snma.hyper

COLLABORATORS

	TITLE : snma.hyper		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		July 1, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

snma.hyper

1.1 Samu Nuojua's Macro Assembler, SNMA v2.04

```
              ****************************************************
          *                                                  *
          *             S   N   M   A    V2.04               *
          *             ~~~~~~~~~~~~~~~~~~~~~~               *
          ****************************************************

     SNMA is 680x0 conditional macro assembler.

1.

              Introduction
                 General issues

2.

              Usage
                 Startup, commandline.

3.

              Features
                 Expressions, directives etc.

4.

              ARexx
                 ARexx interface

5.

              SnmaOpts
                 SnmaOpts option setter.

6.

              Author
                 Address.
```

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. (Look at
          things to note
          )
```

1.1

```
          Copyright and disclaimer
            1.2
          What you need
            1.3
          How to Install
            1.4
          Good & Bad things
            1.5
          History
            1.6
          Bug reports
            1.7
          Misc.
            1.8
          Thanks
```

1.3   1.1 Copyright ©

```
SNMA stands for Samu Nuojua's Macro Assembler.

SNMA is © copyright 1993-1995 by Samu Nuojua.

All SNMA documents are © copyright 1993-1995 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.   note
            ~~   ~~~~~
  - dos.library   37
  - intuition.library   36
  - utility.library   36
  - icon.library   any   *   WB support
  - mathieeedoubbas.library any   *   fp support
  - mathieeedoubtrans.library any   *   fp support
  - rexxsyslib.library   36   *   ARexx support

  Libraries marked with a * are not neccessarily required. The 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 (free ones also) and
snma should work with the ones which can deal with standard hunks.

   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  Rexx:  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 (good for You).
-  Most   common
                directives
                are supported.
-  Macros are supported.
-  All 680x0, 6888x, 68851 and 030/040 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
-  ARexx port

   Bad:
-  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).
-  Some sort of beginner's help would be good (sources...).
-  Only output format is Amiga object code or executable.
-  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.

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 in 2.x systems. It is faster in
  3.x systems (where SetVBuf() works).

  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.

Snma now reports its memory usage (excluding loading the program
itself, and all indirect allocations – like opening file).

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 written SNMA entirely in assembler – first I wrote most
of it with a68k and then minor parts with snma itself. Now I'm
using snma only – it is good enough for that.

SNMA operates pretty stable on my system (A2000, GVP A3001 28Mhz
68030/882, 4M fast 1Mchip, OS3.1). 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 Mahony   for sending me source code, testing and suggestions.

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 use(d)
while developing SNMA:

– Charlie Gibbs, A68k
– Matt Dillon, DME and DLink
– Aaron Digulla, XDME
– Steffan Becker, ToolManager
– Jorrit Tyberghein, PowerVisor.
– SAS/C for their excellent compiler (SnmaOpts).

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
  LB=LONGBRA AREXX/S PORTNAME/K QUIET/S PAGELEN/K/N LNM=LSTNOMAC/S
  LOD=LSTONLYDATA/S EXEOBJ/S SR=SHORTRELOC32 KEEPOBJ/S MOVEM/T
  EC=ERRCOLOR/K/N DEBUG/S OL=OPTLEVEL/K/N TD=TYPESDIR/T
  CS=CASESEN/T ISC=IMMSIZECHECK/T CPU/K/N OCT/T OF=OPTFILE

Where:

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

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

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

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

EQUATE/K   defines the equate file name to be created.   An empty
E   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/K   Listing file to be created. Minimal formatting may be
    L   done by using the PAGELEN option.
        alias: -l[<name>]

    QUICKOPT/T   Quick optimizing flag,   default: on
    Q

    PCOPT/T   pc-relative optimizing,   default: off
    P

    ADDRESSOPT/T   effective address optimizing   default: on
    A

    BASEFORCE/T   Auto-force (Bd,An)->(disp16,An) default: on
    B

    SYMBOL/S   Write symbol data hunk   default: off
    S

    LONGBRA/T   Long branches (wo/size field)   default: off
    LB

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

    PORTNAME/K   The name of the Arexx port snma will use.
        Default: SNMA.

    QUIET/S   Disables informational output.

    PAGELEN/K/N   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/S   Don't include macro expansions in the listing file.
    LNM   Default: off.

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

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

    SHORTRELOC32/S   write HUNK_RELOC32SHORT instead of HUNK_RELOC32 when-
    SR   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/S        Don't delete object file if there are errors in source
             file.
             Default: off.

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

    ERRCOLOR/K/N     ANSI color code for the error highlightning.
    EC               Default: none.

    DEBUG/S          Include LINE DEBUG hunks to all sections. See also

                     debug
                     directive and
                     snmaopt
                     d.
             Default: off.

    OPTLEVEL/K/N     Number of maximum optimizing passes.  One optimization
    OL               may make more optimization possible.  You  can  define
             the  maximum  number of  levels  snma will take.  SNMA
             will stop optimizing  when  there  is  nothing more to
             optimize.
             Default: 1.

    TYPESDIR/T       Toggle some directives which override some
                     types.i macros.
                     TD               Default: off.

    CASESEN/T        Toggle to use case-sensetine (ON) or -insesensitive (OFF)
    CS               symbols.
             Default: on.

    OPTFILE/K        Name of the file where to read options.
    OF               See
                     option file format
                     .

    IMMSIZECHECK/T   Toggle immediate data size checking on/off.
    ISC              Default: on.

    CPU/K/N          Initial CPUID value. See
                     CPU
                     directive.
             Default: -1

    OCT/T            Toggle to allow @ character to be used as octal number
             specifier. This used to be hardcoded, but since SAS/C
             uses it in __regargs function names, this has been changed.
             To allow old behaviour, set this flag on.
             Default: Off.


    See
```

```
            Options

                .
```

SNMA does not check the stack. I haven't had problems with a stack
size of 4000 bytes. You can overflow the   stack with very nested
includes. One level 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:). You can specify some options this
way in snma, too.

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:'.

To use style old flags, just specify -<flag> in the commandline.

For example:
 ->snma source.asm -l -e

  Request listing and equate file.

->snma source.asm -iincdir: -hhdr.i

  Set include directory as "incdir:" and read hdr.i as header file.

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 2.1.3 Option file format

```
                 Option files is simple text file where different options are in
  separate lines. Snma reads this file always when assembly is started,
  in ARexx mode, too.

  If the "OPTFILE <name>" keyword is specified snma tries to handle
  that file, else snma does the following:

      1) If ENV:SNMAOPTIONS exists it reads and handles it.
      2) Then if SNMAOPTIONS file is found from the same directory
         where the source file is, it is handled.

  Option file is intended to be used with
              snmaopts
               options setter
  program which provides simple GUI to create option files.
  (Heh, hard to guess where I got the inspiration for this :).

  Option file format:

  <This may very well change in future>

  QUICKOPT    = <ON | OFF>                 opt modes
  PCOPT       = <ON | OFF>
  ADDRESSOPT  = <ON | OFF>
  BASEFORCE   = <ON | OFF>
  MOVEM       = <ON | OFF>
  LBRA        = <ON | OFF>
  OPTLEVEL    = <number>
  SYMBOL      = <ON | OFF>                 create symbol hunks
```

```
LINED      = <ON | OFF>              Create LINE DEBUG hunks
EXEOBJ     = <ON | OFF>              Create executable
SR32       = <ON | OFF>              ShortReloc32
KO         = <ON | OFF>              Keep Object
EQUF       = <ON | OFF>              create equate file
LISTF      = <ON | OFF>              create listing file
LNM        = <ON | OFF>              no macros to listing file
LOD        = <ON | OFF>              Only lines which define data
CS         = <ON | OFF>              Case-sensetive symbols
TD         = <ON | OFF>              TypesDirectives
QUIET      = <ON | OFF>              Be quiet
EC         = <number>               Error Color
CPU        = <number>               Initial CPU value
-h         = <filename>             Header file
-i         = <directory>            Incdir
```

1.16   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.
    Trying to run snma twice with the same ARexx port name will now ask if
    you want to quit snma (ARexx host). If not, nothing happens.
    If you want to start multiple SNMAs, define different port name to
    each instance. This is easiest to do with creating project icons (do
    not create actual file - only the .info file) and specifying neccessary
    tooltypes in them (in this case AREXX and PORTNAME=<>). Set default
    tool to snma.

    See also 4. 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?

Simple, click the icon you started snma from and ask "Quit" to the
requester which appears.

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, including
snmaoptions file handling.

1.17   2.3 Starting the ARexx SNMA host

                SNMA can be started as an ARexx host and I suggest it be used
that way. Of course, lots of people are not used that and stick in
more traditional way – no problem.

Startup from:

     – Shell   use AREXX template

     – WB   use AREXX tooltype

  See

              ARexx
              section (4.).

1.18   3. Features of SNMA

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

  3.1
              Source code format
               3.2
              Symbols
               3.3
              Expressions
               3.4

1.19   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 backslash in front of it. \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.

1.20   3.2 Symbols in SNMA

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

Absolute, relative, register and register list symbols are
case-sensetive by default, but you can chnage this behaviour with
"snmaopt c-" or with CASESEN=OFF commandline toggle.

    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.

              Local symbol 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.
    Names of macros are always case-insensitive.

1.21 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
```

M68xxx symbols are meant to be used with the
          cpu
          directive.

See also
              register names
            .

1.22 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 DTT0 DTT1 ITT0 ITT1 URP IC DC IC/DC DC/IC
BC

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.

1.23 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 ($), octal
   (@) or in binary (%) form. First character tells which type to use.
   To use octal numbers, OCT flag must be set on in commandline options.

```
   3) operators (in the order of precedence)                    precedence

-               Unary minus                              9
~               bitwise NOT (one's complement)           9
<<              left shift                               6
>>              right shift                              6
^^              power                                   6
&               bitwise AND                             5
! or |          bitwise OR                              5
*               multiply                                4
/               divide                                  4
//              modulo                                  4
+               add                                     3
-               subtract                                3

boolean operators (TRUE <> 0, false = 0)                2
=, ==           equal to
!=, <>          not equal to
<               less than
<=, =<          less than or equal to
>               greater than
>=, =>          greater than or equal to
```

```
   4) Parenthesis are (). They may 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 rela-
 tive symbols. See the table below.

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

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

1.24 Local symbol example

   The following example demonstrates local and global symbols.
   (Global means global to one source file here).

```
; 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------------------------------------------
```

1.25 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. (Mode is probably optimized to word size
    later). In some other addressing mode, like (Aku.w,An,Xn)
    the .w suffix is inside the parenthesis. The mode (Aku.w) is
    just stripped down version from that mode and I think it is
    better to use consistent system to refer things.

   Currently you can't force to byte like "num.b(a0,d0.w)" but you can
   do it by disabling 020+ code generation by using mc68000 directive.
   This is a little ugly, I know. Maybe I'll add a ".b" suffix in some
   day.

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
(err.. may have – that .l or .w – it is good practise to use them)
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.

1.26 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

DEBUG
    Write line debug hunk

DS
    Define Space

DX
    Define BSS space
```

```
ELSEIF
    Conditional assembly toggle

END
    End of source file

ENDC
    End conditional assembly

ENDIF
    Alias for endc

ENDM
    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

SMALLDATA
   Enable small data for section

XDEF
    Export symbol

XREF
    Import symbol
```

```
                       types.i
                          types.i macros as directives
```

1.27   3.5.1 types.i directives

SNMA has most of the exec/types.i macros defined as directives.
Although this doesn't save very much assembling time it reduces
memory usage because snma doesn't need to create macro lines.

You can enable/disable this feature with 'snmaopt T' flag or
with a 'typesdir' toggle in a startup template. By default this
feature is disabled.

As you can see, you probably need to include types.i because
there is only structure definition stuff at the moment. (This
does not take into consideration what kind of modifications
people have made to the their types.i file)

Currently , the following exec/types.i macros are supported:

```
 aptr             <name>[,how many]    (32 bits)
 byte             <name>[,how many]    (8 bits)
 bool             <name>[,how many]    (16 bits)
 cptr             <name>[,how many]    (32 bits)
 double           <name>[,how many]    (64 bits)
 float            <name>[,how many]    (32 bits)
 fptr             <name>[,how many]    (32 bits)
 label            <name>
 long             <name>[,how many]    (32 bits)
 rptr             <name>[,how many]    (16 bits)
 struct           <name>,<how many>
 structure        <name>[,offset]      (offset=0 if omitted)
 short            <name>[,how many]    (16 bits)
 uword            <name>[,how many]    (16 bits)
 ubyte            <name>[,how many]    (8 bits)
 ulong            <name>[,how many]    (32 bits)
 ushort           <name>[,how many]    (16 bits)
 word             <name>[,how many]    (16 bits)
```

[,how many] field is extension to the exec/types.i macros.
'Structure' doesn't require initial offset (basic macro does).

Notice that there are no data type checks so most of these directives
do same thing (the ones which have same size).

Example:
```
    structure    MyDataStruct
  aptr         ptr                ;ptr  = 0
  byte         col1               ;col1 = 4
  byte         col2               ;col2 = 5
  word         mask               ;mask = 6
```

```
aptr          planes,8          ;planes= 8
long          count             ;count = 8+8*4 = 40
label         sizeof            ;sizeof= 44
```

1.28 CLRFO directive

```
          CLRFO
```

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

1.29 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.

1.30 CNUL directive

```
              CNUL    offset,alignment
```

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

1.31 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
M040        68040 MMU
```

If you want to use CPU keyword, the value for it must be calculated
'by-hand'. See the values from
            Pre-defined symbols
                and bitwise OR them together, or use SnmaOpts and option file ←
                 (a lot
easier).

The documentation for CPU command may have been little mis-leading.
When one cpu type is allowed, instructions and address modes for that
type are enabled. It does not mean that other instructions are
disabled. So if you have both M68000 and M68020 set, snma can generate
020 opcodes and address modes as well as M68000 ones. Snma starts in
'swallow all' mode, allowing all instructions and address modes to be
generated (if you don't specify cpu in options).

Pay attention to the '040s MMU commands. There are some overlaps with
'030: same instruction name must produce different code for '030 and
'040. For example "PFLUSHA" is different in '030 and in '040. If M040
is set, '040 MMU opcodes are produced, no matter of M030 flag.
So, by default, snma generates 040 MMU commands.

Examples:

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

CPU M68020!M68030!M68881    enables FPU instructions also.


CPU M68040!F040             '040 without floating point supporting
    routines, ie. only those fpu instructions
    which are in silicon.
```

```
CPU M68040!F040!M68882        '040 floating point software installed.
     (which emulates missing silicon)
```

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.

1.32 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.

1.33 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
               .
If value is omitted, snma fills area with nulls.

1.34   DEBUG directive

DEBUG

Writes LINE debug hunk(s) to to the output file. Only affects the
section where this directive is specified. If DEBUG commandline switch
is used, LINE debug hunk is written to the all sections. "SNMAOPT D"
can be used to switch snma to write the full path of the source name,
which may be required with some debuggers.

This feature allows source level debugging to be used with debuggers
which support that.

1.35   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
             .

1.36   DX directive

DX.<size>   defines BSS space to be allocated after code or data
    hunk. This directive only takes affect when producing
    an executable with snma. DX is like DS except that the
    size is reserved only from the hunk header. When the
    AmigaOS loads the executable it allocates memory as is
    specified in the size field of the hunk header. This
    size can be larger than the actual hunk size. Before
    2.0 this memory range was not cleared and it must be
    done (if desired) by startup code. After 2.0 AmigaOS

```
clears this area.

By using this method you can reduce the size of the
executable and make more effecient BSS data references
using combined DATA/BSS base register.
```

1.37 ELSEIF directive

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

1.38 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.
```

1.39 ENDC directive

```
ENDC (alias ENDIF)

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

1.40 ENDM directive

```
ENDM

Ends current macro definition.
```

1.41 ENDR directive

```
               ENDR

Ends current repeat block.
See
          REPT
            .
```

1.42 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
```

1.43 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
```

1.44   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
                    .

1.45   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.

1.46   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
```

1.47   IDNT directive

```
 IDNT     <name>
```

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

1.48   IFC directive

```
                 IFC     'string1','string2'
```

If string1 = string2 DO assemble.

See also
             IFNC
           .

1.49   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          <expression>   If true (not equal)
```

```
IFT          <expression>    If true (not equal)
IFF          <expression>    if false (equal)
IFNT         <expression>    if not true - false (equal)
```

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

1.50   IFD directive

```
                IFD <symbol>
```

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

See also
                IFND
                .

1.51   IFNC directive

```
                IFNC     'string1','string2'
```

If string1 <> string2 DO assembly.

See also
                IFC
                .

1.52   IFND directive

```
                IFND <symbol>
```

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

See also
                IFD
                .

1.53   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.

1.54 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.

1.55 INCLUDE directive

```
              INCLUDE <file>
```

  Starts 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
              .

Addition to include file handling
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Snma can now handle includes which I have decided to call "symbol tables"
shortly STab files. STab files are include files which store the
information organized so that snma can handle them faster than normal
include files.

Important !
STab files are just symbol and macro tables, there are no assembly
controlling information (directives) nor any instructions, like normal
include files can have. Usually, include files just define things so
there is no problem here, but this is good thing to keep in mind.

Where to get these STab files ?

You have to use snma to create them. It is pretty easy if you follow the
next example.

  1) Start snma in ARexx mode (if not already running).

  ->run snma arexx

  2) Create file which includes all the files you want to include
to the STab file.

  ->list <incdir> files all lformat=" include %p%n" > <include file>

where <incdir> is the directory you want to include the files.
     <include file> is the produced include file.

  3) Add the include files which the file produced in step 1 will include
     to the snma's global symbol table. I assume that you have copied
     "addgb.rexx" ARexx macro to your rexx: directory.

  ->rx addgb <include file>

  4) Produce the STab file.

  ->rx "address snma makest <stabfile.st>"

    where <stabfile.st> is the name of the STab file you can include much
    faster later on.

  Quick repeat:

Assume that you have system includes in the directory called
"incdir:" and you want to create file named "systab.st" , which
contains all the stuff include files contains in this directory. "->"
represents shell prompt.

```
1) ->run snma arexx
2) ->list incdir: files all lformat=" include %p%n" > ia.asm
3) ->rx addgb ia.asm -iincdir:
4) ->rx "address snma makest systab.st"
```

3.1 Include files - without comments - take about 441 kB of space,
while the systab.st file - produced from the above - takes 219kB.

Shortly, add the include files to the snma's global symbol table and
then execute snma's ARexx command MAKEST which will write the STab file
taking the symbol and macro definitions from the snma's global symbol
table.

You can, of course, call these files whatever you want and set the file
extension to whatever pleases you.

1.56   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.

1.57   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)
        v $VER   (dd.mm.yy)
```

```
      DATE() uses dos/datestamp.
      v format uses CND format, and replaces "-" with
      ".".

 \*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
```

1.58 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.
```

1.59 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.
```

1.60 NOLIST directive

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

1.61 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

D        Write  the  full  path of the source file to the line
   debug hunks.  See also DEBUG.
   Default: Off.

T        SNMA has some
              exec/types.i macros
              defined as directives.
   Default: off

I        Immediate data field siz checks. Checks if data fits to given field.
   Default:on

Flags are case-insensitive.

Example:

SNMAOPT S,P+,b-

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

1.62 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 or by using snmaopt directive.
```

1.63   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
```


1.64   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.

1.65   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)
```

1.66   RSRESET directive

```
                 RSRESET
```

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

1.67   RSSET directive

```
                 RSSET    <absexpression>
```

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

1.68   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.

1.69   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.

1.70   SETFO directive

```
             SETFO <absexpression>
```

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

1.71   SMALLDATA directive

```
SMALLDATA [<An>]
```

Enables small data model for the current section. If the
section has any base relative address modes which have An
as a base register, snma handles them as being references
to data area – ie. using HUNK_DREL or EXT_DREF hunks.

If register An is omitted , the default is a4.

To use small data model, you need linker a which can deal
with HUNK_DREL and EXT_DREF hunks. Please refer to your
linker documentation for more information on how to use
the small data model. There are certain rules as to how to
name sections which define data and so on.

1.72   XDEF directive

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

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

1.73   XREF directive

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

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

1.74 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.
```


1.75 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.

1.76 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.

1.77   Bcc instruction

```
Bcc instruction. (Also bsr).

bhs   = bcc
blo   = bcs

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

1.78   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, which can
be turned off, if desired.

1.79 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.

1.80 3.7.4 Notes about the directives

The OPT directive is changed to "SNMAOPT" to avoid any con ←
              -
fusion with other assemblers which may have implemented the
OPT directive differently.

The CPU directive works a little differently than the MACHINE
directive found in some assemblers. So, to avoid any confu-
sion I implemented a
                CPU
                directive.

1.81 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.

1.82 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.

1.83 3.9 Known bugs

Yes, there are known bugs.

   -    Outer displacement can't be relative (label).

   -    Power operator associates from left to right.

   -    Branch optimizing will sometimes optimize branch (Bcc) although
it shouldn't. This will happen when you have right combination
of cnops and/or byte strings between branch. I'm not sure what this
right combination is and therefore unable (at the moment) to fix
the bug. So if snma complains something like "Must be 8-bit signed
value" you have to add the .w extension (Bcc.w) to the branch
instruction.

Altough the cases when this happens are rare it is still annoying
bug. Sorry about that.

   -    Hmmm... probably something else, too.

1.84 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^)
```

1.85   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).
```

1.86   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
4.2.12
                MAKEST
                        Make symbol table file
```

1.87 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.
```

1.88   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.
```

1.89   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.
```

1.90   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.

1.91 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          number of data sections
<STEM>.BSS           number of bss sections
<STEM>.CODESIZE      number of bytes in code sections
<STEM>.DATASIZE      number of bytes in date sections
<STEM>.BSSSIZE       number of bytes in bss sections
<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.
```

1.92 4.2.6 SNMA, ARexx: QUIT command

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

1.93 4.2.7 SNMA, ARexx: SET command

```
                Command:    SET Q=QUICKOPT/T P=PCOPT/T A=ADDRESSOPT/T
~~~~~~~         B=BASEFORCE/T S=SYMBOL/T LB=LONGBRA/T
    TABS/K/N KS=KEEPSOURCE/T OF=OUTFILE/T RE=RXERR/T
    LNM=LSTNOMAC/T LOD=LSTONLYDATA/T SR=SHORTRELOC32/T
    KO=KEEPOBJ/T MOVEM/T EC=ERRCOLOR/K/N DEBUG/T
    OL=OPTLEVEL/K/N TD=TYPESDIR/T CS=CASESEN/T CPU/K/N
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
            or negative SNMA simply ignores it.
            Default: 8.

    KEEPSOURCE
    KS         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 next 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
    OF         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
    RE         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
 LNM         Default: OFF.

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

 SHORTRELOC32 Use short reloc32 whenever possible.
 SR          Default: OFF.

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

 MOVEM       movem optimizing.
            Default: ON.

 ERRCOLOR    Error highlighting.
 EC          Default: none.

 DEBUG       Toggle to enable debug line hunks
            Default: OFF.

 OPTLEVEL    How many times to run optimizer.
```

```
    OL          Default: 1.


    TYPESDIR    Some macros implemented as directives.
    TS          Default: OFF.

    CASESEN     Case-sensetive symbols.
    CS          Default: ON.

    CPU         Initial CPUID value.
        Default: -1.
    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
                    .


    Also, if snma is able to find option file, the values
    in it override the ones set by "SET" ARexx command.
    Option file doesn't have all options which "SET" ARexx
    command has.

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

1.94   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 ADDGB'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)

1.95   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).
```

1.96   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.
```

1.97   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).
```

1.98   4.2.12 SNMA, ARexx: MAKEST command

```
              Command:    MAKEST
~~~~~~~
```

```
Function:
~~~~~~~~     Produces symbol table file from the symbols and macros
which are in the snma's global symbol table. This is the
other way (another is to use global symbol tables) - but
not mutually exclusive - to reduce include file handling
time.
```

```
The include file produced is in a format which is much
quicker to handle, but it lacks all the assembly directives
and possible instructions.
```

```
You can, for example, create one file which will contain
all the symbols and macros from the system include files,
which can be loaded faster.
```

```
See
          INCLUDE
          directive documention.
```

```
I added this command because I realised that not at all
people are using ARexx interface and I found even by myself
that loading symbol tables to memory in every assembly time
have its strong points, especially when developing software
which is full of bugs. So, now it can be done much quicker.
```

1.99   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.
```

1.100   5. SnmaOpts

```
              SnmaOpts is a program for setting snma's option files.

   5.1
              Shell startup
              5.2
              Workbench startup
              5.3
              Menus
              5.4
              Gadgets
```

1.101   5.1 SnmaOpts - Shell startup

```
   FORMAT       snmaopts   [OPTFILE=<file>]
        [PUBSCREEN=<screen>]
        [WINX=<x>] [WINY=<y>]
        [FONT=<font>] [FONTY=<fontsize>]
        [QUIET]
```

```
<file>      Name of option file. Specifying this, SnmaOpts
       reads only this file.

<screen>    Name of public screen to open window.
       Default: default public screen.

<x> <y>     Window's place in screen.
       Default: Visible portion of screen.

<font>      Name of the font to use in window. Use .font
       extension.
       Default: Screen's font, fallback to system's
       monospaced font or topaz 8, if window doesn't
       fit to visible portion of screen.

<fontsize>  Y size of the <font>.
       Default: 8.

QUIET       Don't give error messages.


TEMPLATE    OF=OPTFILE/K,PUBSCREEN/K,WINX/K/N,WINY/K/N,
FONT/K,FONTY/K/N,QUIET/K


FUNCTION
  SnmaOpts reads option file in. If you give <file> options are
  read from  that file. Without file <file>, options are read
  first from ENV: and then from the current directory, from the
  file SNMAOPTIONS.
```

1.102 5.2 SnmaOpts - Workbench startup

SnmaOpts can be started from the Workbech. You can pass it argument by
the extended selection (click SnmaOpts icon once, then hold down shift
key, and double click icon you want to pass as an argument). SnmaOpts
ignores all but first argument, because there is no point to pass many
arguments.

```
   TOOLTYPES:

PUBSCREEN=<screen>
WINX=<x>
WINY=<y>
FONT=<font>
FONTY=<fontsize>
QUIET
```

The function of tooltypes is exactly same as the shell commandline options.

1.103   5.3 SnmaOpts - Menus

```
Project           shortcut        function
=======           ========        ========
Save              S               Save SNMAOPTIONS to current directory
Save Def          D               Save SNMAOPTIONS to ENV: and ENVARC:
Save As...        A               Save asking filename

Load              L               Load SNMAOPTIONS from current directory
Load Def          E               Load SNMAOPTIONS from ENV:
Load As...        O               Load asking filename
Restore Defs                      Restore defaults

About                             Display version

Quit              Q               Quit SnmaOpts


Settings
========
This menu is currently disabled. Settings are read from the .info file
of SnmaOpts or from the .info file of argument file (if started from
WB). You can alter the settings by using Workbench's Information menu
item, and manually set TOOLTYPES.
```

1.104   5.4 SnmaOpts- Gadgets

```
                On the right side, there are always three gadgets:

Save              Saves SNMAOPTIONS to current directory and quits.

Save default      Saves SNMAOPTIOS to ENV: and ENVARC: ,and quits

Cancel            Quits


On the left side, there are group of gadgets (page) which can be
cycled with the top gadget "Page". These gadgets display the state of
option file and they can be used to alter options.
There are five pages: Optimizing, Result files,  Debug & cpu, Misc and
Include. Please refer to other sections of this manual to find out the
meaning of the options to snma. Good starting point is section
          2.1
            startup from shell or
          2.1.3
          option file format.
```

1.105   6. Author

```
The author lives in Helsinki, Finland.

If  you are using snma, by sending mail or a postcard  you will
```

motivate him to work with snma.

Send your comments, flames or whatever to

E-mail: snuojua@cc.helsinki.fi

or

smail: Samu Nuojua
  Harustie 8 B 15
  00980 Helsinki
  FINLAND

  Although snma is freeware, donations are always welcome.
  If you use snma to create something nice, how about sending me a copy ?

  Finally, thanks to Satu for the patience.