PowerD

COLLABORATORS

	TITLE : PowerD	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 25, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

PowerD

1.1 main

```
PowerD v0.06 (21.11.1999) by Martin Kuchinka
(please, excuse my poor english)

About this document:
 Information

Important information
 Rules of programming in PowerD
 I need some help
 Um, what does it do???

How to write Your own programs in PowerD:
 Immediate values
 Immediate strings
 Constant definition
 Variable definition
 Macro definition
 Types, Pointers, Arrays
 OBJECT definition
 Equations
 Single variable operators
 Constant Equations
 Function using
 Returning values
 Procedure definition
 Using MODULEs
 External MODULEs
 Exceptions
 Global data
 LOOP definition
 FOR definition
 WHILE definition
 REPEAT definition
 IF definition
 SELECT definition
 DO keyword
 THEN keyword
 EXIT/EXITIF keyword
```

1.2 info

Writing this document was/is most difficult work on whole PowerD, I'm not a writer, so
please excuse my bad english and in some(many) parts quite short description. I think,
that if someone want to start programming, (s)he will learn most from examples ...

Here follows some information what describes the dosument format:

Labels are black and bold.
Code parts are white. Special keywords are bold.
Optionals are closed in black []. If there are more optionals, they are separated by /.
Requied optionals are closed in black () and separated by black /.

```
PROC main()
  DEF[L/UL] a,b=10
  FOR a:=0 (TO/DTO) b [STEP 2]
    PrintF('a=\d\n',a)
  ENDFOR
ENDPROC
```

1.3 PowerD.guide - Rules of programming in PowerD

```
  Here you will get hints, how to write as most efficient as possible, how to ↩
     use new
features etc. PowerD contains many improvements what other languages misses, but ↩
     there
are no elements of other languages like in AmigaE (list, ada, etc.), PowerD has ↩
    very
flexible syntax, and allows you to short you source a lot, most important thing ↩
    is DO
keyword I think:

  AmigaE:

b:=Rnd(10)-10
WHILE a<20
EXIT b=15
ENDWHILE
IF b=15 THEN PrintF('Yes\n')

  PowerD:

b:=Rnd(10)-10
WHILE a<20
EXITIF b=15 DO PrintF('Yes\n')
ENDWHILE
```

1.4 help

```
  Some parts of this document are currently not done, just wait. Something ↩
     written in
this document does not work, please email me about it, PowerD is quite large ( ↩
    about 8000
lines=270kb of source code written in AmigaE) and I'm only man, I can't know ↩
    everything,
if You will tell me about that error, I will eliminate it as soon as possible.

  As I said, this is quite large project, it takes very much time (especially ↩
     finding
errors). So, if You want to help me by writing support programs, please email me ↩
    . I
very lack an inteligent c-header-to-d-module converter, it is extremely boring ↩
    work to
rewrite all those c-headers manually. In near future I want to add gui something ↩
     like
```

StormC has. Maybe VisualD. Everyone who can/want/will help me is welcome.

If you have some ideas/bug-reports/suggestions/etc. please email me.

1.5 what

If you see something like:
PowerD v0.0: Generating(100)...
it compiles your source.

If you see something like:
PowerD v0.0: Generating(100) in intuition/intuition...
it reads modules.

Then you can see something like:
PowerD v0.0: Generating...
it regenerates lists of OBJECTs and adds links between OBJECTs.

Then you can see something like:
PowerD v0.0: Writing(12%)...
it writes assembler source.

Then you can see something like:
PowerD v0.0: Cleaning...
it frees all memory allocations.

Then you can see something like:
PowerD v0.0: Compiling...
it executes phxass to compile assembler source.

Then you can see something like:
PowerD v0.0: Linking...
it executes phxlnk to link startup header, object files and link libraries.

Then you can see something like:
PowerD v0.0: Done.
if everything went ok, or:
PowerD v0.0: Not Done.
if not.

1.6 values

Decimal values:

[-][0..9][0..9]...
limitation: min: -(2^31), max: +(2^31)-1
eg.: 1, -12, 123, 0002, -01234

Hexadecimal values:

```
  [-]$[0..9|a..f][0..9|a..f]...
limitation:
    signed: min: $80000000, max: $7fffffff
  unsigned: min: $00000000, max: $ffffffff
eg.: $1, -$32, $ffab, $abcdef01, $002d


Binary values:

  [-]%[0|1][0|1]... (you can use upto 32 bits)
eg.: %1, %00001101, %10101011, %11001100


ASCII values:

  [-]"#"
where # is arbitrary string maximaly 4 characters long
eg.: "A", "AHOJ", "J\no\0", "ok23", "1234"


Float values:

  [-]#1"."#2[e#3|E#3]
where #1 is number before point, #2 is number after point, #3 is exponent
(see: Types for limitations)
In PowerD are all float numbers converted to DOUBLEs, and then it is used
how is it better (DOUBLEs, FLOATs, LONGs, ...)
```

1.7 strings

```
  Special characters:

\\        - backslash "\"
\a or ''  - apostrophe "'" (\a only for AmigaE compatibility)
\b        - return (ascii 13)
\e        - escape (ascii 27)
\n        - linefeed (ascii 10)
\q or "   - double quote """ (\q only for AmigaE compatibility)
\t        - tabulator (ascii 9)
\v        - vertical tabulator (ascii 11)
\!        - bell (ascii 7)
\0        - zero byte (ascii 0), end of string

\jx       - single character where x is number (0-255) of character you want.


  Formating characters:

\d        - decimal number
\h        - hexadecimal number
\c        - single character
\u        - unsigned decimal number
\s        - string

\l        - used before \s, \h, \d, \u, means left justified
```

```
\r        - used before \s, \h, \d, \u, means right justified
\z        - used before \h, \d, \u with field definition (see below) creates ↩
   leading
            zeros

You can ofcourse use c-like string format, but the string must start and stop ↩
   with "'"
(apostrophe), not """ (double quote)
```

Field definition in strings (usable only after \s, \d, \h and \u):

```
[#]        - where # is number of characters to be used for a formating
             character
```

Examples of normal strings:

```
'bla'
'Hello world!\n'
```

Examples of formatting strings (use them with PrintF(), StringF() and similar functions):

```
PrintF('a+b=\d\n',a+b)
PrintF('file ''\s'' not found.\n',filename)
PrintF('Address is $\z\h[8]\n',adr)
```

Examples of \jx using:

```
'Hello\j10'           // is the same as 'Hello\n'
'Test \j12345'        // is the same as 'Test {45'
'\j999'               // is the same as 'c9'
```

1.8 const

Description:

Constants are in PowerD defined with one of following keyword: CONST, ENUM, SET, FLAG.

```
Constants can be defined nearly everywhere you like in/out-side a procedure.
```

Syntax of CONST keyword:

```
CONST name=value[,name2=value2]...

The values can be LONGs or DOUBLEs or their equations
```

Examples:

```
CONST COUNT=10,
```

```
      LISTSIZE=COUNT*SIZEOF_LONG,
      PI=3.1415926
```

Syntax of ENUM keyword:

```
ENUM name[=value][,name2[=value2]]...

ENUM generates list of constants where each next constant is increased by one. ↩
   Values
must be LONGs.
```

Examples:

```
ENUM YES=-1,NO,MAYBE              // YES=-1,NO=0,MAYBE=1
ENUM WHAT,IS,YOUR,NAME,           // WHAT=0,IS=1,YOUR=2,NAME=3
     MY=10,NAME,IS,PRINCE         // MY=10,NAME=11,IS=12,PRINCE=13
```

Syntax of SET keyword:

```
SET name[=value][,name2[=value2]]...

where values are for 32bit numbers from 0 to 31. Each next constant has its bit ↩
   shifted
left by one. (respectively it is multiplied by two)
```

Examples:

```
SET VERTICAL,                     // VERTICAL=1
    SMOOTH,                       // SMOOTH=2
    DIRTY                         // DIRTY=4
SET CLEAN=5,                      // CLEAN=32
    FAKE,                         // FAKE=64,
    SLOW=10                       // SLOW=1024
```

Syntax of FLAG keyword:

```
FLAG n_ame[=value][,n_ame2[=value2]]...

where n_ame is normal name, but it MUST contain "_" character (eg.: AG_Member, ↩
   FI_Open).
FLAG generates two constants from each the first is same as in SET case, but "F" ↩
    is
added before the first "_" character and the second is like ENUM, but "B" is ↩
   added
before the first "_" character. Values are the same as in SET case.
```

Examples:

```
FLAG CAR_Fast,                    // CARF_Fast=1,      CARB_Fast=0
     CAR_Auto,                    // CARF_Auto=2,      CARB_Auto=1
     CAR_Comfort,                 // CARF_Comfort=4,   CARB_Comfort=2
     CAR_Expensive                // CARF_Expensive=8, CARB_Expensive=3
```

1.9 def

```
  Description:

Variables can be defined with "DEF" keyword, nearly every where you like,  ↩
   outside a
procedure they are global, inside a procedure they are local.

  Syntax:

DEF[L/UL/W/UW/B/UB/F/D/S] name[[field]][=default][:type]...

DEF name                  // name is VOID/LONG
DEF name:type             // name is type
DEF name[]:type           // name is PTR TO type
DEF name[][]:type         // name is PTR TO PTR TO type
DEF name[a]:type          // name is array of a types
DEF name[a,b]:type        // name is array of a*b types with width of a
DEF name[:a]:type         // name is PTR TO type of width a
DEF name[:a][:b]:type     // name is PTR TO type of width a and height b
etc.

(See  Types  to get info about ":" character between brackets)


EDEF name[:type]          // for external variables (see  Multiple source  ↩
   projects )


  Simplier/Faster variable definition:

DEFL  name        is same as  DEF name:LONG
DEFUL name        is same as  DEF name:ULONG
DEFW  name        is same as  DEF name:WORD
DEFUW name        is same as  DEF name:UWORD
DEFB  name        is same as  DEF name:BYTE
DEFUB name        is same as  DEF name:UBYTE
DEFF  name        is same as  DEF name:FLOAT
DEFD  name        is same as  DEF name:DOUBLE
DEFS  name[x]     is same as  DEF name[x]:STRING


  Default values:

  Syntax:

DEF name=value[:type]...  // where value is a number/constant/list/string

in local variables is also possible:

DEF name=result[:type]... // where result can be other variable, equation or  ↩
   something
                          // what returns a value/pointer

Each variable can have its initial value/list/string:
```

```
DEF num=123:LONG,
    float=456.789:FLOAT,
    name='Hello World!\n':PTR TO CHAR,
    list=[12,23,34,45,56]:UWORD

or more complex:

DEF num=12*3+232/6:LONG,
    float=31/2.2+76.3:FLOAT,
    name='Hello '+
         'Amigans!\n',
    list=[[1,2,3]:LONG,[4,5,6]:LONG,[7,8,9]:LONG]:PTR TO LONG

Variables do not must be defined before they are used, if they are global it is
definitely unimportant, if they are local there are some limitations:

PROC main()
   PrintF('n=\d, m=\d\n',n,m)
   DEF n=1
ENDPROC
DEF m=2

is same as:

PROC main()
   DEF n
   PrintF('n=\d, m=\d\n',n,m)
   n:=1
ENDPROC
DEF m=2

it means, that if you want to give to variable default value (n=1), the value ↩
   will
be given on the place where it is defined (in our case after it was printed)
```

1.10 macro

```
It can be used same way as in AmigaE or C/C++. In PowerD must be #define keyword ↩
    used
only as global macros (outside a PROC) Macros are replaced only between PROC and ↩
    ENDPROC.
This means that You can't use macros like eg. here:

PROC a(b,c) IS macro(b,c)

PROC a(b,c)
ENDPROC macro(b,c)

You have to do it this way:

PROC a(b,c)
   DEF r
   r:=macro(b,c)
ENDPROC r
```

```
  each leading "#" MUST start at the beginning of the line (right after linefeed), ↩
      rest of
  keyword (define, if, endif, ...) must be the first word on line:

  Good:

#ifdef DEBUG
#  define DODEBUG
#else
#  define NODEBUG
#endif

  Bad (syntax error):

#ifdef DEBUG
  #define DODEBUG
#else
  #define NODEBUG
#endif


    Known keywords:

  #define name data
    - each occurence of 'name' will be replaced with 'data'
    - example:

    #define Hello PrintF('Hello\n')
    PROC main()
       Hello
    ENDPROC

    is the same as:

    PROC main()
       PrintF('Hello\n')
    ENDPROC

  #define name(args) data ...args ... data
    - where args is list of names eg.: #define name(a,b,c,d)
    - example:

    #define AddThree(a,b,c) (a*b*c)
    a:=AddThree(1,2,3)

    is the same as:

    a:=1*2*3
```

1.11 types

```
    Known types:
```

```
Name:   Short:      Length:   Min:                    Max:                    ↩
   Epsilon(Accuracy):

BYTE    B           1         -128                    +127                    1
UBYTE   UB          1          0                       255                    1
WORD    W           2         -32768                  +32767                  1
UWORD   UW          2          0                       65535                  1
LONG    L           4         -2147483648             +2147483647             1
ULONG   UL          4          0                       4294967296             1
FLOAT   F           4          1.17549435e-38          3.40282347e+38          ↩
   1.19209290e-07
DOUBLE  D           8          2.225073858507201e-308  1.797693134862316e+308  ↩
   2.2204460492503131e-16
BOOL    -           2          0                       non zero               -
PTR     -           4                  32bit address                           -

PTR TO BYTE         4                  32bit address                           -
...
PTR TO PTR TO BYTE  4                  32bit address                           -
...
CHAR    B           1            only for AmigaE compatibility                 1
INT     W           2            only for AmigaE compatibility                 1
```

  Multiple pointers:

```
If you would like to use more then two dimensional fields, you cant do it like  ↩
   above:
  field:PTR TO PTR TO PTR TO PTR TO ...
You have to do it like here:
  field[][][][]:LONG
  field[][]:PTR TO PTR TO LONG
(these are the same)
```

  Multiple arrays:

```
DEF field[10,20]:LONG
field[3,4]:=123

is the same as:

DEF field[10*20]:LONG
field[4*10+3]:=123
```

  Multiple arrays through pointers:

```
The two examples above allocates 10*20*SIZEOF_LONG bytes of memory, but you can  ↩
   do it
also without memory allocation (good when using fields as arguments in  ↩
   PROCedures):
Is enough to add before the first field size specification character ":"

DEF field[:10,:20]:LONG
...
memory allocation for field
```

```
...
field[3,4]:=123

is the same as:

DEF field:PTR TO LONG
...
memory allocation for field
...
field[4*10+3]:=123

This allocates nothing, but stores information about field width.
```

1.12 object

```
  Description:

Object is something like field of types or typed memory.


  Syntax:

OBJECT name [OF objectname]
  var[[size]][:type],
  ...


  Multiple name:

Each item in object can have upto 16 names, all of these must be separated by ↩
   '|' sign.

OBJECT Point
  X|x|R|r:FLOAT,
  Y|y|G|g:FLOAT,
  Z|z|B|b:FLOAT


  Unions:
  This is very useful, if you want to use one object to store different types of ↩
      values
in same object but different memory block.

OBJECT Help
  Type:UWORD,             // help type
  NEWUNION AmigaGuide     // amigaguide help
    File:PTR TO UBYTE,    // file name
    Node:PTR TO UBYTE     // node name
  UNION LocalHelp         // inlined help
    Text:PTR TO UBYTE,    // pointer to text
    Length:UWORD          // length of the text
  ENDUNION,               // end of the union
  HelpTitle:PTR TO UBYTE  // title of the help
```

```
  This will generate have length of 14 bytes: Type has 2 bytes, each UNION ↩
     between
NEWUNION and ENDUNION has the same start offset (in this case it is 2). Each ↩
    UNION
starts on even address, so if the address is odd, one byte is skipped. Then ↩
    PowerD
finds the longest UNION and adds it's length to the UNION offset (in this case ↩
    has
AmigaGuide 8 bytes and LocalHelp 6 bytes, 8 bytes used).  Next item starts on ↩
    this
address.

ATTENTION: see the commas, those have to be used exactly.

  Pad bytes:

Each non BYTE/UBYTE item must start on even address:

OBJECT xxx               // SIZEOF_xxx = 6 bytes
  a:BYTE,                // offset=0
  b:BYTE,                // offset=1
  c:BYTE,                // offset=2
  d:WORD                 // offset=4


  Linked objects:

OBJECT PointList OF Point
  Next:PTR TO Point,
  Prev:PTR TO Point

is the same as:

OBJECT PointList
  X|x|R|r:FLOAT,
  Y|y|G|g:FLOAT,
  Z|z|B|b:FLOAT,
  Next:PTR TO Point,
  Prev:PTR TO Point


  Object sizes:

  With each object is generates one constant called SIZEOF_xxx, where xxx is ↩
     object
name, this constant contains the object length in bytes.
```

1.13 equa

```
You can use equations with decimal numbers only, float numbers only and ↩
    combinations


  Operator priorities:
```

```
Operators with higher priority will be processed before operators with lower  ↩
   priority:
x:=1+2*3       // 2 will be multiplied with 3, result will be added to 1 and  ↩
   result
               // will be copied to x.
x:-=1<<2*3     // 1 will be shifted by 2 to the left, result will be multiplied  ↩
   by 3
               // and result will be subtracted from x.
```

Operator	Name	Priority	Comment
+	Plus	1	
-	Minus	1	
*	Multiply	2	
/	Divide	2	
\	Modulo	2	
|	Bit OR	3	Possible only with decimals
!	Bit EOR	3	Possible only with decimals
&	Bit AND	3	Possible only with decimals
<<	Shift Left	4	Possible only with decimals
>>	Shift Right	4	Possible only with decimals
<| or |<	Rotate Left	4	Possible only with decimals
>| or |>	Rotate Right	4	Possible only with decimals

Assigning operators:

Operator	Name	Comment
:=	Copy	
+=	Add	
-=	Subtract	
*=	Multiply	
/=	Divide	
\=	Modulo	
|=	Bit OR	Possible only with decimals
!=	Bit EOR	Possible only with decimals
&=	Bit AND	Possible only with decimals
~=	Copy NOTed	Possible only with decimals
<<=	Shift Left	Possible only with decimals
>>=	Shift Right	Possible only with decimals
<|= or |<=	Rotate Left	Possible only with decimals
>|= or |>=	Rotate Right	Possible only with decimals
:=:	Swap	This is only for same types

Equation examples:

```
DEF a,b,c
a:=10                       // a=10
b:=a\4                      // b=2
c:=a>>2                     // c=2
a+=b+3*c                    // a=18
a:=:c                       // c=18, a=2
b:=a+3+c-=12                // c=6, b=11
```

1.14 single

```
Operator      Name            Comment
+             Useless         Only for you :^)
-             NEGation
~             NOTation(?)
&             Address
++            Addition        Possible multiple subtractions (see below)
--            Subtraction     Possible multiple subtractions (see below)


  Negation:

a:=-4
b:=-a                         // b=4


  Notation:

Returns inversed (bit) variable

a:=16                         // a=$00000010
b:=~a                         // b=$fffffef
b~=a                          // b=$fffffef


  Address:

Returns address of the variable

b:=&a                         // b contains address of a

ATTENTION:
b&=a                          // this is not an address, but Bit AND


  On address:

Returns long on address in the variable

b:=^a                         // b contains long on address in a


  Post/Pre addition/subtraction:

If ++ or -- are after the variable then returned value is the contain of  ↩
   variable, then
is the (number of ++ or -- minus 1)*1 added/subtracted to/from the variable.
If ++ or -- are before the variable then is the (number of ++ or -- minus 1)*1  ↩
   added/
subtracted to/from the variable and result is the returning value.

a:=10                         // a=10
a--                           // a=9
a----                         // a=6
a++                           // a=7
```

```
b:=a++                          // b=7, a=8
b:=+++a                         // b=10, a=10
```

1.15 constequa

Constant equations are the same as Equations , but destination and all members must be constants. Only possible assign operator is "=" and it can be used only with CONST, ENUM, SET and FLAG keywords.

Constant functions:

Syntax	Name	Comment
SIN(a)	Sinus	Floats only
COS(a)	Cosinus	Floats only
TAN(a)	Tangents	Floats only
ASIN(a)	Arcus sinus	Floats only
ACOS(a)	Arcus cosinus	Floats only
ATAN(a)	Arcus tangents	Floats only
SINH(a)	Hyperbolic sinus	Floats only
COSH(a)	Hyperbolic cosinus	Floats only
TANH(a)	Hyperbolic tangents	Floats only
EXP(a)	Exponent	Floats only
LN(a)	Natural logarithm	Floats only
LOG(a)	Logarithm with base of 10	Floats only
RAD(a)	Degree to radian	Returns only float
ABS(a)	Absolute value	
NEG(a)	Negate value	
FLOOR(a)	Floor value	Floats only
CEIL(a)	Ceil value	Floats only
POW(a,b)	Power	Floats only
SQRT(a)	Square root	Floats only
FAC(a)	Factorial	

1.16 func

Description:

PowerD can use currently three types of functions, first are library functions , second are procedures and third are linked library functions. Each of these is defined in other way, but they all can be used alike. Functions can be stand alone like:

```
Function(a,b,c)
```

or functions, that returns one or more values:

```
x:=Function(a,b,c)
x,y,z:=Function(a,b,c)
```

1.17 return

Description:

In PowerD is able to return one or more values from not only functions, it is possible from FOR, WHILE, IF etc. Each of these has rather different syntax for returning values.

1.18 proc

Description:
How to describe procedure? I really don't know...

Syntaxes:

```
PROC name([list of typed arguments])[(list of typed results)] IS result

PROC name([list of typed arguments])[(list of typed results)]
  code
[EXCEPT/EXCEPTDO
  ecode]
ENDPROC [result]

APROC name([list of typed registers])[(list of typed results)]
  assembler only code
ENDPROC
```

Everything between APROC and ENDPROC, is COPIED into output assembler source code, so
if you do a mistake, PowerD will not show an error!!!, only while Compiling... pass
PhxAss will leave with error code of 20. I will add some processor for assembler routines in future, so currently be carefull with it.

Examples:

Following example shows, how useful may be default return values. These are the same:

```
PROC test()(DOUBLE,DOUBLE)
  RETURN a,1.1
ENDPROC 1.0,1.1

PROC test()(DOUBLE=1.0,DOUBLE=1.1)
  RETURN a
ENDPROC

APROC compute(d0,d1,d2)(LONG)
  add.l d1,d0
  and.l d2,d0
ENDPROC
```

1.19 module

```
  Description:

  With 'MODULE' keyword you can insert any of #?.m or #?.d files. When you use ↩
     more
modules with same name, only the first one will be processed. This keyword can ↩
   be used
only outside of procedures. Modules should be in 'DMODULES:' assigned directory ↩
    (see:
installation), but it is possible to insert before module name full path leading ↩
     with
'*' sign (see below).


  Syntax:

MODULE 'module1','*module2',...


  Examples:

MODULE 'dos'
  will try to open file 'DMODULES:dos' or 'DMODULES:dos.m' or 'DMODULES:dos.d', ↩
     whereas

MODULE '*HD5:Sources/module.m'
  will try to open only 'HD5:Sources/module.m'.
```

1.20 emodule

```
  Description:

  Externam modules are normal modules, which contains information about external
object/library files. Currently global variables, procedures and linked library
functions.

  Global variables:

  Global variables in external files are defined with 'EDEF' keyword, with this ↩
     syntax:

EDEF list of typed variables

where variable is external variable name (with following sizes if it is an array ↩
   ) and
type is normal type.


  External procedures:

  External procedures are defines in the same was as normal procedures, but ↩
     leading
```

```
with 'EPROC' instead of 'PROC'. All arguments must be defined and all return  ↩
    types
must be defined:

EPROC procname(list of typed arguments)[(list of types)]

where procname is external procedure name, vars are variable names (this should  ↩
    be
what you like), types must be the same as in procedures.
  If you write an external definition of c compiled function, use LPROC keyword.


  Linked library functions:

  Linked library functions are defined in same way as external procedures, but  ↩
     leading
with 'LPROC' instead of 'EPROC'.
```

1.21 except

```
  Description:

  Exceptions may be very useful if you do a very complex program. If Raise()  ↩
     function
called, arguments will be set as exception and exceptioninfo and it will jump  ↩
    into the
last processed procedure EXCEPT part. If you call Raise() function in except  ↩
    code part,
it will do the same, but into the last previous procedure with EXCEPT part.

  Syntax:

PROC xxx()
  code
EXCEPT
  excepted-code
ENDPROC

  If somewhere in code a Raise() function is used, the excepted-code will be  ↩
     processed,
if nowhere excepted-code will be skipped. If you use EXCEPTDO instead of EXCEPT  ↩
    keyword,
excepted-code wont be skipped, it will be processed right after code. The  ↩
    following two
pieces of code are the same:

EXCEPTDO

and

  Raise(0,0)
EXCEPT
```

1.22 global

Description:

This is useful for including a binary data/file that will be available within program's code.

Syntax:

```
BYTE   list or string
WORD   list
LONG   list
```

If string (BYTE only) wrote, no zero character will be added to the end, you have to add manualy '\0'.

BINARY list of file names

Here will be placed listed files.

To be more usefull, you can sign these static fields with labels.

Example:

```
rawdata: BINARY 'ram:data.raw'
BYTE '\0$VER: v0.1\0'
```

1.23 oo

1.24 loop

Description:
LOOP is the infinite loop, it means that everything between the LOOP and ENDLOOP keywords will repeat until RETURN, EXIT or EXITIF keywords are processed.

Syntaxes:

```
LOOP
  code
ENDLOOP

LOOP DO commands // see DO keyword
```

Returning values:

```
a:=LOOP           // loop is repeated until condition is true and then is b  ↩
    copied to a
   EXITIF condition IS b // see EXIT/EXITIF
```

```
   ENDLOOP
```

You can also return multiple return values.

1.25 for

Description:

FOR is a loop, where its variable is after each loop increased/decreased by one.
You can also use floats.

Syntax:

```
FOR a (TO/DTO) b [STEP c]
  code
ENDFOR [list]

FOR a (TO/DTO) b [STEP c] DO commands [IS list]  // see DO keyword
```

where 'a' is something like: n, n:=2, n:=i*j, etc.

Returning values:

Watch 'list' in syntax part.

a:=FOR...

Early exit:

See EXIT/EXITIF

1.26 while

Description:

Code between WHILE and ENDWHILE will be repeated until condition after WHILE is true.

Syntaxes:

```
WHILE[N] condition
  code
ENDWHILE
  While condition is TRUE, code is processed, else program continues after ' ↩
     ENDWHILE'
```

```
WHILE[N] condition DO commands     // see DO
  While condition is TRUE, code is processed, else program continues on next  ↩
     line.

WHILE[N] condition1
  code1
ELSEWHILE[N] condition2
  code2
ALLWAYS
  code3
ENDWHILE
  While condition1 is TRUE, code1 and code3 are processed, if condition1 is  ↩
     FALSE and
condition2=TRUE, code2 and code3 are processed, if both conditions are FALSE,  ↩
    loop
is stopped and program continues after 'ENDWHILE'. It is ofcourse possible to  ↩
    insert
more ELSEWHILEs or remove ALLWAYS.

If You add 'N' after WHILE or ELSEWHILE, the result of contition will be negated ↩
    :

WHILE a>b
ELSEWHILE a=0

is the same as

WHILEN a<=b
ELSEWHILEN a


  Returning values:

  WHILE loop can return list of values, just add the return list after 'ENDWHILE ↩
     '
keyword:

ENDWHILE list


  Early exit:

EXIT/EXITIF keyword
```

1.27 repeat

```
  Description:

  The code between REPEAT and UNTIL keywords will be processed until the  ↩
     condition is
false, if the condition is true, it will terminate. It is also possible to use  ↩
    EXITIF
keyword for early termination. If DO keyword is used, commands will be processed ↩
     while
```

```
terminating the loop. If IS keyword is used, the loop can return a list of  ↩
   values.


  Syntax:

REPEAT
  code
UNTIL[N] condition [DO commands] [IS list]

If You add 'N' after UNTIL, the result of contition will be negated:

UNTIL a>b
UNTIL a=0

is the same as

UNTILN a<=b
UNTILN a
```

1.28 if

```
  Description:

  What to say about if? Just try it.

  Syntax:

IF[N] condition THEN commands [ELSE commands]  // THEN/ELSE can be used as DO

IF[N] condition
  code
[ELSEIF[N] condition
  code]
...
[ELSE
  code]
ENDIF

or with DO keyword only

IF[N] condition     DO commands
ELSEIF[N] condition DO commands
ELSE                DO commands


If You add 'N' after IF or ELSEIF, the result of contition will be negated:

IF a>b
IF a=NIL

is the same as

IFN a<=b
IFN a
```

Example:

```
IF age<10  DO PrintF('Too young!\n')
ELSEIF age<70
  PrintF('Yes, what is your name?: ')
  ReadStr(stdout,name)
ELSE DO PrintF('Too old!\n')
```

1.29 select

Description:

Syntax:

```
SELECT a
CASE b
  code
  [IS list]
[CASE c,d,e    DO commands [IS list]]
[CASE f TO g,h [IS list]]
...
DEFAULT [DO commands/
  code]
ENDSELECT [list]
```

where a, b, ... are equations, functions, constants or something what returns a value.

Examples:

```
SELECT age
CASE 0 TO 17
  PrintF('Young\n')
CASE 18 TO 50
  PrintF('Adult\n')
CASE 51 TO 120
  PrintF('Old\n')
DEFAULT
  PrintF('What???\n')
ENDSELECT

name:=SELECT Person.ID
  CASE 1 IS 'Paul'
  CASE 2 IS 'Jenny'
  CASE 3 IS 'Peter'
  CASE 4 IS 'Mark'
  ENDSELECT 'unknown'
```

1.30 do

Description:

DO keyword in PowerD is quite different from AmigaE, it is not limited to only one
command. You can add after DO howmany commands you like, but they must be separated by
semicolon (';')

Syntax:

DO command1; command2; command3

where commands are functions, equations, etc. Everythink you like.

1.31 then

Description:

Syntax:

1.32 exit

Description:

Via this keywords you can stop loops early. This keywords can be used in LOOP, IF,
WHILE, REPEAT, SELECT and FOR loops

The WHILE loop can be stopped before it reaches its end via 'EXITIF' keyword:

EXITIF[N] condition [DO code] [IS list]
EXIT [DO code] [IS list]

While condition is FALSE nothing happens, if TRUE, code will be processed and list of
values will be returned.

1.33 jump

Description:

Via JUMP keyword You can skip from one part of procedure to another, but be sure that
label exists, PowerD currently doesn't check it. Never JUMP into loops.

```
  Syntax:

JUMP label

label can be defined everywhere in a procedure:

label:


  Comment:

  I'm sorry for every body who missed this command in PowerD, PowerD never ↩
     missed it,
but I have just forgot to include it in documentation. (Thanx to Marco ↩
    Antoniazzi)
```

1.34 library

```
  Description:
Libraries are on Amiga used very often, they contains many useful functions.
It is ofcourse possible to use them in PowerD.


  Syntax:

LIBRARY NameBase
  Function([list of typed arguments])[(list of returning variables)][=function ↩
     offset],
  Function2(),
  Function3()


  Function arguments:
Each argument starts with register (eg.: d0,d1,a0,a1,fp0,fp1,...), then should ↩
    follow
a type. It is also possible to use 'LIST OF' keyword, that is used for
inline lists. This must be last argument, since it may contain different number ↩
    of
arguments. (eg.: PrintF('\d*\d=\d\n',a,b,a*b) where a, b, a*b are arguments of ↩
    list)


  Default argument values:
If you want to use default arguments in library functions like in procedures, ↩
    just
insert after register '=value', where value is a number or a constant.


  Return values:
All functions in Amiga libraries currently returns maximally one value in D0 ↩
    register.
This way you can create your own libraries that wont be so limited. If you wont ↩
    define
return register/type, register (D0:VOID) will be used.
```

```
  Library offsets:
Initial library offset is -30 (default Amiga library first function offset).
After each function is this offset decreased by 6.


  Examples:

LIBRARY DrawBase
  DrawPixel(a0:PTR TO RastPort,d0:WORD,d1:WORD),
  DrawLine(a0:PTR TO RastPort,d0:WORD,d1:WORD,d2:WORD,d3:WORD),
  ReadPixel(a0:PTR TO RastPort,d0:WORD,d1:WORD)(d0:WORD)=-48,
  VTextF(a0:PTR TO RastPort,d0:WORD,d1:WORD,a1:PTR TO UBYTE,a2=NIL:PTR TO LONG),
  TextF(a0:PTR TO RastPort,d0:WORD,d1:WORD,a1:PTR TO UBYTE,a2:LIST OF LONG)=-54
```

1.35 linklib

```
  Description:

  Linked libraries are on amiga used in many programming languages except AmigaE ↩
     , so I
added linked library support into PowerD. Linked libraries can contain many more ↩
    or less
useful functions. The defference between linked libraries and normal libraries  ↩
   is that
linked library will add its functions into the your code, so the executables  ↩
   will be
quite longer instead of normal libraries's (usually in libs:) functions will be  ↩
   only
called, those functions needs only to open the library. On other platforms than  ↩
   Amiga
are linked libraries more often (somewhere only possible :^().


  Calling functions from linked libraries:

Calling is absolutely the same as calling procedures, only definition slightly  ↩
   defferent.


  Definition of functions from linked libraries:

See  How to create LinkLib
```

1.36 ifunc

```
PowerD has currently no hardcoded internal functions, all functions are in  ↩
   PowerD.lib.


  Inline functions:
```

```
FAbs(a)
Sin(a)
Cos(a)
Tan(a)
Tanh(a)
Sqrt(a)
ASin(a)
ACos(a)
ATan(a)
ATanh(a)
EtoX(a)
EtoXm1(a)
GetExp(a)
GetMan(a)
Int(a)
IntRZ(a)
Log(a)
Log2(a)
Ln(a)
Lnp1(a)
TenToX(a)
TwoToX(a)

  These functions are currently inline and hardcoded as fpu instruction.


  Linked library functions:

See linked libraries


  PowerD library functions:

 String/EString functions
 Math functions
 Intuition functions
 DOS support functions
 Miscelaneous functions
```

1.37 pdlstr

```
  Note:
Everywhere is written estring or estr MUST be E-Strings, not normal strings. If ↩
    you
wont fulfil it, your program may in better case do strange shings and in worse ↩
    case
crash your computer.


  NewEStr(length)
This allocated memory and header for an EString with a length.

  RemEStr(estring)
This frees memory and header of estring.
```

```
  EStrCopy(estring,string,length=-1)
This function copies length characters from string to estring. If length=-1,
whole str is copied.

  StrCopy(string,str,length=-1)
This function copies length characters from str to string. If length=-1,
whole str is copied.
Be sure that the string is long enough.

  EStrAdd(estring,string,length=-1)
This function adds string of length to the end of the estring. If length=-1,
whole string is copied.

  StrAdd(string,str,length=-1)
This function adds str of length to the end of the string. If length=-1,
whole str is copied.
Be sure that the string is long enough.

  EStrLen(estring)
This function returns length of the estring. It is much faster than StrLen(),
but it can be used only with E-Strings.

  StrLen(string)
This function returns length of the string. It can be used also for E-Strings,
but it is much slower than EStrLen().

  EStrMax(estring)
This function returns maximum length of the estring excluding last zero byte.

  SetEStr(estring,length)
This function sets estring's length to length. It is needed if you do some
operations with the estring withour E-String functions.

  ReEStr(estring)
Same as SetEStr() but length is got via zero byte finding.

  EStringF(estring,formatstr,arguments)
This function generates formated estring. Where arguments are same types
as used in formatstr.

  StringF(string,formatstr,arguments)
This function generates formated string. Where arguments are same types
as used in formatstr.
Be sure that the string is long enough.

  LowerStr(string)
All characters of string are converted to lower case.

  UpperStr(string)
All characters of string are converted to upper case.

  InStr(string,str,startpos=0)
This functions return position of str in string starting at position defined
by startpos or -1 if not found.

  MidEStr(estring,string,startpos,length=-1)
This functions copies length characters from string started at startpos to the
```

```
estring. If length=-1 all characters are copied.

  RightEStr(estring,estr,length)
This functions copies length right characters from estr string into the estring.

  StrCmp(str1,str2,length=-1)
This compares str1 and str2 of the length and returns -1 if str1=str2 else 0.
If length=-1 whole string is compared.

  OStrCmp(str1,str2,length=-1)
This compares str1 and str2 of the length and returns 1 if str1<str2, 0 if str1= ↩
   str2
and -1 if str1>str2. If length=-1 whole string is compared.

  ReadEStr(fh,estring)
This reads string from filehandle fh. String is read byte by byte until "\n" or  ↩
   "\0"
reached. All characters are copied into estring.

  TrimStr(string)
"\n", "\t", " " and similar characters will be skiped in the string and returned ↩
   .

str:='\t  \nHello\n'
str:=TrimStr(str)

now str contain 'Hello\n' only.
Note: if source was E-String, result is no an E-String.

  IsAlpha(byte)
Returns TRUE if byte is an alphabetical letter, otherwise FALSE.

  IsNum(byte)
Returns TRUE if byte is a number letter, otherwise FALSE.

  IsHexNum(byte)
Returns TRUE if byte is a hexa-decimal number letter, otherwise FALSE.

  Val(str:PTR TO CHAR,startpos=0)(LONG)
  This functions returns a number which is generated from the str. Currently is  ↩
     able
to use binary(eg: %1011100), hexadecimal (eg: $12ab34dc) and decimal (eg: 123)  ↩
   numbers.
If you specify startpos the number generation will start on this position. If  ↩
   string
contains illegal characters this will probably return an illegal value. If the  ↩
   str
begins with ' ', '\n' or '\t' characters, all of these will be skipped.

  RealVal(str:PTR TO CHAR,startpos=0)(DOUBLE)
  This function is similar to Val(), but it is usable only with floats.  ↩
     Currently is
able only to convert strings with format of '[-]x.y', so no exponent alloved. If ↩
    the
str begins with ' ', '\n' or '\t' characters, all of these will be skipped.

  RealStr(str:PTR TO CHAR,num:DOUBLE,count=1)(PTR TO CHAR)
```

```
  This function generates str from given num with count of digits after the  ↩
     point.
Currently does not allow exponents.

  RealEStr(estr:PTR TO CHAR,num:DOUBLE,count=1)(PTR TO CHAR)
Same as RealStr(), only generates an E-String.
```

1.38 pdlmath

```
  Abs(a)
Returns absolute value of a.

  And(a,b)
Returns a&b.

  Bounds(a,min,max)
Bounds a with min and max and returns the result. It is the same as:
res:=IF a<min THEN min ELSE IF a>max THEN max ELSE a

  EOr(a,b)
Returns a!b.

  Even(a)
Returns -1 if a is even else 0.

  Max(a,b)
Returns the bigger value of a and b.

  Min(a,b)
Returns the smaller value of a and b.

  Neg(a)
Returns negated a.

  Not(a)
Returns noted a.

  Odd(a)
Returns -1 if a is odd else 0.

  Or(a,b)
Returns a|b.

  Rol(a,b)
Returns a rotated left by b bits.

  Ror(a,b)
Returns a rotated right by b bits.

  Shl(a,b)
Returns a shifted left by b bits.

  Shr(a,b)
Returns a shifted right by b bits.
```

```
  Sign(a)
Returns 1 if a>0, -1 if a<0, else 0

  Pow(a:DOUBLE,b:DOUBLE)
Returns a^b. If b=0, 1 is returned.
```

1.39 pdlintui

```
  WaitIMessage(w:PTR TO Window)(LONG,LONG,LONG,LONG)
This function waits for a message in window specified by w and returns four ↩
    values.
The first is class, second is code, third is qual and fourth is iaddress.

  Mouse()
This function returns %001 if left mouse button is pressed, %010 if right mouse ↩
    button is
pressed and %100 if middle mouse button is pressed. It can return it's ↩
    combinations.

  MouseX(w:PTR TO Window)
Returns mouse horizontal position in window w.

  MouseY(w:PTR TO Window)
Returns mouse vertical position in window w.

  MouseXY(w:PTR TO Window)(LONG,LONG)
Returns mouse position in window w.
```

1.40 pdldos

```
  FileLength(name:PTR TO CHAR)(LONG)
Returns length of file specified by name or -1 if file doesn't exists.

  Inp(fh)(LONG)
Returns byte read from file handle specified by the fh or -1 if an error occured ↩
    .

  Out(fh,byte)
Writes byte to file handle specified by the fh.
```

1.41 pdlmisc

```
  CtrlC()
  CtrlD()
  CtrlE()
  CtrlF()
There functions checks if ctrl+c etc. key combination is pressed. If yes -1 else ↩
    0 is
returned.
```

```
  Long(a)
  Word(a)
  Byte(a) (020+)
Returns byte/word/long value what is on address specified by a.

  PutLong(a,b)
  PutWord(a,b)
  PutByte(a,b)
Writes byte/word/long value specified by b to address specified by a.

  KickVersion(requied)
Returns TRUE if requied is lower or equal to your system version, else returns  ↩
   FALSE.

  Rnd(top)
Returns a pseudo random number from range 0 to top-1. If the top value is lower
then zero, new seed is set. top must be a 16bit number.

  RndQ(seed)
This is quite faster than Rnd(), but it returns a pseudo random 32bit number.  ↩
   Use the
result of this function for next seed of this function to get random numbers.
```

1.42 iconst

```
  These constants are set allways before compilation:

TRUE  = -1
FALSE = 0
NIL   = 0
PI    = 3.141592653589


  Special (changeable) constants:

OSVERSION = requied version of operation system (see: Options
STRLEN    = length of last used string

PrintF('Hello\n')
len:=STRLEN                    // len contains number 6


  Type size constants:

SIZEOF_BYTE   = 1
SIZEOF_UBYTE  = 1
SIZEOF_WORD   = 2
SIZEOF_UWORD  = 2
SIZEOF_LONG   = 4
SIZEOF_ULONG  = 4
SIZEOF_FLOAT  = 4
SIZEOF_DOUBLE = 8
SIZEOF_PTR    = 4
SIZEOF_BOOL   = 2
SIZEOF_VOID   = 4
```

```
  OBJECT size constants:

SIZEOF_bla = size of OBJECT named 'bla'
```

1.43 opt

```
Options are in PowerD introduced by keywrd 'OPT', it can be defined everywhere ↩
   outside
PROCedures.

HEAD/K       // startup #?.o file, this file have to be in 'd:lib' or it have
             // to begin with '*' and full object path (default: 'startup.o')
  // It is better to use modules in dmodules:startup after 'MODULE' keyword to ↩
     avoid
  // external variable definitions instead of use 'OPT HEAD' in your source.
NOHEAD/S     // this switch disables adding a head (default: enabled)
LINK/M       // this introduces #?.o or #?.lib file what should be linked to ↩
   executable
             // this should be used more times
OBJECT/K     // no executable, but #?.o file will be created with given name
             // (default: '<progname>.o')
PRIVATE/S    // enables private data in objects
NOSTD/S      // disables standard.lib linking

  Examples:

OPT NOHEAD,LINK='algos.o',LINK='d:lib/amiga.lib',DEST='calc'

  This compiles source without a head to #?.o file and link this #?.o file with ↩
     algos.o
and amiga.lib files into the 'calc' executable.
```

1.44 cli

```
SOURCE/A              - PowerD source file
DEST                  - Destination executable file
TOOBJECT/K            - Destination object file
GM=GENMODULE/K        - See: external modules
CO=CHECKONLY/S        - Only first pass, check for syntax errors
NS=NOSOURCE/S         - Disables source writing after errors
AI=ASMINFO/S          - Generates more readable assembler source with some ↩
   information
DS=DEBUGSYM/S         - Compiles source with debug symbols (only way, how to ↩
   debug)
SDV=STARTDEBUGVALUE/N - Each compiled source starts with label counter from 0, ↩
   this
                        can be changed. (useful for multiple source compiling)
NU=NOUNUSED/S         - Disable writing of list of unused variables/procedures
I=INFO/S              - When compilation is finished, some information is wrote ↩
   to
```

```
                          stdout.
  AUTHOR/S              - This is quite useless, but who knows...
```

1.45 error

```
    PowerD contains quite many error messages, and there is no a syntax error (be  ↩
       happy),
  if you do a syntax error, it will tell you what (or at least where) you did  ↩
     wrong.
    Currently if you do some mistake, an error message is repeated until you press ↩
        Ctrl+C
  and stop compilation. These errors are something like forgotten apostrophe,  ↩
     bracket,
  comma etc. PowerD currently shows bad line numbers, so don't look at it :^(,  ↩
     PowerD
  tries to show a piece of source where an error occured, this may be sometimes  ↩
     strange
  mainly in Writing pass.
```

1.46 syntax

```
    Tabulators should have size of 3. Preprocessor keywords must be at the  ↩
       beginning of
  the line.
```

1.47 diff

```
  PowerD is based on AmigaE syntax, but PowerD has the syntax more flexible, here  ↩
     will
follow differences between AmigaE and PowerD:
(It isn't everything and in future I'll expand it)

PowerD:                                 AmigaE:

  Operators:

a\b                                     Mod(a,b)
a|b                                     a OR b
a&b                                     a AND b
a!b                                     Eor(a,b)
a<<b                                    Shl(a,b)
a>>b                                    Shr(a,b)
a|<b                                    -
a|>b                                    -
~a                                      Not(a)
&a                                      {a}
a:=:b                                   tmp:=a; a:=b; b:=tmp
=>,>=                                   >=
=<,<=                                   <=
b:=--a                                  b:=a--
```

```
b:=++a                                  b:=a; a++
- (future)                              `a
- (future)                              {a}    where 'a' if a function
SIZEOF_x                                SIZEOF x


  Structures:

PROC x()                                PROC x()
PROC x()(LONG)...ENDPROC a              PROC x()...ENDPROC a
PROC x()(LONG,LONG=2)...ENDPROC a       PROC x()...ENDPROC a,2

EXITIF b                                EXIT b

FOR x:=a TO b STEP 2                    FOR x:=a TO b STEP 2
FOR x:=a TO b STEP c                    ?
FOR x:=0.1 TO 1.2 STEP 0.2              ?

SELECT a                                SELECT a
                                        SELECT a OF b
SELECT s                                -
CASE 1 TO 10 DO s; ...                  CASE 1..10; s; ...
CASE s                                  -
DEFAULT DO s; ...                       DEFAULT; s; ...
ENDSELECT                               ENDSELECT

IFN s                                   IF s=FALSE
WHILEN s                                WHILE s=FALSE


  Constants:

FLAG A_1,A_2,A_3                        ENUM AB_1,AB_2,AB_3
                                        SET AF_1,AF_2,AF_3
0.123456      (FLOAT)                   0.123456
0.123456789   (DOUBLE)                  ?


  Objects, Types:

OBJECT x                                OBJECT x
  a:BYTE,b:UBYTE,                         a:CHAR,b:CHAR
  c:BOOL,d:FLOAT                          c:INT,d:LONG
                                        ENDOBJECT

DEF                                     DEF
a:BYTE,b:WORD,c:BOOL                    a,b,c
a[10]:LONG                              a[10]:ARRAY OF LONG
a[]:LONG                                a:PTR TO LONG
```

1.48 ccode

Compile your c source in to an object file and write an external module of it.

1.49 asmcode

```
  Description:

  From 0.05, you are able to write inlined assembler routines, it is currently ↩
     VERY
limited, so you can't work with DEFined variables, this will be allowed in next ↩
    release
of PowerD.

  PowerD currently doesn't check if your assembler syntax is right, it only ↩
     COPIES the
assembler part of your source code to destination assembler source, so be ↩
    carefull with
the assembler syntax, if you do an mistake, PhxAss will leave with error code of ↩
     20.

  Since asselbler doesn't allow '//' and '/* */' comments, you have to use ';' ↩
     or '*'
as comment beginnings.

  I'll eliminate all of this disadvantages in next releases.


  Syntax:

ASM
  here are your assembler routines
ENDASM
```

1.50 createhead

```
  Description:

  Header (resp. startup file) is a piece of code what is run at the beginning of ↩
      the
executable. It usually makes some variable initiallisations, library opening etc ↩
    . Each
header should call function called 'main()'. It is usually written in assembler ↩
    to get
the best perfomance, but it can be written in PowerD as good.


  Header module:

  If you wrote a header, you should write also a module what contains in the ↩
     header
initialised but external variables (via 'EDEF') and OPT HEAD='xxx.o' where xxx
is the header object file name in 'd:lib', or '*' and full path.


  Example:
```

```
OPT HEAD='startup_tri.o'

MODULE 'dos','exec','intuition','graphics'

EDEF DOSBase,IntuitionBase,GfxBase
```

1.51 createlib

```
  How to use a linked library:

  The best way how to use linked libraries is to define it's functions in a ↩
     module. To
be more simple for programmer is better to add in to the module line containing:

OPT LINK='linklibrary'

where linklibrary is your linked library name if it is in 'd:lib' drawer, if not ↩
   , add
'*' before the linklibrary with full path (eg: 'hd5:lib/mylib.lib'). This will ↩
   add the
link library into the list of the link objects and after compilation everything ↩
   will be
linked.
  This module should be in 'dmodules:lib' drawer.


  Function definition:

  PowerD makes it possible to use different types of functions. First function ↩
     type is
normal linked library function which allows LIST using. All used arguments are ↩
   loaded
into the stack in inverse order and then the function is called:

LPROC procname(args)(results)

LPROC is mostly used by C compilers, it parses arguments inverted from its ↩
   definition:
  x(a,b,c)
this moves to stack c then, b and then a, if You use a list as a last argument, ↩
   all list
items will be inserted to stack in same order:
  LPROC x(u,v,w:LIST OF LONG)
  x(a,b,c,d,e,f)
this moves to stack f, e, d, c, b and a. This is quite dull when you want use ↩
   something
like this:
  LPROC x(a,b,c,d)
  n:=0
  x(n++,n++,n++,n++)
this will copy to a 3, to b 2, to c 1 and to d 0.

  Second is PowerD procedure compatible stack loading. It loads arguments in ↩
     right order
```

```
but don't (currently) allows the LISTs. These are also used for external  ↩
    procedures:

EPROC procname(args)(results)

This is quite more intelligent, but it doesn't allow inline lists as a last  ↩
    argument,
you can of course use normal lists (closed in: []).
  EPROC x(a,b,c,d)
  n:=0
  x(n++,n++,n++,n++)
works correctly, moves 0 to a, 1 to b, 2 to c and 3 to d.

  Last one is best suited for assembler routines, which doesn't use stack (it is ↩
      slower
than registers). Here is the limitation gave by count of registers on the cpu ( ↩
    MC68k
allows 8 data, 5 address registers (don't use a6 and a7) and 8 float registers.  ↩
    PPC
allows about 25 data/address registers and 31 float registers.):

RPROC procname(args with registers)(results)['asm']

  As you can see, it is possible to add an assembler source after RPROC  ↩
      definition, this
way you can generate inlined functions. Be careful about the assembler source  ↩
    you must
adhere right syntax:
- each line starts with tabulator '\t' or some spaces or a label
- each line must be ended with linefeed '\n'
- you must use right instruction set
- it is absolutely not affected by PowerD, it will be only copied instead of the
  function call
- it is normal string, so you can use multi line strings

  How to build linked library:

  First you have to generate object files, each object file should contain only  ↩
      one
function. This may be done with OPT OBJECT in PowerD case. If you want to create ↩
     an
assembler routines, use export keyword (usually xdef). Copy all these objects  ↩
    into a
single drawer, open shell, set there the drawer and enter 'join #?.o as xxx.lib ↩
    ', where
xxx is your library name. Then copy this file into 'd:lib' drawer. Finaly you  ↩
    should
write a module for this lib.
  This way is possible to generate linked libraries from PowerD, Assembler, C  ↩
      etc.
objects, just select right function definition (LPROC/EPROC/RPROC).


  Examples:

LPROC printf(fmt:PTR TO CHAR,args:LIST)
```

```
EPROC PrintF(fmt:PTR TO CHAR,args:PTR)

RPROC PrintF(a0:PTR TO CHAR,a1:LIST)
```

1.52 createlibrary

```
  Currently it is not possible to make libraries in PowerD. When I will have enough
information about it, I will of course include it.
```

1.53 why

```
It is simple, I wanted to use fpu, but AmigaE wont work with it directly, so I used
bettermath modules (by Michal Bartczak) and later I did my own fpu modules, but it was
quite frustrating when I wanted add a to b to use a function. All that time I got an
idea to create my owm programming language, because I was sure that AmigaE will never
use fpu and newer processors.

At the beginning of the 1998 I started to play with equation reading, simple compilers,
etc. In the middle of 1998 I started working on PowerD.

In march 99 Wouter van Oortmerssen (author of AmigaE) stopped developing if AmigaE. At
this time I was sure to continue my developing. (Many times I wanted to stop it because
of unsolvable problems, but lastly I solved them all :^))

When I got Amiga (1993), I got some 3d raytracing programs and I was lost...  I
wanted to create my own 3d raytracing program that will be better then all other, but in
which programming language should I do it? I could work only with AMOS basic, and it is
not good to do 3d raytracing program in basic. I started to learn assembler (1994), it
is also not good programming language to create so big project (it could have many MB
of assembler source). Then I tried Pascal and C. One is worse then the second. Later (in
1995) I tried AmigaE. Wow, really fast and short programs! I started to work on my own
3d raytracing program. I registered it in 1996. Everything looked good, but I bought
Blizzard 1230/50 with 16MB of RAM and FPU. AmigaE wasn't enough at this time. I tried C,
but returning max one value per function is incredibly few (AmigaE can return 3 values).
So I did something in AmigaE and something in C.
```

This was the time to create a language what will be good for everything!

1.54 install

Unpack the archive, copy everything into a drawer on your harddisk, add to your startup-sequence:

```
  Assign D: <drawer>
  Assign DMODULES: D:Modules
  Path D: D:Bin ADD
```

where <drawer> is path of PowerD directory.

Or click on the Install icon.

1.55 features

```
  Features when comparing with C/C++:

+ multiple return values (8 for m68k, \ensuremath{\pm}25 for ppc)
+ lists can be defined/used everywhere you like/need, not on the definition
+ more readable syntax
\ensuremath{\pm} some people says that "{}" is better then eg.: WHILE ENDWHILE,  ↩
    I think it is shorter
  not better


  Features when comparing with AmigaE:

+ more return values
+ more operators (like <<, >>, <|, >|, etc.)
+ more assign operators (like +=, *=, etc)
+ more intelligent equation computing (PowerD: 1+2*3=7, AmigaE: 1+2*3=9)
+ names can contain high/low letters
+ more types (FLOAT, DOUBLE, BOOL, etc.)
+ fpu using
+ compilation to object files
+ automatic generation of external modules
+ linked library functions using
+ inline lists (OpenWindowTags/OpenWindowTagList)
+ IFN, WHILEN, ... for reverse condition (IF a<>10 is the same as IFN a=10)
- slower


  Features:

- if you use floating point computations in source it will require fpu
```

1.56 future

```
  In next release:

- default settings file for custom powerd configuration
- new OPTions CPRE, APRE, EPRE, DPRE for C-like, PhxAss-like, E-like and D-like
  precedences in equations
- OPTions in modules will be local only, GOPT for global options


  In progress or near future:

- binary modules support for much faster module reading
- Complete preprocessor
- Object oriented programming
- Inteligent optimizations
- Elimination of bugs
- To be more "fool-proof"
- Link libraries with useful functions
  - audio/video/picture loading/playing/showing/saving
  - 3d functions for 3d games
- PowerPC support
- Library compiling


  In plan:

- AmigaNG support (really not sure)
- AmirageK2/QNX Neutrino support
- PowerPC G4/AltiVec (?) support
- VisualD interface
- Debugger
- Get some (small) money for it
- Better manual (this is the most dificult)
- To be modular (eg.: add a module to generate code for other processor)


  In vision:

- Enterprise support (NCC1701D or better requied)


  This I will probably never do:

- Use math libraries instead of fpu instructions
- Windows95/98/2000/3000/4000/NT version


If you have some ideas, send me an e-mail.
```

1.57 history

```
  Version 0.06 (21.11.1999):
- some bugs removed (:=: didn't work in 0.05 and 0.05b, etc.)
```

```
- improved LOOP command, idea by Marco Antoniazzi
- improded this document, I forgot to include here many PowerD abilities:
  - JUMP
  - IFN, ELSEIFN documented
  - WHILEN, ELSEWHILEN documented
  - UNTILN documented
- line numbers on errors are now exact (I hope)

  Version 0.05b (16.11.1999):
- x++/x-- added/subtracted two instead of one and made wrong things...
- pad bytes in lists now works

  Version 0.05 (15.11.1999):
- assignations changed from eg: :*= to *= to be more compatible
- new: ASM and APROC()
- many bugs removed
- added differences between AmigaE and PowerD in this documantation.

  Version 0.04 (7.11.1999):
- improved constant finding (up to 28 times faster!!!)
- added more modules
- removed heavy bug in object reading (eg: example GadToolsTest.d took about 800 ↩
    kB of
  memory and about 24000 allocations, now about 350 kB and 14000 allocations,  ↩
     compiling
  time was about 90 seconds on unexpanded A1200, now takes about 35 seconds)
- added INFO/S switch in cli.
(development is currently quite slow because my blizzard ppc is broken down, and ↩
    I have
to develop powerd on unexpanded A1200 with hd)

  Version 0.03 (10.10.1999):
- added some functions (Val(), RealVal(), RealStr(), etc.) to PowerD.lib
- removed many more or less important bugs


Following versions I uploaded to aminet, but there were problems with main  ↩
   german site,
and I'm not sure if someone got it. I think, it's no so important, because in  ↩
   this time
I eliminated (very) many bugs.
  Version 0.02 (7.10.1999):
- now works with linked functions what has not arguments
- improved returning values
- added support for 192 and higher characters ascii names, you can use now  ↩
   variables/
  procedures named like: ÖÜÄ, testování, etc. See 0-255, this is good
  for non-english programmers.
- added unions in object definition. See UNIONs
- removed some enforcer hits and small bugs


  Version 0.01 (30.9.1999):
First public release, history until this version is top secret.
```

1.58 bugs

```
If you found some bugs, send me an e-mail.

  Known bugs list:
- If you create too deep equations (too many different priorities, it can crash ↩
   or
  generate bad code)
```

1.59 limits

```
Each OBJECT member name can have maximally 16 synonyms.
Count of return values is limited by count of data registers (68k=8,ppc=cca.25)
```

1.60 requires

```
  Requirements:

An Amiga or a compatible computer
OS 3.0 (V39+)
HD
If you work with floats, it requires fpu.
PhxAss (by Frank Wille)
PhxLnk (by Frank Wille)
  Recomendations:

Lots of RAM, 16 or 32 MB should be enough for very large projects.
Fasted CPU, 030 should be enough.
FPU, 68881 should be enough :^)
  Succefully Tested configurations:

A1200+HD+3.0
A600+MTec030/40+882/40+2MB+16MB+3.1+HD
A1200+Blizzard1230IV/50+882/50+16MB+3.1+HD
A1200+Blizzard1240/40+32MB+3.0+HD
A1200+Blizzard603e/160+040/25+64MB+3.1+HD
```

1.61 register

```
  PowerD is currently FREEWARE.

If you like it, please e-mail me.

Rewards are also welcome.

I didn't do it for money, but living(programming) without it is quite difficult.
```

1.62 thanx

```
phase5    - for their wonderful blizzards and for staying with (classic) Amiga
gw2k      - I really don't know if thanx or not at this moment
m*crosoft - for producing still the worst operating system :^)

  Here should follow betatesters (if I will have some), bug reporters, and ↩
     everybody who
will help me.

Mauro Fontana    - for his bug reports and ideas
Marco Antoniazzi - for his bug reports and ideas
```

1.63 author

```
  Snail mail:

Martin  Kuchinka
Amforová 1930/17
Praha 5,  155 00
 Czech Republic


  E-Mail:

kuchinka@k332.feld.cvut.cz


  My Amiga configuration:

Amiga: A1200
  CPU: MC68040/25, PPC603e/160
  RAM: 64 MB Fast
   HD: Seagate Medalist 3 GB
   CD: GoldStar 6x
Modem: Rockwell 33k6 bps
Video: AGA, old VGA monitor
Audio: Paula, JVC PC-V66 (Hyper-Bass Sound)
```

1.64 ascii

Value	ASCII	Value	ASCII	Value	ASCII	Value	ASCII
0	"\0"	64	"@"	128	"\j128"	192	"À"
1	"\j001"	65	"A"	129	"\j129"	193	"Á"
2	"\j002"	66	"B"	130	"\j130"	194	"Â"
3	"\j003"	67	"C"	131	"\j131"	195	"Ã"
4	"\j004"	68	"D"	132	"\j132"	196	"Ä"
5	"\j005"	69	"E"	133	"\j133"	197	"Å"
6	"\j006"	70	"F"	134	"\j134"	198	"Æ"
7	"\!"	71	"G"	135	"\j135"	199	"Ç"
8	"\j008"	72	"H"	136	"\j136"	200	"È"
9	"\t"	73	"I"	137	"\j137"	201	"É"

```
10   "\n"          74   "J"          138   "\j138"       202   "Ê"
11   "\v"          75   "K"          139   "\j139"       203   "Ë"
12   "\j012"       76   "L"          130   "\j130"       204   "Ì"
13   "\b"          77   "M"          141   "\j141"       205   "Í"
14   "\j014"       78   "N"          142   "\j142"       206   "Î"
15   "\j015"       79   "O"          143   "\j143"       207   "Ï"

16   "\j016"       80   "P"          144   "\j144"       208   "Ð"
17   "\j017"       81   "Q"          145   "\j145"       209   "Ñ"
18   "\j018"       82   "R"          146   "\j146"       210   "Ò"
19   "\j019"       83   "S"          147   "\j147"       211   "Ó"
20   "\j020"       84   "T"          148   "\j148"       212   "Ó"
21   "\j021"       85   "U"          149   "\j149"       213   "Õ"
22   "\j022"       86   "V"          150   "\j150"       214   "Ö"
23   "\j023"       87   "W"          151   "\j151"       215   "$\times$"
24   "\j024"       88   "X"          152   "\j152"       216   "Ø"
25   "\j025"       89   "Y"          153   "\j153"       217   "Ù"
26   "\j026"       90   "Z"          154   "\j154"       218   "Ú"
27   "\e"          91   "["          155   "\j155"       219   "Û"
28   "\j028"       92   "\"          156   "\j156"       220   "Ü"
29   "\j029"       93   "]"          157   "\j157"       221   "Ý"
30   "\j030"       94   "^"          158   "\j158"       222   "Þ"
31   "\j031"       95   "_"          159   "\j159"       223   "ß"

32   " "           96   "`"          160   "~"           224   "à"
33   "!"           97   "a"          161   "¡"           225   "á"
34   """           98   "b"          162   "¢"           226   "â"
35   "#"           99   "c"          163   "£"           227   "ã"
36   "$"          100   "d"          164   "¤"           228   "ä"
37   "%"          101   "e"          165   "$\yen$"          229   "å"
38   "&"          102   "f"          166   "¦"           230   "æ"
39   "′"          103   "g"          167   "§"           231   "ç"
40   "("          104   "h"          168   "¨"           232   "è"
41   ")"          105   "i"          169   "©"           233   "é"
42   "*"          106   "j"          170   "ª"           234   "ê"
43   "+"          107   "k"          171   "«"           235   "ë"
44   ","          108   "l"          172   "\ensuremath{\lnot}"           ↩
   236   "ì"
45   "-"          109   "m"          173   " "           237   "í"
46   "."          110   "n"          174   "®"           238   "î"
47   "/"          111   "o"          175   "¯"           239   "ï"

48   "0"          112   "p"          176   "\textdegree{}"          240   " ↩
   ð"
49   "1"          113   "q"          177   "\ensuremath{\pm}"          241  ↩
     "ñ"
50   "2"          114   "r"          178   "$^2$"          242   "ò"
51   "3"          115   "s"          179   "$^3$"          243   "ó"
52   "4"          116   "t"          180   "´"           244   "ô"
53   "5"          117   "u"          181   "$\mathrm{\mu}$"          245    ↩
   "õ"
54   "6"          118   "v"          182   "¶"           246   "ö"
55   "7"          119   "w"          183   "·"           247   "$\div$"
56   "8"          120   "x"          184   "¸"           248   "ø"
57   "9"          121   "y"          185   "$^1$"          249   "ù"
58   ":"          122   "z"          186   "º"           250   "ú"
59   ";"          123   "{"          187   "»"           251   "û"
```

60	"<"	124	"|"	188	"¼"	252	"ü"
61	"="	125	"}"	189	"½"	253	"ý"
62	">"	126	"~"	190	"¾"	254	"þ"
63	"?"	127	"\j127"	191	"¿"	255	"ÿ"