ESA v1.5 Documentation

COLLABORATORS

	TITLE : ESA v1.5 Documentation		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 13, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ESA v1.5 Documentation

1.1   ESA v1.5 doc (30.10.1998)

```
                    Extended Syntax Assembly v1.5 (30.10.1998)

          © 1998 Simone Bevilacqua


     DISCLAIMER & Distribution
     some legal stuff

     Requirements & Installation
     did you buy another 32Mb Simm?

     Introduction
     got time to waste?!? Read here!!!


     Features
     what can it do?

     Usage
     how to run it?

     Grammar & Constructions
     what you can write and what you can't

     General Notes
     things you have to know

     Error Messages
     what's wrong, now?!?


     Bugs
     oh, no!

     History
     what has happened till now

     Future
```

```
                    what's still to be done?

                    Author
                    some notes about me...

                    Greetz & Thanx
                    ciao!
```

1.2 DISCLAIMER and Distribution

```
              DISCLAIMER

 **********************************************************************
 * THIS PROGRAM IS PROVIDED "AS-IS" WITHOUT WARRANTY OF ANY KIND       *
 * EITHER EXPRESSED OR IMPLIED.                                        *
 *
              I
               ACCEPT NO RESPONSABILITY OR LIABILITY FOR ANY DAMAGE OR PROBLEM *
 * DERIVING FROM THE USE OF THIS PROGRAM:  USE AT YOUR OWN RISK!!!     *
 **********************************************************************


Distribution

 This program is FREEWARE, therefore IT CANNOT BE SOLD FOR PROFIT.
 So, only the distribution charges (i.e.: disk, postage, handling, etc.)
 can be applied.

 No fee is required from
               me
               , but donations of any kind (something like
 the 1st original tankobon of "Dr.Slump & Arale chan" would be just a
 dream... ;) will be gladly accepted.

 If distributed on a coverdisk, please send a copy of the mag!!!

 ALL the following files *MUST* be included in the same package (regard-
 less of the form it comes in):

  ESA/                    (main dir)

   ESA                    main executable
   ESA.guide              this manual

   examples/              (examples dir)

    tab.txt               tabulation indicator
    MergeSort.ei          example source
    QuickSort.ei          example source

    sss/                  (dir of a complete example program)

     sss.guide            prog's documentation
     do                   script for quick compiling
     defs.i               standard asm source
```

```
    main.esa              ESA source code
    misc.ei               ESA include file
    opt.ei                ESA include file
    split.ei              ESA include file
    data.i                standard asm source
```

1.3 Requirements & Installation

```
                Requirements
```

```
 ESA requires a 020+ CPU.
 It should work also on KS1.3, but I can give no guarantee.
 About 90kb + 40kb (or as much as specified with
                -b
                ) + enough room
 for all the
                source files
                 of the program to compile are needed.
```

```
Installation
```

```
 It doesn't need to be installed, just put it anywhere on your HD
 (preferably on your commands path).
```

1.4 Introduction

```
                Introduction
```

```
Oh... so you're wondering why I wrote this prog...
No special purpose indeed... I came from a long period during which
I just studied and didn't code anything (coding is kinda disease...
you know when you start, but don't know when you'll finish... sadly
this doesn't help with exams...). At the end of this interminable
period of forced coding inactivity, I would've coded just anything.
And that's what happened. ESA was the 1st idea which came to my mind
and so I immediately started it, getting up in the depths of night.
OK, I guess you can imagine perfectly how I felt like, so I'll try
to be brief.
Between one project and another, I continued (slowly) developing
this program, even though, when the "creative attack" was over, I
was no longer much convinced about it. Yes, an interesting piece of
software to produce, but - I was wondering - will it somehow come in
handy? I didn't find an answer... I wish that somebody of you will
find it useful or (this would please me even more) that it will
help someone to approach the assembly language...
                what do you think
```

```
                about it?!?
                 I wouldn't be surprised of hearing comments of the kind:
«Junk. Afraid of asm? Stop complaining about its "difficulty" and go
on with an high level language. No need of this "extension" at all.»
```

No. I wouldn't be surprised, because that's EXACTLY what *I* think.
Can't believe it? It doesn't matter. The only other thing, apart from
the pleasure of coding a program that I personally found interesting
to code, which pushed me to complete my work is the fact that I've
learned that in this world there's always somebody who likes what you
wouldn't have ever believed that could appeal to anyone (phew! Cor-
rect? If not, I hope you can get the general sense the same!!!).

1.5 Features

                Features

The job of this program is to take a "strange" assembly source and con-
vert it to a "standard" one, ready to be assembled by your favourite
assembler. A kinda asm-preprocessing, in short.

So now – you're surely wondering – what can this prog do, precisely?
Well, as its name suggests, it handles "extended" asm sources (read be-
low to see how), so that, in the end, it can be said that a new, enri-
ched (if you like, this can also be read as: "at an higher level" – but
that's *not* what I want at all) assembly language comes out of it.
In a nutshell: ESA takes an "extended" asm source as input and outputs
a standard 020+ asm source.

The simplest feature is the possibility of writing
                several assembly

                instructions on a single line
                . While this does *not* ease the reading,
sometimes it can help since it permits to have more code than usual on
a single page.

Surely this is not all that ESA can offer.
In fact, it allows you to use some constructions for the program flow
control, which are typical of high-level languages.
Normally you have inline asm inside C, Pascal, Basic, etc.;
ESA, instead, gives inline C, Pascal, Basic, etc. inside asm, with all
the consequent advantages (yeah! we can mess around with CPU's and HW's
registers, variables, the stack, etc. in total freedom!).

Besides, there are some facilities for the program's structure design:
yes, I'm referring to procedures and functions...
All I'm talking about is described in detail
                here
                .

Obviously, any construction can be used in nested form (there's only a

                very loose limitation...
)!

Finally, ESA treats the include files of any kind (i.e.: both the "old"
"#?.i"s and ESA's "#?.ei"s) in a "special" way:
it's well worth having a look at
                these info about this

!

1.6 Using ESA

```
             Using ESA

Run it from both CLI or WB (no tooltypes support... do you really
wanna launch it from icon?!? I can't believe it!!!).

SYNTAX
        esa [OPTIONS] source [dest]

ARGS
        source    : asm source file to convert
        dest      : output filename
                     (def.: source="file.esa" -> dest="file.s"
                            source="anything" -> dest="anything.s")

OPTIONS
        -sS    {S}: 'S' is the instructions' separator (def.: S='§')
                 with this you can decide how to separate
              two or

              more instructions on the same line
                       -c     {D}: include comments in the output file
                 (normally they are omitted)

        -lC    {D}: 'C'=first char of labels (def.: '.')
                 each label produced by ESA will start with 'C'

        -bSIZE {M}: work buffer of SIZE bytes (SIZE=>4096; def.: 40Kb)
                 (the bigger the faster... less accesses to disk!)

        -q     {M}: quiet mode (no message will be given)

NOTES
        - {S}=source option, {D}=dest option, {M}=misc option
        - the options can be placed anywhere in the command line
        - the options and their args can be separated by spaces
        - press CTRL-C to break execution anytime
```

1.7 ESA Grammar & Constructions (back to school...)

```
             ESA Grammar & Constructions (back to school...)

Although ESA makes asm coding a little "easier", to use it without pro-
blems you *do* need to know at least the basics of 68k asm (and of the
Amiga, of course).
Yet, certainly you don't need to be a master...
so don't let this messy manual fool you: the formal definitions of the
grammar are a bit scary, but in the end everything is extra-simple.
```

The fundamental thing to bear in mind is that you can mix pure 68k as-
sembly and ESA code wherever and whenever you want.
To know how to write ESA code, just read on...

Urgh... quite hard to explain clearly and deeply how the syntax works!
Anyway, once you've understood the general sense, everything should co-
me easy (at least I hope).
To start, I advice you to have a good look at
                 this quite formal list

                 of the valid types
                  of the grammar: if something somewhere is not clear
go on the same (don't worry!) taking some glances at the examples in
any of the sections below, and then go back for better understanding.

     logic:
                 bool
                 boolean evaluation

     loops:
                 do ... loop
                 a bit of AMOS, too!

                 exit
                 exiting loops

                 expire ... nexp
                 68k "dbra"

                 for ... to ... step ... next
                 what to say?!?

                 repeat ... until ...
                 just like Pascal!

                 while ... ewhile
                 BASIC's "while"..."wend"

 decisions:
                 on ... goto ...
                 jump table (branches)

                 on ... gosub ...
                 jump table (subroutines)

                 switch .. -> .. def .. eswitch
                 much better than C's!

                 when .. owhen .. othw .. ewhen
                 "if".."else[if]".."endif"

 functions:
                 function ... efunc
                 defining functions

                 FUNCNAME[]
                 calling functions

```
                    pop
                    exiting functions

procedures:
                    procedure ... eproc
                    defining procedures

                    PROCNAME[]
                    calling procedures

                    pop
                    exiting procedures

directives:
                    incdir & include
                    using external sources
```

1.8 General Notes

```
                General Notes
```

This section gives you a few hints about:

```
            correct use
            problems with generated code

            speed
            performance of generated code

            misc notes
            interesting things
```

1.9 Correct Use

```
            Correct Use
```

The most important thing you have to bear in mind in order to get fully
working code is that you can't use the stack pointer (sp) freely inside

```
            ESA constructions
            (avoid dirty sp tricks!): in fact, the code produced
```
needs to mess a lot with the sp, so don't be surprised if crashes hap-
pen when (sp)-like modes are used inside expressions. Just think about
something else and let ESA take total control of the sp inside its own
constructions.

Remember: the stack is heavily used by ESA generated code!

Another thing to remember is that constructions nesting is permitted to
a certain degree: the biggest nest possible is 64 entries long.

Pay attention! There is *no* check... instead of inserting checks, I'd
prefer to enlarge the internal stack (even doubled would be still very
small) used for this purpose in order to avoid the consequent slowdown.

                    Let me know
                     if you feel too constrained.

Finally, I advice you to increase the default stack size (4096 bytes)
when working with long & complex sources.

1.10   How Do I Get the Best Performance?

                    How Do I Get the Best Performance?

Basic, simple, speedy, flexible... but hard to work with due to the
length of the use procedures.

This applies to almost everything in this world.
And particularly to the hardware/software worlds.
Often, to make things a little bit shorter, simplicity, speed and fle-
xibility are sacrificed.
And this is exactly what (naturally) happens with ESA.

****
WHEN WRITING TIME-CRITIC ROUTINES, DON'T RELY ON ESA CODE'S SPEED!!!
****

There's not much to add. You gotta write them by hand (and that's not
so much bad...).
The reason is that to allow total flexibility to the various constru-
ctions, the code has got to be as much general as possible, and, conse-
quently, slower than it could be if hand written.

ESA's
                    add-ons
                     affect the speed in different degrees:

 -
                    procedures
                     and
                    functions
                     cause a very little speed loss (sometimes
   no loss at all)

 - the
                    for
                     and
                    expire
                     constructions also cause a minor speed loss,
   (
                    expire
                     , in particular, thanks to its nature (simple), is often as
   fast as hand written code). Be careful, though, when using a vari-
   able for the counter of
                    for...next

                        : in small loops the overhead
   could be quite heavy!

  – the real beasts are all the others, as they include the evaluation
    of
            boolean expressions
            .
    Here I'd like to spend a couple of words (you can skip this...):
    writing code which automatically generates pieces of code to evalu-
    ate (almost) all kinds of boolean expressions, *without* having the
    possibility of using registers, is a tough thing (I looked at it as
    a challenge... I really enjoyed writing the code about this part –
    – I wonder if there's any theory about this... if you know, please

            contact me
            ); it isn't easy to get rid of the difficulties that this
    problem presents (mainly because there is no availability of regi-
    sters), since not only variables (like in high level languages) but
    also the registers themselves have to be handled (carefully) as bo-
    olean and integer variables in the expressions.
    The result is that the code produced for boolean expressions' evalu-
    ation looks ugly (and it is, indeed), altough I put in as many opti-
    mizations as possible (for example: "not" ("
            ~
            ") is treated in a ve-
    ry smart way, making large use of the "De Morgan" rules): so, if you
    need speed, avoid automatically generated boolean expressions.

 My advice is: use
            procs
            ,
            funcs
            ,
            fors
             on so on almost everywhere, but
do pay attention when a
            boolean expression
             pops up!!!

1.11 Miscellaneous Notes

            Miscellaneous Notes

 These notes come in no particular order.
 If you have followed a link then you should be automatically pointed to
 the relevant section (unless you're at the bottom of the page... this
 is a problem of the amigaguide viewers!)

 – some constructions produce jumps to labels generated automatically:
   if they are local (=start with '.') and if between these jumps you
   use any global definition, probably the assembler will fail with an
   error of the kind: "undefined symbol"

 – default size is ".l" (except where differently stated);

- place spaces/TABs wherever you want, except between the arguments
  and their own sizes;

- remember that ESA makes mainly *syntactical* checks, *semantics* is
  left to the assembler: so, if you write an invalid expression, ESA
  won't warn you at all (give a look at
                this simple example
)!!!

- since
                var
                 accepts almost anything, it's up to you to avoid
  weird things...

- ESA is *case sensitive* for speed's sake!

- remarks must start with '*' or ';' if they are at the beginning of a
  line or are not preceded by any instruction/directive;
  otherwise ';' is the only char which marks a comment (in this case it
  has to be used after a TAB or space);

- comments can be put only at the end of any
                sequence of instructions
                ;

- all spaces and TABs in the arguments will be removed (except if en-
  closed between "" or '');

- when ESA is halted by
                an error during pass 2
                , the output file holds
  all the code generated until that moment

- as shown in the examples scattered in the
                grammar
                 chapter, sometimes
  ESA doesn't seem able to align properly the asm instructions in their
  column... weird, huh?!? Well, this is not a bug, it's another "tribu-
  te" to speed!!! For the same reason, a negated exclusive or (~eor)
  makes some capitalized letters appear in the code ("EOR")!!!

- the labels generated by ESA have this format: CXXXXXXX, where XXXXXXX
  is a number in hexadecimal notation and C is generally '.' (or the
  char you have selected with the
                -l option
                ); otherwise, it can be ei-
  ther 'p' for
                global procedures
                 or 'f' for
                global functions
                .
  In theory, up to 3*268435456 different labels can be generated, but
  once passed the 268435455 mark, it's highly likely to produce repeti-
  tions... but who's gonna pass it, anyway?!?

- for those who are going to deeply and critically analyze the code
  produced: somewhere you'll find things like "(-6,sp)" where, in-

stead, it should have been "(-5,sp)". Don't worry. This is due to a
"bug" in the MC68k which decreases [increases] sp by 2 when using a
byte size and a predecrement [postincrement] addressing mode!

- notice on
              error reports
              : rarely (in just *one* particular case -
challenge (no prize): find it!) the printing of the string which ge-
nerated the error could be somehow corrupted (truncated or partially
modified in the middle, etc.); this is *not* a bug: it's because du-
ring pass1 some integer values are directly written in the source (to
speed up several things): since it happens not so often, I chose not
to fix this problem (to avoid a little slowdown and an increase of
memory needs)

- lines longer than 2048 characters could cause malfunctioning (even
GURUs!!!) when the work buffer is almost full

- little discussion on the kind of brackets used for funcs/procs or
boolean expressions: yes, I was *forced* to use '[',']' or '{','}',
respectively. Wanna know why?!?

Look at this: " ~(a0) " [this is a
              boolean expression
              ]
What does it mean to you?
 1. logical complement of the data stored at the address in a0
 2. logical complement of the data stored in a0
If I had used '(',')', both answers would have been right.
Using the ungraceful '{'s any ambiguity is swept away:
 1. ~(a0) = ~{(a0)}
 2. ~a0 = ~{a0}

About functions: " move.l MyLabel(a0),d0 "
What's your pick?
 1. load in d0 the value at the address calculated as a0+MyLabel
 2. load in d0 the value returned by the function MyLabel() with the
    parameter a0
Again, those would've been both right.
But those unusual brackets help us once again:
 1. move.l MyLabel(a0),d0 = move.l (MyLabel,a0),d0
 2. move.l MyLabel[a0],d0

And what about procs?
Honestly, there is no problem with them, thanks to the way they are

              called
              . But how could I mix together '['s and '('s ?

- not to complicate too much the code which checks the syntactical cor-
rectness of
              vars
              , "-(ax)+" is accepted even if wrong bigtime!

1.12 Error Messages

```
              Error Messages
```

As you may have guessed, this section covers the errors reported by
ESA and all the related stuff. I've not been too fussy, so the same
error could be given for a number of different mistakes. My advice
is to check the syntax, the prob is almost always there!

Error reports take the form of:

```
 "ERROR " ERRNO ": " ERRTEXT
```

or (when needed):

```
 "ERROR " ERRNO ": " ERRTEXT " at line " LINENO " of " FILENAME ":"
 ">" CODELINE
```

where:

 - ERRNO is the number of the error found (it will also be returned as
   the AmigaOS fail returncode)
 - ERRTEXT is the concise explanation of what happened
 - LINENO is the line which the error occurred at
 - FILENAME is the file which contains the error (only the file
   part of the path is printed)
 - CODELINE is the wrong line in the source

```
(there's also another little
              notice about this
              ...)
```

Errors are grouped into 3 classes; below you can find a few info
about them (no description/info given for self-explaining messages):

```
              pass 1
              reports during pass 1

              pass 2
              reports during pass 2

              misc
              general messages
```

You may also find useful an ordered
              list of all messages
              .

1.13 Pass 1 Errors

```
              Pass 1 Errors
```

 1: user break

```
          - this is your own business...
 2: couldn't load source file
 4: not enough memory
    - ESA either didn't find enough room to load a
              source file
               or
      failed to allocate dinamically one of the little structures used
      for
              procedures
               and
              functions
               definitions!
12: wrong syntax in
              procedure declaration
               13: wrong syntax in
              function declaration
               24: too many
              nested includes
                  - max recursion degree for
              include files
               is 64 - and you've just
      passed beyond!
25: couldn't access source directory
    - ESA couldn't get the lock to the dir of a
              source/include file
               33: directory not found
    -
              incdir
               specifies a directory which cannot be reached from the
      current directory
```

1.14 Pass 2 Errors

```
              Pass 2 Errors

 1: user break
    - this is your own business...
 5: unexpected end of file
    - there is a construction of the type: "begin"..."end" which
      hasn't been closed (i.e. "end" part missing) before the end
      of the source file
 6: unexpected end mark
    - ESA met an "end" statement used for the constructions of the
      kind: "begin"..."end" which wasn't the one it was waiting for.
      Pay attention to the
              nested constructions
               in your source
 7: unsignificant string after ESA declaration
    - side comments must start with ';'
    - no string is allowed after an ESA construction, unless separated
      by the
              separator char
                8: wrong syntax in
              boolexpr
                9: wrong syntax in
```

```
                bool
                 declaration
10: wrong syntax in
                expire
                 declaration
11: wrong
                condition code
                 in
                nexp
                 declaration
14: wrong size in
                pop
                 declaration
15:
                pop
                 statement not inside a
                procedure
                /
                function
                       -
                pop
                 doesn't work for loops
16: unknown
                procedure
                 17: unknown
                function
                 18: wrong syntax in
                procedure call
                 19: wrong syntax in
                function call
                 20: arguments mismatch in
                procedure
                /
                function
                 call
    - you passed less or more arguments than expected from the decla-
      ration of the
                procedure
                /
                function
                 21: wrong syntax in
                until
                 declaration
22: wrong syntax in
                while
                 declaration
23: wrong syntax in
                when
                 declaration
26: wrong syntax in on...
                goto
                /
                gosub
                ... declaration
27: wrong syntax in
                for...to...step
                 declaration
```

```
28: byte size in conjunction with address register
    - CTR has a byte size in the
                for...to...step
                 declaration and END
      or STP is an address register (this applies also to
                functions
                '
    return values!)
    - you simply wrote "ax.b"!
29: wrong size in
                next
                 declaration
30:
                othw
                 not inside
                when...ewhen
                 31: wrong syntax in
                switch
                 declaration
32: wrong value declaration after
                ->
                 34: error inside
                switch...eswitch
                     - at least 1 "->" is needed (indepentently of
                def
                 case}
    -
                def
                 must be the last case statement
35:
                othw
                 repetition
    - othw has already been declared inside the current
                when...ewhen
                 36:
                owhen
                 not inside
                when...ewhen
                 37:
                othw
                 already specified before
    -
                owhen
                 can't be declared after
                othw
                 38: wrong size in
                loop
                 declaration
39: wrong size in
                exit
                 declaration
40: not enough loops to
                exit
                 41: cannot
                exit

                procedures
```

```
                /
                functions
                      - you have to use
                pop
                !
 42: bad
                efunc
                 return value
```

1.15 General Errors

```
                General Errors
```

```
 3: couldn't open dest file
 4: not enough memory
    - ESA failed to allocate the work buffers.
      Try freeing some memory or decreasing the
                work buffer size
```

1.16 Errors List

```
                Errors List
```

```
no class    text

  1

                1
                2
                  : user break
  2
                1
                    : couldn't load source file
  3
                m
                : couldn't open dest file
  4
                1

                m
                : not enough memory
  5
                2
                  : unexpected end of file
  6
                2
                  : unexpected end mark
  7
                2
                  : unsignificant string after ESA declaration
  8
                2
                  : wrong syntax in
```

```
                boolexpr
                  9
                2
                  : wrong syntax in
                bool
                 declaration
10
                2
                  : wrong syntax in
                expire
                 declaration
11
                2
                  : wrong
                condition code
                 in
                nexp
                 declaration
12
                1
                    : wrong syntax in
                procedure declaration
                 13
                1
                    : wrong syntax in
                function declaration
                 14
                2
                  : wrong size in
                pop
                 declaration
15
                2
                  :
                pop
                 statement not inside a
                procedure
                /
                function
                 16
                2
                  : unknown
                procedure
                 17
                2
                  : unknown
                function
                 18
                2
                  : wrong syntax in
                procedure call
                 19
                2
                  : wrong syntax in
                function call
                 20
                2
```

```
              : arguments mismatch in
procedure
/
function
 call
21
2
  : wrong syntax in
until
 declaration
22
2
  : wrong syntax in
while
 declaration
23
2
  : wrong syntax in
when
 declaration
24
1
    : too many
nested includes
 25
1
    : couldn't access source directory
26
2
  : wrong syntax in on...
goto
/
gosub
... declaration
27
2
  : wrong syntax in
for...to...step
 declaration
28
2
  : byte size in conjunction with address register
29
2
  : wrong size in
next
 declaration
30
2
  :
othw
 not inside
when...ewhen
 31
2
  : wrong syntax in
switch
```

```
            declaration
32
            2
              : wrong value declaration after
            ->
             33
            1
                : directory not found
34
            2
              : error inside
            switch...eswitch
             35
            2
              :
            othw
             repetition
36
            2
              :
            owhen
             not inside
            when...ewhen
             37
            2
              :
            othw
             already specified before
38
            2
              : wrong size in
            loop
             declaration
39
            2
              : wrong size in
            exit
             declaration
40
            2
              : not enough loops to
            exit
             41
            2
              : cannot
            exit

            procedures
            /
            functions
             42
            2
              : bad
            efunc
             return value
```

1.17 Bugs

                    Bugs

 Some versions of ESA have been tested (not so deeply) on:

  - Unexpanded A1200
  - A1200 + BZ1230-IV
  - A1200 + BZ1260
  - A4000/040
  - A4000 + CS060-II

 No known bug at the moment.

 If you think you have found any, please
                    send me
                     a detailed bug report.
 Machine specs ain't strictly necessary, the most important thing is the
 part of code which you think to be responsible for the bad behaviour of
 ESA and the (bad) code generated.

 After this, just hope for a prompt fix!!!

1.18 History

                    History

 v1.5 (30.10.1998)

  -
                    efunc
                     extended
  - little optimization in
                    boolexpr
                     check code
  - little manual retouches

  Well, no bugfixes this time... it seems I'm almost done with this prog
  (at least I wish so)!

 v1.4 (25.10.1998)

  - as I feared, the "frantic" changes in the previous version led to a
    number of mistakes:
    1. the usual "bne" <-> "beq" error in type detection code
    2.
                    >> and <<
                     were considered
                    cmpops
                     if used in
                    mathexprs
                     inside

```
                    boolexprs
                        3.
                    predecrement/postincrement
                     modes weren't recognized correctly as

                    var
                    , because '+' and '-' were considered separator chars
      4. negative
                    symbols
                     weren't accepted (this should have been fixed
        much time ago, but I simply forgot to do it!!!)
      5. '.' was recognized as an "empty"
                    symbol
                       - removed superfluous TAB+ENTER in the code produced by
                    switch
                       - several optimizations (particularly in the grammar handling  ←
                         code)
   - manual update
```

All the bugs fixed in the last two versions (including this one) have
been discovered while writing the program "sss" (contained in the ar-
chive "sss.lha" in the directory "examples" of this
```
                    distribution
                    .
```
Please, Mr.Murphy, stop tormenting me...

v1.3 (23.10.1998)

```
   - brackets changed again!
                    Procs
                     and
                    funcs
                     now use '[','],': nicer and
      more practical (no SHIFT - one keystroke less) (sorry if you have
      already defined many {}-procs, but there was also a serious reason:
      the '{'s produced some conflicts with boolexprs and resolving them
      in another way would have been less efficient... and less stylish!!!
   - bugfixes:
    1. by changing the brackets used for procs/funcs (in v1.2) I introdu-
       ced several bugs (ex.: funcs were handled incorrectly inside bool-
       exprs; during debugging I even found one which should have screwed
       up everything, but all misteriously worked perfectly!!!).
    2. silly flaws in
                    do
                    ,
                    repeat
                     and
                    expire
                     code which, in some
       combinations, messed up the labels
    3. little correction to include handling
    4. few minutes before going to the uni computer lab (and just after
       getting up...) to upload this version, I realized that due to the
       last changes the grammar code had to be modified!!! So I turned on
       my Amiga and made this fix "on the fly", with one hand on the key-
       board and the other putting on my shoes...
```

```
  - little change in
                when...ewhen
                 routines to make generated code a
    little more readable if compiling interrupts in the middle of that
    construction
  - small optimizations
  - oh damn! I fear I'll never stop updating this .guide!!!
```

Several important parts of the code had to be modified in a hurry, I
just hope I didn't throw in any other bugs... I've been fighting for
the whole night!!!

v1.2 (16.10.1998)

```
  - major changes in parsing routine (optimised)
  - the elegant form "name(args)" for proc/func calls has been dropped
    in favour of the awkward form "name{args}"...
    ...but now
                calls to undefined functions can be detected
                !!!
  -
                do...loop
                 added
  - "exit" renamed "
                pop
                "
  - (new)
                exit
                 added!
  - some flaws fixed
  - elapsed time report added
  - usual boring changes to this manual
```

Although this is not a definitive version, I decided to release it be-
cause I'm going away for a few days and, when I'll be back, I'll be
very busy with studies...
Since it's complete (and bugfree, I hope) now, there's no reason to
delay the release for an undefined period of time

v1.1 (12.10.1998)

```
  -
                switch
                 100% working: now nesting is permitted and "beq" replaced
    the wrong "bne" (little moment of absent-mindedness of mine...)
  -
                switch
                 and
                when...ewhen
                 capabilities extended (explicit condition
    declaration and
                owhen
                , respectively)
  -
                for...next
```

```
                      default step set to -1 when using
                   dwto
                    (I just forgot
   about it before...)
 - bugfixes:
   1. source file loading
   2.
                   incdir
                    (after pass1 this directive wasn't preserved)
   3.
                   until
                    ("bne"<->"beq"... same as
                   switch
                   !)
   4. parameters loading in
                   proc
                   /
                   func
                    calls
 -
                   includes
                    handling improved (now names between " or ' are accepted)
 - misc optimizations
 -
                   grammar definition of type imm
                    extended (I totally forgot the forms
   of the kind: #"symb" or #'symb')
 -
                   grammar definition of type args
                    changed (compatible with previous)
 -
                   AmigaOS fail returncode
                    added
 - default
                   work buffer size
                    changed (10Kb -> 40Kb)
 - manual deeply revised/updated

 WOW! it seems I'm almost finished with it!!!


v1.0 (05.10.1998)

 -
                   switch
                    included at 99%
 -
                   size types
                    extended ({dsize, asize, jsize} instead of {size})
 - better handling of regs' sizes ("ax.b" somewhere would have been
   used as a
                   val
                    instead of causing an error)
 -
                   procedures
                    and
                   functions
```

```
                     declaration syntax slightly changed:
      "PROCNAME,loc()" has become a much more meaningful: "loc:PROCNAME()"
    - bugfixes:
      1.
                    error reports
                        2.
                    othw
                        3.
                    include
                        4. type detection code (probably introduced in v0.9b!), "/"  ←
                            recogni
         tion as a
                    matop
                      - manual revised/updated ;)
```

 Not released, although it's the 1st (almost) complete version

v0.9b (14.09.1998)

 -
                    incdir
                     handling added

 For some unknown reasons the upload of this version failed several
 times: hence it's never been publically released!!!

v0.9 (15.07.1998)

 First public release.
 For time reasons
                    switch
                     and
                    incdir
                     couldn't be implemented.

1.19 Future

                    Future

First, let me say that I don't think I'll have much time to spend on im-
proving this program. Too bad this *doesn't depend on me*.
I just can ensure that I'll do my best to fix all the
                    bugs
                     you'll find
(as soon as I'll have the time) and add those easy, minor improvements
which could make ESA a little more friendly.

Speaking about "real" additions/expansions or whatever...

To be honest, I'm not willing at all to add more constructions, for one
simple, plain reason: I don't wanna end up writing a new language.
If you need to pass to an even higher level, than switch to C or E or
anything else.

```
ESA has already a few features which at the beginning I didn't plan nor
want to implement (which ones? procedures, functions... and something
else), 'coz I considered too "advanced"...
Well, now you got'em, enjoy and let's forget about this.
But, pleeeeeze, don't ask me to add other magic commands, unless they're
are really something special...
However, don't be discouraged by what I just said:
got an idea? Just
                gimme a call
                 and let's see if I fancy it.
Maybe it turns out to be that damn nice feature ESA was missing!
```

1.20 Hi there!

```
Hi there!

 I *do* want your feedback.
 Let me know what you think and if you have any problems/ideas or need
 some explanations/hints.

 Write to:

  bevilacq@cli.di.unipi.it

 I can also be reached by snail mail at the following addresses:

  (during "normal" periods)

  Simone Bevilacqua
  P.za  Garibaldi 9
  56100 Pisa (PI)
  ITALY

  (during uni vacation periods – "safer" address!!!)

  Simone Bevilacqua
  Via A.Volta 6
  86010 Ferrazzano (CB)
  ITALY
```

1.21 Greetz and Thanx

```
Greetz and Thanx

 Thanks to all the Amiga coders still around and in particular to:

  Michele Berionne, Pietro Ghizzoni: help with uploading and testing;
  Fabio Bizzetti: testing;
  Frank Wille: testing and... his magic PhxAss!!!

 Mega greetings to my family and all my friends!!!
```

(the next time I'd like to scribble here something like:
"WOOOOOOOOOOOooooooooooOOOOOOOOOOOOW!FINALLYMYPATIENCEHASBEENREWARDED!!!
THANXALOTAMIGAINC./INT.FORHAVINGGIVENTHEWORLDANEWBEAUTYANDAHOPETOSAVEIT
FROMTHOSEDIRTYM$OFT/INTEL'SHANDS!!!"

Now I wonder: will that moment ever come before 2177a.d.?!?
PLEASE, MAKE IT SO, MAKE IT SOON!!!)

1.22   Include Files Handling

                    Include Files Handling

ESA processes the include files listed in the source so that you can
freely build your own "libraries" of
                    functions
                    /
                    procedures
                    .
It will recursively (max depth: 64) parse the includes, producing a
single output file without *any* include statement. Of course, each
include file will be included and compiled just once (BTW: as a side
effect, this will ease the assembler's task, as it will have to load
only a single source).
Please note that "IF" directives are simply ignored, so this kind of
declarations:

  IFND EXEC_TYPES_I
  include "exec/types.i"
  ENDC

would be compiled as:

  IFND EXEC_TYPES_I

  ENDC

if "exec/types.i" has already been included (even if specified with a
different path).

The directory which will be scanned to find the include files listed
in a source is the source's one (when no full path is declared - this
applies recursively also to includes).

The above rule is void if an "incdir" directive is found: in that case,
any other subsequent include statement in the source containing that
"incdir" will refer to the specified directory.

Dir/file names can be enclosed in "" or ''.

Please note that it doesn't make any sense to compile ESA include files
(my proposal is to call them "#?.ei" for convention) separately from
the source[sources] which makes[make] use of them because ESA generates
unique labels only when all the source files are available.

    ****

```
* WARNING: DUE TO TIME REASONS, VERY FEW TESTS HAVE BEEN DONE!        *
*          IF SOMETHING STRANGE HAPPENS (ESPECIALLY WITH "incdir") IT *
*          COULD BE WELL A
                BUG
                  (though I had no problem)!              *
    ***********************************************************************
```

1.23 Multiple Instructions on a Single Line

                    Multiple Instructions on a Single Line

ESA allows you to put several instructions and/or ESA commands (with
their arguments, if required), separated by a special char, on a single
line.

Let's make an example:

          lea.l buffer,a0 § bool d1=d2,d0.b § add.b d0,d0

I stopped at the 3rd instruction, but there can be as many instructions
as you want... but then you'll find yourself scrolling the screen hori-
zontally rather than vertically! Not a great deal!!!

As you can see, the instructions are separated by " §" (note: the lea-
ding ' ' is compulsory, the following not), which is the default sepa-
rator. If you wish to change it, use the
                  -s option
                    .

WARNING: don't put labels after an instruction using the separator
          (they would be exchanged for instructions)!

1.24 Conventions and Types

                    CONVENTIONS USED IN THE WHOLE TEXT

```
...         = ESA and/or asm code
[xyz]       = xyz is optional
ID:type     = ID is an identifier of the type specified
"xyz"       = xyz is a string of characters
'xyz'       = as above (less frequent)
```

Also, have a look at the
                  misc notes
                    .

TYPES

  0.
                  logop
                  : "&" | "|" | "^"

```
   1.
                   cmpop
                 :  "<" | ">" | "<=" | ">=" | "=" |
                   "«" | "»" | "«=" | "»=" | "<>"
   2.
                   matop
                 :  "+" | "-" | "*" | "/" | "//" | "<<" | ">>"
   3.
                   dsize
                 :  ".l" | ".w" | ".b"
   4.
                   asize
                 :  ".l" | ".w"
   5.
                   jsize
                 :  ".l" | ".w" | ".b" | ".s"
   6. dreg        :  "d0" | "d1" | ... | "d7" |
                   dreg dsize
                     7. areg      :  "a0" | "a1" | ... | "a7" |
                   areg asize
                     8.
                   reg
                 :  dreg | areg
   9.
                   regslist
                 :  reg | reg"/"regslist |
                   dreg"-"dreg | dreg"-"dreg"/"regslist |
                   areg"-"areg | areg"-"areg"/"regslist
  10.
                   sym
                 :  any symbol accepted by the assembler
  11. var         :
                   ea
                   [size] except imm
  12.
                   boolexpr
                 :  rval | rval cmpop rval | boolexpr logop boolexpr |
                   imm cmpop rval | "
                 ~
                 " boolexpr | "{" boolexpr "}"
  13.
                   mathexpr
                 :  sym matop sym | sym matop mathexpr |
                   mathexpr matop sym | mathexpr matop mathexpr |
                   "(" mathexpr ")"
  14. imm         :  "#"sym | "#"mathexpr | "#'?'" | '#"?"'
                   (where "?" is a string 1,2 or 4 characters long)
  15. val         :  imm | var | func
  16.
                   rval
                 :  var | func
  17. args        :  val | val "," args
  18. func        :  any valid ESA
                   function call
                     19.
                   cc
                 :  "eq" | "ne" | "vc" | "vs" | "pl" | "mi" |
```

```
                    "lo" | "ls" | "hi" | "hs" | "cc" | "cs" |
                    "lt" | "le" | "gt" | "ge" | "t"  | "f"
```

1.25 Effective Address

```
Effective Address

 ea = any valid addressing mode

 ESA won't make any check on several addressing modes, so eas correct-
 ness is in your hands.
```

1.26 Logical Operators

```
                Logical Operators

 "&" = and
 "|" = or
 "^" = exclusive or

 These operators work on boolean basis:
 they are *not* bitwise operators operators, but just know 0 and <>0.

 Please note that "~" (not), being an unary logic operator, can be used
 only in some positions in
                boolean expressions
                  .
```

1.27 Comparison Operators and Condition Codes

```
                Comparison Operators and Condition Codes

 Here's the list of the operators which can be used in
                boolexprs
                  (with the corresponding condition codes):

  op     cc     meaning

  "="     eq     equal to
  "<>"    ne     not equal
  "<"     lt     less than       (signed)
  ">"     gt     greater than    (signed)
  "<="    le     less or equal   (signed)
  ">="    ge     greater or equal (signed)
  "«"     lo     lower than      (unsigned)
  "»"     hi     higher than     (unsigned)
  "«="    ls     lower or same   (unsigned)
  "»="    hs     higher or same  (unsigned)

 Other valid condition codes are:
```

```
cc     meaning

t      true
f      false
vc     overflow clear
vs     overflow set
cc     carry clear
cs     carry set
pl     plus
mi     minus
```

1.28 Mathemathical Operators

```
Mathemathical Operators

 "+"  = addition
 "-"  = subtraction
 "*"  = multiplication
 "/"  = division
 "//" = modulo
 "<<" = shift left
 ">>" = shift right

 These are the ones accepted by PhxAss;
 dunno other assemblers.
```

1.29 Sizes

```
Sizes

 ".b", ".s" = byte
 ".w"       = word
 ".l"       = long
```

1.30 A Little Mistake in the Grammar...

```
              A Little Mistake in the Grammar

 According to the definition adopted in the
              conventions
              , a thing in
 the shape of: "d0.b.b.w" is a *correct*  dreg.
 Actually, this is *not* true, but that's just a simplification in the
 grammar (to make it a bit more readable).
```

1.31   Registers

```
Registers

 Only data & address registers can be used, sorry.
 (For now) forget about ssp, sr, and so on...
 If you try to use one of them, it will be treated just like a normal
 symbol!
```

1.32   Registers Lists

```
              Registers Lists

 This is the type used for movems in 68k asm.
 With ESA it assumes a more versatile aspect: in fact you can declare
 also the size of any argument.
 This, obviously, doesn't applies to movems (sizes are discarded, ".l"
 is used as default), but has a great importance in
              procs
               and
              funcs
               calls.

 A declaration of the kind: "a0.w/d3.b-d5" is perfectly legal and means,
 if included in a call:

  - load a0 with a 2 bytes long value
  - load d3, d4, d5 with 1 byte long values

 The same would have happened if the declaration had been:
  "a0.w/d3.b-d5.w"
 since only the 1st size, in "dx.y-di.j" or "ax.y-ai.j" statements, is
 taken into account (y here).

 Moreover, as the syntax shows, it's possible to mix in any order aregs
 and dregs: "a3.w / d0-d2 / a5 - a7 / d5 / a1" is still valid (but *NO*
 check is performed on repetitions! An "a5" in the place of "a1" would
 not cause any error!).
```

1.33   Symbols

```
              Symbols

 Here are listed all the chars which can be used in symbols (labels).
 If you think that someone is missing, just
              drop me a line
               .

 0 1 2 3 4 5 6 7 8 9
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
@ $ \ _ $^1$ $^2$ $^3$ ¢ ¼ ½ ¾ · ì è à ù § ò å \textdegree{} © ® þ ¤ $\mathrm{\ ←
    mu}$ ¡
ø ¶ æ ß ð £ £ \ensuremath{\pm} $\times$ ç ª °
```

The chars '.' and '-' are allowed only at the beginning of a symbol.

ESA will only partially check the correctness of symbols, so it can
happen that unvalid symbols are used without any warning.

1.34 Boolean Expressions

                Boolean Expressions

                Click here
                 for some hints on how to use these expressions in the most
effective way.
Also have a look at the
                boolean
                 and
                comparison
                 operators.

The arguments of boolean expressions are treated in this way:
 false=0, true<>0.
Yet, after the execution of the evaluation code, it will always be:
 false=0, true=-1 (255);
that's why it's possible to write expressions like: "a0.w & Sendo.b",
whose code would be:

```
        cmpa.w      #0,a0
        sne.b       -(sp)
        tst.b       Sendo
        sne.b       -(sp)
        move.l      d0,(-4,sp)
        move.b      (sp)+,d0
        and.b       d0,(sp)
        move.l      (-6,sp),d0
```

The size used in comparisons is the one of the 1st register or, when
there's no reg, of the 1st argument:

 code produced for "Hanamichi.w=Kaede.b":

```
        move.l      d0,(-6,sp)
        move.w      Hanamichi,d0
         cmp.w      Kaede,d0
        seq.b       -(sp)
        move.l      (-4,sp),d0
```

 code produced for "d5.b=Haruko.l" or "Haruko.l=d5.b":

```
         cmp.b      Haruko,d5
        seq.b       -(sp)
```

As an additional note, when an argument is an address register only
".w" and ".l" can be used, thus it's impossible to write something like
"a5.b = Senbe";
on the other hand, a statement of the kind "d0.b > a3.w" will make use
of ".w", since aregs have priority over dregs.

OK. Why don't you use the same size in both arguments ;)

As you can see, the best code is obtained when at least one argument
is a register:

 code produced for "Ronzaman<d1":

```
        cmp.l       Ronzaman,d1
        sgt.b       -(sp)
```

 code produced for "a5.w »= Suppaman":

```
        cmpa.w      Suppaman,a5
        shs.b       -(sp)
```

 code produced for "Suppaman.b »= Ronzaman":

```
        move.l      d0,(-6,sp)
        move.b      Suppaman,d0
         cmp.b      Ronzaman,d0
        shs.b       -(sp)
        move.l      (-4,sp),d0
```

"Style" note: boolean expression can be contained inside '{' and '}'.
I know it isn't stylish, but there's
                a very serious reason
                 behind.

1.35 Mathemathical Expressions

                Mathemathical Expressions

These are made of constats/symbols and
                math operators
                 .
As always, ESA will check only their syntactical correctness:

 - ((say+hello-to-Pippo)

 this will be reported as wrong (FYI (if you're a very curious dude):
 (say+hello-to-Pippo) will be accepted and used. Upon completion of all
 the operations with it, going on with the parsing, the second ')' will
 not be found and an error will be generated);

 - ApplePie/0

 this, instead, won't cause any warning, even if the assembler will
 clearly scream out loud that divisions by 0 are a little hard to do...

1.36 Restricted Values

```
              Restricted Values

 This type is defined for (almost) exclusive use in
              boolexprs
              .
 As the name suggests, it's a restricted version of val, lacking of the

              imm type
              .
```

1.37 boolean evaluation

```
bool

 SYNTAX

  "bool" BL:boolexpr "," DEST:var


 MEANING

  1. evaluates BL
  2. writes its value (true, false) to DEST


 NOTES

  - the default size used for DEST is *byte*;
  - to obtain the fastest results, use the default size, especially
    if DEST is not a dreg (see below);
  - if DEST is an areg without explicit size, ".w" is used as default;


 EXAMPLE 0

  ESA asm:

         bool { {Suppaman=d4} & Slump} | {~{d4=d5}}, d2.l

  68k asm:

          cmp.l     Suppaman,d4
         seq.b      -(sp)
         tst.l      Slump
         sne.b      -(sp)
         move.l     d0,(-4,sp)
         move.b     (sp)+,d0
         and.b      d0,(sp)
         move.l     (-6,sp),d0
          cmp.l     d5,d4
         sne.b      -(sp)
         move.l     d0,(-4,sp)
```

```
        move.b      (sp)+,d0
         or.b       d0,(sp)
        move.l      (-6,sp),d0            ;BL evaluation
        move.b      (sp)+,d2             ;.l size doesn't affect
        extb.l      d2                   ;much the speed...
```

EXAMPLE 1

 ESA asm:

```
        bool Makusa,ObabaHaru.w

        bool Makusa,ObabaHaru.b          ;default size
```

 68k asm:

```
        tst.l       Makusa               ;1st "bool"
        sne.b       -(sp)
        move.l      d0,(-4,sp)
        move.b      (sp)+,d0
        extb.l      d0
        move.w      d0,ObabaHaru
        move.l      (-6,sp),d0           ;slooow...

        tst.l       Makusa               ;2nd "bool"
        sne.b       -(sp)
        move.b      (sp)+,ObabaHaru      ;much faster, huh?!?
```

1.38   a bit of AMOS, too!

```
do ... loop
```

 SYNTAX

```
  "do"
    ...
    ...
    ...
  "loop"[SZ:jsize]
```

 MEANING

  1. executes the code between "do" and "loop"
  2. repeats 1 forever

 NOTES

  - SZ is the size for the bra instruction used (default: none);

EXAMPLE 0

```
   ESA asm:

           do                  ;here's a nice
            addq.l    #1,d0    ;way of wasting
           loop.s             ;processor time...

   68k asm:

.0000000
           addq.l    #1,d0
           bra.s     .0000000
```

1.39   exiting loops

```
exit
```

```
 SYNTAX

  "exit"[SZ:jsize][","  CNT:imm]
```

```
 MEANING

  1. exits from the last CNT loops entered
     (if CNT undeclared, then CNT=1 by default)
```

```
 NOTES

  - SZ is the size to be used for the bra (default: none);
  - CNT is the number of loops you wish to exit from (CNT>0; default: 1)
  - if used also inside a begin...end-type construction, this will be
    "broken", too (except if it's a proc or func: that would generate an
    error)!
```

```
 EXAMPLE 0

  ESA asm:

           do
            repeat
             while d0
              expire d1=#23
               for d2=#0 upto #10        ;this example does nothing!
                exit.s #5                ;exit all the loops at once!
               next
              nexp
             ewhile
            until d3
           loop

   68k asm:

.0000000                                                        ;do label
```

```
.0000001                                                         ;repeat label

.0000002    tst.l      d0                                       ;while condition
            sne.b      -(sp)
            tst.b      (sp)+
            beq        .0000003
            move.w     #23,d1
.0000004                                                         ;expire label
            move.l     #0,d2                                    ;for args loading
            move.l     #10,.0000005
            move.l     #1,.0000005+4
            bra.s      .0000006
.0000005    dc.l       0,0
.0000006    cmp.l      .0000005,d2
            bgt        .0000007
            bra.s      .0000008                                 ;this is exit!!!
            add.l      .0000005+4,d2
            bra        .0000006                                 ;next
.0000007
            dbra       d1,.0000004                              ;nexp
            bra        .0000002                                 ;ewhile
.0000003                                                         ;until condition
            tst.l      d3
            sne.b      -(sp)
            tst.b      (sp)+
            beq        .0000001
            bra        .0000000                                 ;loop
.0000008


  EXAMPLE 1

    ESA asm:

            do                          ;looks like a rather *WorRyiNG*
             when.s #1000=d0.b          ;delay-loop!!!
              exit.s
             othw
              addq.l #1,d0
             ewhen
            loop.s

    68k asm:

.0000000
            cmpi.b     #1000,d0
            seq.b      -(sp)
            tst.b      (sp)+
            beq.s      .0000002
            bra.s      .0000003          ;exits when...ewhen, too
            bra.s      .0000001
.0000002
            addq.l     #1,d0
.0000001
            bra.s      .0000000
```

```
.0000003
```

1.40   68k 'dbra'

```
expire ... nexp

 SYNTAX

  "expire" DX:dreg "=" ST:val
      ...
      ...
      ...
  "nexp" ["," COND:cc]


 MEANING 0 (when COND not declared)

  1. assigns to DX the value of ST
  2. executes the code
  3. decrements DX by 1
  4. if DX=>0, goes to 2


 MEANING 1 (when COND declared)

  1. assigns to DX the value of ST
  2. executes the code
  3. if COND is satisfied then the execution contines with the first
     instruction after "nexp"
  4. else decrements DX by 1
  5. if DX=>0, goes to 2


 NOTES

  - since the instruction used is dbcc, the size of DX and ST is always
    word (any specification is ignored);
  - if DX=ST, no assignment is done, so that you can use a register ini-
    tialized externally;


 EXAMPLE 0

  ESA asm:

          lea.l      Buffer,a0
.air      expire d7 = BufLen
           clr.b     (a0)+
          nexp

  68k asm:

          lea.l      Buffer,a0
.air      move.w     BufLen,d7      ;counter initialization
.0000000
```

```
        clr.b       (a0)+
        dbra        d7,.0000000
```

  EXAMPLE 1

   ESA asm:

```
        expire d3=d3
         nop § nop § tst.l d1    ;ran out of fantasy...
        nexp,pl
```

   68k asm:

```
.0000001
        nop                         ;no init here!
        nop
        tst.l       d1
        dbpl        d3,.0000001   ;dbra with COND
```


1.41   what to say?!?

```
for ... to ... step ... next
```

 SYNTAX

```
  "for" CTR:var "=" ST:val "upto"|"dwto" END:val ["step" STP:val]
      ...
      ...
      ...
  "next"[SZ:jsize]
```


 MEANING 0 ("upto", STP>0)

```
  1. assigns the value of ST to the counter CTR
  2. if CTR>END, goes to 6
  3. executes the code "..."
  4. adds STP to CTR
  5. goes to 2
  6. first instruction after "next"
```


 MEANING 1 ("dwto", STP<0)

```
  2. if CTR<END, goes to 6
```


 NOTES

```
  - defaults: STP= 1 if "upto";
              STP=-1 if "dwto";
  - *NEVER* use STP=0!!! No check!
  - SZ is the size of the bcc instruction used (default: none);
  - size of CTR is its own;
```

```
       size of ST, END and STP is forced to be equal to CTR's;
    - never use "upto" with negative STP or "dwto" with positive STP!
    - it is necessary to declare the direction with "upto"/"dwto" because
      statically STP's sign is unknown. Direct checks in the generated co-
      de would produce even more unefficient code...
```

EXAMPLE 0

  ESA asm:

```
          for d4.b=#100 upto d6
           clr.l      (a0)+
          next.s
```

  68k asm:

```
          move.b     #100,d4           ;load CTR with ST
          move.b     d6,.0000002       ;store END
          move.b     #1,.0000002+4     ;default STP
          bra.s      .0000003
.0000002  dc.l       0,0               ;local variables (END,STP)
.0000003  cmp.b      .0000002,d4       ;compare CTR with END
          bgt        .0000004          ;exit if CTR>END
          clr.l      (a0)+
          add.b      .0000002+4,d4     ;update CTR
          bra.s      .0000003          ;repeat the loop
.0000004
```

EXAMPLE 1

  ESA asm:

```
          for tmp.w = d3 dwto #23 step NegStep[]
           move.l     (a1)+,(a2)+
          next

          bra        WhoKnowsWhere

          function NegStep[]:d1
          bsr        _rnd
          neg.l      d0
          efunc
```

  68k asm:

```
          move.w     d3,tmp            ;load CTR with ST
          move.w     #23,.0000002      ;store END
          bsr        f0000000          ;call NegStep[]
          move.w     d1,.0000002+4     ;store function result (STP)
          bra.s      .0000003
.0000002  dc.l       0,0               ;local variables (END,STP)
.0000003  move.l     a0,-(sp)          ;this quite complex way of
          exg.l      d0,a0             ;performing the boundary
          move.w     tmp,d0            ;check is caused by the fact
          cmp.w      .0000002,d0       ;that CTR is not a reg!
```

```
        exg.l       d0,a0
        movea.l     (sp)+,a0
        blt         .0000004            ;exit if CTR<END
        move.l      (a1)+,(a2)+
        move.l      d0,-(sp)            ;again, things get complicated!
        move.w      tmp,d0              ;using a reg for CTR would
        add.w       .0000002+4,d0       ;noticeably speed up this
        move.w      d0,tmp              ;part (see above)!
        move.l      (sp)+,d0
        bra         .0000003
.0000004

        bra         WhoKnowsWhere

f0000000                                ;NegStep[]
        bsr         _rnd
        neg.l       d0
f0000001  rts
```

1.42   just like Pascal!

```
repeat ... until ...
```

 SYNTAX

```
  "repeat"
      ...
      ...
      ...
  "until"[SZ:jsize] BL:boolexpr
```


 MEANING

```
  1. executes the code "..."
  2. evaluates BL
  3. if BL is false, goes to 1, else exits
```


 NOTES

```
  - the code is always executed at least once;
  - SZ is the size of the bcc instruction used (default: none);
```


 EXAMPLE

  ESA asm:

```
        moveq.l     #1,d0
        repeat
         add.b      d0,d0
        until.s #16=d0.b            ;silly, but works...
```

  68k asm:

```
            moveq.l    #1,d0

.000000A
            add.b      d0,d0
            cmpi.b     #16,d0
            seq.b      -(sp)            ;BL evaluation
            tst.b      (sp)+
            beq.s      .000000A         ;until
```

1.43   BASIC's 'while' ... 'wend'

```
while ... ewhile
```

 SYNTAX

```
  "while"[SZ:jsize] BL:boolexpr
       ...
       ...
       ...
   "ewhile"
```

 MEANING

```
  1. evaluates BL
  2. if BL is false, goes to 5
  3. executes the code "..."
  4. goes to 1
  5. 1st instruction after "ewhile"
```

 NOTES

```
  - if the 1st time BL is false, the code is never executed;
  - SZ is the size of the bcc instruction used (default: none);
```

 EXAMPLE

  ESA asm:

```
            while.s {Arale<d7.w}&{#Gacchan>d3}
             addq.l  #1,Arale
             add.l   Arale,d3
            ewhile                      ;don't try to find a meaning...
```

  68k asm:

```
.000000D    cmp.w    Arale,d7
            sgt.b    -(sp)
            cmpi.l   #Gacchan,d3
            slt.b    -(sp)
            move.l   d0,(-4,sp)
            move.b   (sp)+,d0
```

```
        and.b      d0,(sp)
        move.l     (-6,sp),d0          ;BL evaluation
        tst.b      (sp)+
        beq.s      .000000E            ;if while fails...
        addq.l     #1,Arale
        add.l      Arale,d3
        bra.s      .000000D            ;repeat loop
.000000E
```

1.44   jump table (branches)

```
on ... goto ...
```

 SYNTAX

  "on" V:val "," RX:reg "goto" ["safe"](S0:sym, S1:sym, ... , Sn:sym)

 MEANING 0 ("safe" not declared)

  1. evaluates V
  2. V=x and x<=n: the execution continues at the address Sx
     V=x and x>n : get ready for a GURU!!!

 MEANING 1 ("safe" declared)

  1. evaluates V
  2. V=x and x<=n: the execution continues atthe address Sx
     V=x and x>n : jumps to the first instruction after "on ... goto"

 NOTES

  - RX is the register which can be freely trashed to perform the jump;
  - RX's size is discarded;
  - V is loaded to RX only if V<>RX (obvious enough...);
  - the size of V can be only ".w" and ".l" (def.: ".w");
  - no check is done on SXes...

 EXAMPLE 0

  ESA asm:

```
        on d5,a6 goto (.shoot, .block, .pass, .jump
                    .steal, .dunk, .run, .fly )  ;very legal!!!
```

  68k asm:

```
        move.w     d5,a6                         ;get V
        jmp        ([.0000000,pc,a6.w*4])
.0000000   dc.l       .shoot,.block,.pass,.jump,.steal,.dunk,.run,.fly
```

```
EXAMPLE 1

  ESA asm:

          on UnitID.w,a2 goto safe (68k,Copper,Blitter,Paula)

  68k asm:

          move.w      UnitID,a2                    ;get V
          cmp.w       #$0004,a2                    ;is it valid?
          bhs         .0000001                     ;if not...
          jmp         ([.0000002,pc,a2.w*4])
.0000002  dc.l        68k,Copper,Blitter,Paula
.0000001
```

1.45   jump table (subroutines)

```
on ... gosub ...
```

```
 SYNTAX

  "on" V:var","RX:reg "gosub" ["safe"](S0:sym, S1:sym, ... , Sn:sym)
```

```
 MEANING 0 ("safe" not declared)

  1. evaluates V
  2. V=x and x<=n: jumps to the subroutine indicated by Sx
     V=x and x>n : get ready for a GURU!!!
  3. the code at the address Sx is expected to return with an "rts"
  4. execution goes on with the first instruction after "on ... gosub"
```

```
 MEANING 1 ("safe" declared)

  1. evaluates V
  2. V=x and x<=n: jumps to the subroutine indicated by Sx
     V=x and x>n : goes to 4
  3. the code at the address Sx is expected to return with an "rts"
  4. execution goes on with the first instruction after "on ... gosub"
```

```
 NOTES

  - RX is the register which can be freely trashed to perform the jump;
  - RX's size is discarded;
  - the size of V can be only ".w" and ".l" (def.:".w");
  - no check is done on SXes...
```

```
 EXAMPLE 0

  ESA asm:

Mangas     on Rumiko.w,a0 gosub (.ataru, .akane, .lum, .ranma)
```

```
   68k asm:

Mangas      move.w      Rumiko,a0
            jsr         ([.0000003,pc,a0.w*4])
            bra         .0000004                        ;skip jump table
.0000003    dc.l        .ataru,.akane,.lum,.ranma
.0000004
```

```
 EXAMPLE 1

  ESA asm:

            on fool.l,a3 gosub safe(
                            this
                            is
                            unquestionably
                            silly
                           )
```

```
   68k asm:

            move.l    fool,a3                 ;".l" is often useless!!!
            cmp.l     #$00000004,a3           ;safety check
            bhs       .0000005
            jsr       ([.0000006,pc,a3.l*4])
            bra       .0000005
.0000006    dc.l      this,is,unquestionably,silly
.0000005
```

```
 EXAMPLE 2

  ESA asm:

MyLife      on WhatIWillDo[],d0 gosub (code,PlayBBall,
                                    sleep,eat,study)
            bra.s     MyLife

            function WhatIWillDo[]:d0   ;d0'll get the def size (".l")
            repeat
            bsr _rnd
            until #4<>d0                ;eh, eh...
            efunc
```

```
   68k asm:

MyLife      bsr       f0000000                ;func call; no RX loaded
            jsr       ([.000000C,pc,d0.l*4])  ;note also the size!!!
            bra       .000000D
.000000C    dc.l      code,PlayBBall,sleep,eat,study
.000000D
            bra.s     MyLife

f0000000                                      ;nothing here because I
                                              ;didn't save any reg
```

```
.000000E
          bsr       _rnd
          cmpi.l    #4,d0
          sne.b     -(sp)
          tst.b     (sp)+
          bne       .000000E
f0000001  rts
```

1.46   much better than C's!

```
                switch ... -> ... eswitch
```

 SYNTAX

```
 "switch"[SZ:jsize] SW:rval
 "->" [CO:cmpop] V1:val
    ...
 ["->" [CO:cmpop] V2:val
    ...

  "->"
    ...

  "->" [CO:cmpop] Vn:val
    ...]
 ["def"
    ... ]
 "eswitch"
```

 MEANING

 1. executes the code contained between the brackets whose Vx is compa-
    red successfully to SW according to the condition CO specified (if
    CO is omitted, '=' is used as default);
    if the case that no condition is satisfied, the default code is
    executed (if "def" declared)
 2. jumps to the 1st instruction after "eswitch"

 NOTES

 - if one or more Vx potentially satisfy their own condition, only the
   code of the 1st one (starting from the top) is executed;
 - SZ is the size to be used for branches (bccs - default: none);
 - the "def" statement must be the last case;
 - to decide the case to execute, a series of comparisons between SW
   and the Vxs have to be done: the rules about their sizes (if diffe-
   rent) are explained
              here
              ;

 EXAMPLE

```
  ESA asm:

        switch.s WhatHasHappened.w

        -> #2
           lea.l  OhDamn,a0
           bsr    Say

        -> a0
           lea.l  WOWILIKEIT,a0
           bsr    Say

        -> >= xz
           bsr    GetUpset

        def
           move.l #"OKOK",answer
        eswitch

  68k asm:

        cmpi.w      #2,WhatHasHappened    ;1st comparison
        seq.b       -(sp)                 ;CO omitted, '=' used
        tst.b       (sp)+
        beq.s       .0000000              ;if not successful, go to next
        lea.l       OhDamn,a0             ;else execute the code inside
        bsr         Say

        bra.s       .0000001              ;then continue after switch
.0000000  cmpa.l    WhatHasHappened,a0    ;2nd comparison - please note
        seq.b       -(sp)                 ;that the size used is .l,
        tst.b       (sp)+                 ;cos aregs' size has priority
        beq.s       .0000002
        lea.l       WOWILIKEIT,a0
        bsr         Say

        bra.s       .0000001
.0000002  move.l    d0,(-6,sp)            ;3rd comparison
        move.w      WhatHasHappened,d0
         cmp.w      xz,d0
        sge.b       -(sp)                 ;CO is ">="
        move.l      (-4,sp),d0
        tst.b       (sp)+
        beq.s       .0000003              ;go to default case
        bsr         GetUpset

        bra.s       .0000001
.0000003
        move.l      #"OKOK",answer
.0000001
```

1.47 'if' ... 'else if' ... 'else' ... 'end if'

```
when ... owhen ... othw ... ewhen
```

SYNTAX

```
"when"[SZ:jsize] BLW:boolexpr
     ...
     ...
     ...
["owhen" BLO:boolexpr]
     ...
     ...
     ...
["othw"]
     ...
     ...
     ...
"ewhen"
```

MEANING

1. evaluates BLW
2. if BLW is true, executes the code between "when" and the following
   "owhen" or "othw" or "ewhen";
   then goes to 8
3. if any "owhen" is declared goes to 6
4. if "othw" is specified, executes the code between "othw" and "ewhen"
5. goes to 8
6. if BLO is true, executes the code between "owhen" and the following
   "owhen" or "othw" or "ewhen";
   after that goes to 8
7. repeats from step 3
8. execution continues after "ewhen"

NOTES

- SZ is the size to be used for branches (bccs – default: none);
- there can be as many "owhen"s as you want;
- "othw" can be declared only once and after any "owhen" statement;

EXAMPLE 0

ESA asm:

```
        when.s  ~{d0.w ^ ~d1.b}
         bsr        OhDamn
        ewhen
```

68k asm:

```
        tst.w      d0
        seq.b      -(sp)
        tst.b      d1
        sne.b      -(sp)
        move.l     d0,(-4,sp)
        move.b     (sp)+,d0
        EOR.b      d0,(sp)
```

```
              not.b        (sp)
              move.l       (-6,sp),d0        ;BL evaluation
              tst.b        (sp)+
              beq.s        .000000F          ;if false condition...
              bsr          OhDamn
.000000F                                     ;...jump here!
```

EXAMPLE 1

  ESA asm:

```
              when rains
               bsr          OpenUmbrella
              othw
               bsr          PutOnSunGlasses
              ewhen
```

  68k asm:

```
              tst.l        rains
              sne.b        -(sp)             ;BL evaluation
              tst.b        (sp)+
              beq          .0000011          ;jump performed when false
              bsr          OpenUmbrella
              bra          .0000010          ;skip "othw" section
.0000011
              bsr          PutOnSunGlasses
.0000010
```

EXAMPLE 2

  ESA asm:

```
              when.s d0=d1
               nop
              owhen d1<d2
               nop § nop
              owhen d3>d4
               nop § nop § nop
              othw
               bsr          DoSomething
              ewhen
```

  68k asm:

```
               cmp.l        d1,d0
              seq.b        -(sp)             ;d0=d1?
              tst.b        (sp)+
              beq.s        .0000001          ;if not...
              nop
              bra.s        .0000000          ;exit
.0000001      cmp.l        d2,d1
              slt.b        -(sp)             ;d1<d2?
              tst.b        (sp)+
              beq          .0000002          ;if not...
```

```
        nop
        nop
        bra.s      .0000000        ;exit
.0000002   cmp.l     d4,d3
        sgt.b      -(sp)           ;d3>d4?
        tst.b      (sp)+
        beq        .0000003        ;if not...
        nop
        nop
        nop
        bra.s      .0000000        ;exit
.0000003
        bsr        DoSomething     ;default case
.0000000
```

1.48   defining functions

```
            function
```

SYNTAX

```
 "function" ["loc:"] NAME:sym "[" [RL1:regslist] "]" ["," RL2:regslist] ":" OUT: ↵
    var
      ...
      ...
      ...
 "efunc" [',' RESULT:val]
```

MEANING

1. a label is defined as the entry point of the function
2. if RL2 is declared, the registers are stored in the stack with a
   movem
3. the code "..." is copied (and processed, of course)
4. if RESULT is specified, it is copied to OUT (with OUT's size)
5. if RL2 is specified, the registers are restored from the values
   previously saved in the stack (another movem)
6. rts is put at the end of the function

NOTES

 - RL1 tells ESA how to assign the arguments when this function is

            called
            ;
 - OUT tells ESA where to get the function's result from;
 - pay attention to RL2 and OUT!!! RL2 *SHOULD NOT* contain OUT, if OUT
   is a reg (*no* check)!!!
 - "function" must be separated from NAME by one or more spaces/TABs,
   otherwise "functionNAME" would be acknowledged as an instruction/
   /macro/etc...
 - the exit point of the function is marked by a label to allow the

```
                   forced exit from the func
                   ;
 - normally functions' labels are global (
                   whatever char has been

                   chosen
                    for labels); instead, if "loc" is declared, the function
   definition will be "local", i.e. its labels will start with '.';
 - NAME can be up to 30 char long;
 - don't put a label on the same line of "function" (why should you
   enter a func in that way?!?);
 - size of OUT is used only if inside a boolexpr;
 - ESA won't check for repetitions of function names;
 -
                   wondering why you have to use '[',']'-type brackets?
                    EXAMPLE 0


   ESA asm:


           function SetDMA[d0.w],d1:d0
           move.w     $dff002,d1
           ori.w      #$8000,d0
           move.w     d0,$dff096
           move.w     d1,d0
           efunc


   68k asm:


f0000000   movem.l    d1,-(sp)                ;save regs in RL2
           move.w     $dff002,d1
           ori.w      #$8000,d0
           move.w     d0,$dff096
           move.w     d1,d0
f0000001   movem.l    (sp)+,d1
           rts



 EXAMPLE 1

   ESA asm:


           function GetMess[], d0-d7/a0-a6 :MessAmount.b
           lea.l      TileTable,a0
           bsr        MessWithRegs
           move.b     (a5),MessAmount
           efunc


   68k asm:


f0000002   movem.l    d0-d7/a0-a6,-(sp)
           lea.l      TileTable,a0
           bsr        MessWithRegs
           move.b     (a5),MessAmount
f0000003   movem.l    (sp)+,d0-d7/a0-a6
           rts
```

```
EXAMPLE 2

  Go
              here
               to learn a way of using local definitions.



EXAMPLE 3

ESA asm:


        function MessWithDMA[],d0:d1
        bsr         _Rnd                    ;let's get a random d0...
        efunc , SetDMA[d0]                  ;... and watch some fireworks!


 68k asm:


f0000004   movem.l    d0,-(sp)
        bsr         _Rnd
        bsr         f0000000    ;see example 0
        move.l      d0,d1       ;return SetDMA[] retcode
f0000005   movem.l    (sp)+,d0
        rts
```

1.49   calling functions

```
              Calling a Function

 SYNTAX

  NAME:sym [SZ:jsize] "[" [ ["sav:"] PARAMS:args] "]"


 MEANING

  1. if "sav:" is declared, stores the RL1 registers (declared in the

              function definition
              ) in the stack
  2. loads to RL1 the parameters passed inside the brackets
  3. executes function code
  4. after the execution of NAME (if "sav:" is declared, the registers
     of RL1 are restored) the program continues with the 1st instruction
     after this call


 NOTES

  - a function can be called only as an argument of an asm instruction
    or ESA construction, i.e. you can't put it in the label/instruction
    fields;
  - SZ is the size to be used for the bsr (default: none);
  - when SZ=".l", the instruction jsr is used instead of bsr.l to easily
    allow calls to other code sections;
  - since ESA is fully orthogonal, funcs can be used everywhere their re-
```

```
   turn type (
              var
              ) is expected to be found;
 - when "sav:" declared make sure that OUT (returned by the function),
   if reg, is not included in RL1;
 - be extremely cautious when calling functions inside other ESA con-
   structs, as you could accidentally trash some variables/registers!
 -
              wondering why you have to use '[',']'-type brackets?
               EXAMPLE 0
```

ESA asm:

```
        move.w
            SetDMA.l[#$f]
            ,OldDMA      ;1st
        move.w    SetDMA[sav:#$f],OldDMA                         ;2nd
```

68k asm:

```
        move.w    #$f,d0              ;load arg
        jsr       f0000000
        move.w    d0,OldDMA           ;1st OK!
        movem.l   d0,-(sp)            ;"sav:" used in the 2nd
        move.w    #$f,d0
        bsr       f0000000
        movem.l   (sp)+,d0            ;WRONG! the result
        move.w    d0,OldDMA           ;is lost!!!
```

EXAMPLE 1

ESA asm:

```
        bool #24=
            GetMess[]
            ,d7         ;compound call!
```

68k asm:

```
        bsr       f0000002           ;execute function
        cmpi.b    #24,MessAmount
        seq.b     -(sp)              ;BL evaluation
        move.b    (sp)+,d7           ;result
```

1.50   premature exit from a procedure or function

pop

 SYNTAX

 "pop"[SZ:jsize]

 MEANING

1. the last procedure/function being defined is forced to terminate
   (a jump to the end label is performed)

 NOTES

 - SZ is the size to be used for the bra (default: none);
 - make sure that the sp is in the same position when the proc/func was
   entered, otherwise a crash is almost sure!
 - if inside a func, don't forget about the return value...

 EXAMPLE 0

  ESA asm:

```
        procedure UpperCase[a0/d0],d0-d1/a0
        IFNE        TEST_ON                     ;if we're in test mode,
        pop.s                                   ;we wanna do nothing...
        ENDIF
        moveq.l     #$df,d1
        subq.l      #1,d0
        expire d0=d0
         and.b        d1,(a0)+
        nexp,eq
        eproc
```

  68k asm:

```
p0000000   movem.l     d0-d1/a0,-(sp)
           IFNE        TEST_ON
           bra.s       p0000001                 ;jump to exit label
           ENDIF
           moveq.l     #$df,d1
           subq.l      #1,d0

.0000002
           and.b       d1,(a0)+
           dbeq        d0,.0000002
p0000001   movem.l     (sp)+,d0-d1/a0
           rts
```


 EXAMPLE 1

  ESA asm:

```
        procedure StrangePlot[a0],d0-d1/a0

        expire d0=#199
         move.b     fx[d0],(a0)+
        nexp

        pop                                     ;fx *MUST* be skipped!!!

        function loc:fx[d1]:d1                  ;local func definition:
```

```
          mulu.w      d1,d1                      ;as StrangePlot[] is glo-
          eori.l      RndSeed,d1                 ;bal, fx[] isn't visible
          efunc                                  ;externally

          eproc
```

```
  68k asm:
```

```
p0000000  movem.l     d0-d1/a0,-(sp)

          move.w      #199,d0
.0000004
          move.l      d0,d1
          bsr         .0000002
          move.b      d1,(a0)+
          dbra        d0,.0000004

          bra         p0000001

.0000002
          mulu.w      d1,d1
          eori.l      RndSeed,d1
.0000003  rts

p0000001  movem.l     (sp)+,d0-d1/a0
          rts
```

1.51   defining procedures

```
                procedure
```

```
 SYNTAX
```

```
  "procedure" ["loc:"] NAME:sym "[" [RL1:regslist] "]" ["," RL2:regslist]
     ...
     ...
     ...
  "eproc"
```

```
 MEANING
```

```
  1. a label is defined as the entry point of the procedure
  2. if RL2 is declared, the registers are stored in the stack with a
     movem
  3. the code "..." is copied (and processed, of course)
  4. if RL2 is specified, the registers are restored from the values
     previously saved in the stack (another movem)
  5. rts is put at the end of the procedure
```

```
 NOTES
```

```
  - RL1 tells ESA how to assign the parameters when this procedure is
```

```
                    called
                    ;
  - movems size is always long;
  - size of RL2 is always ".l";
  - "procedure" must be separated from NAME by one or more spaces/TABs,
    otherwise "procedureNAME" would be acknowledged as an instruction/
    /macro/etc...
  - the exit point of the procedure is marked by a label to allow the

                    forced exit from the proc
                    ;
  - normally procedures' labels are global (
                    whatever char has been

                    chosen
                     for labels); instead, if "loc" is declared, the procedure
    definition will be "local", i.e. its labels will start with '.';
  - NAME can be up to 30 char long;
  - don't put a label on the same line of "procedure" (why should you
    enter a proc in that way?!?);
  - ESA won't check for repetitions of procedure names;
  -
                    wondering why you have to use '[','[']'-type brackets?
                     EXAMPLE 0


  ESA asm:


          procedure loc: WaitMouse[]
.w        btst.b      #6,$bfe001
          bne.s       .w
          eproc


  68k asm:


.0000002                                        ;local labels
.w        btst.b      #6,$bfe001
          bne.s       .w
.0000003  rts



 EXAMPLE 1


  ESA asm:


          procedure SlowClr[a0/d0.b],a0/d1
          move.l      d0,d1
          lsr.l       #2,d1
          subq.l      #1,d1
.c        clr.l       (a0)+
          dbra        d1,.c          ;from "Writing Bad Code", Chapter 1
          eproc


  68k asm:


p0000000  movem.l     a0/d1,-(sp)    ;save regs in RL2
          move.l      d0,d1
          lsr.l       #2,d1
```

```
            subq.l      #1,d1
.c          clr.l       (a0)+
            dbra        d1,.c
p0000001    movem.l     (sp)+,a0/d1
            rts
```

EXAMPLE 2

  Go

              here
               to learn a way of using local definitions.

1.52   calling procedures

                Calling a Procedure

 SYNTAX

  NAME:sym [SZ:jsize] "[" [["sav:"] PARAMS:args] "]"

 MEANING

  1. if "sav:" is declared, stores the RL1 registers (declared in the

                procedure definition
) in the stack
  2. loads to RL1 the parameters passed inside the brackets
  3. executes the proc code
  4. after the execution of NAME (if "sav:" is declared, the registers
     of RL1 are restored) the program continues with the 1st instruction
     after this call

 NOTES

  – procedure calls can only be put in the instruction field;
  – SZ is the size to be used for the bsr (default: none);
  – when SZ=".l", the instruction jsr is used instead of bsr.l to easily
    allow calls to other code sections;
  – if one of the args matches exactly the corrispondent destination re-
    gister in RL1, no "move" is done!
  –
                wondering why you have to use '[','] '-type brackets?
                 EXAMPLE 0

  ESA asm:

```
            WaitMouse.s[]
            bra         SomewhereElse       ;avoid "collisions" with procs

            procedure loc:WaitMouse[]
.w          btst.b      #6,$bfe001
            bne.s       .w
```

```
          eproc

  68k asm:

          bsr.s       .0000000
          bra         SomewhereElse

.0000000
.w        btst.b      #6,$bfe001
          bne.s       .w
.0000001  rts
```

 EXAMPLE 1

  ESA asm:

```
          SlowClr[ sav: #buffer , d1]
          bra         SomewhereElse

          procedure SlowClr[a0/d0.b],a0/d1
          move.l      d0,d1
          lsr.l       #2,d1
          subq.l      #1,d1
.c        clr.l       (a0)+
          dbra        d1,.c           ;from "Writing Bad Code", Chapter 1
          eproc

  68k asm:

          movem.l     a0/d0,-(sp)     ;"sav:" -> save regs in RL1
          move.l      #buffer,a0
          move.b      d1,d0           ;.b according to declaration
          bsr         p0000000        ;call proc
          movem.l     (sp)+,a0/d0
          bra         SomewhereElse

p0000000  movem.l     a0/d1,-(sp)
          move.l      d0,d1
          lsr.l       #2,d1
          subq.l      #1,d1
.c        clr.l       (a0)+
          dbra        d1,.c
p0000001  movem.l     (sp)+,a0/d1
          rts
```

 EXAMPLE 2

  ESA asm:

```
          SlowClr.l[sav:#Buffer,d0]      ;same proc as above

  68k asm:

          movem.l  a0/d0,-(sp)
          move.l   #Buffer,a0            ;only a0 loaded!
```

```
        jsr     p0000002                ;jsr instead of bsr
        movem.l (sp)+,a0/d0
```