

**PowerD**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> PowerD		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		January 19, 2025	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>PowerD</b>	<b>1</b>
1.1	main . . . . .	1
1.2	PowerD.guide - Read This First . . . . .	2
1.3	PowerD.guide - Important information . . . . .	2
1.4	PowerD.guide - Rules of programming in PowerD . . . . .	3
1.5	PowerD.guide - I need some help . . . . .	3
1.6	PowerD.guide - Um, what does it do??? . . . . .	4
1.7	PowerD.guide - Immediate values . . . . .	4
1.8	PowerD.guide - Immediate strings . . . . .	6
1.9	PowerD.guide - Constant definition . . . . .	7
1.10	PowerD.guide - Variable definition . . . . .	9
1.11	PowerD.guide - Macro definition . . . . .	10
1.12	PowerD.guide - Using types . . . . .	12
1.13	PowerD.guide - OBJECT definition . . . . .	13
1.14	PowerD.guide - Equations . . . . .	15
1.15	PowerD.guide - Single variable operators . . . . .	16
1.16	PowerD.guide - Constant equations . . . . .	17
1.17	PowerD.guide - Function using . . . . .	18
1.18	PowerD.guide - Returning values . . . . .	18
1.19	PowerD.guide - PROC definition . . . . .	18
1.20	PowerD.guide - REPROC returning values . . . . .	19
1.21	PowerD.guide - Using MODULEs . . . . .	19
1.22	PowerD.guide - External MODULEs . . . . .	20
1.23	PowerD.guide - Exceptions . . . . .	21
1.24	PowerD.guide - Global data . . . . .	21
1.25	PowerD.guide - OO programming . . . . .	22
1.26	PowerD.guide - Polymorphism . . . . .	23
1.27	PowerD.guide - LOOP definition . . . . .	24
1.28	PowerD.guide - FOR definition . . . . .	25
1.29	PowerD.guide - WHILE definition . . . . .	25

1.30	PowerD.guide - REPEAT definition . . . . .	26
1.31	PowerD.guide - IF definition . . . . .	27
1.32	PowerD.guide - SELECT definition . . . . .	28
1.33	PowerD.guide - DO keyword . . . . .	28
1.34	PowerD.guide - THEN keyword . . . . .	29
1.35	PowerD.guide - EXIT/EXITIF keyword . . . . .	29
1.36	PowerD.guide - JUMP keyword . . . . .	29
1.37	PowerD.guide - NEW/END/INC/DEC/NEG/NOT... . . . .	30
1.38	PowerD.guide - LIBRARY definition . . . . .	31
1.39	PowerD.guide - Linked libraries . . . . .	32
1.40	PowerD.guide - Internal functions . . . . .	33
1.41	PowerD.lib - String/EString functions . . . . .	33
1.42	PowerD.lib - Math functions . . . . .	36
1.43	PowerD.lib - Intuition functions . . . . .	37
1.44	PowerD.lib - Graphics functions . . . . .	38
1.45	PowerD.lib - DOS support functions . . . . .	39
1.46	PowerD.lib - Miscelaneous functions . . . . .	39
1.47	PowerD.guide - Internal constants . . . . .	40
1.48	PowerD.guide - Options . . . . .	41
1.49	PowerD.guide - Preset user OPTions . . . . .	44
1.50	PowerD.guide - NOFPU . . . . .	44
1.51	PowerD.guide - CLI/Shell Syntax . . . . .	45
1.52	PowerD.guide - Error messages/warnings . . . . .	45
1.53	PowerD.guide - PowerD syntax . . . . .	49
1.54	PowerD.guide - PowerD and AmigaE differences . . . . .	50
1.55	PowerD.guide - How to use C code . . . . .	51
1.56	PowerD.guide - How to use Assembler code . . . . .	51
1.57	PowerD.guide - How to create Header . . . . .	52
1.58	PowerD.guide - How to create linked library . . . . .	52
1.59	PowerD.guide - How to create Library . . . . .	54
1.60	PowerD.guide - How to create binary module . . . . .	54
1.61	PowerD.guide - Why I did it . . . . .	55
1.62	PowerD.guide - Instalation . . . . .	56
1.63	PowerD.guide - Features . . . . .	56
1.64	PowerD.guide - Future . . . . .	57
1.65	PowerD.guide - History . . . . .	58
1.66	PowerD.guide - Bugs . . . . .	61
1.67	PowerD.guide - Limitations . . . . .	61
1.68	PowerD.guide - Requirements . . . . .	61

---

1.69	PowerD.guide - Registration . . . . .	62
1.70	PowerD.guide - Thanks . . . . .	62
1.71	PowerD.guide - Author's Address . . . . .	62
1.72	PowerD.guide - ASCII Table . . . . .	63
1.73	PowerD.guide - Index . . . . .	64

# Chapter 1

## PowerD

### 1.1 main

PowerD v0.12 (4.2.2000) by Martin Kuchinka  
(dc.e: 12345 lines, 396396 bytes)  
(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
- UPDATED Macro definition                      UPDATED
- Types, Pointers, Arrays
- OBJECT definition
- Equations
- Single variable operators
- Constant Equations
- Function using
- Returning values
- Procedure definition
- Using MODULEs
- External MODULEs
- Exceptions
- Global data
- OO programming
- Polymorphism
- LOOP definition
- FOR definition
- WHILE definition
- REPEAT definition
- IF definition
- SELECT definition

---

DO keyword  
THEN keyword  
EXIT/EXITIF keyword  
JUMP keyword  
NEW/END/INC/DEC/NEG/NOT...  
LIBRARY definition  
Linked libraries  
Internal functions  
Internal constants

Options  
CLI/Shell Syntax  
Error messages/warnings

Support programs:  
fd2m  
pr2m

Tutorial for novice programmers only:  
PowerD syntax  
PowerD and AmigaE differences

For experienced programmers only:  
How to use C code  
How to use Assembler code  
How to create Header  
How to create LinkLib  
How to create Library  
NEW How to create binary module NEW

Others:  
Why I did it  
Instalation  
Features  
Future, next release  
README History README  
README Bugs README  
Limitations  
Requirements  
Registration  
Thanks  
Author's address

## 1.2 PowerD.guide - Read This First

## 1.3 PowerD.guide - Important information

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
```

## 1.4 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
```

## 1.5 PowerD.guide - I need some 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 12k  
lines=400kb 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.

## 1.6 PowerD.guide - Um, what does it do???

If you see something like:  
PowerD v0.1: Generating(100)...  
it converts your source to my tables.

If you see something like:  
PowerD v0.1: Generating(100) in intuition/intuition...  
it reads modules.

Then you can see something like:  
PowerD v0.1: Generating...  
it regenerates lists of OBJECTs and adds links between OBJECTs.

Then you can see something like:  
PowerD v0.1: Writing(12%)...  
it writes/optimizes assembler source.

Then you can see something like:  
PowerD v0.1: Cleaning...  
it frees all memory allocations.

Then you can see something like:  
PowerD v0.1: Compiling...  
it executes phxass to compile assembler source.

Then you can see something like:  
PowerD v0.1: Linking...  
it executes phxlnk to link startup header, object files and link libraries.

Then you can see something like:  
PowerD v0.1: Done.  
if everything went ok, or:  
PowerD v0.1: Not Done.  
if not.

## 1.7 PowerD.guide - Immediate 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: $800000000, max: $7fffffff
    unsigned: min: $000000000, 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
```

Octal values:

```
[-]${0..7}[0..7]...
eg.: $123, $12345670
```

ASCII values:

```
[-]"#"
where # is arbitrary string maximally 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 needed (DOUBLES, FLOATs, LONGs, ...)
```

Value separator:

From v0.10 You are able to use dot character (".") in numbers as separators. ↵  
 This  
 will be usefull for 64bit values where \$fedc.ba98.7654.3210 is more readable ↵  
 then  
 \$fedcba9876543210. This separator can be used only with binary, hexadecimal and  
 octal numbers (leading with: \$, %, \$).  
 From v0.11 You are able to use also decimal/float number separator (","=ascii ↵  
 184)  
 (on german keyboard: alt+m). This enables sth like: 1,000,123.001=1000123.001, ↵  
 this also  
 improves number reading.

## 1.8 PowerD.guide - Immediate 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

\j#     - single character where # is number (0-255) of character you want.
\x      - extended formating, see below

```

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

```

Extended formating:

Each extension starts with \x, this must be followed by one of following ←  
 characted:

```

t      - full actual time (hh:mm:ss)
d      - full actual date (yy-mmm-dd)
s      - actual second
m      - actual minute
h      - actual hour
Dn     - actual day number
DN     - actual day number (2 digits)

```

---

```

Ds      - actual day short name (like Mon, Tue, ...)
Df      - actual day full name (like Monday, Tuesday, ...)
Mn      - actual month number
MN      - actual month number (2 digits)
Ms      - actual month short name (like Jan, Feb, ...)
y       - actual year (4 digits)
Y       - actual year (2 digits)
v       - compiler version string
V       - compiler version without '$VER: ' string
c       - compiling machine cpu (like MC68LC040)
C       - shorter compiling machine cpu (like LC040)
f       - compiling machine fpu (like MC68882 or none)
F       - shorter compiling machine fpu (like 882)

```

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 extended formatting strings

'Date: \xDn.\Mn\xy\n' will produce sth like 'Year: 4.2.2000'

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'

```

## 1.9 PowerD.guide - Constant definition

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
```

## 1.10 PowerD.guide - Variable definition

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

## 1.11 PowerD.guide - Macro definition

Useful information:

Macros are in PowerD quite limited, macros can be defined (via #define keyword) only in global part of code (outside of procedures). This way defined macros can be used only inside of procedures (between PROC and ENDPROC keywords). Pay attention about this, because preprocessor affects only code between PROC and ENDPROC keywords. You can't use sth like this: ←

```
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)
    RETURN macro(b,c)
ENDPROC
```

From v0.12 You can't use macros inside strings. If You use it, the macro wont be recognized. If You need it use sth like: 'str1'+MACRO+'str2', where MACRO is sth like 'str'.

Description:

```
#define name data
#define name(args) data with args
#ifdef name
#ifdef name
#else
#endif
```

```
#define name data
- each occurrence 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
```

```
#ifdef name
- if macro called name is already defined, the following code will be used.
```

```
#ifndef name
- if macro called name is already defined, the following code will be skipped.
```

```
#define XXX
```

```
PROC main()
#ifdef XXX
```

---



```

    Printf('XXX defined!\n')
#else
    Printf('XXX not defined!\n')
#endif
ENDPROC

```

Compile this with and without the first line and launch it, I thing it is clear ↵  
:)

## 1.12 PowerD.guide - Using 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  $10 \times 20 \times \text{SIZEOF\_LONG}$  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.

## 1.13 PowerD.guide - OBJECT definition

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.

## 1.14 PowerD.guide - Equations

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.
```

This is true only if You use DPRE OPTION.

Operator	Name	Priority	Comment
		D,C,A,E	See OPTIONS.
, OR	Logical OR	1,1,1,1	
&&, AND	Logical AND	1,1,1,1	
NOR	Logical NOR	1,1,1,1	(a NOR b) equals to Not(a OR b)
NAND	Logical NAND	1,1,1,1	(a NAND b) equals to Not(a AND b)
=,<,>,>,<,>=,<=	Conditions	2,2,2,2	
+	Plus	3,5,3,3	
-	Minus	3,5,3,3	
*	Multiply	4,6,5,3	
/	Divide	4,6,5,3	
\	Modulo	4,6,5,3	Floats only with fpu. See NOFPU
	Bit OR	5,4,4,3	Possible only with decimals
!	Bit EOR	5,4,4,3	Possible only with decimals
&	Bit AND	5,4,4,3	Possible only with decimals
<<	Shift Left	6,3,6,3	Possible only with decimals
>>	Shift Right	6,3,6,3	Possible only with decimals
<  or  <	Rotate Left	6,3,6,3	Possible only with decimals
>  or  >	Rotate Right	6,3,6,3	Possible only with decimals

The priority is for each of DPRE, CPRE, APRE and EPRE different, this is to be ←  
 more  
 compatible with other languages (I know APRE is quite useless, but...)

Assigning operators:

Operator	Name	Comment
:=	Copy	
+=	Add	
-=	Subtract	
*=	Multiply	

/=	Divide	
\=	Modulo	Floats only with fpu. See NOFPU
=	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
```

## 1.15 PowerD.guide - Single variable operators

Operator	Name	Comment
+	Useless	Only for you :^)
-	negation	
~	inversion	
&	address	
++	addition	Possible multiple subtractions (see below)
--	subtraction	Possible multiple subtractions (see below)

Negation:

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

Inversion:

Returns inversed (bit) variable

```
a:=16           // a=$00000010
b:=~a           // b=$ffffffef
b~=a            // b=$ffffffef
```

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
```

## 1.16 PowerD.guide - Constant equations

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 in:

- CONST, ENUM, SET, FLAG keywords to define constants.
- default variable sizes like: DEF a[ constant\_equation ]:LONG
- default values in arguments in functions and procedures
- default return values in procedures
- OBJECT item sizes
- global list items
- binary data values

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	

## 1.17 PowerD.guide - Function using

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)
```

## 1.18 PowerD.guide - Returning values

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.

## 1.19 PowerD.guide - PROC definition

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

```

## 1.20 PowerD.guide - REPROC returning values

Description:

## 1.21 PowerD.guide - Using MODULEs

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'.
```

## 1.22 PowerD.guide - External MODULES

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'.

## 1.23 PowerD.guide - Exceptions

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
```

## 1.24 PowerD.guide - Global data

Description:

This is useful for including a binary data/file that will be available within  
program's code. If such data list will be in a procedure, this list will be ←  
overjumped  
to avoid mr guru.

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 manually '\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'
```

## 1.25 PowerD.guide - OO programming

Description:

Object Oriented programming (OOP) is currently very limited in PowerD. If You define an OOP variable like:

```
DEF xyz:PTR TO obj
```

where obj is defined in same way as normal OBJECT. This allows You to define OOP functions to all OBJECTs (like Window, IntuiMessage, etc.). These functions must be defined as following:

```
PROC name(args)(result) OF obj
```

where name, args and result are defined in same way as normal procedures and obj means, that this function will be attached to OBJECT called obj. If function is attached to an OBJECT, it allows You to use OBJECT's items as normal variables:

```
OBJECT testobj
  name:PTR TO CHAR,
  count:LONG,
  weight:DOUBLE

PROC SetName(new:PTR TO CHAR) OF testobj
  name:=new
ENDPROC
PROC Reset() OF testobj
  name:='Unnamed'
  count:=100
  weight:=12.3
ENDPROC
PROC Total() (DOUBLE) OF testobj IS count*weight

PROC main()
  DEF ob:testobj
  ob.Reset()
```

```

ob.SetName('MarK')
ob.weight:=78.1
Printf('Total: \d\n',ob.Total())
ENDPROC

```

## 1.26 PowerD.guide - Polymorphism

Description:

Polymorphism works in D in two ways. At the first You can use it for calling ↵  
different  
procedures with same name (but different arguments) and at the second You can ↵  
use it via  
object oriented programming.

Definition:

```

TPROC procname(list of typed args)

the rest is same as in normal procedures.

```

Difference between PROC and TPROC:

It is very simple. If You define procedure via TPROC, no type conversions will ↵  
be done  
for argument parsing, so the types of arguments must equal:

```

TPROC xxx(x:LONG,y:FLOAT)
TPROC xxx(x:FLOAT,y:FLOAT)
TPROC xxx(a:PTR TO CHAR)
TPROC xxx(a:PTR TO obj)

xxx(1.0,2.3)          // this will call the second procedure
xxx(1,2.3)            // this will call the first procedure
xxx([1,2,3,4]:obj)    // this will call the fourth procedure
xxx('Hello')          // this will call the third procedure

```

If You define TPROCs and PROCs with same names, everything depends on storage ↵  
order in  
memory, so be very carefull if You use this.

Allowed arguments:

As You can imagine, not all arguments are allowed. Only allowed are variables, ↵  
functions, constants, numbers, strings and pointers.  
Equations are not allowed! Also everything like IF, SELECT etc that can return a ↵  
value  
is not allowed.

## 1.27 PowerD.guide - LOOP definition

### 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

```
LOOPexp
    code
ENDLOOP
```

LOOP exp DO commands // see DO keyword

where exp can be constant, expression, etc.

### 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.

### Examples:

```
LOOP 10
    Printf('Hello\n')
ENDLOOP
```

This will write 'Hello' ten times.

```
LOOP a:=5
    Printf('Hello(\d)\n',a)
ENDLOOP
```

This will write:

```
Hello(5)
Hello(4)
Hello(3)
Hello(2)
Hello(1)
```

## 1.28 PowerD.guide - FOR definition

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
```

```
FOR a (TO/DTO) b [STEP c] command [IS list]
```

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

## 1.29 PowerD.guide - WHILE definition

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
ALWAYS
  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 ALWAYS.

```

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

## 1.30 PowerD.guide - REPEAT definition

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
```

### 1.31 PowerD.guide - IF definition

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    Printf('Too old!\n')

```

## 1.32 PowerD.guide - SELECT definition

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'

```

## 1.33 PowerD.guide - DO keyword

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. Everything you like.

### 1.34 PowerD.guide - THEN keyword

Description:

Syntax:

### 1.35 PowerD.guide - EXIT/EXITIF keyword

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/GIVES/GIVING/RETURNING) list]  
 EXIT [DO code] [(IS/GIVES/GIVING/RETURNING) list]

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

Comment:

GIVES, GIVING and RETURNING keywords are all the same as IS keyword, it is  
 allowed only  
 to be more readable.

### 1.36 PowerD.guide - JUMP keyword

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)

### 1.37 PowerD.guide - NEW/END/INC/DEC/NEG/NOT...

Description:

These are similar to single operator operations, but this can be used with list of variables like here:

NEG a,b,c

is the same as

a:=-a

b:=-b

c:=-c

Syntax:

NEG a is the same as a:=-a

NOT a is the same as a:=~a

INC a is the same as ++a

DEC a is the same as --a

Description of NEW:

NEW calls function to allocate a chunk of memory, with size given as below, and writes pointer of this chunk to given variable. If allocation fails, "MEM" exception is raised.

DEF a:PTR TO obj,b=20,c=3,d:PTR TO DOUBLE

NEW a

// equals to IF (a:=AllocVec(SIZEOF\_obj, MEMF\_PUBLIC|MEMF\_CLEAR))=NIL THEN Raise ("MEM")

NEW a[10]

```
// equals to IF (a:=AllocVec(10*SIZEOF_obj, MEMF_PUBLIC|MEMF_CLEAR))=NIL THEN ↵
    Raise("MEM")

NEW a[b*c+2]
// equals to IF (a:=AllocVec((b*c+2)*SIZEOF_obj, MEMF_PUBLIC|MEMF_CLEAR))=NIL ↵
    THEN Raise("MEM")

NEW d[10]
// equals to IF (d:=AllocVec(10*SIZEOF_DOUBLE, MEMF_PUBLIC|MEMF_CLEAR))=NIL THEN ↵
    Raise("MEM")
```

you can also use list of allocations like:

```
NEW d[4],a[b]
```

Description of END:

END must be used to deallocate a NEW allocated chunk of memory.

```
NEW d[4],a[b]
...
END a,d
```

## 1.38 PowerD.guide - LIBRARY definition

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
```

## 1.39 PowerD.guide - Linked libraries

**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

## 1.40 PowerD.guide - Internal functions

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

Inline functions:

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

These functions are currently inline and hardcoded as fpu instruction. Some functions wont be compiled with NOFPU OPTions.

Linked library functions:

See linked libraries

PowerD library functions:

- String/EString functions
- Math functions
- Intuition functions
- Graphics support functions
- DOS support functions
- Miscelaneous functions

## 1.41 PowerD.lib - String/EString functions

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 formatted estring. Where arguments are same types as used in formatstr.

`StringF(string, formatstr, arguments)`

This function generates formatted 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 skipped 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.

```
IsHex(byte)
```

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

```
IsBin(byte)
```

Returns TRUE if byte is a binary 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 allowed. 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.

## 1.42 PowerD.lib - Math functions

Abs(a)  
 Returns absolute value of a.

And(a,b)  
 Returns a&b.

BitCount(value)  
 Returns # of bits contained in 32bit value.  
 BitCount(\$0f0) returns 4 : \$0f0=%0000.1111.0000  
 BitCount(\$124) returns 3 : \$124=%0001.0010.0100

BitSize(value)  
 This returns size of bit field contained in the 32bit value  
 BitSize(\$2c) returns 4 : \$2c=%0010.1100  
                   size is this :       ^^ ^^  
 BitSize(5) returns 3 : 5=%101  
                   size is this :     ^^^  
 BitSize(\$124) returns 7 : \$124=%0001.0010.0100  
                   size is this :       ^ ^^^^ ^^  
 BizSize(a) equals to HiBit(a)-LoBit(a)

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.

HiBit(value)  
 This returns position of highest active bit the 32bit value

LoBit(value)  
 This returns position of lowest active bit the 32bit value

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.

## 1.43 PowerD.lib - Intuition functions

OpenW(l,t,w,h,i,f,n=NIL,s=NIL,st=1,g=NIL,ta=NIL) (PTR TO Window)

This function returns pointer to opened window if everything went ok or NIL if ↩  
no. If so  
stdrast variable is set to this window, and all graphics functions will be drawn ↩  
in this  
window. This function is AmigaE's equivalent.

Arguments:

l - left edge of the window  
t - top edge of the window  
w - width of the window  
h - height of the window  
i - idcmp flags  
f - window flags  
n - window name  
s - pointer to screen to open window on  
st - screen type (1 to open window on WB screen, 15 on custom screen)  
g - window gadgets  
ta - window tagitems (see 'intuition/intuition.m')

CloseW(w:PTR TO Window)

This function closes OpenW() opened window, this also clears stdrast variable if ↩  
it is  
this's window.

SetStdRast(rp:PTR TO RastPort) (PTR TO RastPort)

This functions setups stdrast variable to the rp and returns old stdrast to ↩  
store.

OpenS(w,h,d,m,n=NIL,t=NIL) (PTR TO Screen)

This function returns pointer to opened screen if everything went ok or NIL if no. This function is AmigaE's equivalent, but it doesn't initialize stdrast, drawing to screen's rastport shouldn't be used, try window!

Arguments:

w - width of the screen

h - height of the screen

d - depth of the screen (1=2 colours, 2=4 colours, 3=8 colours, ...)

m - screenmode (see 'graphics/modeid')

\$0000 - lores

\$8000 - hires

\$0004 - laced

\$8020 - super hires

\$0800 - ham

\$0080 - ehb

etc.

if You want more specific screen like multiscan productivity, use

SA\_DisplayID tag

in tag item.

t - screen tagitems (see 'intuition/screens.m')

CloseS(s:PTR TO Screen)

This function closes OpenS() opened screen.

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.

## 1.44 PowerD.lib - Graphics functions

All of these functions uses variable stdrast which will be initializes by opening window with OpenW() function or via SetStdRast() function.

Colour(a,b=-1)

This functions sets default colour a and b (b is currently useless)

Plot(x,y,c=-1)

This function draws pixel coloured with colour c, if no colour specified, ←  
default colour  
will be used.

Line(xa,ya,xb,yb,c=-1)

This functions draws line from xa,ya to xb,yb by colour c, if colour not ←  
specified  
default colour will be used.

Box(xa,ya,xb,yb,c=-1)

This functions draws rectangle from xa,ya to xb,yb by colour c, if colour not ←  
specified  
default colour will be used.

Circle(x,y,r,c=-1)

This function draws circle with center on coords x,y and with radius r coloured ←  
with  
colour c, if no colour specified, default colour will be used.

Ellipse(x,y,rx,ry,c=-1)

This function draws circle with center on coords x,y and with radiuses rx and ry  
coloured with colour c, if no colour specified, default colour will be used.

SetColour(s,n,r,g,b)

This function sets colour n of screen s to value given by r,g,b.

## 1.45 PowerD.lib - DOS support functions

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.

## 1.46 PowerD.lib - Miscellaneous functions

AllocVecPooled(pool,size)

This functions calls exec's AllocPooled() and stores also it's size, it is ←  
possible to  
free such chunk of memory if You don't know it's length. See also FreeVecPooled ←  
().

FreeVecPooled(pool,mem)

This function frees memory allocated by AllocVecPooled().

SizePooled(mem)

This function returns size of memory chunk allocated by AllocVecPooled().

CtrlC()

CtrlD()

CtrlE()

CtrlF()

These functions check 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.

ULong(a)

UWord(a)

UByte(a)

Returns unsigned byte/word/long value what is on address specified by a.

ULong(a) equals to Long(a) (both return the same)

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  
than 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.

## 1.47 PowerD.guide - Internal constants

These constants are set always before compilation:

TRUE = -1

FALSE = 0

NIL = 0

PI = 3.141592653589

OLDFILE = 1005 // for file opening

NEWFILE = 1006 // for file opening

Special (changeable) constants:

```
OSVERSION = required 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'
```

## 1.48 PowerD.guide - Options

Options are in PowerD introduced by keyword 'OPT' or 'GOPT', it can be defined everywhere outside PROCedures.

OPT defines local file options. If You define this in a module and use this ↵  
 module,  
 all of the options will be used only in the module, not in source where do You ↵  
 use the  
 module via MODULE keyword. Some of option keywords are always global since as ↵  
 local  
 they are nonsensefull, all of these are signed with \*. If GOPT used, all options  
 will be known as global and will be used everywhere.

If You use keywords like HEAD or NOHEAD, always the last one will be used.

HEAD/K\* (default: 'startup.o')

Sets startup #?.o file, this file should be located in 'd:lib' directory or it ↵  
 's name

must begin with '\*' charecter and full path of custom startup object file.

This automatically switches NOHEAD swtich off.

(example: OPT HEAD='\*hd2:myheads/best.o')

NOHEAD/S\* (default: head is enabled)

This switch disables adding linkable startup head.

If You use this You must open all libraries and set all variables.

LINK/K\* (default: no linkable file)

This allows You to define linking files in source code. By this way Yu can ↵  
 define all

object (#?.o) and link-library (#?.lib) files. This keyword can be used more ←  
times, so

You can define more linkable files.

This automatically sets OBJECT output name to default.

(example: OPT LINK='\*hd1:lib/math040.lib',LINK='amiga.lib')

OBJECT/K/S\* (default: '<programe>.o')

This sets output object file name. If You define only 'name.o' it will be ←  
located in

current directory, if You define it with path (eg: 'hd0:objects/prog.o') it ←  
will

locate output file in directory given with path.

NOSTD/S\* (default: reads powerd.m module)

This switch disables reading of 'dmodules:lib/powerd.m' module. It means: if ←  
You

enable this switch, You wont be able to use functions located in this module. ←  
I think

removing this is only needed if You have another set of default functions for ←  
PowerD.

DEST/K\* (default: '<programe>' without extension '.d')

This is the same as OBJECT/K keyword, it only allows You to set executable ←  
name after

linking pass.

PRIVATE/S (default: only public data allowed)

This enables using of private data in the source.

DPRE/S, CPRE/S, APRE/S, EPRE/S (default: DPRE enabled)

These keywords sets precedence of signs/operators, default is DPRE.

See Equations for more information.

NOSOURCES/S (default: writing of source with errors enabled)

This switch allows You to switch off writing of source code with errors.

AMIGAE/S (default: disabled)

This switch raises compatibility with AmigaE programming language:

- object names and object item names are changed to lower case
- EPRE switch is switched on
- HANDLE keyword in PROC definition for exceptions allowed (it will be ←  
skipped)
- EXCEPT DO changed to EXCEPTDO
- EXIT keyword changed TO EXITIF

OSVERSION/N (default: 0)

This sets minimal os version required, it is currently quite useless.

NOFPU/S (default: fpu is enabled)

This switch allows You to use floating point computations without a ←  
mathematical

coprocessor (FPU). This converts fpu instructions to non-fpu instructions and ←  
use

mathieeedoubbas.library and mathieeedoubtrans.library. I added this only to be ←  
able to

compile and try sources also on Amigas without fpu, the generated code is very ←  
slow,

so use it only if there is no another way.  
 Always the better way to use NOFPU is to use a module instead of the NOFPU keyword in  
 the source. See NOFPU to get more information about it.

PowerD non-fpu instruction converter is not done, so use it with care!!!!

CPU/N,FPU/N (default: 68020,68881)

This allows You to select a cpu to generate code for. 68000 and 68010 makes currently  
 same code as 68020 and 68030. If 68040 or 68060 entered then coprocessor is also  
 enabled. If You have eg: 68LC040, use: OPT CPU=68040,NOFPU. If You have 68030 and  
 68882 then use: OPT CPU=68030,FPU=68882. Be sure than there is currently no difference  
 between 68881/882 and 68040/060. These optimizations will be added in future. If You  
 enter FPU=0, it is the same as NOFPU.

MODULE/K/S (default: no binary module)

This option causes binary module production. If MODULE keyword is alone, then PowerD  
 will generate module called '<modname>.b'. If MODULE='xxx.b' is used, then PowerD will  
 generate module called 'xxx.b'.  
 If You want use this option, then insert it to the first line of your source, or  
 before all PowerD keywords, else the generated module will contain all constants, etc.  
 from all used MODULEs.  
 If You want to create your own binary module, see: binary module

OPTIMIZE/N (default: 0)

This set bits needed for optimizations:  
 - bit 0 - all unnecessary tst instructions removed  
 - bit 1 - muls/divs/moveq optimizations

If only OPTIMIZE (without number) used, optimizations will be set to -1.

All optimizations will be enabled by -1 (\$ffff.ffff) value.

and You can use your own OPTions.

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.

OPT OBJECT='hd2:objects/proggy.o',DEST='hd2:proggy'

This will generate the object file into 'hd2:objects/proggy.o' and then it will be



everything linked into 'hd2:proggy'

## 1.49 PowerD.guide - Preset user OPTions

Description:

This allows You to predefine Your own OPT keywords, currently are supported ↵  
only  
single word keywords.  
Most of these definitions should be defined in dmodules:powerd/options.m file. ↵  
This  
file is supported from v0.09 it is always loaded before your source is compiled, ↵  
so  
never do anything with this file, if You don't know what does it do!!!  
Between SETOPT and ENDOPT can be everything You like, like constant ↵  
definitions,  
variable definitions, another OPTions, MODULEs, PROCedures etc.  
This allows You to use OPT DOSONLY instead of MODULE 'startup/startup\_dos' and ↵  
it is  
shorter, isn't it?

Syntax:

```
SETOPT name
    put here everything you like
ENDOPT
```

Example:

This should defined be defined in dmodules:powerd/options.m file.

```
SETOPT IEEE
    MODULE 'startup/startup_ieee'
ENDOPT
```

and if You use in your code:

```
OPT IEEE
```

the module 'startup/startup\_ieee' will be processed.

## 1.50 PowerD.guide - NOFPU

If this OPTion set You must do something more to be able to compile non fpu ↵  
sources.  
The first thing is to open mathieeedoubbas.library and mathieeedoubtrans.library ↵  
. Then  
You shouldn't use normal PowerD.lib, because some functions use fpu only code. ↵  
So set  
NOSTD option and LINK='d:lib/powerd\_ieee.lib' to be linked instead. All these ↵  
operations

are made in dmodules:startup/startup\_ieee.m and dmodules:startup/ ←  
 startup\_dos\_ieee.m so  
 put eye on them.

See also d:lib/startup\_ieee.ass and d:lib/startup\_dos\_ieee.ass.

If 'AI=ASMINFO/S' cli switch enabled, all fpu instructions will appear as a ←  
 comment.

Currently works with this option only following functions and operators:

Sin(), Cos(), Tan(), ASin(), ACos(), ATan(), Sinh(), Cosh(), Tanh(), Sqrt(), Pow ←  
 (),  
 FAbs(), Ln(), Log()  
 other functions are fpu only!!!  
 +,-,\*,/ (no \ (modulo))

## 1.51 PowerD.guide - CLI/Shell Syntax

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)	
	It now works better, so You can debug also linked files
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
O=OPTIMIZE/N	- same as OPT link opt} OPTIMIZE
I=INFO/S	- When compilation is finished, some information is wrote ←
to	
	stdout.
CPU/N	- same as OPT CPU
FPU/N	- same as OPT FPU
NOFPU/S	- same as OPT NOFPU
AUTHOR/S	- This is quite useless, but who knows...

## 1.52 PowerD.guide - Error messages/warnings

Some error messages have no comment. Such are clean for everyone or I haven't ←  
 time to  
 comment it.

Not enough memory

#### Letter expected

You probably used sth else than a name, this error also appear when You forgot to write a name. ↵

#### Unexpected end of file

Your program structure coherency is probably damaged, check if all IFs, FORs, brackets, ENDPROCs etc are where they should be. ↵

#### '=' expected

Some OPTions (like HEAD, OBJECT, etc) requies '='  
CONST requies '=' after a constant name

#### Unknown identifier

You forgot to define a variable or constant, or You wrote it badly.

#### Unknown function

This function or procedure is not defined, of You wrote it's name badly.

#### Unexpected character

You probably forgot ",", " or ")" in a list or in a function.  
Or You wrote sth else where a number is expected.  
Or You wrote an illegal character between " and " (two double quotes).

#### Unknown OO function

This OO function is not joined to the OBJECT.

#### Unknown constant

This constant doesn't exist. This appears only where constants/numbers are expected. ↵

#### Unknown label

This label is not defined. See if your JUMP has right label.

Incomplete IF-ELSEIF-ELSE-ENDIF  
Incomplete WHILE-ELSEWHILE-ALWAYS-ENDWHILE  
Incomplete FOR-ENDFOR  
Incomplete SELECT-CASE-DEFAULT-ENDSELECT

Incomplete LOOP-ENDLOOP  
Incomplete REPEAT-UNTIL  
Incomplete SUBDEF-SUBEND

One of these errors appears if You, for example, replaced ENDIF with ENDSELECT ↵  
etc.

Missing apostrophe

You probably forgot apostrophe (') in front of a string (usually with MODULE ↵  
keyword)

Unknown OPT

Given OPTion is not available, see OPT or dmodules:powerd/options.m.

Unmatched brackets

You counts of "(" and ")" or "[" and "]" are not same. Check if You haven't ↵  
forgotten  
one.

Unexpected end of string

You forgot to add apostrophe (') to the end of given string.

Unexpected end of OBJECT

You probably added a comma (",") after last OBJECT item.

Unexpected end of constant list

You probably added a comma (",") after last constant.

Unknown processor, generating for MC68030+MC68882

You set illegal cpu or fpu type.

Unknown OBJECT

Given OBJECT is not defined or the name is wrong.

UNION can be defined only after NEWUNION

You started new union with UNION instead of NEWUNION.

Divide by zero

---

Modulo by zero

CONST mode requied for

You probably used sth else than CONST keyword wif given constant.

Too many arguments

Not enough arguments

Too many return values

Not enough return values

Too many names of one item

Count of names per item in OBJECT is limited to 16.

At least one argument in TPROC requied

Typed procedures expects at least one argument. (else they aren't typed :)

Octal numbers consists only from "0" to "7" numbers

You wrote illegal character in octal number. (leading with \$)

OBJECTs must be global

OBJECTs can be defined only between PROCedures, not in them.

This MODULE requies newer compiler

Given binary module is compiled with newer version of PowerD.

It can't be used with this version.

Assembler error occured, please contact the author

This error appears when PowerD made bad code. If You won't repair it in the #?.d ↵  
source.

Send bad source and asm output and some more info to author.

Linker error occured, please contact the author

There is probably problems with linked libraries, check if they have all needed  
functions, or if You didn't forget OPT LINK='libname'.

This error appeared in v0.11 with binary modules, which contained LINK.

Syntax Error

This error is available from v0.12 and it appears when You have forgotten a ↵  
comma in a

list of constants, defines etc.

---

']' expected  
'(' expected after  
)' expected  
'\ ' expected  
'd'h' or 's' expected after '\ '  
Incoherent program structure  
This MUST return a value(s)  
Retyping requies an OBJECT name  
Unknown error, contact author  
Unknown OBJECT/UNION member  
' or ')' expected in the list  
' or ']' expected after an index  
'TO' or 'DTO' expected  
Illegal condition  
LONG expected  
CASE or DEFAULT must be after SELECT  
Unknown constant function  
Can't return multiple values  
'\_' not found in FLAG name  
FLAG/SET requires decimal number(no float)  
Can't swap a CONST  
Can't return a value  
CASE expected  
CASE, DEFAULT or ENDSELECT expected  
Useless SELECT structure  
ENDSELECT expected  
ENDIF expected  
ENDWHILE expected  
ENDLOOP expected  
ENDFOR expected  
UNTIL expected  
STRING/PTR can be copyed only with ':='  
Assembler source code string expected after '='  
Illegal type conversion  
This operator can't be used with floats  
This operator can't be used with constants  
This operator can't be used in this case  
Swapping different types  
Only CONSTant possible  
CONSTant requied as argument  
Variable declared 2 or more times  
Macro is too large (max. is 16384)  
STRING type must have a size  
Unknown \x extension  
Unable to read file  
Unable to open file  
Requied arguments missing  
Unable to open source file  
Unused variables in  
Unused procedures

## 1.53 PowerD.guide - PowerD syntax

---

Tabulators should have size of 3.

## 1.54 PowerD.guide - PowerD and AmigaE differences

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

0.123456 (FLOAT)

SET AF\_1,AF\_2,AF\_3

0.123456789 (DOUBLE)

0.123456

?

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

## 1.55 PowerD.guide - How to use C code

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

## 1.56 PowerD.guide - How to use Assembler code

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 asselblder 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

```

## 1.57 PowerD.guide - How to create Header

Description:

Header (resp. startup file) is a piece of code what is run at the beginning of the executable. It usually makes some variable initialisations, library opening etc. Each header should call function called 'main()'. It is usually written in assembler to get the best performance, 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

```

## 1.58 PowerD.guide - How to create linked library

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. Finally 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)
```

## 1.59 PowerD.guide - How to create Library

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

## 1.60 PowerD.guide - How to create binary module

Warning:

Reading binary modules is about 2-10 times faster then ascii modules, but it  
 may cause  
 some problems, so if some appear, use only ascii modules.

Is probably, that I will rewrite module format in future (to be much faster),  
 so don't  
 delete your old ascii modules (better: backup them :).

---

Compiling:

If You define OPT MODULE in a #?.d file, it will be compiled as a module into  
 #?.b  
 file. Second (better) way is to compile #?.m file (don't forget the '.m'). It  
 does the  
 same. Eg.:

Limitations:

Binary modules can currently contain only:

MODULE definitions,  
 CONSTants,  
 OBJECT definitions (no unions currently allowed!!!),  
 #define definitions,  
 EDEF definitions,  
 LINK OPTions and  
 HEAD OPTion

## 1.61 PowerD.guide - Why I did it

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 of  
 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!

## 1.62 PowerD.guide - Instalation

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.

## 1.63 PowerD.guide - Features

Features when comparing with C/C++:

- + multiple return values (8 for m68k,  $2^5$  for ppc)
- + lists can be defined/used everywhere you like/need, not on the definition only
- + more readable syntax
- + binary modules
- + much faster
- + requies less memory
- \ensurermath{\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)
- + changable introduction of precedence
- + names can contain high/low letters in all cases
- + for object oriented programming you don't have to use self.#?, you can use only #?
- + better polymorphism
- + 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

## 1.64 PowerD.guide - Future

In next release:

- new feature REPROC
- new feature SUB
- complete preprocessor
- move Raise() functions into powerd.lib
- finish binary module support (object unions)
- removation of all bugs (I hope :)
- new assembler writer with many new optimizations

In progress or near future:

- use of math libraries instead of fpu instructions
- more of object oriented programming
- intelligent optimizations
- elimination of bugs
- to be more "fool-proof"
- 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 difficult)
- To be modular (eg.: add a module to generate code for other processor)
- link libraries with useful functions
  - audio/video/picture loading/playing/showing/saving
  - 3d functions for 3d games

In vision:

- Enterprise support (NCC1701D or better requied :)

This I will probably never do:

- Windows95/98/2000/3000/4000/NT version
-

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

## 1.65 PowerD.guide - History

- Version 0.12 (4.2.2000) (dc.e: 12345 lines, 396396 bytes):
- removed bug in optimizations
  - removed some bugs in docs :)
  - powerd.lib functions added into index of this document
  - some new things added into index
  - added few new functions (intuition, graphics, pools):
    - OpenW(), CloseW(), OpenS(), CloseS(), SetStdRast()
    - Plot(), Line(), Box(), Circle(), Ellipse(), Colour(), SetColour()
    - AllocVecPooled(), FreeVecPooled(), SizePooled()
  - removed some bugs in binary modules
    - saved binary modules has now slightly different format
    - added EDEFs and #defines
    - added dmod tool, to show binary modules
  - improved macros, now You can use #ifdef, #ifndef, #else and #endif
  - preprocessor can from now convert: xxx=.yyy to xxx:=xxx.yyy
    - this slightly slowed down generate pass, because preprocessor is now called always ←
  - I rewrote many functions to don't use stack, but registers = shorter and faster from powerd.lib ←
  - Even() and Odd() flipped :)
  - local e-strings now works again
  - improved asm string writer, now You can use eg: '\0\0\0\0', to allocate 4 bytes long string of zeros ←
  - main code reader routine is rewritten to be faster
  - added Syntax Error (sorry :)
  - I finally added description of some errors in this document, very incomplete
  - this error currently quits powerd
  - added many new string extensions
  - removed some enforcer hits

- Version 0.11 (17.1.2000) (dc.e: 11066 lines, 361271 bytes):
- added , (ascii 184) decimal number separator
  - improved LOOP x where x can be now constant/number
  - added binary module support (still very limited)
    - added MODULE OPTion
  - improved procedure/function finding routines, up to 28 times faster
  - added few new functions:
    - UByte(), UWord(), ULong(), HiBit(), LoHit(), BitCount(), BitSize()
  - improved startup files, arg variable now work
    - added startup\_dosarg.m module and DOSARGONLY OPTion to allow arg variable with dos opening only ←
  - inlined IF now work better
  - added one new example
  - no more linker error like line too long or similar
  - removed some bugs in modules and added ExecBase variable
  - removed some other bugs

Version 0.10 (5.1.2000) (dc.e: 9563 lines, 320559 bytes):

---

This release has nothing from big improvements, because some heavy bugs appeared ←  
, so  
please wait, again...  
- added . number separator  
- now You can use sth like a:=&main where main is a procedure  
- added octal number support  
- now should work nofpu floats correctly  
- removed heavy bug with strings (didn't work \s, \d, etc right after apostrophe ←  
)  
- removed some bugs

Version 0.09 (31.12.1999):

- added NEWFILE/OLDFILE constants
- added new OPTion and cli argument called OPTIMIZE for optimizations
- new SETOPT/ENDOPT keywords allows You to set your custom OPTions
- improved DEBUGSYM cli switch
- now works inlined IF, WHILE, etc. again (like: a:=IF b THEN c ELSE d)
- now odd byte/word array or string length allowed
- added some optimizations including output of gained bytes, but don't trust it ←  
too much
- removed many bugs with nofpu floats (but not all :( )
- some examples added
- some new functions
- some bugs removed

22.12.1999 - My Develop partition died. I lost all of my developed software, ←  
but I  
backuped whole PowerD source and it's datas two days ago :), so this will be ←  
only small  
slow down until I will get all needed software back :(. And I still don't have ←  
my  
Blizzard...

Version 0.08 (20.12.1999):

- added some OO features
- added NOFPU, CPU and FPU OPTions and CLI arguments.
- added #?\_ieee.m startup modules to be used instead of NOFPU option
- added powerd\_ieee.lib (without fpu requirements)
- added small asm code optimizer
- added TPROC definition for polymorphism.
- now is allowed 'ELSE command' instead of 'ELSE DO command'
- now is allowed 'FOR a TO b command' instead of 'FOR a TO b DO command'
- removed bug: a+=x where x is a variable didn't work (thanx to Mauro Fontana)
- removed enforcer hits on errors
- and many small improvements

Version 0.07 (5.12.1999):

- ALWAYS changed to ALWAYS, really silly mistake :)
  - DPRE, CPRE, EPRE, APRE OPTions added.
  - added && and || in conditions (AND and OR works ofcourse too)
  - totally rewrote equation/condition generator, now is allows bigger freedom of programming.
  - removed enforcer hits
  - improved OPTions, now global and local options, DEST works, OBJECT works
  - added: NEW, END, NEG, etc. keywords
  - index added to this document
-



Version 0.06 (21.11.1999):

- some bugs removed (:=: didn't work in 0.05 and 0.05b)
- improved LOOP command, idea by Marco Antoniazzi
- improved 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):

- assignments changed from eg: := to \*= to be more compatible
- new: ASM and APROC()
- many bugs removed
- added differences between AmigaE and PowerD in this documentation.

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 not 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 :)

## 1.66 PowerD.guide - Bugs

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

Known bugs list:

dc:

- if You use AND/OR/|| instead of &| compiler will generate bad code, usually ↵  
     phxass  
     fails on line containing sth like and.l d32,d0

- powerd compiles wrong sth like:

OBJECT rgb

    r,g,b

OBJECT img

    x,y,

    pix:PTR TO rgb

DEF img:PTR TO img,pix:PTR TO rgb,c

pix:=img.pix[y\*img.w+x]                      // this currently doesn't work !!!

c:=img.pix[y\*img.w+x].r                      // but this works fine !!!

- powerd sometimes on some mistakes in the source code jumps into infinite ↵  
     unbreakable  
     loop.

powerd.lib/powerd\_ieee.lib:

## 1.67 PowerD.guide - Limitations

Each OBJECT member name can have maximally 16 synonyms.

Count of return values is limited by count of data/fpu registers (68k=8(+8fpu), ↵  
     ppc=cca.25(+32fpu)).

Length of each name used in PowerD should be shorter then 64 characters.

## 1.68 PowerD.guide - Requirements

Requirements:

An Amiga or a compatible computer  
 OS 3.0 (V39+)

---

HD

2 megs of memory, but at least 4 are strongly recommended

PhxAss (by Frank Wille)

PhxLnk (by Frank Wille)

Recommendations:

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 :^)

Successfully 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+Blizzard1240/40+64MB+3.0+HD

A1200+Blizzard603e/160+040/25+64MB+3.1+HD

## 1.69 PowerD.guide - Registration

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.

## 1.70 PowerD.guide - Thanks

phase5 - for their wonderful blizzards and for staying with (classic) Amiga

gw2k - I really don't know if thanx or not at this moment

microsoft - for producing still the worst operating system :^)

Special thanks to:

The most important one:

Petr Voralek for lending me his B1240 when I had plain A1200

Tomasz Wiszkowski for his advices, comments, nice emails, Creative and more

Mauro Fontana for his bug reports and ideas

Marco Antoniazzi for his bug reports and ideas

Przemyslaw Szczygieski for his oo advices

and to every body who sent me supporting emails.

## 1.71 PowerD.guide - Author's Address

---

Snail mail:

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

E-Mail:

kuchinka@k332.feld.cvut.cz (preffered)  
kuchinka@student.fsid.cvut.cz  
kuchinka@student.fsik.cvut.cz  
kuchinka@pruvodce.cz

WWW:

<http://lide.pruvodce.cz/kuchinka>

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)

## 1.72 PowerD.guide - ASCII Table

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"	140	"\j140"	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	"j"	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	"ç"	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	" ←
ø "							
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	←
"ö"							
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	" ° "	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"

## 1.73 PowerD.guide - Index

#define	macro definition
#ifdef	macro definition
#ifndef	macro definition
#else	macro definition
#endif	macro definition
++	single: post/pre incrementation
--	single: post/pre decrementation
&	single: address
^	single: long on address
~	single: inverse
-	single: negative
+	single: useless
-	operator: minus
+	operator: plus
*	operator: multiple
/	operator: divide
\	operator: modulo
	operator: bit or
!	operator: bit exclusive or
<<	operator: shift left
>>	operator: shift right
< or <	operator: shift left
> or >	operator: shift right
=	condit: equal
<	condit: less
>	condit: bigger
<= or ==	condit: less or equal
>= or ==	condit: bigger or equal
<>	condit: not equal
&& or AND	condit: logic and
or OR	condit: logic or
ABS()	constant: absolute
Abs()	internal function: absolute value
ACOS()	constant: arcus cosinus
AllocVecPooled()	internal function: allocate memory
ALWAYS	WHILE definition
AMIGAE	OPT: AmigaE compatibility
AND	condit: logic and
And()	internal function: bit and
APRE	OPT: assembler like precedences
APROC	assembler-only procedure
APTR	same as PTR
ASIN()	constant: arcus sinus
ASM	inline assembler code
ATAN()	constant: arcus tangents
BitCount()	internal function: count of bits in long
BitSize()	internal function: size of bit field in long
Bounds()	internal function: bound a value
Box()	internal function: draw a box
Byte()	internal function: get a byte
BPTR	same as PTR
BYTE	global data definition
BYTE	type
CASE	SELECT definition
CEIL()	constant: ceil value
CHAR	same as UBYTE

Circle()	internal function: draw a circle
CloseS()	internal function: close a screen
CloseW()	internal function: close a window
Colour()	internal function: set default colour
CONST	CONST definition
COS()	constant: cosinus
COSH()	constant: hyper cosinus
CPRE	OPT: c/c++ like precedences
CPTR	same as PTR
CPU	OPT: selects processor for assembler generator
CtrlC()	internal function: check if Ctrl+C pressed
CtrlD()	internal function: check if Ctrl+D pressed
CtrlE()	internal function: check if Ctrl+E pressed
CtrlF()	internal function: check if Ctrl+F pressed
DEC	multivariable decrementation
DEF	variable definition
DEFAULT	SELECT definition
DEFB	BYTE variable definition
DEFD	DOUBLE variable definition
DEFF	FLOAT variable definition
DEFL	LONG variable definition
DEFUB	UBYTE variable definition
DEFUL	ULONG variable definition
DEFUW	UWORD variable definition
DEFW	WORD variable definition
DO	multiple commands on single line
DOUBLE	global data definition
DOUBLE	type
DPRE	OPT: default/PowerD precedences
DREPROC	DREPROC definition
DTO	FOR definition
EDEF	external object variable definition
Ellipse()	internal function: draw an ellipse
ELSE	IF definition
ELSEIF	IF definition
ELSEIFN	IF definition
ELSEWHILE	WHILE definition
ELSEWHILEN	WHILE definition
END	multiple variable deallocation
ENDASM	end of inline assembler code
ENDFOR	FOR definition
ENDIF	IF definition
ENDLOOP	LOOP definition
ENDOPT	SETOPT definition
ENDPROC	end of procedure definition
ENDSELECT	SELECT definition
ENDUNION	OBJECT: unions
ENDWHILE	WHILE definition
ENUM	ENUM constant definition
EOr()	internal function: bit exclusive or
EPRE	OPT: AmigaE link precedences
EPROC	external object PROC definition
EStrAdd()	internal function: add to e-string
EStrCopy()	internal function: copy to e-string
EStringF()	internal function: format e-string
EStrLen()	internal function: length of an e-string
EStrMax()	internal function: maximal length of an e-string

Even()	internal function: returns TRUE in value is even
EXCEPT	procedure exception definition
EXCEPTDO	procedure exception definition
EXIT	early exit from loops
EXITIF	early exit from loops
EXITIFN	early exit from loops
EXP()	constant: exponent
FAC()	constant: factorial
FALSE	internal constant: 0
FileLength()	internal function: returns file length
FLAG	FLAG constant definition
FLOAT	type
FLOOR()	constant: floor value
FOR	FOR definition
FPU	OPT: selects coprocessor for assembler generator
FreeVecPooled()	internal function: frees a pooled memory
GIVES	EXITIF: same as IS
GIVING	EXITIF: same as IS
GOPT	global options
HEAD	OPT: selects linkable head file
HiBit()	internal function: highest bit in long
IF	IF definition
IFN	IF definition
INC	multiple variable incrementation
Inp()	internal function: reads byte from a file
INCBIN	include binary file
InStr()	internal function: position of str b in str a
INT	same as WORD
IS	return a list of values (it can be used in many cases)
IsAlpha()	internal function: true if the character is a letter
IsBin()	internal function: true if the character is a binary number
IsHex()	internal function: true if the character is a hexadecimal ←
number	
IsNum()	internal function: true if the character is a decimal number
JUMP	jump to a label
KickVersion()	internal function: true if given version is lower of equal to ←
system	
LIBRARY	LIBRARY definition
Line()	internal function: draw a line
LINK	OPT: add a link to linking list
LIST	inlined list of arguments
LIST OF	inlined list of types arguments
LN()	constant: natural logarithm
LoBit()	internal function: lowest bit in long
LOG()	constant: decimal logarithm
LONG	global data definition
LONG	type
Long()	internal function: get a long
LOOP	LOOP definition
LowerStr()	internal function: translates str to lower case
LPROC	external object C compatible procedure definition
Max()	internal function: bigger of two
MODULE	module definition
MODULE	OPT: generates binary module
MidEstr()	internal function: copy somewhere to e-string
Min()	internal function: smaller of two
Mouse()	internal function: check if mouse pressed



MouseX()	internal function: mouse position x
MouseXY()	internal function: mouse position x,y
MouseY()	internal function: mouse position y
NAND	condit: logic negative and
NEG	multiple variable negation
NEG()	constant: negative value
Neg()	internal function: negative value
NEW	multiple variable memory allocation
NewEStr()	internal function: allocate new e-string
NEWFILE	internal constant: 1006
NEWUNION	OBJECT: unions
NIL	internal constant: 0
NOFPU	OPT: disables coprocessor for assembler generator
NOHEAD	OPT: link executable file without a head
NOR	condit: logic negative or
NOSOURCES	OPT: don't write source lines after errors
NOSTD	OPT: don't read d:lib/powerd.m module
NOT	multiple variable not-ation
Not()	internal function: inverse value
OBJECT	OBJECT definition
OBJECT	OPT: force generate object instead of object file
Odd()	internal function: returns TRUE in value is odd
OF	OBJECT: linking objects
OLDFILE	internal constant: 1005
OpenS()	internal function: open a screen
OpenW()	internal function: open a window
OPT	local file options
OPTIMIZE	OPT: enable optimizations
OR	condit: logic or
Or()	internal function: bit or
OStrCmp()	internal function: compares two strings ordered
OSVERSION	OPT: minimal operating system version
Out()	internal function: writes byte to a file
PI	internal constant: 3.141592653589
Plot()	internal function: draw a pixel
POW()	constant: power
Pow()	internal function: power
PRIVATE	OPT: enable private data
PRIVATE	OBJECT: private data generation
PROC	procedure definition
PTR	type
PTR TO	type
PUBLIC	OBJECT: public data generation
PutByte()	internal function: put a byte/ubyte
PutLong()	internal function: put a long/ulong
PutWord()	internal function: put a word/uword
RAD()	constant: radian value
ReadEStr()	internal function: read string from file/stdio
RealEStr()	internal function: convert double into an e-string
RealStr()	internal function: convert double into a string
RealVal()	internal function: convert string into a double
ReEStr()	internal function: update changed e-string
RemEStr()	internal function: free allocated e-string
RightEStr()	internal function: copy some chars from right side of e- ↔ string to an e-string
REPEAT	REPEAT definition
REPROC	REPROC better procedure return

RETURN	RETURN values of procedures
RETURNING	EXITIF: same as IS
Rnd()	internal function: random number
RndQ()	internal function: random number quick
Rol()	internal function: roll number left
Ror()	internal function: roll number righth
RPROC	arguments use registers instead of stack
SELECT	SELECT definition
SET	SET constant definition
SetColour()	internal function: set screen colour
SetEStr()	internal function: set e-string length
SETOPT	Preset user OPTions
SetStdRast()	internal function: set default rastport
Shl()	internal function: shift number left
Shr()	internal function: shift number righth
Sign()	internal function: sign of a value
SIN()	constant: sinus
SINGLE	same as FLOAT
SINH()	constant: hyper sinus
SIZEOF	size of type/object
SizePooled()	internal function: size of AllocVecPooled() memory chunk
SQRT()	square root
StrAdd()	internal function: add to string
StrCmp()	internal function: compare two strings
StrCopy()	internal function: copy to string
STRING	e-string definition
StringF()	internal function: format string
StrLen()	internal function: length of a string
TAN()	constant: tangent
TANH()	constant: hyper tangent
THEN	IF definition
TO	FOR definition
TO	SELECT: arrays
TrimStr()	internal function: skip white spaces in a string
TRUE	internal constant: -1
UBYTE	type
UByte()	internal function: get an unsigned byte
ULONG	type
ULong()	internal function: get an (unsigned) long
UNION	type
UNTIL	REPEAT definition
UNTILN	REPEAT definition
UWORD	type
UWord()	internal function: get an unsigned word
Val()	internal function: convert string to number
WaitIMessage()	internal function: wait for window's message
WHILE	WHILE definition
WHILEN	WHILE definition
WORD	type
Word()	internal function: get a word