not defined the default size is word.

## 1.31 DCB directive

Define Constant Block

```
dcb.n   <abs expression>,<value>
```

Reserves space for the given data type (n). All entries are set to a single value which is given as the second argument. See
Data types
.

## 1.32 DS directive

Define Space

```
DS.n    <abs expression>
```

Defines storage for the given data type (n).

```
ds.b    12    reserves space for 12 bytes
ds.l    3   reserves space for 3 longs (12 bytes)
ds.x    4*4   reserves space for 16 extended type fp number.
  (192 bytes)
```

<abs expression> must be evaluated and it cannot contain relative symbols (or undefined by far).

All reserved space is set to null.
See
Data types
.

## 1.33 ELSEIF directive

```
ELSEIF toggles  conditional assembly.  If  assembly was off it
toggles it on and vice versa.

Warning! This one does NOT WORK as it should, when conditional
assembly nests.  (I should fix that...).
```

## 1.34 End Directive

```
End directive ends assembly. It is not required to be at the end of the
source file.  When the source file ends it also ends assembly. Anything
after the END directive is not assembled.
```

## 1.35 ENDC directive

```
ENDC (alias ENDIF)

Toggles off conditional assembly if nest counts match.
```

## 1.36 ENDM directive

```
ENDM

Ends current macro definition.
```

## 1.37 ENDR directive

```
              ENDR

Ends current repeat block.
See
             REPT
            .
```

## 1.38 EQU directive

```
                EQUate.

symbol EQU <expression>

Sets the value of the symbol to <expression>.  Symbol and  <expression>
are both required.  = is equivalent to equ.
```

See
Expressions
.

Floating point numbers are supported by specifying a size field to the EQU.

See
Data types
.

Only floating point data types are supported in equ if the n suffix is present. (You can’t ’equ.b 5’).

When a symbol is used as part of an fpu instruction simple type checking is done, but only between floating point types. The following code will cause an ERROR:

```
NUM     equ.d   +11.234343
  fmove.s #NUM,fp0
```

To fix this, change NUM to single or fmove to double.

```
NUM     equ.d   +11.234343
  fmove.d #NUM,fp0    declared type same as used type !
```

The following code will not cause an error, because fmove uses the long type which is not floating point type (okay, it would be good to check anyway).

```
 NUM    equ.d   12.23232
  fmove.l #NUM,fp0
```

## 1.39 EQUR directive

EQUR means register equate.
It allows registers to be addressed as symbols.
Register equates must be defined before use.

Example:

```
count   equr    d0
  move.l  #0,count      ; means move.l #0,d0
```

## 1.40 EVEN directive

EVEN aligns PC to be divisible by two if it is not already. It ↩
does the
same thing as cnop 0,2 See
CNOP

.

## 1.41 FAIL directive

FAIL flags the assembler to stop assembling. It is used to flag user errors. It may be used if a macro call won't get enough parameters for example.

## 1.42 FO directive

```
              Label FO.<size> <absexpression>
```

Define frame offset. Useful with the link instruction. This is something like
RS
but decreases the counter _FO and then assigns the value to the symbol (label).

Sizes other than byte are aligned to the word boundary.
See also
CLRFO
and
SETFO
.

It's good practise to keep your stack long word aligned. FO doesn't ensure it (how about some FOLONG directive ?).

Little example:

```
  clrFO     ; reset _FO (0)
long1 fo.l  1   ;=-4
byte1 fo.b  1   ;=-5
word1 fo.w  1   ;=-8

  link  #_FO,a5       ; _FO = -8

    ;-- set all local variables to 0.

  clr.l long1(a5)
  clr.b byte1(a5)
  clr.w word1(a5)

  unlk  a5
  rts
```

The stack looks something like the following after link:

```
  a5
      offset
      ~~~~~~
 SP->  -8  [  ]  word     word1
       -6   [ ]  byte     pad byte
       -5   [ ]  byte     byte1
       -4      [    ]  long     long1
 a5 ->  0      [    ]  long     old a5
```

## 1.43 IDNT directive

```
  IDNT  <name>

  Sets the name of the program unit to <name>.
```

## 1.44 IFC directive

```
                  IFC     'string1','string2'

    If string1 = string2  DO assemble.

    See also
                IFNC
               .
```

## 1.45 IFcc directives

```
                  IFcc  <expression>

    IFcc is a  conditional  assembly control directive.  cc is the condition.
    The expression is tested against the value of zero.

    directive          means
    (condition)
    ~~~~~~~~~~~          ~~~~~
    IFEQ  <expression>   EQual
    IFNE  <expression>   NEqual
    IFGT  <expression>   Greater Than
    IFGE  <expression>   Greater or Equal
    IFLT  <expression>   Lower Than
    IFLE  <expression>   Lower or Equal

    If the condition is true, assembly is continued.  If the condition is
    false  assembly is turned off.  The
                ENDC
                directive ends conditional
    assembly.
```

## 1.46 IFD directive

```
                 IFD <symbol>

Conditional assembly trigger.
IF symbol is defined,  do assembly, else don't.

See also
             IFND
            .
```

## 1.47 IFNC directive

```
                 IFNC     'string1','string2'

If string1 <> string2 DO assembly.

See also
             IFC
            .
```

## 1.48 IFND directive

```
                 IFND <symbol>

Conditional assembly trigger.
IF symbol is not defined,  do assembly, else don't.

See also
             IFD
            .
```

## 1.49 INCBIN directive

```
INCBIN  <file>

Incbin directive  includes the  named file  into the code  in its binary
form. No assembling is done on the file. If you had a file named bin and
it contained following data in hex form (4 bytes long file):

0BAD BEEF

Now..

  IncBin bin

would do same as
```

```
dc.l $0BADBEEF.
```

If the length  of the  file is  not even, an extra null byte is added to the end  of the  data when it  is set to the  produced code.  This makes sure  that the  program counter stays  aligned. This is just the same as the Incbin without this feature (in SNMA this feature is always on):

```
incbin  <file>
even
```

Thus, above ’even’ is done automatically (always) by ’incbin’ and is unnecessary in SNMA.

## 1.50 INCDIR directive

```
INCDIR  <mydir1>[,mydir2,mydir3...]
```

INCDIR  adds directories to the directory list where the INCLUDE files are retrieved from.

INCBIN uses this list, too.

## 1.51 INCLUDE directive

```
INCLUDE <file>
```

Startsx to assemble <file>.  After it has been assembled, snma continues assembling after the INCLUDE directive.

Include files are looked for first in the current directory, and then in in the directory list which can be defined in
command line
or with the

INCDIR
directive.  If an include file is not found during assembly the assembly is terminated immediately (fatal error).

When snma is in ARexx mode it can have global include tables.  If <file> is  already in the  global table it  is skipped  (not  even  loaded into memory). See
ADDGB
.

## 1.52 LIST directive

```
                LIST  turns on listing file  generation.  You can disable ↩
                   portions
    (like includes)  of a listing file to  be generated with the
    LIST and
                 NOLIST
                 directives.  These do not nest.
```

## 1.53 MACRO directive

```
    <Symbol> MACRO


    Starts macro definition. Code inside a macro definition is not assembled
    until the macro is called.  The NARG symbol is set to the number of argu-
    ments passed  to the macro, when  the macro is  called.  Macro names are
    case-insensitive.


    The Macro call may have up to 99 parameters.  A produced macro line must
    fit into 256 bytes, however.  If a parameter is enclosed between "<" and
    ">", it can contain  any characters  (including  commas,  spaces,  tabs)
    except the ">" (which always ends the parameter started with "<") and LF
    ($a).

    Backslash ("\") has  special meaning  in macro definition.  If character
    after it is:

    1)  01-99   insert argument number <number>
    2)  0[0]    insert size field of macro call
    3)  @<label>  produces unique label (like local labels).
    4)  *<function()>   executes special function


    Argument number (1) may be  defined with one or two digits.  If you want
    to produce something like <argument><number> with arguments below 10 you
    must set the leading 0, otherwise it is not required.

;------------------------------------------------------
    simple examples:

BURGER  macro
  dc.b  "\011"
  endm

  BURGER  HAM     ; produces dc.b "HAM1"


Do  macro
  move.\0 \1,\2
  endm

  Do.b  d0,d1     ; produces move.b d0,d1
;------------------------------------------------------
```

```
Special  functions (4)  insert the resulting  string into the macro.  If
there is an error the actual function call is inserted. These 'functions'
do not nest,  BUT you can  define macro arguments in  the argument  of a
function (like \*valof(\1)).


The parenthesis  are always  required  because there must be some way to
tell when the 'function' ends.

\*VALOF(expression)         inserts the  numerical  value of  expression.
    This  expression is only solved during pass1.
    If expression has relative components, optim-
    ization may change  it but the change is NOT
    reflected  here.  Also,  symbols  in the exp-
    ression must already be defined.




\*DATE(format)              inserts the current date string of the system.
    Format char     format type
    ~~~~~~~~~~~     ~~~~~~~~~~~
    d        DOS     (dd-mmm-yy)
    i        INT     (yy-mmm-dd)
    u        USA     (mm-dd-yy)
    c        CND     (dd-mm-yy)

    DATE() uses dos/datestamp.

\*TIME()                    inserts current system time
    TIME() uses dos/datestamp.

\*VAR(name)                 inserts local or global (ENV:) dos variable
    using dos/GetVar().

\*STRLEN(string)            inserts length of string
\*UPPER(string)             converts string to uppercase
\*LOWER(string)             converts string to lowercase




DATE    macro
  dc.b    'Assembling Date: \*DATE(d)',0
  endm
```

## 1.54 mc680x0 directives

```
              The following directives are shortcuts to the cpu directive.

mc68000
mc68010
mc68020
```

```
mc68030
mc68040
mc68881
mc68882

These do the  same as
               CPU
               with a single  argument,  which is one of
the types of cpu. MC68881 (mc68882) enables also 68020 (68030). Using
the CPU directive, you can control arbitrarily the value of CPUID.

mc68040 is bit useless now because I still don't know the 68040
specific instructions.
```

## 1.55 MEXIT directive

```
MEXIT

Exit from macro definition.
When macro is called  it may be useful to  exit from macro expansion
before actual macro ends.

Used usually in conditional macros.
```

## 1.56 NOLIST directive

```
              NOLIST  directive turns off listing file generation.
  See also
               LIST
               directive.
```

## 1.57 SNMAOPT directive

```
                 SNMAOPT flag[,flag,flag...]


  where flags are:

  Q     Quick optimizing.  Move->moveq,  add->addq, sub->subq
    whenever possible.
    Default: On

  P     Absolute long addressing -> program  counter relative
    whenever possible.
    Default: Off

  A     Effective address  optimizing. 0(An)->(An), BD and OD
    optimizing  if  0  or  word.   (BD,An)->dip16(An)  if
    possible.  Optimizes  address modes  as quick as they
    can be.
    Default: On

  B     Auto-force (BD,An) -> disp16(An). You can override by
    disabling this  flag or using .l suffix in the symbol.
    (BD.l,An)  makes  the  addressing  mode  always  long
    irrespective of the B flag.  This  feature is present
    because routines in the run-time libraries can all be
    called by  the (disp16,An) mode.  Displacement can be
    an  xref'd symbol  which is  solved during  link time.
    Because symbol is xref'd it must be treated as 32 bit
    because in  theory it  can be this way.   All library
    calls however can use (disp16,An) mode. In using this
    option you needn't add the .w suffix to all calls.
    Default: On

  S     With this flag symbol hunks are written to the object
    file.  This is handy when using symbolic debuggers.
    Default: Off

  L     Long branches.  Enable  long branches  if there is no
    size field in Bcc instructions.
    Default: Off

  E     Produce executable (instead of producing object code).
    Source  file cannot  have xref statements.  Sorry, no
    DATA+BSS coagulation yet.
    Default: Off

  R     RELOC32SHORT to executables, implemented as 1015.
    Default: Off

  J     'jsr <ea>, rts'  pair  to the  jmp <ea> if rts is not
    referenced.  See
               Opt j
               example.
    Default: Off

  M     Movem to move if only one reg.
    Default: On


  Flags are case-insensitive.

  Example:

  OPT S,P+,b-

  Write  symbol  hunks,  Optimize  Absolute long -> pc-relative,
  Disable Auto-forcing of (BD,An).
```

## 1.58 OPT J example

```
  Little example:
...
jsr SubRoutine
rts
  Can be converted to
...
jmp SubRoutine
  because SubRoutine (usually) ends to rts.

  Be careful with this one - if you pass parameters via
  stack or do  something  else that depends on a return
  address being in stack, DO NOT USE this.

  If  you have enabled this optimizing, you can locally
  disable it by setting a label to the rts.
```

## 1.59 REG directive

```
Symbol  REG <reg-list>

REG  directive  specifies the  register list used by the movem
instruction. List may contain a symbolic register name defined
by equr.  Symbolic  register  lists must be defined before use.
Symbols defined by the REG directive can be used only with the
movem instruction. <reg-list> may be omitted. If <reg-list> is
empty,  the movem  instruction,  which uses empty list, is not
generated.


example:

list    reg     d0-d3/a0-a2/a5
  movem.l list,-(sp)           push registers onto stack
  nop
  movem.l (sp)+,list           pop  registers from stack
```

## 1.60 REPT directive

```
                REPT  <num>
Starts repeat block. <num> specifies how many times repeat
block is repeated.

Example

    rept    100       ; clear 400 bytes
    clr.l   (a0)+
    endr

See
               ENDR
```

.

Don’t define things inside a repeat block, use include or something similar.

## 1.61 RS directive

```
Label RS.<size>  <absexpression>
```

RS directive can be used to define structure offsets. Label is always required (for obvious reasons). Size field is one of the allowed. See
data types
. Expression
must be absolute and defined before use.

Simple example:

```
  rsreset     ; reset _RS

num1  rs.l  1   ; num1 = 0
double  rs.d  1   ; double = 4
byte  rs.b  3   ; byte = 12  (3 bytes)
word  rs.w  1   ; word = 16  (auto-align to word boundary)
```

## 1.62 RSRESET directive

```
RSRESET
```

Reset the value of _RS symbol to 0. Equivalent to
RSSET
0.
See also
RS
.

## 1.63 RSSET directive

```
RSSET <absexpression>
```

Set _RS value. _RS can be set also with the SET directive, but DO NOT ’equ’ it. See also
RS
.

## 1.64 SECTION directive

```
    SECTION <name>[,<type>[,<mem type>]]

    Start a new section.  Name of the section is set to <name>.
    The Type of section is one of the following:
  CODE  CODE_C  CODE_F
  DATA  DATA_C  DATA_F
  BSS BSS_C BSS_F

    _C extension specifies mem type (C=CHIP, F=FAST).

    <mem type>  specifies  the type of  the memory where the hunk
    should  be loaded.   It can be specified with <type> field by
    extension  or with separate third  argument which  is CHIP or
    FAST.
```

## 1.65 SET directive

```
    <Symbol>  SET <number>

    With SET  you  can define a symbol whose value can be changed
    arbitrarily later on. You can 'equ' set symbols but then they
    become absolute symbols which cannot be changed again.
```

## 1.66 SETFO directive

```
                  SETFO <absexpression>

  Sets _FO variable to the <absexpression>.
  See also
               FO
              .
```

## 1.67 XDEF directive

```
  XDEF  <symbol>[,<symbol>[,<symbol>...]]

  Define  external  symbol.  Defines a  symbol value  to be
  visible to other modules. (Export)
```

## 1.68 XREF directive

```
XREF  <symbol>[,<symbol>[,<symbol>...]]

External  reference  to  symbol.   Defines a  label to be
imported from other modules.
```

## 1.69 3.6 Data types

```
  Supported data types.

  All data types are NOT supported by ALL instructions.

  Directives which support:
  DC.n
  DCB.n
  DS.n
  EQU.n (all  integers  are 32  bit values and they are defined
   without a suffix).
  RS.n

  n is one of these.           size
  ~~~~~~~~~~~~~~~~~           ~~~~
  b = byte           ( 1 byte )
  w = word           ( 2 bytes)
  l = long word        ( 4 bytes)
  s = single precision floating point number   ( 4 bytes)
  d = double precision floating point number   ( 8 bytes)
  x = extended precision floating point number (12 bytes)
  p = packed floating point number     (12 bytes)
Used K-factor is 17 with packed type.
```

## 1.70 3.7 Things to Note

```
                Things to note when using SNMA.


             3.7.1 Instructions

             3.7.2 Expressions

             3.7.3 Include files

             3.7.4 Directives

             3.7.5 Misc.
```

## 1.71 3.7.1 Notes about instructions

```
                 Bcc
                 (bsr, bra, beq,...) instructions
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    - If  there  is no  size field (like bra or bsr) SNMA attempts
      to optimize the branch to the shortest possible. If the size
      field  is given (like bsr.w)  SNMA does  not try to optimize
      the  instruction.  When  generating  jump  tables, I  wanted
      branches  to be  'forced' to  word size without altering the
      snmaopt  directive (I don't remember those option chars even
      myself 8).


    If the size field is omitted:

    - LONGRA flag tells SNMA whether to use long or word size.
      LONGBRA=on enables long branches in these cases and
      LONGBRA=off disables longs, forcing SNMA to use word size
      branches.

    - If the  value is an  external reference, SNMA won't optimize
      it.  Instead,  depending  the on  LONGBRA  flag (and the CPU
      mode) it may be set to long or word.

    - Currently, if you  Bcc to another module and SNMA ends up at
      a long branch, things go wrong.  I have to  dig out  what is
      in  those  new hunks - about  which my old Bantam book knows
      nothing  at all.  Until  then be  warned... (a change to jsr
      could  be  made but I'm searching for a more elegant  method,
      like RELRELOC32) SNMA should really warn if this happens.
```

## 1.72 Bcc instruction

```
    Bcc instruction. (Also bsr).

    bhs   = bcc
    blo   = bcs

    Sizes:  .s .b   (byte)
      .w      (word)
      .l      (long)
```

## 1.73 3.7.2 Notes about the expressions

```
    The Check of the data sizes
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~

    SNMA  checks  that specified  data fits  where it is set.  For
```

```
example if  you have the  instruction "move.w #100000,d0" SNMA
tells you that value won’t fit as word.  Positive number range
is  unsigned and negative signed (All internal math is 32 bit).

Remember that when using the NOT (~) operation all 32 bits are
affected, not just parts of it.

Range checking can  be annoying if you are using NOTs with bit
masks  and sizes  other than  long word.  (Hint:  you can mask
values  with AND) Checking  of the ranges by SNMA reveals some
bugs (value  too big to fit in a  byte) which were not noticed
when using a68k, so I think it is a useful feature.
```

## 1.74 3.7.3 Notes about the include files

```
              SNMA assembles V37 includes just fine.

The V40  includes  which I got  from NativeDeveloperKit3.1
have very few problems with SNMA.

Ooops... about  devices/conunit...  <> is a macro argument
encloser (or something  like that) and is now supported by
snma.

NOTE: I haven’t tried all  the possible macros, so I don’t
claim all macros  will work as intended.  If you have some
problems, a
               bug report
               is welcome.

StripC  tool which is in the NDK3.1 won’t  handle the file
exec/macros.i, BTW.  (I just wonder why)
```

## 1.75 3.7.4 Notes about the directives

```
The SNMAOPT directive format is not neccesarily the same as in
other assemblers. The CPU directive works a little differently
than the MACHINE  directive  found in some assemblers.  So, to
avoid any confusion I implemented a CPU directive.
```

## 1.76 3.7.5 Misc. notes

```
Alignement problems
~~~~~~~~~~~~~~~~~~~
SNMA does not produce mis-aligned code.  One thing to mention:

;-----------
s dc.b ’arg’    line 1
```

```
sl = *-s      line 2
  dc.w  0     line 3
;-----------
```

The value of 'sl' will be 4 instead of the intended 3. SNMA 'adjusts' dc.b in this case with one null byte. To prevent this (so that 'sl' will get the right value), add EVEN or CNOP between line2 and line3. (It's good practice to have EVEN or CNOP 0,4 after 'DC.B's anyway).

One pass assembler
~~~~~~~~~~~~~~~~~~

SNMA does not read and parse the source twice. So things like conditional assembly relating to the value of PC are probably going to break under snma. You don't have to declare normal symbols before you use them (with the exception of register equates and lists), snma is smart enough to create its private structures for them, and then solves them in pass2. Sorry if that pass2 is confusing. In snma, it just means that the source is parsed and then snma starts all optimizing after which it writes requested files.

## 1.77 3.8 Error messages of SNMA

Error messages in SNMA are quite self explanatory. An error ↩ may be a "DEAD END" error which means that assembly cannot continue. An object file is NOT produced if there is even one error. If there is an old object file it is deleted so you don't link it by accident. There are a couple of warnings too. Grammar of the error messages is quite awful, by the way, but I hope you get an idea of what caused the error (I hope you understand this guide, too).

The first character of the possible cause of the error is under-scored. This might sometimes be in the wrong spot, so don't wonder if you think the error is somewhere else (it can be). I added this feature later on and there are quite a lot of places I should adjust it so that the error column will be in the right spot always. When I find it is in the wrong place I usually (or sometimes) try to fix the problem, which is not quite a big one.

When the error is in a macro you get two messages in the SHELL interface. The first one tells you where the error is in the macro and the second, where the macro call is which caused the error. The error count is incremented by one, because there is only one error, except in
AREXX
mode where both are counted.

When an error is found in pass 2 (solving undefined symbols...) you get only one error message when it's in a macro. This needs some work but, you can see where the error is quite easily even now (well, not always...8-( )
~~~~~~~~~~~~~~~~~~

```
    If you get an error which says its an internal bug or something
    similar, a
                 bug report
                 is welcome and appreciated.

    You can control the error message mechanism with
                 AREXX/SET
                    command.
```

## 1.78 4. SNMA, ARexx

```
                  The ARexx  interface is implemented as a command host.  The
  PORTNAME keyword in the  template can be used to change the
  basename of the ARexx port. Default basename is SNMA.

  RexxMaster must be running when using snma in ARexx mode.

  SNMA itself cannot execute any ARexx macros, it is a simple
  command host.


  4.1
                 General
                  4.2
                 Commands
                  4.3
                 Examples
                  SNMA's  ARexx  port is the first one I wrote so it may need
  little more polishing, but it works the way I want. Totally
  different question: is that way the right one?... 8^)
```

## 1.79 4.1 general ARexx stuff

```
                    GENERAL AREXX THINGS
    ~~~~~~~~~~~~~~~~~~~~

    I recommend you use snma as an ARexx host.  Calling snma direct
    from the text editor and  using a global symbol table cuts down
    assembly time.  If you have difficulties  in writing  interface
    macros for the text editor you use, you can always  utilize the
    ShellAsm.rexx macro.  The interface is basically the same as in
    snma's shell mode.  That way you  can still  use  global tables.
    Also, snma is always in memory so, no  load time in this method
    either.

    It is possible to display error  messages as in shell mode when
    snma is in ARexx mode.  SNMA writes the messages to the default
    output if you want (you can  always redirect the default output
```

```
    wherever you want, for example to con:).  See
                 AREXX/SET
                    There are lots of possibilities in using snma, be creative! ( ↩
                       As
    always).
```

## 1.80 4.2 ARexx commands of SNMA

```
                 Many commands use RexxVariableInterface to pass information
  back.


  4.2.1
                 ASM
                          Assemble file
  4.2.2
                 CHDIR
                          Change working directory
  4.2.3
                 FREE
                          Free resources of last assembly
  4.2.4
                 GETERR
                          Get the errors
  4.2.5
                 INFO
                          Get info about the last assembly
  4.2.6
                 QUIT
                          Quit SNMA (ARexx)
  4.2.7
                 SET
                          SET attributes
  4.2.8
                 ADDGB
                          Add global include
  4.2.9
                 REMGB
                          Remove global include
  4.2.10
                 SEEGB
                          See global includes
  4.2.11
                 SELFCHECK
                        Check snma's code
```

## 1.81 4.2.1 SNMA,ARexx: ASM command

```
                    Command:
    ~~~~~~~ ASM SOURCEFILE/A,OBJ=O/K,INCLUDE=I/K,HEADER=H/K,
```

```
    LISTING=L/K,EQUATE=E/K

Template:
 ~~~~~~~
All the keywords are the same as in SNMA commandline.
SNMA ARexx commandline does not have PORTNAME, AREXX
or
             SNMAOPT
             flags.  You can  set SNMAOPT flags with
the
             SET
             command.


Results:
~~~~~~~
ASM command triggers SNMA to assemble SOURCEFILE. To
To find  out how  assembly  went, see/use the
             INFO
                command.
```

## 1.82 4.2.2 SNMA, ARexx: CHDIR command

```
Command:  CHDIR DIRNAME/A
~~~~~~~
Template: DIRNAME is the name of the directory
~~~~~~~~
Function:
~~~~~~~~
Changes  the  working  directory of SNMA.   You  can
set the working  directory to be the  same directory
where your source file is with this command.  If you
have the include files  in the same directory as the
prrogram source this becomes quite a helpful command.
If DIRNAME does not exist or is a file, SNMA opens a
Requester to tell you.

Why is this named CHDIR instead of CD ? To avoid any
confusion between this and AmigaDOS CD.
```

## 1.83 4.2.3 SNMA, ARexx: FREE command

```
Command:  FREE
~~~~~~~
Template: none
~~~~~~~~
Function:
~~~~~~~~
Frees  all   resources  opened  by  the  ASM command.
After you execute  this  command  you  will lose all
information  about the  previous assembly, including
```

```
    errors.

    "Resources"  here means  memory.  All  file handling
    (opening/closing) is internal to the ASM command.
```

## 1.84 4.2.4 SNMA, ARexx: GETERR command

```
                    Command:  GETERR  NUMBER/N REMOVE/S WARN/S STEM/K
    ~~~~~~~
    Template:
    ~~~~~~~~
  NUMBER  error/warning  number.  SNMA  keeps  track of  which
    error was requested last.  If the  number is omitted
    the next error is returned. If GETERR tries to fetch
    an  error  which  does  not exist it returns special
    values (see below).


  REMOVE  Toggling  removes error from  the  list.  Errors are
    numbered  such that the  first  one is always number
    one  and  if  you  remove number one, the second one
    becomes number one and so on.

  WARN  Toggling causes GETERR to return WARNINGS instead of
    the ERRORS.

  STEM  variable specifies variable base name.


    Results:
    ~~~~~~~
      <STEM>.LINENUM
      <STEM>.LINETXT
      <STEM>.FILENAME
      <STEM>.ERRTXT
      <STEM>.COLUMN

  <STEM>.LINENUM
    The line number of the error.  If this is NULL there
    is no such error as the one requested.

  <STEM>.LINETXT
    String  which  holds  the source code line where the
    error is.

  <STEM>.FILENAME
    Name of the source code file.

  <STEM>.ERRTXT
    Error description text.

  <STEM>.COLUMN
    Column number where SNMA  thinks  the  error  is. If
    your  source  code  has TABS in  it you  may need to
    change the TAB  value  with
```

```
             SET
             command to get the
right column.
Default TAB is 8 (as AmigaDOS uses).

<STEM>.ERRNUM
  Which error this was. Handy if you want to know when
  using GETERR to fetch next error.  Number 1 is first.

  Function:
  ~~~~~~~~  GetErr is  used  to fetch  errors  SNMA found in the
  source code. Warnings are fetched with GetErr, too.

  Errors  are  stored  in  a list.  The first error is
  first in the list, second is second and so on.   The
  Warnings are in a separate list.

  Exactly how many  errors is in the list can be found
  with the
             INFO
             command from the <STEM>.ERRORS field.
  Warnings are in the <STEM>.WARNINGS field.

  GETERR REMOVE changes  these values if you call INFO
  after  GETERR REMOVE, but  does not change them when
  you call GETERR.

  If  you  call  GETERR with an  illegal  error number
  (an error  which  does  not  exist) you will get the
  following results:

  LINENUM=0
  LINETEXT=' '
  FILENAME=' '
  ERRTXT='No more errors'
  ERRIND=0

  LINENUM=0 is  not  generated  anywhere  else because
  line numbers start at one(1), so it is safe to check
  this field in an ARexx macro if  the  error fetching
  was successful.

  If you  call GETERR to  get the next error and there
  is  no text  error (all  errors are handled) the sit-
  uation is same as above.
```

## 1.85 4.2.5 SNMA, ARexx: INFO command

```
Command:  INFO  <STEM>
~~~~~~~

Template:
~~~~~~~~
<STEM> is the stem variable where values are put.
```

```
    Results:
    ~~~~~~~
    <STEM>.STATUS     ok, warn, error, fail
    <STEM>.LINES      How many lines we assembled
    <STEM>.ERRORS     number of errors
    <STEM>.WARNINGS     number of warnings
    <STEM>.CODE     number of code sections
    <STEM>.DATA         data
    <STEM>.BSS          bss
    <STEM>.CODESIZE     number of bytes in code sections
    <STEM>.DATASIZE            data
    <STEM>.BSSSIZE             bss
    <STEM>.FAILSTR      possible failure string
            (if STATUS="FAIL")


  .STATUS     is one of the following strings:

        OK    = assembly went just fine / nothing
          assembled yet
        WARN  = last assembly resulted warn
        ERROR = last assembly resulted error
        FAIL  = last assembly resulted failure

        if .STATUS is "FAIL" .FAILSTR has the failure
        description.

  .LINES      tells  how many  lines SNMA assembled.  (Include
        files are also counted for this).

  .ERRORS     How  many  errors, if any.  Failures don't count
        here.

  .WARNINGS   How many warnings there were, if any.

  .CODE     Number of CODE sections.

  .DATA  and .BSS are equivalent with .CODE

  .CODESIZE
        The number of bytes section(s) take. This is the
        sum of all CODE sections.

  .DATASIZE and .BSSSIZE are equivalent with .CODESIZE.

  .FAILSTR    Possible failure description.  See .STATUS.
```

## 1.86 4.2.6 SNMA, ARexx: QUIT command

```
    Command:      QUIT
    ~~~~~~~
    Template:     none
    ~~~~~~~~
    Function:     Quits SNMA
```

```
~~~~~~~~
```

## 1.87 4.2.7 SNMA, ARexx: SET command

```
                Command:  SET Q=QUICKOPT/T P=PCOPT/T A=ADDRESSOPT/T
~~~~~~~     B=BASEFORCE/T S=SYMBOL/T L=LONGBRA/T
    TABS/K/N KS=KEEPSOURCE/T OF=OUTFILE/T RE=RXERR/T
    LNM=LSTNOMAC/T LOD=LSTONLYDATA/T
Template:
~~~~~~~~  QUICKOPT    Q flag  move-moveq and so on
PCOPT     P flag  absolute long ->pc-relative
ADDRESS     A flag  address mode optimizing
BASEFORCE   B flag  ensure disp16(an) mode
SYMBOL      S flag  write symbol hunk
LONGBRA     L flag  enable long branches (w/o .l
        suffix)

TABS
      Number  which  specifies your  current TAB
      setting.  SNMA  needs  this  value for the
      GETERR.COLUMN  field. If the value is NULL
      the value  is NULL or negative SNMA simply
      ignores it.
      Default: 8.

KEEPSOURCE
      If off  snma frees all the  source code it
      allocated  in  the   last assembly and you
      cannot  print the  line of  the error, but
      when working from a  text editor  that  is
      not necessary.  When this flag is on, snma
      frees the source when the text assembly is
      started (or via FREE command), keeping the
      source code in memory between the previous
      assembly and the next one.
      one.
      Default: ON.

OUTFILE
      Toggle  to enable/disable normal  snma out-
      put.  With this is OFF snma will not write
      assembly  messages to  the  default output.
      When ON  snma writes to the default output.
      Default: OFF.

RXERR
      Enable/disable  ARexx errors. Normally, in
      ARexx mode this flag is kept  on. If it is
      off  snma will  NOT  generate  ARexx error
      structures. INFO and GETERR commands think
      that  there were no errors if this flag is
      off, even if there are errors.
      Default: ON.

LSTNOMAC    no macro expansions to the listing file
```

```
      Default: OFF.

LSTONLYDATA only those lines which define data.
      Default: OFF.

SHORTRELOC32 Use short reloc32 whenever possible.
      Default: OFF.

KEEPOBJ     Do  not delete object file when  there are
      errors in the source file.
      Default: OFF.

MOVEM     movem optimizing.
      Default: ON.

Function:
~~~~~~~~~
 To  change  default  settings of  SNMA.  The
             SNMAOPT
                 directive overrides  those flags which can be set with
 SNMAOPT.  Some  flags  are  the  same as in  the shell

             template
            .


 The  ARexx SET  command has nothing to do with the SET
 directive.
```

## 1.88 4.2.8 SNMA, ARexx: ADDGB command

```
                Command:  ADDGB
~~~~~~~     SOURCEFILE/A,OBJ=O/K,INCLUDE=I/K,HEADER=H/K,
    LISTING=L/K,EQUATE=E/K

Template:
~~~~~~~~  Template is same as in the
             ASM
             command.

Function:
~~~~~~~~  To add symbols and macros to the global table. A single
ADDGB can add any number of include files to the global
table.

ADDGB works like
             ASM
             with the following exceptions:

1) No code is generated
2) Second pass is not executed
3) Absolute  symbols  and  macros are transfered to the
   global table at the end of assembly.
4) Include files are added to the internal include file
```

list to prevent them from being included again.

Equates, sets, floating point equates and macros are transferred to the global table.

The idea of the global table is to lower include file loading and processing time. You need do this process only once after which SNMA will find the symbol (or macro) from the global symbol table if it is there. If you try to
INCLUDE
a file which has been ADDFB'd the include file will be skipped. If you have includes in your source the include files which are ADDGB'd are not processed.

SNMA locks the files you add to the global table. This prevents you from modifying the (still readable) files, as the change doesn't modify the global table anyway. The
REMGB
command can be used to remove a file from the global table.

Important! The main sourcefile symbols and macros are NOT transferred to the global table. A file you handle with this command usually contains some INCLUDE directives which SNMA processes. This enables you to process your current sourcefiles with ADDGB.

Errors are reported just like in the ASM command.

If you have includes which produce code and you want that code to be included DO NOT use the ADDGB command to include that file.

The Global table can be used only in ARexx mode.

So, what does all that mean in practice? There is a little example in snma/examples. Look at it.

The current implementation of include file skipping tries to lock the file, even if it is in the global table and if it doesn't succeed it fails. For the hard disk users there is nothing to worry about, but the floppy users may find this frustrating. In theory, you should be able to add the include file to the global table from the floppy and then remove the actual disk where they were loaded, but as snma tries to lock these files it can't find them. Any floppy users out there? It is not a big task to change the include handling to allow the above situation. It is big enough to avoid if nobody really needs it though. (Hint: Invest in a hard disk. Yeah, I know, THAT requires money 8)

## 1.89 4.2.9 SNMA, ARexx: REMGB command

```
                Command:  REMGB FILENAME
~~~~~~~

Template:
~~~~~~~~~  FILENAME       Name of  include file you want to remove.
         This removes  only symbols  added in the
         main include file.  If this  include has
         included  other  include files they will
         not be removed. If the FILENAME has been
         omitted, the global table is cleared and
         the symbols freed.

Function:
~~~~~~~~~  To remove symbols and macros for one file (or all) that
have been  included in the global  tables.  You can see
what  files are in the global  tables  with the
             SEEGB
                command.

I have the following alias defined in my shell-startup:

alias remgb "rx 'address SNMA remgb "[]"'

Then just typing remgb <name> will remove the file from
the global table if present there.

Symbols  which  have  been  declared  by using  the SET
directive  belong  to  the  first  file  which  defined
them.  When you remove that file, the symbol is removed
from the global table even if  some other include  file
has  changed it (i.e.  declared by using SET).
```

## 1.90 4.2.10 SNMA, ARexx: SEEGB command

```
Command:  SEEGB STEM

Template:
~~~~~~~~~  STEM     stem variable name.
     STEM.COUNT will hold the number of filenames
     STEM.0     filename 0
     STEM.1     filename 1
     STEM.n     filename n


Function:
~~~~~~~~~  To  get information  as to what files are in the global
symbol table. If you don't specify a STEM variable this
command is a no-op.
```

## 1.91 4.2.11 SNMA, ARexx: SELFCHECK command

```
Command:  SELFCHECK
~~~~~~~

Function:
~~~~~~~~  Calculates a new checksum from the snma's  own code and
compares it with that calculated in startup. If there's
a difference snma will report it.  Programs tend to run
away every now and then  when  developing.  Using  this
command you  can make sure snma is still in good health.
(Yes, sometimes memory protection would be nice).
```

## 1.92 4.3 ARexx examples

```
The ARexx directory of the snma  package has  many ARexx macros
(programs).  You may use them as they are or modify them anyway
you like.


ShellAsm.rexx is a simple macro to call from the shell and it
    does basically the  same as SNMA  in shell mode.
    This one  looks a little better than that found
    in the V1.70 release.

addgb.rexx  adds symbols to the global table. Basically the
    same as ShellAsm.rexx.

seegb.rexx  displays  what  include  files  are in the snma
    global symbol table.


Other macros are for the XDME text editor.  Those who have some
other text editor (which supports ARexx) may found them helpful
as an example in writing their own interface macros.

asm.xdme      assembles file
GetErr.xdme     fetches error information
snmainfo.xdme     displays information about last assembly

These  macros  need a little more work but they do well for now.
The GetErr macro also needs some work with 'user macro' errors.


Feel  free to modify these  macros  anyway  you please or write
completely new ones.  My taste is not neccessarily yours.

If  you  modify these files  and distribute them, please change
the names and state what they are supposed to do clearly enough.
```

## 1.93 5. Author

```
The author lives in Helsinki, Finland.

If  you are using snma, by sending mail or a postcard  you will
motivate  him to work even  harder with snma.  By stating which
Amiga model you use the author knows  that snma has no problems
(has it? It shouldn't) with different Amiga models.  By stating
where you  got the snma package, you will be giving interesting
information too. (Paranoid? Who? Me? Why? 8')

Send your comments, flames or whatever to

E-mail: snuojua@cc.helsinki.fi

or

smail:  Samu Nuojua
  Harustie 8 B 15
  00980 Helsinki
  FINLAND



  Well, although snma is freeware, donations are always welcome.


  Finally, thanks to Satu for the patience.
```