

**proasm**

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

	<i>TITLE :</i> proasm	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY		October 23, 2022
		<i>SIGNATURE</i>

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

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>proasm</b>	<b>1</b>
1.1	pro.guide	1
1.2	about this manual	17
1.3	notices	18
1.4	acknowledgments	19
1.5	author	20
1.6	registration	20
1.7	pro.guide/Introduction	23
1.8	installation	26
1.9	starting proasm from the shell	28
1.10	starting proasm from workbench	32
1.11	pro.guide/Config File	33
1.12	pro.guide/Hello World	34
1.13	pro.guide/Source Line Format	36
1.14	pro.guide/Expressions	40
1.15	pro.guide/Registers	45
1.16	pro.guide/Addressing Modes	46
1.17	pro.guide/Instruction Set	49
1.18	pro.guide/END	50
1.19	pro.guide/Include Files	51
1.20	pro.guide/OPT Y	51
1.21	incbin	52
1.22	pro.guide/INCEQU	52
1.23	header	53
1.24	pro.guide/MACLIB	53
1.25	pro.guide/INCDIR	54
1.26	pro.guide/Macros	55
1.27	macro	56
1.28	pro.guide/Symbols and Keywords for Macros	57
1.29	pro.guide/Substituting textual symbols in symbol names	60

---

---

1.30	pro.guide/Substituting subsections of strings . . . . .	61
1.31	pro.guide/Conditional Assembly . . . . .	62
1.32	pro.guide/Repeating Text . . . . .	68
1.33	pro.guide/Equates . . . . .	71
1.34	pro.guide/EQUR . . . . .	72
1.35	reg . . . . .	74
1.36	pro.guide/Structure Offsets . . . . .	77
1.37	pro.guide/Case Sensitivity . . . . .	81
1.38	pro.guide/OPT C . . . . .	81
1.39	pro.guide/OPT U . . . . .	82
1.40	pro.guide/Syntax Options . . . . .	82
1.41	pro.guide/OPT I . . . . .	83
1.42	pro.guide/OPT P . . . . .	83
1.43	pro.guide/OPT NOTYPE . . . . .	83
1.44	pro.guide/Processor Options . . . . .	84
1.45	pro.guide/SUPER . . . . .	87
1.46	pro.guide/READMODWRITE . . . . .	87
1.47	pro.guide/SETKFACTOR . . . . .	87
1.48	pro.guide/DEFAULT . . . . .	88
1.49	pro.guide/Optimization . . . . .	91
1.50	pro.guide/Assembler Message Control . . . . .	99
1.51	pro.guide/Controlling the Message Output . . . . .	100
1.52	pro.guide/BASE . . . . .	103
1.53	pro.guide/Absolute Assembly . . . . .	107
1.54	pro.guide/Data Output Directives . . . . .	108
1.55	pro.guide/Initialized Data with Restricted Range . . . . .	109
1.56	pro.guide/Declaring Data Blocks . . . . .	110
1.57	pro.guide/Uninitialized Data Blocks . . . . .	111
1.58	pro.guide/Defining Strings . . . . .	112
1.59	pro.guide/Alignment Padding . . . . .	113
1.60	pro.guide/ALIGN . . . . .	114
1.61	pro.guide/Convenience Pseudo-Opcodes . . . . .	118
1.62	pro.guide/Controlling the Output File . . . . .	121
1.63	pro.guide/Sections . . . . .	124
1.64	pro.guide/SMALLOBJ . . . . .	128
1.65	pro.guide/Debugging Information . . . . .	129
1.66	pro.guide/Object Modules . . . . .	132
1.67	pro.guide/Defining and Referencing External Symbols . . . . .	132
1.68	pro.guide/Output File Attributes . . . . .	134

---

---

1.69	pro.guide/Auxiliary Output Files . . . . .	136
1.70	pro.guide/OPT NOLIST . . . . .	139
1.71	pro.guide/LLEN . . . . .	140
1.72	pro.guide/OPT . . . . .	143
1.73	pro.guide/OPT XPK . . . . .	152
1.74	pro.guide/ESCAPESTR . . . . .	154
1.75	pro.guide/VERBOSE . . . . .	156
1.76	pro.guide/RCRESET . . . . .	157
1.77	pro.guide/LOCKSYM . . . . .	159
1.78	labseg . . . . .	159
1.79	pro.guide/Special Symbols . . . . .	161
1.80	pro.guide/Support Libraries . . . . .	168
1.81	pro.guide/Errors . . . . .	169
1.82	pro.guide/Warnings . . . . .	192
1.83	pro.guide/AmigaDOS Error Codes . . . . .	201
1.84	pro.guide/Instruction Set Summary . . . . .	205
1.85	pro.guide/ProOpts Directives Summary . . . . .	205
1.86	pro.guide/Bibliography . . . . .	205
1.87	pro.guide/Directive Index . . . . .	207
1.88	pro.guide/Concept Index . . . . .	223

---

# Chapter 1

## proasm

### 1.1 pro.guide

```
=====
                                P R O A S M
Motorola 680x0-series Professional Macro Assembler
```

```
                                v1.74
=====
```

```
                                Copyright © 1989-1996 by Daniel Weber
                                ProAsm is Shareware
```

About This Manual

TABLE OF CONTENTS

-----

Notices

Copyrights

Trademarks

Disclaimer

Acknowledgments

Registration Information

The Author

Introduction

o Product Overview

o The Manual

o Conventions Used in this Manual

o Manual Updates

o System Requirements

Installation

o

---

---

- Installation
  - Running ProAsm
    - o Starting ProAsm from the Shell
      - o Starting ProAsm from Workbench
- Configuration File
- Hello World
  - Assembly Language Syntax
    - o Source Line Format
      - . Labels
        - . Local Labels
        - . Scope of Labels
      - . Mnemonic
      - . Operands
      - . Comment

- Expressions
- . Constants
  - . Integer Constants
  - . Floating-point Constants
- . Numeric Symbols
  - . Absolute Symbols
  - . Relocatable Symbols
  - . External Symbols
- . Operators
- . Textual Symbols
- . Forward References to Numeric and Textual Symbols
- . Assembly Location Counter
- Registers
- . Basic Registers
  - . Control Registers
  - . Stack Pointer
  - . Floating-point Registers
  - . MMU Registers
- Addressing Modes
- o Instruction Set
  - Directives
    - o Source Code Input
      - . Terminating the Input File
- END
- ENDSRC
- . Include Files
- INCLUDE
- OPT Y

---

OPT NOINCONCE

OPT INCONCE

INCEQU

INCBIN

IBYTES

HEADER

MACLIB

. Specifying Include Directories

INCDIR

INCPATH

Macros

MACRO

ENDM

ENDMAC

MEXIT

. Symbols and KEYwords for Macro ←  
Definitions

\n

\(n)

\0

\#

\@

\\*VAL, \\*VALOF, \\*V, \\*D

\\*HEX, \\*HEXOF, \\*H

\\*BIN, \\*BINO, \\*B

\\*OCT, \\*OCTOF, \\*O

. Substituting textual symbols in symbol ←  
names

\\*STRING, \\*S

\\*STRLEN

. Substituting subsections of strings

\\*LEFT, \\*L



---

\*RIGHT, \*R	.
\*MID, \*M	.
	. Substituting subsections of strings
\*UPPER	.
\*LOWER	.
Conditional Assembly	o
IFD	.
IFND	.
IFEQ	.
IFNE	.
IFGE	.
IFHI	.
IFLE	.
IFLS	.
IFLT	.
IFMI	.
IFPL	.
IFVC	.
IFVS	.
IFHS	.
IFLO	.
IFC	.
IFNC	.
IF1	.
IF2	.
IFU	.
IFNU	.
ELSE	.
ELSEIF	.

---

ENDC	.
ENDIF	.
IIF	.
ENDASM	.
ASM	.
Repeating Text	o
REPT	.
ENDR	.
REPEAT	.
UNTILcc	.
EXIT	.
	o Defining Symbols
. EQU	
. EQUATE	
. =	
. ==	
. DEFINE	
. SET	
. SETVAL	
. FEQU	
. FSET	
.	
EQR	.
EQSTR	.
FEQR	.
SETR	.
FSETR	.
REG	.
EQURL	.
SETREG	.
SETRL	.
FREG	.
FSETRL	.
Structure Offsets	o
	.

---

```

RS.x
SO.x
FO.x
RSRESET
CLRSO
CLRFO
RSSET
SETSO
SETFO
RSVAL
SOVAL
FOVAL
Case Sensitivity
OPT C
CASEON
CASEOFF
OPT CASE
OPT NOCASE
OPT LOCALU
OPT LOCALDOT
OPT U
OPT U1
OPT U2
RELAX
NEWSYNTAX
OLDSYNTAX
OPT CHKIMM
OPT NOCHKIMM

```

- o Local Label Introducer
- o Assembly Control

---

```

OPT I
OPT CHKPC
OPT NOCHKPC
OPT P
    . Type Checking
OPT TYPE
OPT NOTYPE
OPT T
    . Processor Options
MC68000
MC68010
MC68020
MC68030
MC68040
MC68060
MC68EC020
MC68EC030
MCRELAX
MC680X0
MC68881
MC68882
MC68851
OPT P=
OPT FPSP40
OPT NOFPSP40
OPT FPSP60
OPT NOFPSP60
OPT SP60
OPT NOSP60
    . Code Control
```

---

```

SUPER
OPT SUPER
OPT SW
READMODWRITE
SETKFACTOR
                                . Assembly Options
DEFAULT
OPT ABL
OPT ABW
OPT BDL
OPT BDW
OPT BRL
OPT BRW
OPT BRS
OPT ODL
OPT ODW
OPT PCBL
OPT PCBW
                                . Optimization
OPT AUTOPC
OPT NOAUTOPC
OPT A
OPT O
OPTIMIZE
NOOPTIM
OPT OPTIMON
OPT OPTIMOFF
OPT Q
MULTIPASS
```

---

```
OPT OPTIMLIB
OPT NOOPTIMLIB
    . Assembler Message Control
FAIL
WARN
    . Controlling the Message Output
FAILAT
OPT E
OPT W
OPT WARN
OPT NOWARN
OPT OW
OPT QW
ODDOK
ODDERROR
ODD2OK
ODD2ERROR
OPT F
OPT Z
OPT CHKBIT
OPT WARNBIT
OPT NOCHKBIT
    . Baserelative Assembly
BASE
NOBASE
BASEREG
    . Absolute Assembly
ORG
ENDORG
RORG
    . Data Output Directives
    . Initialized Data
```

---

```
DC.x      .
          .
DB        .
          .
DW        .
          .
DL        .
          . Initialized Data with Restricted Range
          .
UB        .
          .
UW        .
          .
UL        .
          .
SB        .
          .
SW        .
          .
SL        .
          .
PB        .
          .
PW        .
          .
PL        .
          .
NB        .
          .
NW        .
          .
NL        .
          . Declaring Data Block
          . DS.x
          . DCB.x
          . BLK.x
          . Uninitialized Data Blocks
          .
DX.x      .
          . Defining Strings
          .
CSTRING  .
          .
CSTR     .
          .
ISTRING  .
          .
ISTR     .
          .
PSTRING  .
          .
PSTR     .
          .
Alignment Padding
          .
CNOP     .
```

ALIGN	.
EVEN	.
DS.W 0	.
QUAD	.
ODD	.
CCNOP	.
ALIGNRS	.
ALIGNSO	.
ALIGNFO	.
Convenience Pseudo-Opcodes	.
PUSH	.
POP	.
PUSHM	.
POPM	.
APUSHM	.
APOPM	.
MEA	.
PFLUSHA (a note)	.
	. Controlling the Output File
	. Output File Name
OUTPUT	.
OBJFILE	.
OBJ	.
	. Output File Format
EXECUTABLE	.
EXE	.
EXEOBJ	.
OPT L	.
LINKABLE	.
LINKOBJ	.



---

BINARY	.
BINARYONLY	.
BINRYONLY	.
ASEG	.
PREASM	.
OPT GENSYM	.
NOOBJ	.
SREC	.
Sections	.
SECTION	.
CODE	.
DATA	.
BSS	.
CSEG	.
DSEG	.
SECTION __OLDSECTION	.
SMALLOBJ	.
NORMOBJ	.
SMALLCODE	.
SMALLDATA	.
SMALLBSS	.
Debugging Information	.
DEBUG	.
OPT DEBUG	.
OPT NODEBUG	.
OPT HCLN	.
OPT NOHCLN	.
ADDSYM	.

---

NOSYM	.
OPT D	.
OPT XDEBUG	.
OPT NOXDEBUG	.
OPT X	.
SECSYM	.
SELSYM	.
OPT NOHCLN	.
Object Modules	.
IDNT	.
IDENTIFY	.
	. Defining and Referencing External Symbols
XREF	.
XDEF	.
PUBLIC	.
AUTOXREF	.
	. Output File Attributes
PURE	.
FILEPROTECT	.
FILENOTE	.
	. Auxiliary Output Files
ERRFILE	.
EQUFILE	.
LISTFILE	.
CREFFILE	.
	. Controlling the Listing
LIST	.
NOLIST	.
OPT LIST	.
OPT NOLIST	.
OPT MD	.

---

---

OPT NOMD	.
OPT MEX	.
OPT NOMEX	.
OPT M	.
LISTSYMS	.
OPT SYMTAB	.
OPT NOSYMTAB	.
OPT S	.
OPT CL	.
OPT NOCL	.
LFCOND	.
SFCOND	.
LISTCHAR	.
	. Formatting the Listing
LLEN	.
PLEN	.
SPC	.
FORMAT	.
PAGE	.
PAGEUP	.
NOPAGE	.
TTL	.
TITLE	.
SUBTTL	.
	. Miscellaneous Directives
OPT	.
OPTION	.
OPT XPK	.
OPT NOXPK	.

---

---

OPT STO	
OPT RCL	.
OPT RESET	.
OPT ESS1	.
OPT ESS2	.
OPT ESS3	.
OPT ESS	.
OPT ESS	.
OPT ESS	.
ESCAPESTR	.
CSYMFMT	.
VERBOSE	.
TIMES	.
QUIET	.
ASMPRI	.
COMMENT	.
PRINTX	.
RCRESET	.
RCSET	.
ERRFLAG	.
LOCKSYM	.
UNLOCKSYM	.
LABSEG	.
LABSEG __OLDLABSEG	.
EQUX	.
Special Symbols	
__PRO	o
__RS	o
__SO	o

---

__FO	o
__MOVEMREGS	o
__MOVEMBYTES	o
__MOVEMLIST	o
__BASE	o
__Vn	o
__PR	o
__CP	o
__DATE	o
__DATE2	o
__DATE3	o
__TIME	o
__DAY	o
__LINENUM	o
__RCODE	o
__LK	o
__NAN	o
__SNAN	o
__INFINITY	o
NARG	o
RARG	o
__MCOUNT	o
Support Libraries	
o Installation and De-installation	
o proasmlang.library	
o proasmoptim.library	
Appendices	
o Errors	
o Warnings	

---

```

AmigaDOS Error Codes
                    o
Instruction Set Summary
                    o
Bibliography
                    Indices
                    o
Directive Index
                    o
Concept Index
                    Additional Manuals:
                    ASX      ProOpts  ProUtils

```

## 1.2 about this manual

About This Manual

=====

Welcome to the AmigaGuide version of the ProAsm manual.

The original manual was written in TeX and texinfo. You are currently looking at the AmigaGuide version of it. I used various tools to convert the primal TeX source into AmigaGuide. They did quite a good job, but the result was over 700KBytes of length! As a consequence I had to shorten the whole guide file by hand. Now it is very possible that when browsing through this manual you will find sentences like:

This node (page) intentionally left blank.

or

This node (page) was intentionally shortened.

Please forgive me for that. I had also to skip various tables that can be found in the original printed manual. Even I skipped and shortened a lot of things, but this manual has still a prideful size of about 470KBytes!

Caused by the conversion of the TeX source of the manual into this AmigaGuide file and the need of some small reorganisation of it, some of the links go nowhere. Sorry for that, but I had not the time to check all links. I did check a lot of them, but unfortunately there are still some of these dead links in the guide file. If you should find any, please feel free to inform me (

Daniel Weber

). To find a specific directive you should

use the

index page

or the

main Page

.

One chapter was completely removed from this guide file and splitted into three other AmigaGuide files:

```
    asx.guide
    proopts.guide
    proutils.guide
```

They cover the associated utility software that comes along with the ProAsm assembler.

If you find some strange (unusal) parts in this AmigaGuide file, please keep in mind, that this guide initially was written as a book. You can order a copy of the printed manual, if you wish. Please read the registration document (register.doc or

```
    Registration
    ) for
```

further information.

Daniel Weber  
Zurich  
November 1995

## 1.3 notices

Copyrights

=====

ProAsm and associated utility software is copyrighted 1989-1996 by Daniel Weber. All rights are reserved worldwide.

```
    ProAsm is Shareware, see
        registration
    .
```

The ProAsm User's Manual is copyrighted 1994-1996 by Daniel Weber and Bryan Ford.

This AmigaGuide file reflects version 1.74 of the ProAsm assembler.

Trademarks

=====

ProAsm and ASX are trademarks of Daniel Weber. MC68000, MC68008, MC68010, MC68020, MC68030, MC68040, MC68060 MC68EC020, MC68EC030, MC68EC040, MC68LC040, MC68881, MC68882, MC68851 and Motorola are trademarks of Motorola, Inc. Amiga is a registered trademark of ESCOM AG. AmigaDOS, Kickstart, and Workbench are trademarks of ESCOM AG. SAS and Lattice are registered trademarks of SAS Institute, Inc. Aztec and Manx are trademarks of Manx Software Systems. ARexx is a trademark of The Wishful Thinking Development Corp. UNIX is a registered trademark

---

of AT&T. OS-9 is a registered trademark of Microware Systems Corporation. Aminet is a registered trademark of Stefan Ossowskis Schatztruhe. All products mentioned in this manual are trademarks of their respective owners.

#### Disclaimer

=====

The information, the ProAsm program, and all the associated utilities are provided "as is" without warranty of any kind, either expressed or implied. The entire risk as to the accuracy of the information herein is assumed by you. Daniel Weber does not warrant, guarantee, or make any representations regarding the use of, or the results of the use of, the information, the ProAsm program, or the associated utilities in terms of correctness, accuracy, reliability, currentness, or otherwise. In no event will Daniel Weber be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the information, the ProAsm program, or the associated utilities even if he has been advised of the possibility of such damages.

## 1.4 acknowledgments

#### Acknowledgments

=====

Without the help and guidance of my co-writer, Bryan Ford, this manual would have been entirely different. His suggestions and corrections led to substantial improvements. Bryan was likewise an invaluable resource for all the questions I used to overwhelm him with.

I am also indebted to Stefan Walter for his sever criticism and various suggestions. He helped me to keep the ball rolling in the early stages of this project. Stefan contributed also several "routine files" among them the great GTF interface system. And his S.I.M. debugging environment helped me through more than a few dark hours of a programmer.

My thanks to Rene Eberhard for all his constructive comments and criticisms. His efforts in finding hidden bugs deserves special thanks as well as investing many hours of work in writing some of the "routine files".

I also thank Susanne Keller, who loaned her energy and time to the careful reading of the entire manuscript.

A special word of "thanks" is due to Anja Gemperli for her support, understanding, and love.

I wish also to thank the following people for their assistance and encouragement:  
Michael Ryffel, Thomas Neubauer, Christian Schneider, U. Dominik Mueller  
Friedmann Buerger, Hansruedi Wenger, and Andreas Bobak.



## 1.5 author

Author  
=====

If you have bugreports, questions, ideas, flames or complaints (constructive criticism is always welcome), or if you just want to contact me, write or send a letter to:

Daniel Weber

Internet:           dweber@amiga.icu.net.ch     (preferred)  
                  dweber@iiic.ethz.ch

Mail:               Daniel Weber  
                  Hoeflistrasse 32  
                  CH-8135 Langnau  
                  Switzerland.

## 1.6 registration

Registration Information  
=====

Please refer to the registration.doc file that comes with the distribution for the ASCII version of the following text.

This program, ProAsm, is distributed as Shareware. It may be freely redistributed, but no charges other than reasonable copying and handling fees may be collected (not over US \$5 per floppy disk, or more than US \$50 per CD). The program may be distributed only as originally release, in the complete archived form.

Please be aware that the registered version of ProAsm and its utilities will not be available from Aminet.

The standard registration covers:

- The ProAsm assembler and associated utilities and numerous routine files.

The distribution contains NO include files from Commodore/ESCOM, as I have no license to include them.

- Free minor updates (probably via E-Mail or a mailing of
-

one disk), and one free major update (complete package). Later major update releases can be ordered for about 30% of the registration fee.

- Additional future registered updates on request will cost 5 SFr./5 DM/5 \$US.
- Possibility to get a full S.I.M. registration for a reduced fee (10 \$US). S.I.M. is a lowlevel debugger and monitor and its unregistered version can be found on Aminet in the dev/debug directory as 'sim172.lha'.

The registered version comes with additional features:

- Removed the code and data size limitation. The non-registered version is limited to 8 kbytes (8192 bytes) output code.
- Complete instruction set: MC68000-MC68060, MC68881/68882, and MC68851.
- All macro directives enabled.
- Multipass optimization enabled.
- All floating-point directives enabled.
- All optimizations enabled.
- Registration notice removed.

To register send the registration fee in one of the following manners:

- Cash, preferred (in BILLS only). The easiest and for you the cheapest way is to send money in an envelope. But this is not the best thing to send through mail.
- A check drawn on a SWISS BANK in Swiss Francs (SFr).
- POSTAL MONEY ORDER, drawn in SFr.
- EuroChecks are also accepted, but they must be in Swiss Francs and the beneficiary should be 'Daniel Weber'.
- An INTERNATIONAL MONEY ORDER, drawn in SFr with the address of a SWISS BANK on the front.
- Private cheques are accepted, but cashing them is very expensive. So make sure to additionally submit the extra charge when sending a private cheque: US\$ 10.-, £ 6.-, DM 10.-,...
- Swiss residents can use my PostCheck account 80-219240-0.

Any currencies are accepted, please use the US Dollar prices as reference.

---

You do not need to send any extra money for return postage, that is part of the registration fee, except otherwise noted.

If you send the registration form via email, and want to know if I have received your money, send another email about two weeks after the registration.

E-Mail: dweber@amiga.icu.net.ch (preferred)  
dweber@iiic.ethz.ch

Mail: Daniel Weber  
Hoeflistrasse 32  
8135 Langnau a/A  
Switzerland

----> CUT HERE <-----> CUT HERE <-----> CUT HERE <-----

=====

Name \_\_\_\_\_

Address \_\_\_\_\_

City\_\_\_\_\_ State\_\_\_\_\_

Country\_\_\_\_\_ PostCode\_\_\_\_\_

E-Mail\_\_\_\_\_

System  A500  A600  A1000  A1200  
 A2000  A2500  A3000 (T)  A4000 (T)

Memory\_\_\_\_\_ CPU\_\_\_\_\_

KickStart version\_\_\_\_\_

Comments (please use additional sheets if necessary)\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

ProAsm Registration: SFr. DM \$US

<input type="checkbox"/>	Shareware Registration.....	30.00	30.00	25.00
<input type="checkbox"/>	Plus latest version on disk.....	5.00	5.00	5.00
<input type="checkbox"/>	Plus full S.I.M. Registration.....	10.00	10.00	10.00
<input type="checkbox"/>	Update to a Major Release.....	10.00	10.00	10.00
	Bound Manual (approx. 322 pages, A5).....	** not available yet **		
	Extra postage outside Europe for			
	AirMail (priotitaire).....	** not available yet **		
	Total Enclosed.....	_____	_____	_____

## Method of Payment:

- Cash (in BILLS only).
- EuroCheque (Swiss Francs only).
- Check drawn on a SWISS BANK in Swiss Francs
- Postal money order (Swiss Francs only).
- International Money Order in Swiss Francs, with the addresss of a Swiss Bank on the front.
- Private Cheque (accepted, but cost extra charges:  
US\$ 10.-, £ 6.-, DM 10.-).
- My PostCheck account 80-219240-0 (for Swiss residents only!).

## Method of Shipment

- per email to
  - the email address mentioned above.
  - email: \_\_\_\_\_
- to the postal address above. (The postage is part of the registration fee.)

=====

## 1.7 pro.guide/Introduction

---

You know what your trouble is?

You're the kind who always reads the handbook. Anything people build, any kind of technology, it's going to have some specific purpose. It's for doing something that somebody already understands. But if it's new technology, it'll open areas nobody's ever thought before. You read the manual, man, and you won't play around with it, not the same way. And you get all funny when somebody else uses it to do something you never thought of.

William Gibson, the winter market 1985

## Product Overview

=====

Welcome to ProAsm!

The ProAsm assembler is a traditional two pass assembler that emits code for the entire Motorola MC68000 Family. ProAsm is a high performance, full-featured assembler with enough powerful features to make it appropriate for all assembly tasks. It produces native 68xxx code, and has special directives to enable the selection of the target processor and the appropriate code optimization for that processor. ProAsm supports both addressing mode syntaxes as defined by Motorola. Programmers find these capabilities of the new syntax mode particularly useful for handling advanced data structures common to sophisticated application and high level languages.

The output produced by ProAsm is either an executable file that can directly be run under AmigaDOS or the Workbench, object modules that are compatible with the Amiga standard linker and BLINK (the replacement linker from 'The Software Distillery'), binary output for ROM-able code (for example), pre-assembled files, or the Motorola S-record format. Besides the normal output files, ProAsm can also generate four types of auxiliary output files which reflect the results of the assembly process: the error file and the equate file, the source listing and the cross-reference listing.

ProAsm has a tremendous number of switchable optimizations including the multipass facility to gain an even more optimized code. ProAsm also has a very rich set of directives including a wide range of synonyms that allow source code written for other assemblers (Public Domain software for example) and the Commodore include files to be assembled. Included as well are directives to deal with structures, repeat loops and similar code elements very easily. Powerful macros with unlimited macro arguments and many macro directives are available to permit code to be easily and clearly arranged.

The rich set of available facilities allows exact control over the performance of the assembler. This control includes features such as optimization, case dependency for symbols, syntax control, and the default behaviour of ProAsm to name a few. More advanced control features such as precise code control and selectable symbol search

---

algorithm are also included.

A valuable feature of ProAsm is the configuration file, which is automatically included in each assembly. You can customize ProAsm to suit your particular wishes by including commonly used macros, code and directives in the configuration file.

No program can be all things to all people. So all assemblers have limitations - ProAsm tries to put them as far as possible not to narrow your creativity. This results in the fact that the most limitations are just limited only by available memory (line length, macro body, macro nesting, macro arguments, nesting of macro directives, repeat and include file nesting,...).

However, ProAsm is an ideal assembler for the professional developer, as well as the high-level language programmer (such as C, Modula,...) who wants to integrate some assembly language code into his programs, and the beginner at assembly language programming.

#### The Manual

=====

This manual is provided as a reference guide to ProAsm and its associated utilities. It contains information on the operation and the use of the directives, pseudo-opcodes, and other pertaining to the effective use of this software.

Two large indices,  
    Concept Index  
    , and  
    Directive Index  
    , allow you to  
find a specific topic quickly and easily.

Please note that this manual makes no effort to be a tutorial for M68000 Assembly Language Programming. For detailed information on the complete instruction set of the M68000 family we recommend the 'M68000 Family Programmer's Reference Manual', from Motorola Inc.

#### Conventions Used in this Manual

=====

This manual refers to the entire Motorola 68000 series of microprocessors as the "M68000". It refers to individual chips as the "MC68000," "MC68010," etc.

M68000 instructions, register names, and symbols are printed in lower case. Assembler directives are printed in upper case to distinguish them from instructions. In reality, case does not matter for instructions, register names, or assembler directives, and only

---

matters for symbols if you specifically request it.

Unless otherwise noted, square brackets [] are used to state that the arguments of a directive they enclose are optional.

#### Manual Updates

=====

The following is only valid for the printed manual. The AmigaGuide version of the manual will be updated directly.

This documentation is updated with Change Pages and a read.me file.

Change Pages are issued for minor changes to the manual as pages or Postscript files or both. The date of issue is printed at the bottom of each page.

A read.me file is provided on the installation diskette. This file contains information about recent changes to the software that are not yet reflected in the manual.

#### System Requirements

=====

ProAsm requires an Amiga with at least 512k of memory. Kickstart and Workbench 1.2 or higher are also required. ProAsm is fully compatible with the entire Amiga family.

## 1.8 installation

### Installation

=====

To install the ProAsm Package correctly, you simply have to copy the complete distribution into a directory (or to a partition) of your choice. The whole package requires about 1.7 MBytes of disk space.

This can be done using the given Installer script, this is also the recommended procedure. However, using the Installer script, you have also the possibility to do a partial installation.

A double-click on 'Install\_english' or 'Install\_deutsch' to start the installation.

You can also install the package manually, by using the following command

---

sequence:

```
makedir <your_directory>/ProAsm
copy ProAsm/ <your_directory>/ProAsm/ all
copy ProAsm.info <your_directory>
delete <your_directory>/ProAsm/install#?
```

Where '*<your\_directory>*' is the path of your desired location where the ProAsm Package should be copied to (e.g: 'work:')

The following two commands copy the configuration files to env: or envarc: respectively:

```
copy ProAsm/env/ env: all
copy ProAsm/env/ envarc: all
```

To assure that the software will work correctly, the following lines should be added to your user-startup file. Use an editor to add these lines to the S:User-Startup :

```
path <your_directory>/ProAsm <your_directory>/ProAsm/c/ add
assign libs: <your_directory>/libs/ add
assign help: <your_directory>/ProAsm/Help add
```

If you intend to do a partial installation manually, you should take the following dependencies into account:

```
ProAsm Assembler:  proasm
                   libs/
                   Catalogs/

ASX                :  ASX
                   ASX.info
                   rexx/
                   env/

ProOpts            :  ProOpts
                   ProOpts.info

Utilities          :  c/

Documentation      :  Help/
                   Help.info

Examples           :  Examples/
                   Examples.info

BLink Archive     :  BLink.lha

Routine Files     :  routines/

Include Files     :  include/
```



## 1.9 starting proasm from the shell

Starting ProAsm from the Shell

=====

ProAsm is driven by keyword options in any order indicating the action to be performed. The general command line format is as follows:

```
ProAsm <source file> [options]
ProAsm -M [options 1] <source file 1> [[options 2] <source file 2> ←
[...]]
```

[options] specifies any optional parameters and <source file> the name of the file to be assembled. The second form allows you to assemble multiple source files in one invocation of ProAsm. If no extension is added to the <source file>, the assembler assumes .s as the extension.

### Command-line Options

-----

Following is the complete list of command-line options supported by ProAsm, in AmigaDOS command-line description format:

```
-A=ADDRESS/S, -B=BINARY/S, -D=DEFINE/S, -E=ERRFILE/K, -F=FILEREQ/S,
-G=CONFIGFILE/K, -H=HEADER/K, -I=INCDIR/K, -K=NOKEYWAIT/S, -L=LINKABLE/S,
-M=MULTISOURCE/S, -MA=MULTISOURCEA/S, -N=NOOBJ/S, -O=OBJFILE/K,
-P=LISTFILE/K, -Q=EQFILE/K, -R=CREFFILE/K, -S=ADDSYM/K, -T=STRING/K,
-U=NOSTDOUT/S, -V0=V0/K, -V1=V1/K, -V2=V2/K, -V3=V3/K, -V4=V4/K, -V5=V5/K,
-V6=V6/K, -V7=V7/K, -V8=V8/K, -V9=V9/K, -W=WINDOW/S, -WS=WINDOWSIZED/K,
-X=EXPSYM/S
```

The meaning of these options is given below. All options can be supplied in either upper or lower case. Switch-style options (-A, -B, etc.) and keyword-style options (ADDRESS, BINARY, etc.) can be mixed arbitrarily. For options that take arguments, if the keyword style is used, the keyword must be separated from the argument by at least one space (e.g. HEADER foo.i). If the switch style is used, a space may be used but is not necessary (either -Hfoo.i or -H foo.i will work). Filename or path arguments that contain a whitespace must be enclosed in double or single quotes (e.g. HEADER "foo bar.i").

A sequence of switch-style options without an argument can be written all alone (e.g. -L -D) or together (e.g. -LD). Even the following is a possible combination -LDE foo.err for -L -D -E foo.err.

ADDRESS address

-A address

This option causes ProAsm to assemble source code from an address in memory rather than from a file. The source code is assumed to start at address, which can be expressed in any base, in the same

way that numeric constants are expressed in the source code itself (e.g. \$1000 is the same as 4096). ProAsm assumes that the source text is a single null-terminated (C-style) string. This option is mainly provided to support text editors that can run an assembler from within the editor and provide the source code directly in memory rather than having to write it to a file first. If you use this option, no source file name may be specified in the command-line.

DEFINE symbol=expression[,symbol=expression[...]]

-S symbol=expression[,symbol=expression[...]]

This option defines the specified symbol with the result of the specified expression as if it had been defined with the EQU directive at the very beginning of the source code (see Equates).

Any amount of whitespace is accepted between the symbol and the equal sign, ← and

between the equal sign and the expression. If more than one symbol definition is given, they can be separated by a comma (,).

BINARY

-B

This option tells the assembler to produce a "raw" binary file, containing only the data explicitly defined in the source code. No linking or debug information can be generated. See Output File Format.

ERRFILE filename

-E filename

This option specifies the name of the error file to be used, and it overrides any ERRFILE directive specified in the actual source code. See

Auxiliary Output Files

, for more information. You may

also use a window specification instead of a filename, e.g.:

```
ProAsm mySource.s ERRFILE con:0/0/640/150/mySource_Errors
```

FILEREQ

-F

This option causes ProAsm to open a file requester in which you can choose the source file to be assembled, rather than specifying its name on the command line. If you cancel the file requester, ProAsm will abort without assembling anything. If you use this option, no source file may be specified in the command-line, unless the MULTISOURCE or MULTISOURCEA option is also used. ProAsm uses the ASL file requester if you have Kickstart version 2.04 or later, or the REQ file requester on an earlier version of Kickstart.

CONFIGFILE filename

-G filename

This option tells ProAsm to use the file filename as ProAsm's configuration file, instead of the default file named ProAsm.config. This can be used, for example, to use a different configuration for individual projects. See

## Config File

,  
for more information.

## HEADER filename

-H filename

The HEADER option tells ProAsm to assemble the file specified as filename before assembling the main source code, as if it had been included with INCLUDE at the very beginning of the source file. See

## Include Files

, for more information about include files. It is still processed after the configuration file, however.

## INCDIR directory[,directory[...]]

-I directory[,directory[...]]

This option adds the specified directory to the list of directories to search for include files. Directory names containing one or more whitespace must be enclosed in double or single quotes. You can specify more than one directory by separating each directory name with a comma (,). See Include Directories, for more information.

## NOKEYWAIT

-K

This option instructs ProAsm not to wait for a key to be pressed at the end of assembly. This only has an effect if the WINDOW or WINDOWSIZE option is also used.

## LINKABLE

-L

This option tells ProAsm to create an object file which can be linked with other object files by a linker such as BLink. See Output File Format, for more information.

## MULTISOURCE

-M

This parameter keyword forces the assembler to accept more than just one source file on the command line. All source files will be assembled sequentially, in the order of their occurrence on the command line. Using this keyword the command line must be of the following syntax:

```
MULTISOURCE [options 1] <source file 1> [[options 2] <source file 2> [...]]
```

For example:

```
ProAsm MULTISOURCE BINARY BinImage.s ADDSYM helloworld.s  
First of all the source code BinImage.s is assembled to a binary image file according to the BINARY flag. Afterward the helloworld.s source file is processed using the ADDSYM keyword.
```

## MULTISOURCEA

-MA

This is an extension of the above described MULTISOURCE keyword. MULTISOURCEA aborts assembly after the source file in which the first error occurred, instead of assembling all source files whether or not any errors occurred.

---

NOOBJ

-N

If this option is used, the assembler will not produce any output file at all. It is useful if you want to run a test assembly to check for syntax errors in the source code. This option overrides any other command-line options or directives in the source code specifying an output file to be produced. See *Output File Format*, for more information on the NOOBJ directive and the different types of output files ProAsm can produce.

OBJFILE filename

-O filename

This option specifies the name of the object file to be produced. The actual format of this output file depends on the output file format selected; see *Output File Format*, for more information. This option overrides any OBJFILE directive contained in the actual source code.

LISTFILE filename

-P filename

This option specifies the filename of the listing file; see

*Auxiliary Output Files*

, for information about listing files. It also automatically enables listing generation, and overrides any LISTFILE directive present in the actual source code.

EQUFILE filename

-Q filename

This option directs ProAsm to generate an equate file in filename. It overrides any EQUFILE directive in the source code. See *The Equate File*, for more information.

CREFFILE filename

-Rfilename

This option causes ProAsm to create a cross reference listing in filename. It overrides any CREFFILE directive in the source code; see

*Auxiliary Output Files*

, for more information.

ADDSYM

-S

This option tells the assembler to add symbol information to the output file (see

*Debugging Information*

). This option is ignored

unless you specify an EXECUTABLE or a LINKABLE output file.

STRING symbol=string[,symbol=string[...]]

-T symbol=string[,symbol=string[...]]

This option is used to assign the specified string to the symbol, as with the EQU directive; see *Textual Symbols*, and

*Equates*

, for

more information. Any number of whitespaces are accepted between

the symbol name and the equal sign, and between the equal sign and the string. If the string contains whitespace it must be enclosed in double or single quotes, or between angle brackets (< and >). If more than one symbol definition is given, they can be separated by a comma (,). In this case, strings that contain a comma should be enclosed within angle brackets (< and >).

#### NOSTDOUT

-U

This tells ProAsm to perform assembly quietly; no output text will be displayed.

Vn=value

-Vn=value

Specifying one of these options will cause the specified value to be used as the definition of the special built-in symbol `__Vn`, where n is between 0 and 9. For example, if you specify `-V0=12345` on the command-line while assembling a particular program, the symbol `__V0` will be defined as if you had included the statement `__V0 EQU 12345` at the beginning of the assembly language source file. See

Special Symbols

, for more information on the `__Vn`

symbols.

#### WINDOW

-W

This option instructs the assembler to send all output text to a window instead to the standard output. This can be useful if you run ProAsm as a background process or from a script file. The window will be of the size of 640x200, or smaller if necessary. After assembly the message 'Press a key to continue' will be displayed in the window and ProAsm will wait for a key to be pressed before terminating, to give you a chance to see any errors or other messages generated during assembly. This behavior can be suppressed by using the `NOKEYWAIT` option. The `WINDOW` option overrides the `NOSTDOUT` option.

#### WINDOWSIZED dimension

-WS dimension

This is exactly the same as `WINDOW`, except that it allows you to specify the window dimensions. The dimension must be in the format `x/y/width/height` (without the quotes), with coordinates in pixels.

#### EXPSYM

-X

This option tells the assembler to add only the information of the exported symbols to the output file; any `ADDSYM` option used previously will be overridden. It is only useful if `LINKABLE` output files are being generated (see Output File Format).

## 1.10 starting proasm from workbench

## Starting ProAsm from Workbench

=====

To start ProAsm from the Workbench, double click on the ProAsm icon to load the assembler from disk. ProAsm will then display a file requester that is used to specify the filename of the source code to be assembled.

All output text is sent to a window of the size of 640x200, or smaller if necessary. After assembly the message 'Press a key to continue' will be displayed in the window and ProAsm will wait for a key to be pressed before terminating, to give you a chance to see any errors or other messages generated during assembly.

## 1.11 pro.guide/Config File

### Configuration File

=====

The ProAsm assembler supports a configuration file that allows you to set assembler options for a project. The configuration file itself can be regarded as an include file that is automatically loaded at the very beginning of each assembly (even before the header files). You can use it to customize ProAsm to suit your particular wishes. It is not restricted in its contents. This allows you to put any assembler directive and pieces of code that you want to have included each time (such as startup codes etc.) in the config file.

The default configfile name is proasm.config. The assembler first looks in the home directory of the main source code, and after that in the ENV: and S: (1) assigns. Searching this way through the directories gives you the possibility to have local and global configurations. Local configurations are stored in the same directory as your main source code and the global configurations in the ENV: assign. For permanent global configuration, copy your config file into the ENVARC: assign.

If you want to use another name than the default config file name, you may use the -G/CONFIGFILE command line option to specify a new name. For example:

```
pro helloworld.s CONFIGFILE asm:myconfig.i
```

The use of the configuration file can be suppressed if a name of file is given that does not exist. For example:

```
pro helloworld.s -G nil:
pro helloworld.s CONFIGFILE qwjhbci
```

As mentioned above, the configuration file is an ASCII file that can be regarded as a variant of an include file. It can be edited using a texteditor or the ProOpts program (see ProOpts, for more details). Remember that the use of a configuration file may have unwished side effects, by the presence of certain directives in the config file. An example to illustrate this is: if the config file contains any symbols, an error could be reported by the assembler, if

it encounters an OPT C- or similar directives, that change the case dependency for symbols.

Below you find two examples of configuration files which may give you an impression how they can look like.

Configuration file #1 is a simple config file, that just sets up often used include pathes:

```
EXEOBJ
INCDIR include:
INCDIR routines:
```

Configuration file #2 shows you a way how a startup code and a standard version string can be put into a config file:

```
EXEOBJ
INCDIR include:
INCDIR routines:

jmp      AutoDetach(pc)
dc.b     "$VER: ",progame," ",version," (" ,__date2,")",0
EVEN
```

```
INCLUDE startup4.r      ; the startup code
```

The startup4.r startup code requires two symbol definitions in the main source code, clistartup and wbstartup, to work properly. Progame and version may be set by the -T/STRING command line option:

```
pro mySource.s -T progame="HelloWorld" -T version="1.00"
```

-----  
 (1) Under Kickstart version 1.3 or earlier, ENV: and ENVARC: may not exist. Use S: instead.

## 1.12 pro.guide/Hello World

```
Hello World
=====
```

Now that you have an idea how to run ProAsm, it is time to assemble the first program. The source code below shows the famous Hello World. If you are new to assembly programming, take a little time to look over this program.

```
*
*
* HelloWorld.s - a small helloworld example in assembler
*
*

        OUTPUT 'ram:HelloWorld'      ; write executable to ram:HelloWorld

;
; exec function offsets
;
```

```

_LVOpenLibrary      EQU      -552
_LVOCloseLibrary    EQU      -414
;
; dos function offsets
;
_LVOutput           EQU      -60
_LVWrite            EQU      -48

;
; program start
;
start:  lea      DosName(pc),a1
        moveq   #0,d0
        move.l  4,a6          ; load exebase
        jsr    _LVOpenLibrary(a6) ; open dos.library
        move.l  d0,DosBase
        beq.s  NoDosLibrary    ; could not open library

        move.l  d0,a6
        jsr    _LVOutput(a6)   ; get StdOut handle (handle in d0)

        move.l  d0,d1
        move.l  #Text,d2
        moveq   #TextLength,d3
        move.l  DosBase,a6
        jsr    _LVWrite(a6)    ; write text to StdOut

Exit:   move.l  DosBase,a1
        move.l  4,a6
        jsr    _LVOCloseLibrary(a6) ; close dos.library

NoDosLibrary:
        moveq   #0,d0          ; set AmigaDOS return code
        rts                    ; exit program

;
; data
;
Text:   DC.B    "Hello World!",$a,0
TextLength EQU    *-Text

DosName: DC.B    "dos.library",0
        EVEN

DosBase: DC.L    0

        END

```

To assemble the above program write the following line into the shell and press RETURN:

```
ProAsm HelloWorld.s
```

After a few ticks the output file is written to RAM:HelloWorld and ready to be executed. Run it by simply typing its name into the shell



followed by a RETURN, and a nice Hello World! will be displayed.

## 1.13 pro.guide/Source Line Format

### Source Line Format

=====

An M68000 assembly language source code file is a normal ASCII text file containing an arbitrary number of lines. Each line has the following general layout:

```
label mnemonic operands comment
```

The following sections describe each of these four fields in detail.

#### Labels

-----

The label field can have two slightly different formats. In the first format, The label starts at the first character of a line, with no whitespace (space or tab characters) in front of it, and is separated from the mnemonic field (if present) with at least one whitespace. In the second format, the label is followed immediately with a colon (:) character. In this case, whitespace may be used at the beginning of the line before the label. In addition, no whitespace is needed before the mnemonic field (the colon is sufficient to separate the two fields), although it may still be present.

For normal instructions and assembler pseudo-ops that generate data (such as DC), the label is optional, and is used to assign the address of the current position to a symbol. If you don't specify a label at all, then you must have at least one whitespace character at the beginning of the line before the mnemonic field.

You may also put a label on a line of its own. In this case, the current position is assigned to the label without doing anything else. You can assign the same position to several different labels by putting each on a separate, otherwise empty line.

To demonstrate how labels are defined, the following code fragments all do the same thing (they produce a program which loops forever uselessly):

```
loop    bra    loop
```

```
loop:  bra    loop
```

```
loop
      bra    loop
```

```
loop:
      bra    loop
```

```
loop:bra loop
```

```

loop: bra    loop

      loop:
      bra    loop

```

Many assembler directives make special use of the label field. For example, the EQU directive assigns a numeric value to the specified label, instead of the current position. In these cases, the label is usually (but not always) required, depending on the directive being used. In this case, a colon can still be used to separate the symbol from the assembler directive, but they must be on the same line.

A valid symbol name may contain up to 256 characters. The first character must be one of the following characters: A through Z (in either upper or lower case), a period (.), a backslash (\), an underscore (\_), or an 'at' symbol (@). The rest of the symbol name can consist of the characters A through Z (also in upper or lower case), the digits 0 through 9, periods (.), and underscores (\_). Symbols are normally case-sensitive, but can be made case-insensitive depending on an option you can set (see

Case Sensitivity

). Names corresponding to

M68000 registers such as D0 are reserved (see

Registers

), as well as

the names of special assembler symbols (see

Special Symbols

).

## Local Labels

.....

Local labels are a special form of labels that are strictly local to the block of assembly instructions and directives between two nonlocal (global) labels. Local labels provide a convenient means of generating labels for loops, branch instructions and such. The use of local labels reduces the possibility of multiply defined labels in a program, and it separates entry point labels from local references, such as the top of a loop.

Local labels cannot be referenced from outside the current assembly block.

Normally a local label begins with a dot (.) or the slash character /. You can use an underscore (\_) to introduce a local label if the OPT U+ option is turned on (see Local Label Introducer). The assembler also accepts local labels of the form n\$, where n is any integer. Valid local labels include:

```

.local
/local
..label
2$
1994$
.2

```

Note that the local label 2\$ is the same as 02\$ or 0000002\$ and so on (leading zeros are ignored).

### Scope of Labels

.....

The section of program text in which a label is defined is called its scope. An ordinary label which tags a location in the program or data is visible only within the current assembly, except otherwise declared by using the XREF directive that makes a label visible to other assembly units at link time.

A local label has a scope that extends between one nonlocal label and the next. Every time a non local label is defined, the previous span of local labels is discarded, and a new local label scope is created. Consider the following example that illustrates the scopes of the different kinds of labels:

```

FirstLabel:                                ; a new local label scope is created
    moveq #15,d7
.loop:   move.b (a0)+,(a1)+                ; first appearance of the local label
.loop
    dbra d7,.loop
SecondLabel:                               ; the above local label scope has gone away,
    move.l a2,a1                          ; and a new scope is created
.loop:   cmp.b (a1)+,d0                    ; another (different) local label .loop
    bne.s .loop                            ; branches to .loop above
ThirdLabel:                                ; the above local label scope has gone away,
    move.l a1,d0                           ; and a new scope is created
    beq.s .loop                            ; generates an error message - no .loop
symbol found
FourthLabel:

```

The scope of the first local label .loop spans from the FirstLabel to the SecondLabel label. The second appearance of the local label .loop has a scope which extends between the SecondLabel and the ThirdLabel label. After the definition of the ThirdLabel label, the branch to .loop will generate an error message because that label is no longer defined within this scope.

### Mnemonic

-----

The mnemonic field contains the name of an instruction (also known as an opcode) or a special assembler directive (a pseudo-opcode) defining exactly what ProAsm is supposed to do with this line. Mnemonics are always case-insensitive: for example, move, Move, and MOVE are the same.

This manual does not describe the instructions ProAsm understands; these can be found in any M68000 assembly language book such as the

'M68000 Programmer's Reference Manual', published by Motorola, Inc. However, ProAsm accepts a large number of pseudo-opcodes which make assembly language programming easier; these are described in Directives. Finally, you can defined your own mnemonics with ProAsm's powerful macro facility (see  
    Macros  
    , for more information).

## Operands

-----

The operands field contains any additional data needed by the selected mnemonic. The field may contain multiple operands separated by commas. In general, whitespace may be used within the operands field, but only within quoted strings (e.g. "Hello world!"). In this case, the whitespace is considered to be part of the string. As a special case, the DC and related directives also allow whitespace immediately after a comma (e.g. 1, 2, 3). This type of whitespace is merely padding and is ignored by ProAsm.

For M68000 instructions, the operands themselves have special formats dictated by the addressing modes in use. The addressing modes are described later in

### Addressing Modes

. For assembler directives, the operands are usually symbolic constants, numeric expressions, or text strings.

## Comment

-----

At the end of any line may be placed a comment, which is completely ignored: comments are purely for the benefit of humans reading the source code. Comments are usually separated from the rest of the source line with a semicolon (;) or asterisk (\*), which explicitly tells ProAsm to ignore everything on the rest of the line starting with the semicolon or asterisk. However, ProAsm automatically considers the contents of a line after a mnemonic and its operands as "comment," so in this case you do not need to use a semicolon at the beginning of the comment. However, for clarity we recommend that you still use one.

You can also reserve a whole line for a comment by making the first non-whitespace character a semicolon or an asterisk.

You can create comments that span many lines by preceding it with /\* and terminating it with \*/. Note that this is the standard comment format in the C language. However, unlike in C, the initial /\* sequence must be first on a line (like \* comments). Additionally, ProAsm ignores all text up to and including the entire line on which the \*/ appears.

Here are some examples of lines containing valid comments, although we do not recommend the last three examples to be used as comments:

```
* This is a comment.
; This is a comment.

                                * This is a comment.
                                ; This is a comment.

foo                                * This is a comment.
foo:                              ; This is a comment.

    nop                            * This is a comment.
    move.l  d0,a0                 ; This is a comment.

    nop                            This is a comment.
    move.l  d0,a0                 This is a comment.

/* This is
   one big
   comment. */
```

## 1.14 pro.guide/Expressions

### Expressions

=====

An expression is a combination of symbols, constants, numbers, labels, and algebraic operations. The expression is used to specify a value which is to be used as an operand.

### Constants

-----

There are two forms of integer constants supported by ProAsm, namely numeric integer constants and string constants. All integer constants are considered absolute quantities when they appear in an expression.

### Integer Constants

.....

ProAsm accepts numeric quantities in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2) radices, or ASCII string. Integer constants can represent quantities up to 32 bits.

### Decimal numbers

Decimal numbers consist of between one and ten decimal digits (in the range 0 through 9). The range of decimal numbers is between -2147483648 and 2147483647.

### Hexadecimal numbers

Hexadecimal constants are preceded by the dollar character (\$) and can then have between one and eight hexadecimal digits. The hexadecimal digits consist of the decimal digits from 0 to 9 and the letters 'a' to 'f' (case does not matter).

### Octal numbers

Octal numbers are preceded by the atSign (@) and can then have one to eleven octal digits. Octal digits consist of the decimal digits from 0 to 7. Note that eleven octal digits can have 33 bits, thus the largest octal number is @37777777777.

### Binary numbers

Binary numbers are indicated by the presence of the percent sign (%) and can then have between one and 32 binary digits. The binary digits are the decimal digits 0 and 1.

### ASCII strings

The ASCII string in an expression is a non-numeric constant. The string data is represented by a sequence of characters enclosed by a pair of matching single or double quotes (' or "). Strings of one to four characters will generate a valid 32 bit numeric value. This can be used as any other integer constant within an expression.

For example, the following strings generate the indicated hexadecimal integer values:

```
'a'      = $00000061
'ab'     = $00006162
'abc'    = $00616263
'abcd'   = $61626364
```

The following example shows all integer constants within an expression:

```
move.l #1994+$7f347+@47946+%100111011101101+"gh",d0
; => move.l #523064,d0
```

### Floating-point Constants

.....

ProAsm accepts two notations for floating-point numbers: the pure fractional number and the engineering (exponential) notation. The fractional notation should already be familiar to you; the decimal-point method of writing numbers:

```
3.1415926
0.1994
32145.1123
-46.732
and so on.
```

The engineering notation includes a fractional part and an exponential part. Taking the same numbers from above, they could be written as:

```
3.1415926E0
1.994E-1
```

```

    3.21451123E4 (or 3.21451123E+4)
    -4.6732E1

```

The E, meaning exponent, can be either written in upper- or lowercase. An exponent of n effectively "moves" the decimal point n places to the right if n is positive and to the left if negative, preserving the correct value of the number.

Notice that when you write a floating-point number in the source code, the assembler also accepts numbers without a fractional part:

```

foo    FEQU    3
; is the same as
foo    FEQU    3.0

```

(The assembler automatically recognizes the integer value (3) as a floating-point number because FEQU was used instead of EQU.)

## Numeric Symbols

-----

A numeric symbol is a symbol that represents a number. The exact value that a numeric symbol actually represents, however, is not always known by the assembler. There are three types of numeric symbols: absolute, relocatable, and external.

### Absolute Symbols

.....

An absolute symbol has an exact, known value. It can be used anywhere a numeric constant can be used, and works in exactly the same way. Absolute symbols are commonly defined with the EQU directive, like this:

```

the_answer    EQU    42

```

### Relocatable Symbols

.....

A relocatable symbol refers to the address of a specific position within the output file the assembler is producing: for example, the address of a specific instruction or data item. However, the assembler may not know exactly where in the computer's address space the output it is producing will be loaded. In such cases, relocatable symbols are used.

While ProAsm does not know exactly where a relocatable symbol will fall in memory, it does know where the symbol will fall relative to other symbols in the same segment. If one relocatable symbol is subtracted from another one in the same segment, they "cancel out" into a known value as if an absolute symbol had been used. In the following example, ProAsm knows the value to put into the last DC.L statement because the two relocatable symbols cancel each other out in the expression:

```

start:
    DC.L    1,2,3
finish:
    DC.L    finish-start    ; => 12

```

An implicit form of "cancelling out" occurs when a relocatable symbol is used as an operand in a PC-relative addressing mode. In the following example, ProAsm knows the exact value to use in the LEA instruction, because the relocatable symbol "cancels out" with the (relocatable) address of the LEA instruction:

```

somewhere:
    ...
    LEA    (somewhere,pc),a0

```

Note that branch instructions ( Bcc) are always PC-relative.

If ProAsm is generating a BINARY or otherwise non-relocatable output file, but no ORG directive is used, "cancelling out" is the only way relocatable references may be used. (If an ORG directive is used, then ProAsm knows exactly where its output will be loaded, so it always uses absolute symbols instead of relocatable symbols.)

If ProAsm is generating a relocatable output file, such as an AmigaDOS executable or a linkable object module, then relocatable symbols can also be used when defining longwords in instructions or data. This causes ProAsm to include in the output file a fixup for the longword in question, which instructs the linker or loader to "fix" the value once the address is known exactly. For example, the following statements are valid only when producing a relocatable file:

```

foo:
    ...
    jmp    foo
    ...
    DC.L    bar+10
    ...
bar:

```

Only one relocatable symbol may be used in an expression this way; all other parts of the expression must be known values (or pairs of relocatable symbols cancelling each other out, which amounts to the same thing). Also, no special arithmetic may be applied to such a relocatable symbol. For example, bar\*2 or -bar would not work in the DC.L above.

## External Symbols

.....

External symbols are symbols declared by the XREF directive, meaning that the symbol is actually defined in a different module to be linked later with the module being generated. In this case, ProAsm does not even know where the symbol is located relative to other symbols: all it knows is that the symbol exists. Because of this,



external symbols are the most restrictive type, and can only be used at all when ProAsm is generating a linkable object file (see Output File Format).

Since the relative positions of symbols are not known to ProAsm, external symbols cannot "cancel out" like relocatable symbols, and a fixup is produced every time one is used. There are three ways an external symbol can be used:

#### 32-bit Absolute

In instructions or data definitions that expect full 32-bit addresses, ProAsm can generate 32-bit absolute fixups (external symbol information), which are "fixed" once the exact address of the symbol is known. These examples use 32-bit absolute fixups:

```
XREF    foo,bar
...
jmp     foo
...
DC.L    bar-15
```

Note that this is the easiest way to use external symbols, and you only need to worry about the other two if you are worried about the size or efficiency of your program.

#### PC-Relative

In instructions expecting PC-relative values, ProAsm can create PC-relative fixups which will be reduced to constants by the linker. Most PC-relative external references are 16-bit, but 8-bit and 32-bit external references are also possible. 8-bit references are not very useful due to their severely limited range, and 32-bit PC-relative references only work in code for 68020 and higher processors, since earlier processors do not have any 32-bit PC-relative addressing modes.

To generate PC-relative references, ProAsm must assume that the segment containing the reference will be merged by the linker into the same segment as the one in which the symbol is actually defined. If this assumption is not satisfied, the linker will produce an error while linking the object files together.

Here are examples that generate 16-bit PC-relative fixups:

```
XREF    foo,bar
...
move.w  (bar+10,pc),d0
jmp     (foo,pc)
```

#### Base-Relative

In instructions expecting 16-bit non-PC-relative values, ProAsm creates 16-bit base-relative fixups. It makes the assumption that the symbol will reside in a special "global data segment" produced by the linker, containing all data accessed this way. The linker creates a special symbol called `_LinkerDB`, which is the base address of all global data. You reference this data by loading the address of `_LinkerDB` into an address register with 32-bit absolute mode, then use this register as the base address for

referencing 16-bit base-relative data. Here are some examples of how this is done:

```

XREF    _LinkerDB
XREF    foo,bar
...
; Load base register
lea     _LinkerDB,a5
...
; 16-bit access to foo
move.w  (foo,a5),d0
...
; 16-bit indirect access to bar
move.w  (indirect,pc),d0
move.l  (d0.w,a5),d1

indirect:
        DC.W    bar

```

## 1.15 pro.guide/Registers

### Registers

=====

All M68000 processors contain the following basic set of registers, which are the ones primarily used during normal programming:

#### D0-D7

General-purpose 32-bit data registers, normally used to hold integer values, counters, etc.

#### A0-A6

General-purpose 32-bit address registers, normally used to hold pointers into main memory.

#### A7 or SP

The last address register is the Stack Pointer, and may be referred to by either A7 or SP.

#### CCR

The 8-bit Condition code Register holds condition codes generated during test and comparison instructions, and is used by conditional branch instructions.

#### SR

The 16-bit Status Register holds various processor status and configuration bits. It is actually a superset of the CCR, which is contained in the lower 8 bits of the SR.

In addition, different members of the M68000 family support various extended registers, listed in the following table. Many of the registers can only be accessed using special instructions. All of the M68000 registers are described fully in the 'M68000 Family Programmer's Reference Manual'.

## Basic Registers

-----

MC680x0, MC68EC030, MC68EC040, MC68LC040: D0-D7, A0-A7, CCR, SR, USP

## Control Registers

-----

MC68000, MC68008: None  
 MC68010: SFC, DFC, VBR  
 MC68020: SFC, DFC, VBR, CACR, CAAR  
 MC68030: SFC, DFC, VBR, CACR, CAAR, CRP, SRP, TC, TT0, TT1, MMUSR  
 MC68030: SFC, DFC, VBR, CACR, CAAR, ACR0, ACR1, ACUSR  
 MC68040, MC68LC040: SFC, DFC, VBR, CACR, URP, SRP, TC, DTT0, DTT1, ITT0, ITT1, MMUSR  
 MC68EC040: SFC, DFC, VBR, CACR, DACR0, DACR1, IACR0, IACR1  
 MC68060: SFC, DFC, VBR, CACR, URP, SRP, TC, DTT0, DTT1, ITT0, ITT1, BUSCR

## Stack Pointer

-----

MC68000, MC68008, MC68010: USP, SSP  
 MC68020, MC68030, MC68040, MC68060: USP, SSP (MSP, ISP)

## Floating-Point Registers

-----

MC68881, MC68882, MC68040, MC68060: FP0-FP7, FPCR, FPIAR, FPSR

## MMU Registers

-----

MC68851: AC, BAC0-BAC7, BAD0-BAD7, CRP, CAL, DRP, PCSR, PSR, SCC, VAL  
 MC68030: CPR, SRP, TC, MMUSR, TT0, TT1  
 MC68040, MC68LC040, MC68060: URP, SRP, TC, DTT0, DTT1, ITT0, ITT1

**1.16 pro.guide/Addressing Modes**

## Addressing Modes

=====

This node (page) was intentionally shortened.

[...]

ProAsm supports both addressing mode syntax as defined by Motorola for the M68000 Family. Below you will find a list of all addressing modes.

The following notation conventions are used in this section:

- Dn  
Any data register (Example: D4 is data register 4)
- An  
Any address register (Example: A1 is address register 1)
- Xn  
Either an address register or a data register
- PC  
Program Counter
- ZDn  
Pseudo register that represents the suppressed data register n.  
Only valid for the program counter and address register relative addressing modes.
- ZAn  
Pseudo register that represents the suppressed address register n.  
Only valid for the address register relative addressing modes.
- ZSP  
Pseudo register that represents the suppressed address register A7.  
Only valid for the address register relative addressing modes.
- ZPC  
Pseudo register that represents the suppressed program counter.  
Only valid for the program counter relative addressing modes.
- d8  
Signed displacement up to 8 bits wide                   (-128-127)
- d16  
Signed displacement up to 16 bits wide               (-32768-32767)
- d32  
Signed displacement up to 32 bits wide
- bd  
Base Displacement up to 32 bits wide
- od  
Outer Displacement up to 32 bits wide
- .size  
Size of an index: either .W for 16 bits or .L for 32 bits
-

- .B, .W, .L  
Fixed size specifiers: byte, word, or long word, respectively
- \*scale  
Scale factor for an index: 1,2,4, or 8, for no scaling, word, long word, or quad word scaling, respectively
- imm  
An immediate value up to 32 bits wide
- ()  
Identifies first-level indirect addressing
- []  
Identifies second-level indirect addressing
- Data Register Direct: Dn
  - Address Register Direct: An
  - Address Register Indirect: (An )
  - Address Register Indirect with Postincrement: (An )+
  - Address Register Indirect with Predecrement: -(An )
  - Address Register Indirect with Displacement: (d16 ,An )
  - (old syntax): d16 (An )
  - Address Register Indirect with Index (8 Bit Displacement): (d8 .B,An,Xn .size \* ← scale )
  - (old syntax): d8 (An ,Xn .size \*scale )
  - Address Register Indirect with Index (Base Displacement): (bd .size
  - Memory Indirect Postindexed: ([bd .size ,An ],Xn .size \*scale ,od.size )
  - Memory Indirect Preindexed: ([bd .size ,An ,Xn .size \*scale ],od.size )
  - Program Counter Indirect with Displacement: (d16 ,PC)
-

- (old syntax): d16 (PC)
  
- Program Counter Indirect with Index (8 Bit Displacement): (d8.B,PC,Xn .size \* ←  
scale )
- (old syntax): d8 (PC,Xn .size \*scale )
  
- Program Counter Indirect with Index (Base Displacement): (bd .size,PC,Xn .size \* ←  
scale )
  
- Program Counter Memory Indirect Postindexed: ([bd .size ,PC],Xn.size \*scale ,od ←  
.size )
  
- Program Counter Memory Indirect Preindexed: ([bd .size ,PC,Xn .size\*scale ],od . ←  
size )
  
- Absolute Short Addressing: (d16 ).W
- (old syntax): d16 .W
  
- Absolute Long Addressing: (d32 )
- : (d32 ).L
- (old syntax): d32
- : d32 .L
  
- Immediate Data: #imm

## 1.17 pro.guide/Instruction Set

### Instruction Set

=====

ProAsm supports the instruction set of the complete M68000 family. For a complete overview refer to  
                   Instruction Set Summary  
                   . This section  
 discusses some peculiarities of the assembler and the instructions:

- \* Some instructions have variant forms that can be specified by the user or will be automatically used by the assembler. The ADD and SUB instructions have ADDA, SUBA, ADDI, SUBI, ADDQ, SUBQ as variants. The CMP instruction has CMPA and CMPM as variants. The AND, OR, and EOR instructions have their immediate variant (ANDI, ORI, and EORI).

The assembler will automatically use the appropriate form if possible. Note that the correct forms are always assembled faster

---

than if the "stubs" have to be converted by the assembler.

- \* A Bcc.B is equivalent to Bcc.S.
- \* If a short branch is specified to the following instruction, the branch will be replaced by a NOP instruction to avoid an execution error. A warning will be reported if such a conversion is made.

Example:

```
bra.s foo ; => converted to NOP
foo: ...
```

- \* There are two extensions to the standard condition codes: HS (higher or same - unsigned) and LO (lower - unsigned). They are equivalent to CC (carry clear) and CS (carry set).

For branch instructions BHS is equivalent to BCC, and BLO is equivalent to BCS. This will also work for the DBcc and the Scc instructions.

- \* If a MOVEQ instruction is used with an immediate source operand from 128-255 a warning is issued. The MOVEQ instruction expects an 8-bit (-128-127) immediate value that, during execution, is sign-extended to long (32-bit). Thus any 8-bit number greater than 127 will automatically become negative when sign-extended to long.

To suppress this warning you have to add a longword size specifier (.L).

- \* A LINK instruction with a positive or odd integer value as second argument reports a warning.
- \* The BTST instruction is the only bit manipulating instruction that allows program counter relative addressing modes.

BTST allows also an immediate addressing mode as destination if a data register is taken as source.

Consider the following example:

```
btst d0, #111100000011101 bne.s foo This example
jumps to foo if the data register d0 contains a value of zero, a
value from 2-4, or from 11-14.
```

## 1.18 pro.guide/END

- Stop reading the input file: END
- : ENDSRC

The END directive indicates the logical end of the source, and the assembler ignores the remainder of the file. This directive is optional; the end-of-file will be detected if no END directive is given. If the assembler detects an unexpected end-of-file (for

example within an expression) an error will be issued.

The END directive may also be used to indicate the end of an included file. If END is encountered in an include file, instead of stopping assembly entirely, ProAsm merely "drops out" of the include file and resumes with the source file that included it.

## 1.19 pro.guide/Include Files

### Include Files

-----

- Include source from file: INCLUDE filename

The INCLUDE directive allows you the inclusion of external files into the program source. The loaded files must be either source or preasm files.

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces.

If no explicit path is given (e.g. DF0:...) the assembler will first search in the current directory for the requested file, and after that in each of the directories defined by the INCDIR/INCPATH directives, and the -I/INCDIR command line option.

The included files have no restrictions on their content. They are only read in the first pass, but processed on all passes.

The INCLUDE directive can be nested as deeply as available memory allows.

Example:

```
INCLUDE "exec/types.i"
INCLUDE "exec/memory.i"
```

## 1.20 pro.guide/OPT Y

- Ignore multiple includes (default): OPT Y+
- : OPT NOINCONCE
- Do not ignore multiple includes: OPT Y-
- : OPT INCONCE

These options control the inclusion of files. The first two forms force the assembler to ignore multiple file inclusion. A file can be included as many times as you like to.

The last two forms tell the assembler to do not ignore multiple includes. The files are only allowed to be included once. Any further inclusion will be skipped by the assembler.

---



These options work together with the following directives:  
INCLUDE, INCEQU, MACLIB, and HEADER.

By default, multiple includes are ignored.

## 1.21 incbin

- Include a binary file: INCBIN filename [,size[,seek]]
- : IBYTES filename [,size[,seek]]

These directives are used to include raw binary data (such as graphic data, sound samples, sprite data, etc.) directly into the object code at the current position. The program counter is updated accordingly.

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces.

If no explicit path is given (e.g. DF0:...) the assembler will first search in the current directory for the requested file, and after that in each of the directories defined by the INCDIR/INCPATH directives, and the -I/INCDIR command line option.

Normally the complete file is loaded into the object code, except one of the optional parameters size or seek is given.

Size represents the number of bytes that are maximally read of the file filename. To avoid complications, the smaller size of both sizes (size and the file size) is taken into account for reading the data. If no size is given, the file size is taken instead.

The seek value defines the position in the file to start the reading. A positive value sets the position relative to the start of the file, and a negative value relative to its end. For example, 20 is the position 20 bytes forward from start, -20 is 20 bytes back from the end of file. If seek is set to zero, reading begins at the first byte of the file.

Consider the following example:

```
INCBIN "TheImage1.raw"           ; include the whole file
INCBIN "TheImage2.raw",1024      ; include max. the first 1024 bytes
INCBIN "TheImage3.raw",512,-512 ; include the last 512 bytes of the ←
    file
INCBIN "TheImage3.raw",,-512     ; same as above
INCBIN "TheImage4.raw",117,25    ; read 117 bytes from byte 25
```

## 1.22 pro.guide/INCEQU

- Include source from file in the first pass only: INCEQU filename  
The INCEQU directive allows you the inclusion of external files into the program source so they are processed in the first pass

only. The loaded files must be either source or preasm files.

Files included using INCEQU have restrictions on their content, because they are read and processed in the first pass only. Only symbol and macro definitions should be used in a file loaded by the INCEQU directive, since that is the only time ProAsm evaluates symbols and macro definitions. Code or data generating directives and temporary symbols are not allowed in such a file. An error is reported if any M68000 instruction or illegal directive is found in the included file.

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces. The INCEQU directive can be nested as deeply as available memory allows.

If no explicit path is given (e.g. DF0:...) the assembler will first search in the current directory for the requested file, and after that in each of the directories defined by the INCDIR/INCPATH directives, and the -I/INCDIR command line option.

## 1.23 header

- Include source from file at the very beginning: HEADER filename  
The HEADER directive allows you the inclusion of external files into the program source before assembling the main source code. The loaded files must be either source or preasm files. This directive must be used before any code or data generating directive, or an error will be reported by the assembler. Note that it is still processed after the configuration file, however.

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces.

If no explicit path is given (e.g. DF0:...) the assembler will first search in the current directory for the requested file, and after that in each of the directories defined by the -I/INCDIR command line option.

The included files have no restrictions on their content. They are read in the first pass only, but processed on all passes.

## 1.24 pro.guide/MACLIB

- Include preasm file: MACLIB filename  
The MACLIB is a special variant of the INCLUDE directive. It allows you only the inclusion of preassembled (preasm) files into the program source.

The filename must be in normal AmigaDOS format and must be

---

enclosed in double or single quotes if it contains any whitespaces.

If no explicit path is given (e.g. DF0:...) the assembler will first search in the current directory for the requested file, and after that in each of the directories defined by the INCDIR/INCPATH directives, and the -I/INCDIR command line option.

The file that is to be included using the MACLIB directive must be a preasm file, otherwise an error will be reported.

This directive is not the only one that allows the inclusion of preasm files. Alternative directives are INCLUDE, INCEQU, and HEADER.

## 1.25 pro.guide/INCDIR

- Define include search path: INCDIR path [,path[,...]]
- : INCPATH path [,path[, ...]]

This directives add the specified directory names to the list of directories that tell the assembler where to look for files to be included by the INCLUDE or INCBIN directives, or their synonyms. The assembler normally looks for includes in the current directory, then in the first directory in the search path, then in the second, and so on.

The INCDIR directive takes as its parameter a sequence of directory names separated by commas (,). Directory names containing whitespaces or commas must be enclosed in double or single quotes.

If you find yourself frequently referring to files with the same directory name, it might be worthwhile adding a line like the following to the top of your source code:

```
INCDIR "horribly/long/directory/name"
```

This way you can refer to the files like this:

```
INCLUDE "myIncludeFile.i"
```

instead of:

```
INCLUDE "horribly/long/directory/name/myIncludeFile.i"
```

We also recommend using multiple INCDIR directives each with just one search path, instead of multiple comma-separated directories in one directive, to provide a better overview of the entire search path.

Example:

```
INCDIR "Include:" ; specify first search path,
INCDIR "asm:myInclude/" ; specify another search path
```

```

INCLUDE "exec/memory.i"           ; Definitions used by the
INCLUDE "intuition/intuition.i"   ; operating system.
INCLUDE "project1/myDefinitions.i" ; Definitions for my project

```

## 1.26 pro.guide/Macros

Macros

=====

Identical or similar instruction sequences may often be repeated in different places in a program. Writing a sequence of instructions repeatedly can be tedious if the sequence is long or must be used many times.

A macro is a shorthand notation for something else. That "something else" may be much longer than the macro name itself, difficult to type, or it can be made more readable by using a macro for it.

A macro can contain M68000 instructions, assembler directives, and other macro calls. Thus, macro substitution is a process of replacing the macro name by its defined substitution code, called the macro body. A macro body is not limited in its size (limited only by available memory).

The example:

```

myMacro      MACRO
              lea    \1,a0
              bsr    WriteStr      ; write string
              ENDM

```

```

...
myMacro titletext

```

will be assembled to:

```

...
lea    titletext,a0
bsr    WriteStr      ; write string

```

This example illustrates the fact that everything in the macro body (including the comments you may write in the definition) replaces the macro name when the macro is invoked.

Typical uses for macros include also the definition of structure elements and structures themselves. For example you want to expand the PSTRING directive (see PSTRING) in a way that it supports strings longer than 255 characters:

```

PSTRING_     MACRO
              dc.w   \*STRLEN(\1)
              dc.b   "\1"
              EVEN
              ENDM

```

```

...
PSTRING_ <Strings up to 65535 characters allowed>
; => DC.W 38 / DC.B "Strings up to 65535 characters allowed"

```

While defining a macro, you must take care of the two following important points:

- \* You can call macros within macros, but you cannot define macros in macros.
- \* If you want to use local symbols within a macro body, then use the `'.'` (`'symbol'`) or a double backslash (`'\'`). The `'\'` will be assembled to a single backslash.

```
foo    MACRO
    ...
    tst.l    d0
    beq.s    \bar
    moveq    #0,d0
    bra.s    .out
\bar:  ...
.out:
    ENDM
```

CAUTION: When you use macros, you should carefully document them. Macros can impair the readability of a program if they are used indiscriminately and unnecessarily. This can make it extremely difficult to understand and to follow the program logic.

## 1.27 macro

- Begin macro definition: `symbol MACRO`
- End macro definition: `ENDM`
- : `ENDMAC`

The block of code between the `MACRO` and the matching `ENDM` directive is the contents of the macro, called the macro body, and may contain parameters. The provided symbol is used as macro name. When the assembler finds the macro name in the opcode field of the source code, the contents of the macro is expanded and inserted into the source code at the point of the macro name. Together with the macro name in the opcode field, you may supply any number of arguments in the operand field.

Arguments supplied in the operand field must be separated by commas. If an argument contains whitespaces or a comma then the argument should be enclosed in a matching set of angle brackets (`<` and `>`).

Consider the following example:

```
foo    MACRO
    DC.B    "\1", "\2"
    ENDM

foo    Argument1,Argument2 ; => DC.B "Argument1", "Argument2"
foo    <A,B,C>, <D,E,F>    ; => DC.B "A,B,C", "D,E,F"
foo    <Hello World>, !    ; => DC.B "Hello World", "!"
```

See

Symbols and Keywords for Macros  
, for more information about

macro arguments and macro directives.

- Exit from macro: MEXIT

This directive can be used to terminate the current macro call prematurely (as though there were no more source code in the macro body). Usually this directive is used in association with a conditional directive. This allows you to have more control over a macro:

```
foo      MACRO
        move.l  \1,d0
        IFEQ   NARG-1      ; just one argument given?
                MEXIT      ; => exit macro.
        ENDC
        move.\0 \2,d1
```

## 1.28 pro.guide/Symbols and Keywords for Macros

### Symbols and Keywords for Macro Definitions

-----

This section describes a number of special symbols and keywords which can be used within macro definitions to build more powerful macros.

Keywords start with a backslash character ( \ ) and can be used anywhere in a line, even embedded within a symbol name, while special symbols such as NARG can only be used where ordinary symbols could be used. The \\*S(symbol) keyword can be used to embed special (or even ordinary) symbols anywhere in the source text; see

Substituting textual symbols in symbol names  
, for more information.

- Argument n to macro: @{i}n

- : \ (n)

Macro invocations can take parameters just like M68000 instructions and assembler directives. A symbol consisting of a backslash and a number greater than zero is substituted with the nth argument to the macro. For example, the following macro generates a synonym for the moveq #0,reg instruction:

```
clrd    MACRO
        moveq  #0,\1
        ENDM
```

After this macro is defined, you can use, for example, clrd d0 instead of moveq #0,d0.

With the @{i}n form, n can be a single digit from 1 to 9 to indicate one of the first nine parameters, or a lowercase letter from a to z to indicate parameters 10-35, respectively. With the \ (n) form, n can be any number or constant numeric expression (e.g. \ (16+i), \ (34), ...).

Macro arguments are textually substituted before they are evaluated, so be careful of possible side-effects. For example, this code sequence does not have the result that was probably intended:

```
times2 MACRO
    moveq    #\1*2,d0
    ENDM

    times2  2+4                ; => moveq #10,d0
```

This is because the text 2+4 is substituted for \1 before the expression is evaluated. The expression that actually gets evaluated is 2+4\*2, which ends up as 10 because of arithmetic precedence rules.

To avoid situations like this, you may want to surround any `@{i}n` symbols used in arithmetic expressions with parentheses. For example, the above piece of code could be fixed like this:

```
times2 MACRO
    moveq    #(\1)*2,d0
    ENDM

    times2  2+4                ; => moveq #12,d0
```

You can take advantage of textual substitution to dynamically construct symbol names within a macro. For example, the following macro, when invoked with part of a symbol name as a parameter, defines three new symbols with names based on the one supplied:

```
one_two_three MACRO
    \1_plus_one EQU    (\1)+1
    \1_plus_two EQU    (\1)+2
    \1_plus_three EQU  (\1)+3
    ENDM
```

- Size tag used to invoke macro: \0

This symbol is replaced by the single-letter size tag used on the macro invocation. If the macro is invoked without any size specifier, word size (.w) is assumed.

For example, this macro definition clears a byte, word, or longword at a0, and post-increments a0 appropriately:

```
stuff0 MACRO
    clr.\0 (a0)+
    ENDM

    stuff0.b        ; stuff a byte with 0
    stuff0.w        ; stuff a word with 0
    stuff0          ; (same as above)
    stuff0.l        ; stuff a longword with 0
```

- Unique number: \#

- Unique number prefixed with \_: @

These symbols are replaced with some arbitrary number which is guaranteed to be unique for every macro invocation. The @ form prefixes the number with an underscore, while the \# form generates the number alone.

These symbols are generally used to generate unique labels within macros that can be used many times. For example, this macro stores the value in d0 into somewhere, but only if d0 is not zero:

```
foo      MACRO
        tst.l   d0
        beq.b  lab
        move.l  d0,somewhere
lab:
        ENDM
```

Unfortunately, this macro can be used only once--if you try to use it more than once, ProAsm will generate a Symbol defined twice error, because lab gets defined once in each macro invocation. This can be fixed by changing the code to look like this:

```
foo      MACRO
        tst.l   d0
        beq.b  lab@
        move.l  d0,somewhere
lab@:
        ENDM
```

In this case, each time the macro is invoked, a new symbol of the form lab\_n is generated, so multiple invocations of the macro do not conflict with each other.

Unfortunately, this macro definition still is not perfect. It will cause problems if it is used in code that defines local symbols, as in the following fragment:

```
        tst.l   d1
        bne.b  .nearby
foo
.nearby
```

In this example, the global label generated in the invocation of foo will "chop" all local labels at that point, and ProAsm will generate an Undefined symbol error. The macro can be fixed once and for all by making its internal symbol local, like this:

```
foo      MACRO
        tst.l   d0
        beq.b  .lab@
        move.l  d0,somewhere
.lab@:
        ENDM
```

- Value of expression as decimal text: \\*VAL
- : \\*VALOF
- : \\*V
- : \\*D



- Value of expression as hexadecimal text: `\*HEX`
- : `\*HEXOF`
- : `\*H`
- Value of expression as binary text: `\*BIN`
- : `\*BINOF`
- : `\*B`
- Value of expression as octal text: `\*OCT`
- : `\*OCTOF`
- : `\*O`

When ProAsm sees one of these symbols, it first calculates expression, which must evaluate to a numeric constant, and then it textually substitutes the computed value of that expression in place of the original symbol. The `\*D(expression)` and `\*V...(expression)` forms substitute the number in decimal, the `\*H...(expression)` forms substitute the value in hexadecimal, the `\*B...(expression)` forms in binary, and the `\*O...(expression)` forms in octal.

For example, the following macro can be used to automatically generate a linked list:

```

counter SET      1                      ; Initial value

defnode MACRO
node_\\*D(counter):                ; This node's label
    DC.L    node_\\*D(counter+1)    ; Pointer to next node
counter SET    counter+1                ; Bump node counter
ENDM

```

Each invocation of `defnode` will then define a new node in a linked list, with the first longword containing a pointer to the next node.

## 1.29 pro.guide/Substituting textual symbols in symbol names

- Text contained in textual symbol: `\*STRING`
  - : `\*S`
- This keyword allows textual symbols such as those defined by the `EQR` directive (see `EQR`) to be used in places where such a symbol could not normally be used; for example, as part of a symbol name. ProAsm first expands the textual symbol, then substitutes it directly into the source code.

Example:

```

foo          EQU    bar

bar_one     EQU    1
bar_two     EQU    2

DC.B    \\*S(foo)_one,\\*S(foo)_two    ; 1, 2

```

- Decimal value of string length: `\*STRLEN`

When ProAsm encounters this macro directive, it first evaluates the length of the given string, and then it textually substitutes the decimal value of resulting string length in place of the original macro directive.

For example, the following macro can be used to automatically generate a linked list of given strings:

```

counter SET      1                                ; Initial value

defstring MACRO
node_\D(counter) :                               ; This node's label
    DC.L      node_\D(counter+1)                 ; Pointer to next node
counter SET      counter+1                       ; Bump node counter

    DC.B      \\*STRLEN(\\1)                       ; insert string length
    DC.B      '\1'                             ; insert given string
    EVEN
    ENDM

defstring Count
defstring Zero

```

### 1.30 pro.guide/Substituting subsections of strings

- Specified number of characters from left: `\*LEFT`
- : `\*L`
- Specified number of characters from right: `\*RIGHT`
- : `\*R`

These macro directives determine the substring out of the given string argument. Expr specifies the number of characters that are taken from left (`\*LEFT()`) or from right (`\*RIGHT()`) respectively of string as substring. Expr is taken as unsigned value, and zero will be similar to an empty string.

If string shorter than the requested length for the substring, the complete string is taken.

The substring then is textually substituted in place of the original macro directive.

Consider the following example:

```

foo      MACRO
    DC.B      "\*LEFT(\\1,\\2)", "\*RIGHT(\\1,\\2)"
    ENDM

foo      ABCDEF, 2
; => DC.B "AB", "EF"
foo      ABCDEF, 4
; => DC.B "ABCD", "CDEF"
foo      ABCDEF, 6

```

```

; => DC.B "ABCDEF", "ABCDEF"
foo      ABCDEF, 8
; => DC.B "ABCDEF", "ABCDEF"

```

- Specified number of characters, starting at position n: `\*MID`
- : `\*M`  
This macro directive textually substitutes the substring, defined by the given arguments, in place of the macro directive.

This substring is ascertained by m characters starting at position n. If there are less than m characters left to the right of position n, the remainder of the string is taken as substring.

The expressions n and m are taken as unsigned values, and at least one expression resulting in zero will produce an empty substring.

For example:

```

...
DC.B      "\*MID(ABCDEF, 3, 2) "
; => DC.B "CD"
DC.B      "\*MID(ABCDEF, 4, 8) "
; => DC.B "DEF"
...

```

- string converted to uppercase: `\*UPPER`
- string converted to lowercase: `\*LOWER`  
These macro directives allow strings to be converted in upper or lowercase, and then the converted strings are textually substituted in place of the original macro directive.

For example, the following macro can be used to generate a list with zero terminated upercased strings:

```

Upper_  MACRO          dc.b      "\*UPPER(\1)", 0          ENDM

Upper_  Appleseed     ; => dc.b "APPLESEED", 0          Upper_
Rumpelstiltskin     ; => dc.b "RUMPELSTILTSKIN", 0        Upper_
Orion              ; => dc.b "ORION", 0

```

## 1.31 pro.guide/Conditional Assembly

Conditional Assembly  
=====

The purpose of conditional assembly directives is to determine whether a section of source code is to be assembled or not.

These directives commonly are used in include files containing symbol and macro definitions to determine if the file has already been included.

This mechanism works like this (the example is taken from the Commodore `exec/types.i` include file):

```

IFND EXEC_TYPES_I
EXEC_TYPES_I SET 1

... ; definition body

ENDC ; EXEC_TYPES_I

```

But what does this code fragment? This code fragment first checks if the symbol `EXEC_TYPES_I` has already been defined. If it has been defined, the whole part between the `IFND` and the `ENDC` directive will be skipped by the assembler. The other way round if the symbol has not been defined, the next line defines the symbol and the definition body gets assembled. The `ENDC` directive informs the assembler that this conditional directive is completed.

This construction prevents multiple definitions of the include file body, while enabling you to include `exec/types.i` wherever and whenever you feel you have to.

All the following described conditional directives have two things in common: they all end with an `ENDC` directive (or by its synonym), and they all control the inclusion or exclusion of one or more groups of code lines. Conditional assembly directives may be nested up to 2147483647 ( ) times.

- Assemble if symbol defined: `IFD` symbol
- Assemble if symbol not defined: `IFND` symbol

Depending on whether or not the symbol in question has been defined, the assembler will include or exclude the source code lines until a matching `ENDC` or `ELSE` directive is found.

Symbol may relate to a symbol of any type.

Consider the following example:

```

IFND DefaultValue
DefaultValue EQU 10
ENDC

```

- Assemble code if expression is equal to zero: `IFEQ` expression
- Assemble code if expression is not equal to zero: `IFNE` expression
- Assemble code if expression is greater than or equal to zero: `IFGE` expression
- Assemble code if expression is greater than zero: `IFGT` expression
- Assemble code if expression is higher than zero: `IFHI` expression
- Assemble code if expression is less than or equal to zero: `IFLE` expression
- Assemble code if expression is lower or same as zero: `IFLS` expression
- Assemble code if expression is less than zero: `IFLT` expression
- Assemble code if expression is minus: `IFMI` expression
- Assemble code if expression is plus: `IFPL` expression

The expression is evaluated and its result is used to determine whether or not the following code lines will be assembled (up until an `ENDC` or `ELSE`).

For example:

```

IFD SymbolA ; SymbolA defined?

```

```

IFMI SymbolA      ; if so, check if its value is negative.
neg.l d0          ; if TRUE, add this code line.
ENDC
ENDC

```

Even more fancy expressions are possible:

```

IFNE      (Counter1<>0)&(FlagB) ; IFNE = ``IF TRUE``
...
ENDC

```

- Assemble code depending on the comparison of both expression: IFEQ e,e
- : IFNE expression,expression
- : IFCC expression,expression
- : IFHS expression,expression
- : IFCS expression,expression
- : IFLO expression,expression
- : IFGE expression,expression
- : IFGT expression,expression
- : IFHI expression,expression
- : IFLE expression,expression
- : IFLS expression,expression
- : IFLT expression,expression
- : IFMI expression,expression
- : IFPL expression,expression
- : IFVC expression,expression
- : IFVS expression,expression

Both expressions are evaluated and the comparison of their results is used to determine whether or not the following code lines will be assembled (up until an ENDC or ELSE).

Both expressions are compared before the condition is checked. If the comparison of the expressions is not true according to the defined condition, assembly is disabled.

Example:

```

IFLE foo,bar
PRINTX 'bar' is less or equal than 'foo'
ENDC

```

The tests for the conditions are equal to the ones for the conditional branch instructions of the M680x0 family:

```

CC
    carry clear

CS
    carry set

EQ
    equal

GE
    greater or equal

```

GT	greater than
HI	higher
HS	carry clear (synonym to CC)
LE	less or equal
LO	carry set (synonym to CS)
LS	lower or same
LT	less than
MI	minus
NE	not equal
PL	plus
VC	overflow clear
VS	overflow set

- Assemble code if strings are the same: `IFC string,string`
  - Assemble code if strings are not the same: `IFNC string,string`
- These directives compare the two given strings. Depending on whether or not the strings are identical, the assembler will include or exclude the source code lines until a matching `ENDC` or `ELSE` directive is found.

If a string contains any whitespaces or comma then it must be enclosed within single or double quotes.

Note that the strings are compared case dependently.

These directives are often used within macro bodies to check the validity of a macro parameter, as shown in the following example:

```
MyMacro MACRO
    IFC '','\1' ; first is a null string
    FAIL *** MyMacro: no parameter given!
    ENDC
    ...
```

ENDM

- Assemble code if within the first pass: IF1
  - Assemble code if within the last pass: IF2
- These directives can be used to let the assembler do certain operations only in the first or last pass of the assembly.

For example:

```
IF1
SymbolA EQU 1
SymbolB EQU 12
ENDC

IF2
PRINTX  *** Pass 2 in progress...
ENDC
```

Please note: Never ever put any code within these two directives, it may cause a difference in the object code size produced between the passes!

- Assemble code if macro is used: IFU macro
  - Assemble code if macro is not used: IFNU macro
- These directives can be used to determine whether the macro in question has been used until now, the assembler will include or exclude the source code lines till a matching ENDC or ELSE directive is found.

For example:

```
MyMacro MACRO
...
ENDM

...
IFU MyMacro
INCLUDE "MyMacro_Routines.s" ; include additional routines
ENDC                               ; used by the MyMacro macro.
```

- Toggle assembly condition: ELSE
  - : ELSEIF
- These directives reverse the condition whether the code is assembled or not from the last IF condition. The ELSE/ELSEIF directive matches the most recent IF directive.

Consider the following example:

```
IFEQ number
PRINTX 'number' is zero.
ELSE
PRINTX 'number' is not zero.
ENDC
```

- Toggle assembly condition and check expression: ELSEIF expression  
The use of the ELSEIF directive together with an expression is an extension to the standard ELSE/ELSEIF directive.

The ELSEIF expression statement is an abbreviation for

```
ELSE
  IFNE    expression
```

It reverses whether the code is assembled or not from the last IF condition. If the toggles to TRUE (the previous condition was FALSE) and the expression is unequal to zero the assembler will include the source code lines until a matching ENDC or ELSE directive is found.

For example:

```
IFND    bar
PRINTX  *** bar not defined!
ELSEIF  bar
add.l   #bar,d0                ; this line is only assembled if ←
      the
ENDC    ; symbol bar is defined and not ←
      zero.
```

- Terminate conditional assembly: ENDC
- : ENDIF  
Returns the assembly status to what it was before the last IF condition. An ENDC/ENDIF directive is required for each IF directive. It matches the most recent IF directive.

- Immediate IFNE: IIF expression instruction  
This is the immediate form of the IFNE directive and it is only effective on instruction. Neither the ENDC nor the ELSE can be used together with this directive.

IIF enables the assembly of instruction if the result of the given expression is unequal to zero.

For example:

```
IIF foo PRINTX  *** foo is unequal to zero.
```

The preceding example is similar to the following:

```
IFNE    foo
PRINTX  *** foo is unequal to zero.
ENDC
```

- Break assembly: ENDASM
- Continue assembly: ASM  
The ENDASM and ASM directives are a special form of conditional assembly directives. They can be used to explicitly exclude the



source code lines between the ENDASM and the next ASM directive.

```
...
ENDASM
; this source code line will not be assembled.
ASM
...
```

A similar construction can be obtained by using one of the standard conditional directives, for example:

```
...
IFEQ 1
; these source code lines will not be assembled,
; since the condition of the IF directive
; is always FALSE.
ENDC
...
```

This can be useful if you want to have old code fragments to be included in the source code but not being assembled, for example as comment or for archive reasons.

These directives can not be nested.

## 1.32 pro.guide/Repeating Text

### Repeating Text

=====

Frequently the same sequence of instructions has to be repeated several times in the same source file, for example if you are generating data tables or some kind of routine that consists of similar parts. To make it easier to define repeated sections of code or data, ProAsm features two kinds of repeat loops: the REPT - ENDR loop and the REPEAT - UNTILcc loop. The assembler will repeat the block between the repeat loop introducer and the matching repeat terminator. After assembling this block the repeat condition will be checked and then the assembly will be recommenced at the beginning of the repeat or the repeat will be terminated.

- Begin repeat loop: REPT expression
- End repeat loop: ENDR

The block of code between the REPT and the matching ENDR directive will be repeated expression number of times. If expression evaluates to zero the assembler will skip the instructions within this loop.

Example:

```
REPT 4
move.l d0, (a0)+
ENDR
; the statement above is the same as the lines below
```

```
    move.l d0, (a0)+
    move.l d0, (a0)+
    move.l d0, (a0)+
    move.l d0, (a0)+
```

- Begin repeat loop: REPEAT
- End repeat loop: UNTILcc expression [,expression ]  
The block of code between the REPEAT and the matching UNTILcc directive will be repeated as long as the result of the expression satisfies the condition cc.

The UNTILcc directive marks the end of the repeat loop and has two or three parameters: a condition and one or two expressions. The given expression is evaluated and according to the cc condition the loop is recommenced or terminated. If two expressions are given the results of both expressions are compared before the condition is checked. The cc may specify the following conditions:

CC	carry clear
CS	carry set
EQ	equal
GE	greater or equal
GT	greater than
HI	higher
LE	less or equal
LS	lower or same
LT	less than
MI	minus
NE	not equal
PL	plus
VC	

---

```
overflow clear
```

VS

```
overflow set
```

The advantage of this loop is that the expressions may involve variables that are undefined when the repetition started. For example, a counting loop may be expressed as:

```

i      EQU      0
REPEAT
...
i      SET      i+1      ; increment the loop counter
UNTILEQ j,n           ; process loop n times
```

The termination condition is checked each time after the source sequence within the loop. As a consequence, the sequence is assembled at least once. The REPEAT loop introduces the danger of a nonterminating loop. Evidently, such loops must be used with care. Consider, for example:

```

REPEAT
i      EQU      i-2
UNTILEQ i
```

It is easy to realize that the above loop only terminates if *i* is higher than zero and even (divisible by two).

Since ProAsm supports also comparison operators within an expression, UNTILcc directives with two given arguments may be reduced into a more obvious statement:

```

UNTILLE i,n
; can be reduced to
UNTILEQ i<=n
```

See Operators, for more information about the comparison operators.

#### - Exit from a repeat loop: REXIT

This directive can be used to terminate the current repeat loop prematurely. Usually this directive is used in association with a conditional directive. This allows you to have more control over a repeat loop:

```

REPT -1                ; loop forever!
IFNE i>=n
  REXIT                ; ...except we can break out here
ENDC
; ...
i SET i+1
ENDR
```

This is a quite similar to a REPEAT-UNTILcc construct, except that this loop will terminate immediately if *i* is already greater or equal to *n* before the repeat loop started. The block of code within a REPEAT-UNTILcc loop will be assembled at least once.

A counting loop may be expanded as:

```

i EQU 0
REPEAT
...
i SET i+1
```

```

        UNTILEQ j=n

; or

        REPT n
        ...
        ENDR

```

### 1.33 pro.guide/Equates

#### Equates

-----

- Define numeric symbol: symbol EQU expression
- : symbol EQUATE expression
- : symbol = expression
- : symbol == expression
- : DEFINE symbol = expression

Equates symbol to the value of the result of expression. The expression is calculated immediately, and must resolve to a numeric constant or a relocatable address. symbol can then be used in place of the value of expression. The symbol may be a forward reference to a symbol defined later in the program, subject to the rules explained in Forward References to Numeric and Textual Symbols.

- Define numeric symbol: symbol SET expression
- : symbol SETVAL expression

These directives assign the result of expression to the symbol. The expression is calculated immediately, and must resolve to a numeric constant or a relocatable address. symbol can then be used in place of the value of expression. These directives are identical to the EQU directives, with the exception that the assignment is temporary and the symbol may be redefined. You can always change the value of a numeric symbol later in the program.

The symbol may be a forward reference to a symbol defined later in the program, subject to the rules explained in Forward References to Numeric and Textual Symbols.

Consider the following example:

```

foo SET 1994
    move.l #foo,d0
foo SET 1995
    move.l #foo,d1

```

will be assembled to:

```

    move.l #1994,d0
    move.l #1995,d0

```

- Define numeric floating-point symbol: `symbol FEQU expression`  
Equates `symbol` to the floating-point value of the result of `expression`. The `expression` is calculated immediately, and must consist of valid floating-point numbers and resolve to a numeric floating-point constant. `symbol` can then be used in place of the value of `expression`.

The symbol may be a forward reference to a symbol defined later in the program, subject to the rules explained in Forward References to Numeric and Textual Symbols.

Example:

```
pi      FEQU    3.14159265359
      ...
      fmove.d #pi,fp0
```

- Define numeric floating-point symbol: `symbol FSET expression`  
Equates `symbol` to the floating-point value of the result of `expression`. The `expression` is calculated immediately, and must consist of valid floating-point numbers and resolve to a numeric floating-point constant. `symbol` can then be used in place of the value of `expression`. This directive is identical to the `FEQU` directive, with the exception that the assignment is temporary and the symbol may be redefined. You can always change the value of a numeric floating-point symbol later in the program by using `FSET`.

The symbol may be a forward reference to a symbol defined later in the program, subject to the rules explained in Forward References to Numeric and Textual Symbols.

Example:

```
foo     FSET    3.14159265359
      fmove.d #foo,fp0
foo     FSET    2.71828182845
      fmove.x #foo,fp1
```

will be assembled to:

```
fmove.d #3.14159265359,fp0
fmove.x #2.71828182845,fp1
```

## 1.34 pro.guide/EQR

- Define textual symbol: `symbol EQUR text`
- : `symbol EQUR <text>`
- : `symbol EQUSTR text`
- : `symbol EQUSTR <text>`
- : `symbol FEQR text`
- : `symbol FEQR <text>`

Attaches text to symbol, without actually evaluating or interpreting the text in any way. When symbol is used in an operand field somewhere else in the program, it is substituted with text before the operands are otherwise evaluated. The second form ( <text>) allows you to include commas, semicolons, or whitespace in the text.

The EQU\* directive is historically meant specifically as an EQU-like directive which applies to M68000 registers, or register lists for movem instructions, instead of numeric values. Hence the name: "EQUate Register." (FEQU, as you may guess, was the same for floating-point registers).

For example, this code will push all the data registers onto the stack:

```
dregs    EQU*    d0-d7
          movem.l dregs,-(sp)
```

ProAsm extends the historical definition of EQU\* to allow you to define a symbol to be equal to any piece of text.(1) For example, this code fragment will work under ProAsm:

```
msg      EQU*    "Hello world"
          DC.B    msg
```

will be assembled to:

```
DC.B    "Hello world"
```

If you want to include commas, semicolons, or whitespace in the EQU\* replacement text, you must surround the entire text string with angle brackets, like this:

```
foo      EQU*    <d0,d1>
          move.l  foo          ; => move d0 to d1
```

See Textual Symbols, for more information on how textual symbols defined by EQU\* are expanded by ProAsm.

- Define textual symbol: symbol SETR text
- : symbol SETR <text>
- : symbol SETSTR text
- : symbol SETSTR <text>
- : symbol FSETR text
- : symbol FSETR <text>

Attaches text to symbol, without actually evaluating or interpreting the text in any way. When symbol is used in an operand field somewhere else in the program, it is substituted with text before the operands are otherwise evaluated. The second form ( <text>) allows you to include commas, semicolons, or whitespace in the text.

These directives are identical to the EQU directive (see EQU),

with the exception that the assignment is temporary and the symbol may be redefined. You can always change the text of a textual symbol later in the program by using one of these directives.

The EQU and SETR directives allow you to define a symbol to be equal to any piece of text. For example, this code fragment will work under ProAsm:

```
msg      EQU      "Hello world"
          DC.B    msg
msg      SETR     "Hello again"
          DC.B    msg
```

will be assembled to:

```
DC.B    "Hello world"
DC.B    "Hello again"
```

If you want to include commas, semicolons, or whitespace in the SETR replacement text, you must surround the entire text string with angle brackets, like this:

```
foo      SETR     <d0,d1>
          move.l  foo          ; => move d0 to d1
```

See Textual Symbols, for more information on how textual symbols defined by EQU or SETR are expanded by ProAsm.

- 
- (1) <C programmers will notice that this directive is similar to the #define directive of the C preprocessor.

## 1.35 reg

- Equate register list mask to a symbol: symbol REG register list
- : symbol EQUURL register list

These directives assign a register list to a symbol that can be used in place of the register list for a MOVEM instruction. The format of the register list is the same as for the MOVEM instruction. The advantage of using it is that if you decide to change the register list, you only have to change the register list on the REG directive instead of changing every MOVEM instruction that uses it.

Example:

```
foo      REG      d0
bar      REG      d0-d3/d5-d6/a0-a6
```

```

        movem.l bar,-(a7)
    ...
    movem.l (a7)+,bar

```

This directive performs the historical function of the EQU directive: it will only work with register lists, not with arbitrary text strings as EQU does.

- Equate register list mask to a symbol: symbol SETREG register list
  - : symbol SETRL register list
- These directives assign a register list to a symbol that can be used in place of the register list for a MOVEM instruction. These directives are identical to the REG directive, with the exception that the assignment is temporary and the symbol may be redefined. You can always change the register list of a register list mask symbol later in the program by using one of these directives.

The format of the register list is the same as for the MOVEM instruction.

For example:

```

    foo    SETR    d0-d2/d5-d6/a0-a6

        movem.l foo,-(a7)
    ...
    movem.l (a7)+,foo

    foo    SETR    d0-a6
    movem.l foo,-(a7)
    ...
    movem.l (a7)+,foo

```

will be assembled to:

```

    movem.l d0-d2/d5-d6/a0-a6,-(a7)
    ...
    movem.l (a7)+,d0-d2/d5-d6/a0-a6

    movem.l d0-a6,-(a7)
    ...
    movem.l (a7)+,d0-a6

```

- Equate floating-point register list mask to a symbol: symbol FREG register list
- This directive assigns a floating-point register list to a symbol that can be used in place of the register list for a FMOVEM instruction. The format of the register list is the same as for the FMOVEM instruction. The advantage of using it is that if you decide to change the register list, you only have to change the register list on the FREG directive instead of changing every FMOVEM instruction in the source code that uses it.



The FREG directive can be used for both, the floating-point data and the floating-point control registers.

Example:

```
foo      FREG      fp0
bar      FREG      fp0-fp3/fp5-fp6 ; floating-point data registers
abcd    FREG      fpcr/fpsr/fpiar ; floating-point control register

      fmovem.x bar,-(a7)
      fmovem.l abcd,-(a7)
      ...
      fmovem.l (a7)+,abcd
      fmovem.x (a7)+,bar
```

This directive performs the historical function of the FEQR directive: it will only work with register lists, not with arbitrary text strings as FEQR does.

- Equate floating-point register list mask to a symbol: symbol FSETRL register list ←

This directive assigns a floating-point register list to a symbol that can be used in place of the register list for a FMOVE instruction. The format of the register list is the same as for the FMOVE instruction. This directive is identical to the FREG directive, with the exception that the assignment is temporary and the symbol may be redefined. You can always change the register list of a register list mask symbol later in the program by using one of these directives.

The FSETRL directive can be used for both, the floating-point data and the floating-point control registers.

Example:

```
foo      FSETRL    fp0-fp3/fp5-fp6 ; floating-point data registers

      fmovem  foo,-(a7)
      ...
      fmovem  (a7)+,foo

foo      FSETRL    fp0-fp6          ; floating-point data registers

      fmovem  foo,-(a7)
      ...
      fmovem  (a7)+,foo
```

will be assembled to:

```
fmovem  fp0-fp3/fp5-fp6,-(a7)
...
fmovem  (a7)+,fp0-fp3/fp5-fp6

fmovem  fp0-fp6,-(a7)
...
fmovem  (a7)+,fp0-fp6
```

## 1.36 pro.guide/Structure Offsets

### Structure Offsets

Highlevel languages usually have language elements to declare a structure:

C/C++

```
struct
```

Pascal/Modula/Oberon

```
RECORD
```

...

Structures in highlevel languages are declarations of a collection of elements as a unit, even if the elements are of different types (heterogeneous structures). The origin of this data structures lies in commercial data processing. (1)

Accessing and defining structures in assembler can demand great toil and effort. A structure as used in the assembly language is a collection of symbols which represent an offset in that structure. To access an element of a structure in assembly language, you typically use the "Address Register Indirect with Displacement" addressing mode or one of its variations (see

Addressing Modes

), with the address of

the beginning of the structure in the address register and the offset of the element you want to access as the displacement. For example, to load the element named LN\_TYPE from a structure pointed to by address register A2, you could use the instruction `move.b (LN_TYPE,a2),d0`.

Accessing structures from assembly language is not difficult; what can be a pain is figuring out the offset of each element and declaring a symbol for it. That's what the assembler directives described in this section are useful for.

You can declare structures in several ways. The most common methods are shown below as examples. The List Node Structure from the `exec/nodes.i` include file from Commodore is taken as a base for the examples.

The simplest, most straightforward way to declare a structure is by using the EQU directive.

```
LN_SUCC EQU    0
LN_PRED EQU    4
LN_TYPE EQU    8
LN_PRI  EQU    9
LN_NAME EQU   10
LN_SIZE EQU   14
```

However, this method requires careful computing of the offset of each element based on the size and offset of the preceding element. Also, it provides no easy way to insert or change the size of an

element: for example, if a new element is to be inserted at the beginning of a long structure, all of the elements following the new one have to be corrected by hand.

Another technique of declaring structures is by using macros as (for example) defined in the `exec/types.i` include file of Commodore.

```

...
BYTE    MACRO
\1      EQU      SOFFSET
SOFFSET SET SOFFSET+1
        ENDM

WORD    MACRO
\1      EQU SOFFSET
SOFFSET SET SOFFSET+2
        ENDM
...

```

More data storage macros can be found in the `exec/types.i` include file from Commodore.

The advantages of using these macros is that the defined structures can be ported very easily to other languages such as C/C++, and that the type of an element can be seen clearly from its declaration in the source code.

```

STRUCTURE LN,0
  APTR    LN_SUCC
  APTR    LN_PRED
  UBYTE   LN_TYPE
  BYTE    LN_PRI
  APTR    LN_NAME
  LABEL   LN_SIZE

```

A disadvantage of using macros to define structures is that macros are textual replacements and therefore it assembles more slowly than the first and the following method.

The easiest method of declaring structures is using the structure offset directives that are described later:

```

RSRESET
LN_SUCC RS.L 1
LN_PRED RS.L 1
LN_TYPE RS.B 1
LN_PRI  RS.B 1
LN_NAME RS.L 1
LN_SIZE RVAL

```

Using these directives, elements can be inserted, removed, or changed with hardly any effort at all.

To make it easier to declare data structures, ProAsm features two different kinds of structure offset directives. These directives do not directly affect your code or data space at all; they only define symbols for use in other parts of the program.

Each of the structure offset counters (`__RS`, `__SO`, and `__FO`) is a special symbol and can only be affected by its directives (see `__RS`).

- : [ symbol ] RS.size expression
- : [ symbol ] SO.size expression

These directives will assign the current value of the structure offset counter symbols (`__RS` and `__SO`, respectively) to the specified symbol, and then increment the counter according to size and expression. The expression is the number of data items of the size `size` the assembler reserves space for; in this way the structure offset directives work much like the `DS` directive. The symbol does not have to be specified; if it is omitted, then the directive only increments the structure offset counter.

The size specifier must be one of the following:

Example:

```

                                RSRESET
pa_next                        RS.L 1
pa_registers                   RS.L 16
pa_fpuregisters               RS.X 16
pa_SIZEOF                      RSVAl
```

Note that the `RS` and `SO` directives work exactly the same way, but they use different and independent structure offset counters. Therefore, you can use either directive to define a structure and get the same results, but in the definition of one particular structure you have to be consistent in which set of directives you use (either just `RS` or just `SO`).

- : [ symbol ] FO.size expression

This directive will decrement the frame offset counter according to size and expression and then assign its value to the optionally given symbol. It can be used to define a list of offset for a stack frame data structure (e.g. for the link instruction). In other respects it works just like the `RS` and `SO` directives.

For example, suppose you define a stack frame data structure this way:

```

                                CLRFO
fx_Pred                        FO.L 1 ; => -4
fx_Succ                        FO.L 1 ; => -8
fx_Flags                       FO.B 1 ; => -9
fx_pad                         ALIGNFO.W ; => -10
fx_SIZEOF                      FOVAL ; => -10
```

Within the source code you could use this stack frame definition as shown in the following code fragment:

```

link    a5,#fx_SIZEOF      ; allocate 10 bytes on the stack
movem.l d0/d1,fx_Pred(a5) ; store fx_Pred and fx_Succ
move.b  #$80,fx_Flags(a5)
...
move.l  fx_Pred(a5),a0
...
unlk   a5                  ; unlink
```

- Clear offset counter `__RS`: RSRESET
- Clear offset counter `__SO`: CLRSO
- Clear offset counter `__FO`: CLRFO

These directives clear their associated structure offset counters to zero. They are typically used at the beginning of a new structure definition.

```
RSRESET
    is equal to RSSET 0
```

```
CLRSO
    equals to SETSO 0
```

```
CLRFO
    is equal to SETFO 0
```

- Set offset counter `__RS`: [ symbol ] RSSET expression
  - Set offset counter `__SO`: [ symbol ] SETSO expression
  - Set offset counter `__FO`: [ symbol ] SETFO expression
- These directives can be used to set the corresponding offset counters to a specific desired value (expression).

An optionally given symbol will be set to zero.

Consider the following example:

```
MH          RSSET   LN_SIZE ; sets initial counter value to LN_SIZE
MH_ATTRIBUTES RS.W   1
MH_FIRST    RS.L   1
MH_LOWER    RS.L   1
MH_UPPER    RS.L   1
MH_FREE     RS.L   1
MH_SIZE     RSVAl
```

This defines the MH structure as an extension of the LN structure, by starting its structure offset counter at LN\_SIZE instead of zero.

The preceding structure is actually the Memory Region Header, defined in the `exec/memory.i` Commodore include file:

```
STRUCTURE   MH, LN_SIZE
            UWORD  MH_ATTRIBUTES
            APTR   MH_FIRST
            APTR   MH_LOWER
            APTR   MH_UPPER
            ULONG  MH_FREE
            LABEL  MH_SIZE
```

- Assign `__RS` offset value to symbol: symbol RSVAl
- Assign `__SO` offset value to symbol: symbol SOVAL

- Assign `__FO` offset value to symbol: `symbol FOVAL`  
These directives can be used to assign the current offset counter value to the given symbol.

```
RSVAL
    is equal to symbol EQU __RS
```

```
SOVAL
    equals to symbol EQU __SO
```

```
FOVAL
    is equal to symbol EQU __FO
```

-----  
(1) Programming language COBOL.

## 1.37 pro.guide/Case Sensitivity

### Case Sensitivity

-----

By default, all symbols in ProAsm are case-sensitive: `foo`, `Foo`, `FOO`, and `FoO` are all different symbols. However, code written for other assemblers may depend on symbols being case-insensitive, for example by making a reference `foo` to a symbol defined as `Foo`. ProAsm provides the following directives to control case sensitivity.

## 1.38 pro.guide/OPT C

- Make symbols case-sensitive (default): `CASEON`
  - : `OPT C+`
  - : `OPT CASE`
  - Make symbols case-insensitive: `CASEOFF`
  - : `OPT C-`
  - : `OPT NOCASE`
  - Make the first part of symbols case-sensitive: `OPT Cn+`
  - Make the last part of symbols case-sensitive: `OPT Cn-`
- The first four forms turn case sensitivity on or off as specified. `OPT Cn+` makes the first `n` characters of every symbol case sensitive, and the rest case-insensitive. `OPT Cn-` does just the opposite: it makes the first `n` characters of every symbol case-insensitive, and the rest case-sensitive. `n` must be in the range of 1 to 256.

These options should only be used before any symbols are defined or an option must be at beginning error will be reported.

---

## 1.39 pro.guide/OPT U

- Use underscore character (`_`) to indicate local labels: `OPT LOCALU`
  - : `OPT U+`
  - Use period (`.`) to indicate local labels (default): `OPT LOCALDOT`
  - : `OPT U-`
- The first two forms should be used if the underscore (`_`) rather than the dot (`.`) should introduce a local label. This option is useful if you assemble source codes developed with other assemblers which use the underline sign to introduce a local label. The character used to introduce a local label defaults to the dot.
- These options should only be used before any symbols are defined or an option must be at beginning error will be issued.

- Enable period (`.`) as local label introducer (default): `OPT U1+`
  - Disable period as local label introducer: `OPT U1-`
  - Enable underscore (`_`) as local label introducer: `OPT U2+`
  - Disable underscore as local label introducer (default): `OPT U2-`
- These options similar to the preceding options, except that they can be used to individually enable or disable each type of local label introducer, so all four combinations are possible.

## 1.40 pro.guide/Syntax Options

### Syntax Options

- 
- Accept both new and old syntax: `RELAX`
  - Accept new syntax only: `NEWSYNTAX`
  - Accept old syntax only: `OLDSYNTAX`
- Normally, ProAsm accepts new M68000 syntax (e.g., `move.l (16,a0),d0`), and old syntax (e.g., `move.l 16(a0),d0`). However, you can use the `NEWSYNTAX` or `OLDSYNTAX` directives to make ProAsm accept only new or old syntax, respectively, and generate an Addressing mode not recognized error if it encounters the "forbidden" syntax. The `RELAX` directive returns ProAsm to its normal state where it allows both new and old syntax. See

### Addressing Modes

, for the exact differences between new and old syntax.

## 1.41 pro.guide/OPT I

- Check absolute addresses for missing #: OPT CHKIMM
- : OPT I+
- Do not check absolute addresses (default): OPT NOCHKIMM
- : OPT I-

When this option is turned on the assembler will report a probably immediate addressing mode error on all instructions using the absolute addressing mode as source operand, unless it is the AbsExecBase (4).

This option may be very useful for detecting typing errors, since a missing pound sign (#) can cause subtle, hard-to-find bugs in which an instruction reads an operand from some arbitrary location in memory when an immediate value was intended.

You may force the assembler to use absolute addresses by appending one of the absolute address size specifiers .L or .W.

Example:

```
OPT      I+
move.l   4,a6           ; no error will be reported.
move.l   123456,d0      ; immediate addressing mode
                        ; was probably meant.
move.l   $f80000.L,d0  ; no error will be reported.
```

## 1.42 pro.guide/OPT P

- Ensure that code is position-independent: OPT CHKPC
- : OPT P+
- Disable check: OPT NOCHKPC
- : OPT P-

If this option is enabled the assembler reports a relocation not allowed error on any instructions that require relocation. Using this option you can make sure that the produced code will be position-independent.

Note that addresses defined by the ORG directive do not produce an error since they do not require relocation.

## 1.43 pro.guide/OPT NOTYPE

- Enable type checking: OPT TYPE
- : OPT T+
- Disable type checking: OPT NOTYPE
- : OPT T-

These options forces the assembler to do or to omit checks for the correct use of symbols and expressions in certain addressing modes.

If type checking is enabled (OPT T+) relocatable symbols are not



allowed in absolute word and address register indirect (indexed or not) addressing modes. These addressing modes require absolute values. However, sometimes type checking can be more of an obstacle than a help, so the checking can be disabled (OPT T-).

Examples:

```

; default
move.l (foo).w,d0      ; => error
move.l foo(a4),d7     ; => ok

OPT      T+
move.l (foo).w,d0     ; => error
move.l foo(a4),d7     ; => error

OPT      T-
move.l (foo).w,d0     ; => ok
move.l foo(a4),d7     ; => ok

foo:

```

By default, the assembler adopts a middle course. The type checking is done only for the absolute word addressing mode. To have the type checking completely enabled or disabled, these options have to be used.

## 1.44 pro.guide/Processor Options

- Select processor specific instruction set: MC68000
- : MC68008
- : MC68010
- : MC68020
- : MC68030
- : MC68040
- : MC68060
- : MC68EC020
- : MC68EC030
- Select complete instruction set of the M68000 family: MCRELAX
- : MC680X0

These directives are used to specify the processor to be programmed for. They tell the assembler the instruction set to be used, valid addressing modes, and the best code optimization possibilities.

You can use MCRELAX or MC680X0 to force the assembler to allow the complete instruction set of the M68000 family.

These directives do not have a negative form. By default, ProAsm is set to MC68000.

- Select allowed coprocessor: MC68881
  - : MC68882
  - : MC68851
- These directives tell the assembler what coprocessor you want to program for to allow their instruction set.

To decline a selected coprocessor you can use the OPT P=-coprocessor option (see OPT P=).

The following example specifies the MC68020 processor together with the MC68882 FPU and MC68851 MMU coprocessors:

```
MC68020          ; only the 68020 instruction set
MC68882          ; plus the 68882 FPU
MC68851          ; and the 68851 MMU instruction set
...
OPT      p=-68851 ; disables the MC68851 instructions
```

- Select processor specific instruction set: OPT P=[- ][processor][coprocessor ]  
This option can be used to specify the processor and coprocessor to be programmed for. It tells the assembler the instruction set to be used, valid addressing modes, and the best code optimization possibilities.

The optional given processor argument must be one of the following:

```
68000
68010
68020
68030
68040
68060
68EC030
68EC020
```

Valid coprocessor arguments are:

```
68881
68882
68851
```

A list of coprocessors can be selected just by separating them using a forward slash (/):

```
OPT      P=68882/68851
```

If more than one processor is selected, the last one is taken into account:

```
OPT      P=68020/68040          ; => MC68040 processor selected
```

The advantage of this option is that you have the possibility to unselect coprocessors by putting a minus sign (-) in front of the first (co)processor. Note that processor selections have no negative form, therefore they cannot be unselected. (By default only the MC68000 processor is specified).

Consider the following examples:

```
OPT      P=68030/68882          ; select MC68030 and MC68882
...

OPT      P=-68040/68882/68851 ; select MC68040, but
...                               ; unselect MC68882 and MC68851

OPT      P=68000                ; select the MC68000 processor,
OPT      P=68851                ; and the MC68851 coprocessor
```

...

```
OPT    P=-68882                ; unselect the MC68882 coprocessor
```

- Enable MC68040 software-supported instructions (default): OPT FPSP40
- Enable MC68060 software-supported instructions (default): OPT FPSP60
- Forbid MC68040 software-supported instructions: OPT NOFPSP40
- Forbid MC68060 software-supported instructions: OPT NOFPSP60

Both the MC68040 and the MC68060 processor have an optimized floating-point unit to directly execute the most commonly used subset of the extensive MC68881/MC68882 instruction set through hardware. The remaining instructions are emulated by Motorola's floating-point software package to ensure complete compatibility to the floating-point coprocessors.

These options can now be used to control the allowance of the software-supported instructions for the specified processor.

By default, all software-supported instructions are enabled.

Example:

```
MC68060                ; select the MC68060 as target processor,
OPT    NOFPSP60        ; but forbid the floating-point emulation.
...
fadd.x  fp1,fp0        ; allowed FPU instruction.
fasin.x fp0            ; a 'MC68060 software-supported instruction ←
    used'
...                    ; error will be reported.
```

- Enable MC68060 integer instruction emulation (default): OPT SP60
  - Forbid emulated instructions: OPT NOSP60
- The MC68060 left some low-use integer instructions unimplemented to streamline internal operations. These unimplemented integer instructions are software emulated to provide user object-code compatibility with the M68000 family.

These options can now be used to control the allowance of the software-supported integer instructions for the MC68060.

By default, the software-supported instructions are enabled.

The unimplemented integer instructions include 64-bit divide and multiply, move peripheral data, cas2, chk2, and cmp2. In addition, CAS used with a misaligned effective address is also unimplemented. The unimplemented integer instructions are:

```
divu.l <ea>,Dr:Dq      64/32 => 32r,32q
divs.l <ea>,Dr:Dq      64/32 => 32r,32q
mulu.l <ea>,Dr:Dq      32*32 => 64
muls.l <ea>,Dr:Dq      32*32 => 64
movep  Dx,(dl6,Ay)     size = W or L
movep  (dl6,Ay),Dx     size = W or L
```

chk2	<ea>,Rn	size = B, W, or L
cmp2	<ea>,Rn	size = B, W, or L
cas2	Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)	size = W or L
cas	Dc,Du,<ea>	size = W or L, misaligned <ea>

Refer to the 'M68000 Family Programmer's Reference Manual' for details on these instructions.

## 1.45 pro.guide/SUPER

- Enable warnings on supervisor-only opcodes (default): SUPER ON
- : OPT SUPER
- : OPT SW+
- Disable warnings: SUPER OFF
- : SUPER
- : OPT NOSUPER
- : OPT SW-

This option tells the assembler either to enable or disable the warnings reported if a privileged instruction is assembled. SUPER OFF should be used if you are writing code that will run in supervisor mode, but it is a good idea to leave SUPER ON for normal use to help avoid accidentally using instructions that are unavailable.

## 1.46 pro.guide/READMODWRITE

- Disable warnings for read/modify/write instructions: READMODWRITE
- : READMODWRITE OFF
- Enable warnings (default): READMODWRITE ON

This directive can be used to control the warnings produced by the CAS, CAS2, or the TAS instructions. These instructions trigger a special bus cycle, intended for multiprocessor support, which many models of the Amiga do not handle correctly. Using these instructions can conflict with system DMA, and are documented by Commodore as being unsuitable for use on the Amiga. The BSET instruction can be used as a substitute for TAS on the Amiga. By default the assembler will report a warning message.

## 1.47 pro.guide/SETKFACTOR

- Set default static k-factor for FPU Packed Decimal: SETKFACTOR expression
- This directive sets the default k-factor used for the FMOVE floating-point instruction if the destination format is Packed Decimal, to specify the format of the decimal string.

The default k-factor is only used if no k-factor (static or dynamic) is given to the FMOVE.P instruction. Format of the

```

instruction with a given and suppressed k-factor:
    fmove.p fpn,<ea>{dx }      ; dynamic k-factor given in a
                                ; data register dx
    fmove.p fpn,<ea>{#k}      ; static k-factor given as value k

    fmove.p fpn,<ea>          ; no k-factor given,
                                ; the assembler uses
                                ; the default k-factor instead.

```

The given expression must be in the range of -64 to +63, where the value defines the format as follows:

-64 to 0

Indicates the number of significant digit to the right of the decimal point (Fortran "F" format).

+1 to +17

Indicates the number of significant digits n the mantissa (Fortran "E" format).

+18 to +63

Sets the OPERR bit in the FPSR exception byte, treated as +17.

The following table gives several examples of how the k-factor value affects the format of the decimal string that is produced by the FPU.

k-Factor	Source Operand Value	Destination String
-5	+12345.678765	+1.234567877 E+4
-3	+12345.678765	+1.2345679 E+4
-1	+12345.678765	+1.23457 E+4
0	+12345.678765	+1.2346 E+4
+1	+12345.678765	+1 E+4
+3	+12345.678765	+1.23 E+4
+5	+12345.678765	+1.2346 E+4

The k-factor value defaults to zero if not specified. Refer to the 'MC68881/MC68882 Floating-Point Coprocessor User's Manual' or any other manual that describes the MC68881/MC68882 coprocessors for detailed information about the k-factor.

## 1.48 pro.guide/DEFAULT

- Define default behaviour: DEFAULT type@,mode

This directive can be used to set the default behaviour of ProAsm. In almost all cases this defines the size the assembler should use for either instructions or addressing modes if forward referenced addresses are found.

The most types are only useful when code utilizing the new addressing modes of the MC68020 or later processors is written.

As default type the following keywords are accepted:

ADRBASEDISP

As mode either WORD or LONG can be given. This sets the default size for the forward referenced address register base

displacements to 16 or 32 bit respectively. By default, the assembler uses 16 bit displacements (`_WORD`).

#### `_BASEDISP`

This type works exactly like `_ADRBASEDISP`, with the exception that it sets the default size for address base displacements (`_ADRBASEDISP`) and program counter relative base displacements (`_PCBASEDIP`). By default, the assembler uses 16 bit displacements (`_WORD`).

#### `_BRANCH`

Using this type the default size for branch instructions (`Bcc` and `BSR`) can be set to a 32, 16, or 8 bit jump offset respectively by selecting `_LONG`, `_WORD`, or `_BYTE` as mode. The `_SHORT` keyword is also supported as a synonym for the `_BYTE` mode. By default, the assembler uses 16 bit jump offset (`_WORD`).

#### `_LINKREF`

This type can be used to select between two different linkable object file formats. Using `_BASERELATIVE` as mode, the assembler generates a standard object module compatible with the Amiga standard linker (`ALINK`) and `BLINK` (the replacement linker from 'The Software Distillery'). Is `_RELATIVE` given as mode, object modules compatible with some older (non-standard) linkers will be generated. By default, `_BASERELATIVE` is used.

#### `_OUTERDISP`

This type works in the same manner as `_ADRBASEDISP`, with the exception that it sets the default size for outer displacements. Refer to `Addressing Modes`, for detailed information about the outer displacement. By default, the assembler uses 16 bit displacements (`_WORD`).

#### `_PCBASEDIP`

This type works precisely in the same manner as `_ADRBASEDISP`, except that it sets the default size for program counter relative base displacements. See `Addressing Modes`, for detailed information about the program counter relative base displacement. By default, the assembler uses 16 bit displacements (`_WORD`).

All the keywords are case-insensitive, and the leading underscore (`_`) of type and mode is optional.

- Set default behaviour: `OPT ABL`
- : `OPT ABW`
- : `OPT BDL`
- : `OPT BDW`

- : OPT BRL
- : OPT BRW
- : OPT BRB
- : OPT BRS
- : OPT ODL
- : OPT ODW
- : OPT PCBL
- : OPT PCBW

These options are abridged versions of the DEFAULT directive and can be used to set the default behaviour of ProAsm.

Here you find a list of these options and their matching DEFAULT directive:

ABL  
    DEFAULT \_ADRBASEDISP,\_LONG

ABW  
    DEFAULT \_ADRBASEDISP,\_WORD

BDL  
    DEFAULT \_BASEDISP,\_LONG

BDW  
    DEFAULT \_BASEDISP,\_WORD

BRL  
    DEFAULT \_BRANCH,\_LONG

BRW  
    DEFAULT \_BRANCH,\_WORD

BRB  
    DEFAULT \_BRANCH,\_BYTE

BRS  
    DEFAULT \_BRANCH,\_SHORT

ODL  
    DEFAULT \_OUTERDISP,\_LONG

ODW  
    DEFAULT \_OUTERDISP,\_WORD

PCBL  
    DEFAULT \_PCBASEDISP,\_LONG

PCBW  
    DEFAULT \_PCBASEDISP,\_WORD

See

    DEFAULT  
    , for more information about the meaning of each  
DEFAULT argument.

## 1.49 pro.guide/Optimization

- Automatic PC-relative addressing: OPT AUTOPC
- : OPT A+
- Disable automatic PC-relative addressing: OPT NOAUTOPC
- : OPT A-
  - If OPT AUTOPC is specified the assembler uses the PC-relative addressing mode where possible, instead of the absolute long addressing. The use of this option may result in considerable reduction of the object code size and runtime.

This transformation is made before the instruction is actually assembled so if the address is out of range an error will be reported. To override this transformation you can force the assembler to use the absolute long addressing mode by appending the .L size specifier, as with the absolute word addressing mode.

OPT NOAUTOPC is the default. See  
 Addressing Modes  
 , for more  
 information about the PC-relative addressing mode.

```

OPT      A+
move.l   foo,d0
; will be assembled to:
move.l   foo(PC),d0

move.l   (foo).L,d0
; will be left unaffected
  
```

- Enable all optimizations: OPTIMIZE
- : OPT O+
- Disable all optimizations (default): NOOPTIM
- : OPT O-
- Enable optimizations n, m, ...: OPTIMON n[,m ...]
- Enable optimization n: OPT On+
- Disable optimizations n, m, ...: OPTIMOFF n[,m ...]
- Disable optimization n: OPT On-

ProAsm is capable of performing various optimizations causing instructions to be smaller and faster. No optimizations are performed by default.

All optimizations performed will be reported by a warning. You can control these warnings using the OW+ or OW- options. If any optimization has been done the assembler will report the number of optimizations made and bytes saved at the end of the assembly.

You may specify single optimizations by inserting a number between the O letter and the plus (+) or minus (-) sign. On+ will enable the optimization and On- is used to disable it.

Example:

```

OPT      o+           ; turns all optimizations on
  
```



```

OPT  o+,o1-,o6- ; turn all optimizations on, except 1 and 6
OPT  o-,o3+,o2+ ; only optimizations 2 and 3 are turned on
OPT  o5+        ; allow the optimization of add and sub

```

The following specific optimizations are available:

01+

01-

Optimizes backwards branches to their short form if in range of -128-0 bytes.

Example:

```

foo:    ...
        bra     foo
        ; will be optimized to
        bra.s  foo
        ; (2 bytes saved)

```

02+

02-

Optimizes the address register indirect with displacement addressing mode to the address register indirect, if the displacement is zero.

Example:

```

bar     EQU     0
        move.l  0(a2),d0
        move.l  bar(a1),d2
        ; will both be optimized to
        move.l  (a2),d0
        move.l  (a1),d2
        ; each optimization saved 2 bytes

```

03+

03-

Absolute long addressing mode within the range of -32768-32767 will be optimized to absolute word.

Example:

```

        move.l  4,a6
        ; will be optimized to
        move.l  4.W,a6
        ; and 2 bytes saved

```

Note that if the address size is given specifically as absolute long (.L) no optimization will be made.

04+

04-

move.l #x,dn statements with an immediate value within the range of -128-127 will be optimized to moveq #x,dn. This optimization saves four bytes.

05+

05-

The add and sub instructions with an immediate number between 1 and 8 as source operand will be optimized to their quick forms.

Example:

```
add.w    #2,counter(a5)
add.l    #4,d0
; can be optimized to
addq.w   #2,counter(a5)
addq.l   #4,d0
; and 2 respectively 4 bytes saved
```

06+

06-

This is not really an optimization. The assembler just reports a warning on all forward branches that can be shortened.

You can optimize it by hand; saves two bytes.

Example:

```
...
beq.s    foo          ; shortened forward branch to foo
...
foo:
```

07+

07-

Backward referencing absolute long addressing modes will be optimized to PC-relative addressing modes if within the range of -32768-0 bytes.

Note that if the absolute long address size is given specifically no optimization will be made.

Example:

```
foo:    ...
        lea    foo,a0
        ; will be optimized to
        lea    foo(PC),a0
        ; and 2 bytes saved, but
        lea    (foo).L,a0
        ; won't be optimized
```

08+

08-

movea.l, adda.l, and suba.l instructions with an immediate value within the range of -32768-32767 will be shortened to word operation size (e.g. movea.w).

Example:

```
adda.l   #$100,a0
; will be shortened to
adda.w   #$100,a0
; and 2 bytes saved
```

09+

09-

cmp, cmpi, and cmpa instructions with an immediate source operand will be optimized to a tst instruction if the

immediate value is zero. This optimization saves either two or four bytes, depending on the used operation size.

Note that a `cmpa` instruction will only be optimized if code generation for a MC68020 or higher processor is enabled. Refer to MC68020 for more information on the processor selection.

013+

013-

A base displacement within the range of -32768-32767 will be optimized to word size.

Note that if the address size is given specifically to the base displacement (`.L`) no optimization will be made. See

Addressing Modes  
, for detailed information about the base displacement.

014+

014-

An outer displacement within the range of -32768-32767 will be optimized to word size.

Note that if the address size is given specifically to the outer displacement (`.L`) no optimization will be made. See

Addressing Modes  
, for detailed information about the outer displacement.

016+

016-

This is not really an optimization. The assembler reports a warning on all forward referencing absolute long addressing modes that can be made PC-relative.

You can optimize it by hand; two bytes can be saved.

017+

017-

This is not really an optimization. A warning will be displayed if an address register indirect with displacement addressing mode can be optimized to an address register indirect, if the forward referenced displacement is zero.

You can optimize it by hand; two bytes can be saved.

018+

018-

This is not really an optimization. The assembler reports a warning on all forward referenced absolute long addressing modes that can be shortened to absolute word.

Note that no warning is displayed for all absolute long addressings with an absolute address size specifier (`.L`).

---

You can optimize it by hand; two bytes can be saved.

- Enable all additional optimizations: OPT Q+
- Disable all additional optimizations (default): OPT Q-
- Enable additional optimization n: OPT Qn+
- Disable additional optimization n: OPT Qn-

Next to the set of the standard optimizations, ProAsm supports various additional optimizations. In the most cases these optimizations convert instructions into other instructions or set of instructions with the same result to gain a smaller and faster code. No optimizations are performed by default.

All optimizations performed will be issued by a warning. You can control these warnings using the QW+ or QW- options. If any optimization has been done the assembler will report the number of optimizations made and bytes saved at the end of the assembly.

You may specify single optimizations by inserting a number between the Q letter and the plus (+) or minus (-) sign. on+ will enable the optimization and on- is used to disable it.

Example:

```
OPT   q+           ; turns all additional optimizations on
OPT   q+,q1-,q7-  ; turn all optimizations on, except 1 and 7
OPT   q-,q3+,q2+  ; only optimizations 2 and 3 are turned on
OPT   q5+         ; allow the optimization of add and sub
```

The following specific optimizations are available:

Q1+

Q1-

Optimizes clr.l with a data register operand to a moveq #0,dn instruction. This optimization saves no bytes, but it is faster than the clear instruction.

Q2+

Q2-

An asl instruction will be optimized to an add instruction if a data register is arithmetically shifted by one bit. It optimizes also the roxl instruction to addx if a data register is rotated by one bit. The result of this optimization is a gain in time.

Example:

```
asl.w   #1,d3           ; => add.w d3,d3
roxl.l  #1,d4           ; => addx.l d4,d4
```

Q3+

Q3-

Optimizes a movea instruction with an immediate source operand of zero to suba. This optimization saves either two or four bytes, depending on the used operation size.

Example

```
movea.l #0,a4          ; => suba.l a4,a4
```

Q4+

Q4-

If this optimization is turned on any adda and suba instruction will be skipped by the assembler if the immediate source operand is zero:

```
suba.l #0,a4          ; => instruction skipped
adda.w #0,a2          ; => instruction skipped
```

An addition or subtraction of zero has no effect to the address register in the destination operand. This optimization saves either four or six bytes, depending on the used operation size.

Q5+

Q5-

A move.l instruction with an immediate source operand and a data register as destination will be optimized to a combination of a moveq and a not.b or not.w instruction.

The immediate value must be within the range of 128-255, 65408-65535, or -65409--65536. This optimization saves two bytes.

Example:

```
move.l #65409,d1
```

will be optimized to:

```
moveq #126,d1
not.w d1
```

Q7+

Q7-

A move.l instruction with an immediate source operand and a data register as destination will be optimized to a combination of a moveq and a subq.l instruction.

The immediate value must be within the range of -136--129. This optimization saves two bytes.

For example:

```
move.l #-132,d0
```

will be optimized to:

```
moveq #128,d0
subq.l #4,d0
```

Q8+

Q8-

A move.l instruction with an immediate source operand and a data register as destination will be optimized to a combination of a moveq and a swap instruction.

The immediate value must have the lower 16 bits set to zero

and be within the range of 65536-8323072 or -8323073--65537.  
This optimization saves two bytes.

Example:

```
move.l  #$10000,d6
```

will be optimized to:

```
moveq   #1,d6
swap    d6,d6
```

Q9+

Q9-

A `move.l` instruction with an immediate source operand and a data register as destination will be optimized to a combination of a `moveq` and a `add.b` instruction.

The immediate value must be even and within the range of 128-254 or -256--130. This optimization saves two bytes.

Example:

```
move.l  #200,d2
```

will be optimized to:

```
moveq   #100,d2
add.b   d2,d2
```

Q11+

Q11-

Optimizes a `move.b` instruction with an immediate source operand to a `st.b` instruction if the immediate value is -1. This optimization saves two bytes.

Example:

```
move.b  #-1,(a0)
```

will be optimized to:

```
st.b    (a0)
```

Caution, the `move.b #-1,<ea>` instruction affects the condition code flags, while the `ST.B <ea>` instruction does not. Refer to the 'M68000 Family Programmer's Reference Manual' for more details about these instructions.

Q12+

Q12-

`eor` and `eori` instruction with an immediate source operand will be optimized to a `not` instruction if the immediate value is zero. This optimization saves either two or four bytes, depending on the used operation size.

For example:

```
eori.w  #-1,foo_flag(a3)
```

will be optimized to:

```
not.w    foo_flag(a3)
```

- Enable multipass optimization: MULTIPASS

The MULTIPASS directive causes the assembler to use more than two passes for an assembly task if needed. This allows to gain an even more optimized code.

Due to the nature of a two pass assembler forward references cannot be optimized. Forward references are only known in the second pass and the assembler is thereby unable to generate an optimized code.

With the possibility to use more than two passes, unknown forward references vanish and multiple forward references "hopes" can be resolved. All these previously unknown references can now be taken into account by optimizations.

The assembler terminates a task, if no more passes have to be done or any circular references were found.

The termination criteria for a multipass assembly are fulfilled when no more optimization can be made and when no label had to be changed due an optimization in the last pass.

Circular references are symbol definitions that assign themselves to themselves:

```

a      EQU    a      ; a circular reference

a      EQU    b      ; another circular reference
b      EQU    c      ; spread over three symbols
c      EQU    a      ;
```

For all of the above written examples ProAsm reports an unresolved symbol found error if multi-passing is enabled.

The MULTIPASS directive will have no effect if never any optimizations were enabled (using the OPT O or OPT Q options or any synonym).

This directive has no negative form. By default, the assembler performs only two passes.

- enable the use of the optimizer library: OPT OPTIMLIB

- disable the use of the optimizer library: OPT NOOPTIMLIB

These options control the use of the optimization support library. This library can be used by ProAsm to recognize more and complex optimization possibilities.

The possible optimizations will be displayed. A short description informs the user about the saved space and whether the

optimization could be made or not. If it could not be made by the library, the way how the optimization can be achieved is described.

If the library could not be opened by OPT OPTIMLIB, a warning will be reported.

See The proasmoptim.library, for more information about the use of this library and its function.

## 1.50 pro.guide/Assembler Message Control

Generating Warnings and Errors

.....

Sometimes you may want to make your source code do some automatic "sanity checking" on macro parameters or other conditional assembly options. The following directives make this possible.

### - Generate an assembler error: FAIL message

This directive generates an error, causing the assembler to display this source line and refuse to produce an output file. For example, this macro generates an error if it is invoked with more or less than four arguments:

```
ListEntry  MACRO
            IFNE    NARG-4
            FAIL    ** wrong number of arguments passed!
            ELSE
            dc.w    \1,\2
            dc.l    \3,\4
            ENDC
            ENDM
```

### - Generate a warning message: WARN message

This is like FAIL, except it is not fatal and does not prevent the assembler from producing an output file; it only displays the source line containing the WARN directive. For example, this macro produces a warning if invoked without any parameters:

```
DataEntry  MACRO
            IFC     '\1',''
            WARN    ** no argument, using default value.
            dc.w    0
            ELSE
            dc.w    \1
            ENDC
            ENDM
```



You can use special backslash symbols such as `@{i}n` within FAIL and WARN messages; they will be expanded before the message is printed. This example demonstrates how it can be useful:

```
Entry      MACRO
           IFLT    \1,4
           FAIL    ** argument must be at least 4, was \1
           ELSE
           dc.w    \1
           ENDC
           ENDM
```

If this macro is called with the parameter 5, ProAsm will fail with this message:

```
FAIL      ** argument must be at least 4, was 5
```

## 1.51 pro.guide/Controlling the Message Output

- Specify maximum number of errors: FAILAT expression

The expression specifies the maximum number of errors to be reported before aborting assembly. An expression of zero sets the limit to unlimited number of errors. This can be used to prevent screenfuls of error message "spewage" from being dumped onto the display because of one minor error in the source code which cascades through and causes many other errors.

- Show errors messages (default): OPT E+

- Suppress error messages: OPT E-

This option enables or disables the report of error messages to the standard output and the error file. It may be useful when the assembler is used within a script file or something similar and the report of error messages would be disturbing.

By default, all errors are displayed.

- Show warning messages (default): OPT WARN

- : OPT W+

- Suppress warning messages: OPT NOWARN

- : OPT W-

Use this option if you wish to display or to suppress the warnings produced by the assembler. See

Warnings

, for detailed information on the warnings generated by ProAsm.

- Show all warnings caused by optimizations (default): OPT OW+
  - Suppress all warnings caused by optimizations: OPT OW-
  - show warnings caused by optimization n: OPT OWn+
  - suppress warnings caused by optimization n: OPT OWn-
- The first two forms are used to control the warnings produced by all optimizations.

You can also control the warnings produced by one particular type of optimization by using either OWn+ or OWn-. The number n is the number denoting the type of optimization; see OPT O, for more details.

- Show all warnings caused by additional optimizations (default): OPT QW+
  - Suppress all warnings caused by additional optimizations: OPT QW-
  - show warnings caused by additional optimization n: OPT QWn+
  - suppress warnings caused by additional optimization n: OPT QWn-
- The first two forms are used to control the warnings produced by all advanced optimizations.

You can also control the warnings produced by one particular type of advanced optimization by using either QWn+ or QWn-. The number n is the number denoting the type of optimization; see OPT Q, for more details.

- Disable misalignment and odd access checks: ODDOK
  - Enable checks: ODDERROR
- These directives can be used to control the check for misalignment and odd accesses.

Designing code for the MC68020 or higher microprocessors it is not always necessary to check for word alignment for data, since these processors can access data at any alignment. However, should the code be downwards compatible the right data alignment has to be ensured.

The ODDERROR directive tells the assembler to report any not word aligned code and data, and any non-byte accesses to odd addresses.

By default, the checks are enabled.

Example:

```

...
ODDOK
move.l  d0,DosBase      ; => ok
add.w   counter(pc),d1  ; => ok

ODDERROR
move.l  d0,DosBase      ; => error
add.w   counter(pc),d1  ; => error
...

```

```

flag:      dc.b    0
counter:   dc.w    2      ; counter is odd aligned => error
DosBase:   dc.l    0

```

- Disable misalignment and odd access checks: ODD2OK
- Enable checks: ODD2ERROR
  - These directives are extensions to the above described ODDOK and ODDERROR.

They enable or disable the same misalignment and odd access checks as the ODDOK and ODDERROR do. In addition the ODD2ERROR directive forces the assembler to report any odd displacement for the address register indirect with displacement and the address register indirect with index addressing modes (see

Addressing Modes  
)

Consider the following example:

```

ODD2ERROR

progstart: lea progstart(pc),a4 ; set program base in A4
           clr.l counter(a4)    ; => ok
           clr.b flag(a4)      ; => ok
           clr.w foo(a4)       ; => error
           ...

counter:   DC.L  0
flag:     DC.B  0
foo:      DC.W  0

```

The main goal of the additional checks are that possible odd accesses through the use of the address register indirect with displacement and address register indirect with index addressing modes can be spotted. But these checks are only useful if the address register is set to a 2-byte boundary. The assembler assumes that the address register holds an even address, since its contents is not known during assembly.

By default, these checks are disabled.

- Display filenames of all loaded files: OPT F+
- Do not display filenames (default): OPT F-
  - This option allows you to enable or disable the report of all names of the files loaded by the either INCLUDE, INCBIN, or one of their synonyms.

This messages will only be issued during the first pass.

- Show the preprocessed source line with messages: OPT Z+
- Show the original source line (default): OPT Z-

Whenever an error or a warning message is displayed, the assembler also displays the corresponding source code line. Using this option you can tell the assembler to display either the original written line or the preprocessed line, with EQUR and similar symbols expanded. It can be very useful to determine the actual cause of an error in the presence of elaborate textual symbols. The original source code line is shown as default.

The following example reports in both cases an undefined symbol error, but different source lines will be displayed.

```
foo    EQUR    D8

OPT    z-      ; display the original written source code line
move.l foo,d0 ; move.l foo,d0 is shown next to the error message

OPT    z+      ; show the assembled source code line
move.l foo,d0 ; move.l D8,d0 is written to the standard output
```

- Report error if bit number not in bounds: OPT CHKBIT
  - Report warning if bit number not in bounds: OPT WARNBIT
  - Do not check bit number (default): OPT NOCHKBIT
- These options tell the assembler to report either an error message, a warning message, or nothing at all, if the bit number specified in the immediate source operand of a bit manipulation instruction is not in bounds. The bit number must be either from 0 to 7 for byte operations or from 0 to 31 for long word operations.

Bit manipulating instructions are the bchg, bclr, bset, and btst instructions. When the destination is a data register, the M68000 processors automatically uses the bit number specified by the modulo 32 bit number. When the destination is a memory location, the operation is a byte operation, and any bit number is modulo 8.

Example:

```
OPT    warnbit
btst   #8,(a0)
; you get a warning from the assembler
OPT    chkbit
bset   #32,d0
; an error message will be displayed
OPT    nochkbit
bclr   #13,14(a4)
; nothing reported, bit 5 will be cleared
```

## 1.52 pro.guide/BASE

- Set base location: BASE expression

This directive forces the assembler to use a base for the address register indirect with displacement (indexed or not) addressing modes.

ProAsm allows the use of relocatable symbols as displacement for the above mentioned addressing modes. If no base is defined (as it is by default), the offset of the relocatable symbols are taken into account for the displacements. (What is exact the same as a base defined to the start of the current section, or a base of zero respectively.)

```
        move.l  foo(a4),d0
        ...
foo:
```

To assure now a correct effective address calculation, the assembler must know the base that is used. The example above works only if the address register a4 is assigned to the beginning of the current section. But what if the address register is set to another location within the program code (but still within the same section)? Moving the base that is hold in the base register (a4 in the example) to another location, allows us to access data in a wider range (from 0-32767 up to -32768-32767).

To tell the assembler the location of the new base you can apply the BASE directive, with the expression set to this new location. The effective address is corrected by subtracting the offset of the base location (expression) from the displacement of the mentioned addressing modes.

The expression describes the new location relative to the start of the current section. An expression of zero forbids the effective address correction. It must be a relocatable expression, absolute expressions are not allowed and an error is reported. (With the exception that zero (0) can be used to disable the use of bases.)

Examples:

```
BASE    progstart+4096
; Set base 4kbytes from program start,
; effective access range is -4096--32767.
```

```
BASE    progbase
; Using a special symbol for the base,
; you have the possibility to set it manually.
```

The advantage of using the BASE directive is that if you decide to change the base location, you only have to change the expression on the BASE directive instead of changing every displacement of a base relative access:

```
start:  ...
; traditional base relative access
lea     start(pc),a4      ; load a4 with the base address
move.l  foo-start(a4),d0 ; base relative access

; new method
BASE    start              ; set the base location to start
BASEREG a4                 ; use only a4 as base register
```

```

        lea    start(pc),a4      ; load a4 with the base address
        move.l foo(a4),d0      ; base relative access
        ...
foo:

```

The following example shows you two different methods to assign the base to the base register:

```

        BASE    progbase

        lea    progbase(pc),a4
        ; common way to set the register

        lea    __base(pc),a4
        ; setting the register using the __BASE symbol
        ...
progbase:

```

This example works only if the base is in the same section as the register assignment.

To handle section overlapping bases correctly, consider the following example:

```

        BASE    Data+32766
        SECTION "code",CODE
        ...
        lea    __base,a4
        ; set base register to the data section.
        ; the same as lea Data+32766,a4
        ...
        move.l d0,DosBase(a4)

        SECTION "data",DATA
Data:
        DosBase:    dc.l    1          ; mark start of data section
ImageData:

```

Note that a base movement done by the BASE directive affects the effective address calculation of all address register indirect with displacement addressing modes for all address registers (except a7). To restrict this correction to one or to a list of address registers, use the BASEREG directive (see its description below).

The \_\_BASE symbol contains the current base address, if any defined. See

Special Symbols

, for the description of the \_\_BASE

symbol. Be aware that when no base is defined this symbol contains the absolute value of zero. Assigning \_\_BASE to a address register in such a situation will load the address register with the value of zero (identical to lea 0,a4).

You can use the BASE directive more than once within a program code. For a temporary change of the base location the following mechanism can be used:

```

        oldBase EQU    __base      ; store current base
        BASE    newbase          ; set new base
        ...

```

```
BASE    oldbase    ; restore previously used base
```

Consider also the following example that uses the OPT T in combination with the BASE and BASEREG directives for a stronger typechecking:

```
BASE    progbase
BASEREG a4
OPT     T+          ; enable typechecking
progbase:
    ...
    lea    progbase(pc),a4
    move.l d0,foo(a4)
    move.l d1,bar(a5)    ; => ERROR
    ...
foo:    dc.l    0
bar:    dc.w    0
```

If in the example above the typechecking would be disabled, no error would be reported by the assembler.

By default, no base is defined.

- Forbid base correction: NOBASE
- : BASE 0

These directives forbid the effective address correction for all address register indirect with displacement (indexed or not) addressing modes.

By setting the base to zero (as NOBASE does) the base offset is resetted to its default, and effective address correction is turned off.

These directives can be used in combination with the BASE directive more than once in the program code if desired.

- Declare base register list mask: BASEREG address register list
- Using the BASEREG directive, a list of address registers (or a single address register) can be defined, that are accepted as base registers.

The address register list is a standard register list, as known from the MOVEM instruction, but only address registers are allowed.

If a BASE directive is used to define a new location for the program base, the displacements of all address register indirect with displacement addressing modes are corrected. Using the BASEREG directive you can select the base registers for which the correction is made.

For example:

```
BASEREG a4    ; select only a4 as base register
```

```

BASEREG a2/a4-a5 ; select a2, a4, and a5 as base registers

BASEREG a0-a7    ; select all registers

BASEREG a0-a6    ; select all registers form a0 to a6 (default)

```

By default, the address registers a0-a6 are accepted for displacement correction.

## 1.53 pro.guide/Absolute Assembly

- Set absolute origin: ORG expression

The ORG directive forces the assembler to enter absolute assembly mode. The location counter is set to the expression in the operand field.

This directive is normally used to set the starting address of ROM based code when the BINARY directive is used (see BINARY).

Note that ProAsm generates a warning when this directive is used if the location counter is unequal to zero. Using the ORG directive while generating executable or linkable object code is not recommended. The Amiga is a multitasking platform and therefore assumption about absolute memory addresses should not be made.

For example:

```

ORG      $F80000          ; set location counter to $F80000
...

ORG      $F80000-$10000  ; set location counter to $F70000
...

```

The location counter defaults to zero at the start of the assembly.

- Terminate absolute assembly mode: ENDORG

This directive can be used to terminate absolute assembly mode introduced by the ORG directive (even if more than one ORG directive was used).

Example:

```

        bsr      foobar
        moveq   #0,d0
        rts
        ...

foo:    ORG      $f70000          ; start absolute code with the
        ...                    ; location counter set to $F70000.
        ENDORG                    ; end absolute code

```



```
foobar: ...
```

- Re-define program origin: RORG expression

The RORG directive defines a new offset for the subsequent code. Expression is the new offset relative to the start of the current section, and it must not contain any forward references.

For example:

```
...
movem.l (a7)+,d1-a6
rts

RORG    1024
...
```

Should the result of the given expression be smaller than the current program counter, the assembler would have to overwrite already generated code. In such a case an error is reported.

## 1.54 pro.guide/Data Output Directives

- Define Bytes: label DC.B value [,value [,...]]

- : label DB value [,value [,...]]

Reserves memory for one or more bytes in the output file, and initializes them to the specified values. Numeric values are stored directly in single bytes. They can be either signed or unsigned, and must be in the range from -128 to 255. (The values -128 to -1 are equivalent to the values 128 to 255.) String operands can be included as well, enclosed in single or double quotes. They can be of arbitrary length, and are stored in order as sequences of bytes.

- Define Words: label DC.W value [,value [,...]]

- : label DW value [,value [,...]]

Reserves 16-bit words. Numeric operands must be in the range from -32768 to 65536. You can use quoted strings, but they must be only one or two characters long. Single quoted characters are zero-extended to 16 bits. (The high-order byte of the word is set to zero, and the low-order byte is set to the character specified.) Strings of two characters are placed in a single 16-bit word with the first character in the upper byte and the second character in the lower byte.

- Define Longwords: label DC.L value [,value [,...]]

- : label DL value [,value [,...]]

Reserves 32-bit long words. Numeric operands can be anywhere in the range -2147483648 to 4294966295. Quoted strings must be from one to four characters in length. They are packed into single longwords, with the last character occupying the least-significant byte in the longword. The longword is zero-extended to 32 bits if the string is less than four bytes in length.

In addition to constants, you can use relocatable values such as labels in DC.L directives, as long as the output file is in a relocatable or linkable format. Such values generate 32-bit relocations or 32-bit external references in the output file.

Here are some examples demonstrating the use of these directives:

```
DC.B      1,2,3,$23,-1,"hello",'world'
; => $01020323 ff68656C 6C6F776F 726C64

DC.W      1,2,3,$1234,-1,"a","b","cd"
; => $00010002 00031234 FFFF0061 00626364

DC.L      1,$1234,-1,"a","bcd",my_label
; => $00000001 00001234 ffffffff 00000061 00626364 (reloc32)
```

Note that if you want a string to be null-terminated (as most operating system functions and C programs expect), you must add the null byte at the end yourself, like this:

```
msg      DC.B      "This is a message.",0
; => $54686973 20697320 61206D65 73736167 652E00
```

## 1.55 pro.guide/Initialized Data with Restricted Range

- Define Unsigned Bytes: label UB value [,value [,...]]
- Define Unsigned Words: label UW value [,value [,...]]
- Define Unsigned Longwords: label UL value [,value [,...]]
 

These directives work like DB, DW, and DL, respectively, except that their operands are restricted to unsigned numbers only: using a negative number as a value generates an error. For example, UB accepts only numbers between 0 and 255, while DB accepts anything between -128 and 255. Note that these directives do not actually restrict the output data you can generate, because large unsigned numbers can always be used instead of negative numbers.
- Define Positive Bytes: label SB value [,value [,...]]
- Define Positive Words: label SW value [,value [,...]]
- Define Positive Longwords: label SL value [,value [,...]]
 

These directives are the converse of UB, UW, and UL; their operands are restricted to signed numbers only. For example, SB only accepts numbers between -128 and 127.

- Define Positive Bytes: label PB value [,value [,...]]
- Define Positive Words: label PW value [,value [,...]]
- Define Positive Longwords: label PL value [,value [,...]]
 

These directives work like UB, UW, and UL, except that their range is further restricted to only numbers whose binary representations have their high bit clear: signed positive numbers. For example, PB only accepts numbers between 0 and 127 instead of 0 and 255.
  
- Define Negative Bytes: label NB value [,value [,...]]
- Define Negative Words: label NW value [,value [,...]]
- Define Negative Longwords: label NL value [,value [,...]]
 

These directives are the converse of PB, PW, and PL: they accept negative numbers only. For example, NB only accepts numbers between -128 and -1.

Examples:

```

DB      -50      ; => $CE
UB      -50      ; => Error!
SB      -50      ; => $CE
PB      -50      ; => Error!
NB      -50      ; => $CE

DB      50       ; => $32
UB      50       ; => $32
SB      50       ; => $32
PB      50       ; => $32
NB      50       ; => Error!

DB      150      ; => $96
UB      150      ; => $96
SB      150      ; => Error!
PB      150      ; => Error!
NB      150      ; => Error!

```

## 1.56 pro.guide/Declaring Data Blocks

- define storage: [ label ] DS.size length [,expression ]
- define constant block: [ label ] DCB.size length [,expression ]
- : [ label ] BLK.size length [,expression ]

These directives can be used to reserve memory space within the object code. The given expression is stored in memory length times with the size of size. The length must be an absolute integer expression. The expression can be an expression of any type, unless the given size contradicts with expression (as in DS.B 4,-1.34E14). If no expression is given a value of zero is used.

The size specifier must be one of the following:

For example:

```

                ALIGN.L
fileinfo:      DS.B   fib_SIZEOF,0 ; dos/dos.i
is the same as:

```

```

                ALIGN.L
fileinfo:      DS.B   fib_SIZEOF      ; dos/dos.i

```

These directives are used within BSS sections to declare storage. If such a directive is found within a BSS section with an expression unequal to zero, a warning is reported.

Defining storage in a BSS section can look like this:

```

                SECTION "bss",BSS
DosBase DS.L   1
GfxBase DS.L   1,0
IntBase BLK.L  1
Flag    DC.B   0
                ALIGN.W
Image   DS.B   1024,0

```

A special form of these directives will align the program counter to an even boundary by using DS.W 0, likewise, DS.L 0 will align the program counter to a longword boundary, and so on for all remaining sizes. See

Alignment Padding  
, for more information.

## 1.57 pro.guide/Uninitialized Data Blocks

- Define uninitialized extra storage: [ label ] DX.size length ←  
[ , 0 ]

This directive can be used to reserve memory space within the object code. The assembler reserves space in the program code depending on the given size specifier size and length (length is the number of data items of the size size).

The length must be an absolute integer expression, and the size specifier must be one of the following:

Memory blocks defined by the DX directive must be at the end of a section (unless within a BSS section) and no other code or data generating directive is allowed after a DX directive.

Unlike the data block declaring directives (see

Declaring Data Blocks

), the DX directive does no memory initialization. If your program requires the memory to be cleared then you must write a small routine to clear that memory block (as a part of the initialization of your program). For example:

```

...
lea    DXStart(pc),a1
move.w #DXEnd-DXStart-1,d7

```

```

.clrdx:  clr.b    (al)+
         dbra    d7,.clrdx
         ...

DXStart:
DosBase: DX.L    1                ; allocate 4 bytes
GfxBase: DX.L    1
ArgStr:  DX.L    1
ArgLen:  DX.B    4                ; allocate 4 bytes

foo:     DX.X    1                ; allocate 12 bytes
bar:     DX.D    3                ; allocate 24 bytes
DXEnd:
        END

```

The advantage of the DX directive is that the program code can be made smaller and, as a consequence, will be loaded faster by the AmigaDOS loader.

## 1.58 pro.guide/Defining Strings

- Define a null-terminated string: [ label ] CSTRING string
  - : [ label ] CSTR string
- This directive is an extension to the DC.B directive. It works similar to the DC.B directive except that an additional null byte is appended after the string. The result is a null-terminated (C-style) string as used for strings in the programming language C.

Consider the following example:

```

CSTRING "Hello World"
; => DC.B "Hello World",0

CSTRING $9b,"1mHello World",$9b,"0m"
; => DC.B $9b,"1mHello World",$9b,"0m",0

```

- Define an OS-9 string: [ label ] ISTRING string
  - : [ label ] ISTR string
- This directive is an extension to the DC.B directive. It works similar to the DC.B directive except that bit 7 of the last byte in the given string is inverted.

For example:

```

ISTRING "error occurred"      ; => DC.B "error occure","d"|$80

```

Since the bit 7 is inverted, characters with an ASCII value higher than 127 can also be handled.

This directive has the advantage that a string intensive program can save a lot of space by using it instead the CSTRING directive

(for example).

It is easy to write string handling routines that can detect the end of such a string. For example a simple string copy routine that just copies the string:

```

;
; A0: pointer to the source i-string
; A1: place the string is copied to
;
copy_string:
    move.b  (a0)+,(a1)+    ; copy character
    bpl.s  copy_string    ; until a negative byte detected (bit 7 ←
        set),
    bchg   #7,-1(a1)      ; invert bit 7 of the last copied byte.
    rts

```

Note that the small routine above does not handle ASCII values greater than 127 (they would be seen as the end of the string).

Within your program it can look like that:

```

...
    lea    title(pc),a0
    lea    buffer,a1
    bsr    copy_string

...
title:  ISTRING "Example Copy_String v1.0"

```

- Define a BCPL string: [ label ] PSTRING string
  - : [ label ] PSTR string
- This directive is an extension to the DC.B directive. It works similar to the DC.B directive except that an additional byte is added in front of the string that contains the length of the given string.

Due to the fact that the length is stored as byte, the string may not be longer than 255 characters.

For example:

```
PSTRING "BCPL Strings" ; => DC.B 12,"BCPL Strings"
```

So defined strings (BCPL strings) were used by the AmigaDOS (up to OS1.3).

## 1.59 pro.guide/Alignment Padding

### Alignment Padding

---

It is often necessary or desirable for certain data structures or other constructs to have certain alignment properties, such as being aligned on a 2-byte or 4-byte boundary. For one thing, all code for

---

the M68000 family has to obey certain alignment conventions:

- \* All machine instructions must start on even-byte boundaries. That should not be a problem unless byte sized data and code get mixed. For instance:
 

```

bra.s    foo
DC.B    "Hello world!",0    ; 13 bytes
foo:    moveq    #0,d0        ; this instructions lies on an uneven ←
        boundary
      
```

The assembler reports a odd address error if an instruction is located at an odd address. To ensure that the program counter is forced to an even address use the EVEN directive or one of its alternate forms:

```

bra.s    foo
DC.B    "Hello world!",0    ; 13 bytes.
EVEN                                ; ensure word boundary!
foo:    moveq    #0,d0
      
```

- \* When working with word or long word sized data you must ensure that they are on an even address. (This is not necessarily for the MC68020 or higher microprocessors, since they can access data at any alignment, but it is recommended for downwards compatibility and for better performance). The assembler gives an odd address error message if an instruction that would access a misaligned word- or longword-sized data item is detected:

```

move.l   myData,d0
...
dc.b    0
myData: dc.l   5                ; myData is misaligned
      
```

Data structures in the Amiga Operating System must be word aligned, except for a few structures that need to lie on a long word boundary. (For detailed information on the data structures of the AmigaOS refer to 'AMIGA ROM Kernel Reference Manual: Includes and Autodocs').

The start of a new section is always automatically long word aligned.

## 1.60 pro.guide/ALIGN

- Align program counter: CNOP expression [,expression]
  - : ALIGN expression [,expression]
- These directives allow you to align code or data to any specified boundary.

If only one argument is supplied with these directives then it is the desired boundary. When both arguments are supplied the first expression is the offset beyond this alignment and the second expression is the desired alignment.

The program counter is padded to the desired boundary by inserting

the required number of null-bytes if necessary. The offset (given by the first expression) will then be added to the program counter and filled with null-bytes.

Example:

```

CNOP    2      ; aligns program counter to word boundary
CNOP    0,4    ; aligns program counter to long word boundary
CNOP    2,4    ; aligns PC to long word boundary and adds 2 ←
             bytes
CNOP    13,24  ; aligns PC to the next 24-byte boundary
             ; and adds 13 bytes

        CNOP    6,2
        ; is similar to
        EVEN
        DS.B    6

```

- Align program counter: ALIGN.size

The ALIGN directive supplied with an operation size is an alternate form for the CNOP directive described above.

The program counter is padded to the desired boundary as specified by size. Null-bytes will be inserted as filler if necessary.

The following operation sizes are allowed for size:

```

.W
    word, align the program counter to word boundary. (equal to
    CNOP 0,2.)

.L
    long word, align program counter to long word boundary.
    (equal to CNOP 0,4.)

.S
    single precision, align program counter to long word boundary.
    (equal to CNOP 0,4.)

.D
    double precision, align the program counter to 64-bit
    boundary. (equal to CNOP 0,8.)

.X
    extended precision, align the program counter to 96-bit
    boundary. (equal to CNOP 0,12.)

.P
    packed, align the program counter to 96-bit boundary. (equal
    to CNOP 0,12.)

.Q
    quad word, align program counter to 64-bit boundary. (equal
    to CNOP 0,8.)

```

---



- Align output to word boundary: EVEN
- : DS.W 0[,pad-value]  
This directive pads the output data to the next word boundary (even address), by inserting a pad byte if necessary. If EVEN or DS.W is used with only one parameter, the pad byte will be zero. If DS.W is used with two parameters, the second parameter will be used as the pad byte. If the output data is already on the desired boundary, the directive will have no effect.

Example:

```
ORG $100
a:      ; at $100
dc.b   "A"
b:      ; at $101
EVEN   ; generates a pad byte
c:      ; at $102
dc.b   "B"
```

- Align program counter to long word boundary: QUAD  
This directive pads the output to the next long word boundary, by inserting the required number of null pad bytes. If the program counter is already on the desired boundary, the directive will have no effect.
- Align output to odd byte boundary: ODD  
The ODD directive forces the assembler to align the output to the next odd byte address (the next address with bit 0 equal to 1). This directive will have no effect if the output is already on the desired boundary.
- Align program counter: CCNOP expression[,expression]  
This directive allows you to align code or data to any specified boundary. It is functionally identical to the CNOP directive above. The only difference is that the program counter is padded to the desired boundary by inserting NOP's (\$4e71) instead of null-bytes. This directive is specially intended to align program code to the desired boundary.

If only one argument is supplied with these directives then it is the desired boundary. When both arguments are supplied the first expression is the offset beyond this alignment and the second expression is the desired alignment.

Example:

```
moveq   #14,d7
```

---

```

        CCNOP    0,4
loop:   ...
        dbra    d7,loop

```

A null-byte will still be inserted if the pad length needed to obtain the desired alignment is odd. Consider the following example:

```

        DC.B    0
        CCNOP    0,4
        rts

```

The code lines above is identical to:

```

        DC.B    0
        DC.B    0
        ; insert a null-byte filler
        ; to obtain an even alignment
        ; for the NOP instruction
        NOP
        NOP
        rts

```

A odd address or offset detected warning will be reported if the offset (given by the first expression) or the current program counter is odd.

- Align `__RS` offset counter: `ALIGNRS expression[,expression]`
- Align `__SO` offset counter: `ALIGNSO expression[,expression]`
- Align `__FO` offset counter: `ALIGNFO expression[,expression]`
- Align DX area: `ALIGNDX expression[,expression]`

These directives are in their function identical to the `CNOP` directive described above except that they are intended for the use with structure, frame offset or DX directives. (See

```

        Structure Offsets
        , or
        Uninitialized Data Blocks
        , for more
information.)

```

No null-bytes are inserted.

Example:

```

        RSRESET
foo     RS.B    1
        ALIGNRS 0,2    ; align __RS counter to word boundary
bar     RS.L    1

```

Note that any of these directives will only work with its counter. Using `ALIGNRS` instead of `ALIGNSO` won't work.

- Align `__RS` offset counter: `ALIGNRS.size`
  - Align `__SO` offset counter: `ALIGNSO.size`
-

- Align \_\_FO offset counter: ALIGNFO.size
  - Align DX area: ALIGNDX.size
- These directives work in the same manner as the ALIGN.size directive described above except that they are intended for the use with the structure offset, the frame offset or the DX directives. (See
- Structure Offsets  
, or  
Uninitialized Data Blocks  
,
- for more information.)

No null-bytes are inserted.

Example:

```
foo      DX.B      3
         ALIGNDX.W      ; align DX counter to word boundary
bar      DX.L      1
```

Note that any of these directives will only work with its counter.

## 1.61 pro.guide/Convenience Pseudo-Opcodes

### Convenience Pseudo-Opcodes

=====

Pseudo-Opcodes are special assembler statements that generate object code but do not correspond to actual M68000 family instructions. These Pseudo-Opcodes are merely implemented for a programmer's sake. The way they are named or the functions they fulfill, are all done to make the life of a programmer easier.

It is not guaranteed that all assemblers support Pseudo-Opcodes or that they are compatible to others. ProAsm understands the most commonly used Pseudo-Opcodes, and in addition it comes with some new ones.

You find the descriptions of all Pseudo-Opcodes known by ProAsm in the following sections.

#### Stack Manipulation

-----

- Move ea on stack: PUSH.size ea
  - Move ea from stack: POP.size ea
- These pseudo-opcodes are abbreviations for MOVE.size instructions when dealing with the stack.

PUSH moves an ea on the stack similar to the MOVE ea,-(SP).

POP is the opposite form of PUSH. It can be used if an ea is

moved from the stack (MOVE (a7)+,ea).

Example:

```
POP.W    (a0)
; => move.w (a0),-(sp)

PUSH.L   (foo,a3,d0.w)
; => move.l (a7)-,(foo,a3,d0.w)
```

Valid sizes are byte, word, and longword. But note, when you use byte as operation size, the stack pointer will automatically align the stack to the next word boundary downwards before the byte is written on the stack.

- Move register list on stack: PUSHM register list
  - Move register list from stack: POPM register list
- These pseudo-ops are alternatives for the MOVEM.L instructions when dealing with the stack.

The PUSHM pseudo-opcode moves the given register list on the stack, and POPM removes the registers from the stack.

The register list is identical to the one used by the MOVEM instruction. The used operation size is longword.

The `_MOVEMREGS`, `_MOVEMBYTES`, and `_MOVEMLIST` symbols are updated if one of these pseudo-ops is used. See

Special Symbols  
, for

more detailed information about these three symbols.

For example:

```
PUSHM    d0-d4/d7/a2-a6
; => movem.l d0-d4/d7/a2-a6,-(a7)

POPM     d0-a6
; => movem.l (a7)+,d0-a6
```

- Push automatically generated register list on stack: APUSHM
- Pop automatically generated register list from stack: APOPM  
[register list ]

These pseudo-ops are special variations of the above described PUSHM and POPM.

They generate automatically a register list of all used registers between an APUSHM and a matching APOPM. The APUSHM will then be replaced by a MOVEM.L instruction that moves these registers on the stack. APOPM will be replaced by a MOVEM.L that moves these registers from the stack.

The optionally given register list (as argument of APOPM)

describes the registers that are excluded from the generated register list. This register list is identical to the one used by the MOVEM instruction. The used operation size is longword. Consider the following example:

```

APUSHM
move.l  d0,a0
move.w  (a0),d1
neg.w   d1
move.l  (a0,d1.w),d0
APOP   d0           ; exclude d0 from the generated register ←
list.

```

will be assembled to:

```

movem.l d1/a0,-(a7)
move.l  d0,a0
move.w  (a0),d1
neg.w   d1
move.l  (a0,d1.w),d0
movem.l (a7)+,d1/a0

```

The `_MOVEMREGS`, `_MOVEMBYTES`, and `_MOVEMLIST` symbols are updated if one of these pseudo-opcodes is used. See

Special Symbols  
, for

more detailed information about these three symbols.

## Move Effective Address

---

- Move effective address: MEA eal,ea2

This is a special pseudo-opcode that is implemented to make program counter relative assembly easier.

It moves the effective address of the first operand (eal) to ea2 using the stack. MEA is splitted into a PEA eal and a MOVE.L (a7)+,ea2 instruction.

For example:

```

MEA     foo(pc),d0

```

will be assembled to:

```

pea     foo(pc)
move.l  (a7)+,d0

```

If the destination (ea2) is an address register or an address register indirect with predecrement with the stack pointer as register (`-(A7)`), the assembler automatically optimizes the MEA pseudo-opcode to a LEA eal,varea2 instruction or a PEA eal instruction respectively:

```

MEA     foo(pc),a2           ; => LEA foo(pc),a2
MEA     foo(pc),-(a7)       ; => PEA foo(pc)

```

The advantage of using the MEA pseudo-opcode lies in the

---

possibility to avoid relocatable references. This is useful for writing program counter relative code. A `move.l #foo,d0` instruction can simply be replaced by `MEA foo(pc),d0` (as far as `foo` is a relocatable symbol).

#### The PFLUSHA instruction

-----

Both the MC68030 and MC68040 processors have an instruction called PFLUSHA. Unfortunately, these instructions have different encodings on the two processors. Therefore, when ProAsm sees a PFLUSHA instruction, it must somehow be told what kind of PFLUSHA instruction to write.

- PFLUSHA instruction: PFLUSHA operands  
If either the MC68030 or MC68040 directive is in effect, ProAsm uses its setting to control the encoding of PFLUSHA. If MCRELAX is in use, ProAsm always generates the MC68030 form.

This is an instruction of the M68000 family, refer to the 'M68000 Family Programmer's Reference Manual' for more information about the PFLUSHA instruction.

- Pseudo-Opcode for the MC68030 PFLUSHA: PFLUSHA30 operands
- Pseudo-Opcode for the MC68040 PFLUSHA: PFLUSHA40 operands  
To allow both forms of PFLUSHA to be produced while using MCRELAX mode, ProAsm also provides these two alternate instruction forms, which always produce the MC68030 or MC68040 forms, respectively.

## 1.62 pro.guide/Controlling the Output File

### Output File Name

-----

- Set assembly output file: OUTPUT filename
- : OBJFILE filename  
This directive is used to specify the filename for the output file. This filename can be overridden by the '-O'/OBJNAME command line option. For example, this line tells ProAsm to place the assembled output into the file `ram:test`:

```
OUTPUT 'ram:test'
```

If more than one OUTPUT directive is present, the first one will define the output name; any others that specify a different filename will generate a warning.

- Set name of linkable object file: OBJ filename  
This directive is equivalent to the following:

```
OUTPUT filename
LINKABLE
```

In other words, it sets the output filename and instructs ProAsm to make it a linkable AmigaDOS object file (see Output File Format).

#### Output File Format

-----

By default, ProAsm creates an AmigaDOS executable.

- Generate an AmigaDOS executable: EXECUTABLE
- : EXE
- : EXEOBJ
- : OPT L-

Any of these directives instruct ProAsm that the output file should be an AmigaDOS executable, ready to load and run from the command line or Workbench. The program may not reference any external symbols, since it will not be linked with anything, and it may not use ORG, because AmigaDOS may load the program at any place in memory. See Executables, for more information about AmigaDOS executables.

- Generate a linkable object module: LINKABLE
- : LINKOBJ
- : OPT L+

These directives instruct ProAsm to create an object file which can be linked with other object files by a linker such as BLink, eventually resulting in an AmigaDOS executable. This is one way to write big programs spread over several source files, although this can also be done using include files (see

Include Files  
)

More importantly, this allows you to link assembly language programs assembled by ProAsm with object files generated by a high-level language compiler, such as a C compiler. See

Object Modules  
, for more information about linkable object modules.

- Generate a binary image: BINARY
- : BINARYONLY
- : BINRYONLY
- : ASEG

These directives instruct ProAsm to output a "raw" binary file,

---

containing only the data explicitly defined in the source code. No linking or debug information can be generated, so XREF may not be used. Labels may not be referenced in 32-bit absolute modes unless ORG is used to specify the precise address at which the image is going to be run (see ORG).

- Generate a preassembled file: PREASM
- : OPT GENSYM

The PREASM directive tells the assembler to generate a preassembled file. Use the OUTPUT or the OBJFILE directive to name the resulting file.

A preassembled file contains specially formatted symbol and macro lists that can immediately be added to the symbol and macro tables without further parsing of directives and expressions, as must be done with non-preassembled include file. The use of files produced using the PREASM directive will result in faster assembly times and less memory usage.

Next to these tables a list of all included files is likewise stored in a preassembled file. The assembler checks all new include files for matches in this list before including them. Only files that do not match will be loaded since all the others are already included in the preasm file.

The following symbol types are stored in the preasm file along with the macro definitions and include file names:

Absolute symbols

(declared with either EQU or SET)

External references

(declared with XREF)

Relocatable symbols

Symbols assigned register lists

Symbols that are defined as synonyms for other symbols

Symbols equated to FPU constants

Symbols assigned to FPU register lists

Temporary symbols

Textual symbols

The following directives and their synonyms will be ignored if used with the PREASM directive: ADDSYM, CSYMFMT, DEBUG, OPT DEBUG, OPT HCLN, OPT X+, SECSYM.

You will get a warning if any code or data producing statements, the XDEF directive, or section statements are used together with the PREASM directive, since they are meaningless in this context.

A preasm file can later be loaded using one of the following directives: INCLUDE, INCEQU, MACLIB, HEADER.



- Suppress output: NOOBJ  
This directive forces the assembler not to create any output file at all. It is useful if you want to run a test assembly to check for syntax errors in the source code.  
  
It is the same as the NOOBJ and -N command line arguments.
- Generate Motorola S-record output: SREC format,record length,address[,name]  
This directive instructs ProAsm to generate a Motorola S-record output file. The S-record format is for encoding programs or data files in a portable format for transportation between computer systems. It is often used for writing embedded M68000 applications. For more information on the S-record format, refer to the 'M68000 Family Programmer's Reference Manual' (see

Bibliography  
)

format is the Motorola Record format type and must be a value from 1 to 3.

record length is the count of character pairs in the record, excluding the type and record length; the value must be from 16 to 255.

address is the load address. ProAsm sets the location counter offset to this address (see ORG).

name can be used to specify the module name. A module name that contains a whitespace must be enclosed in double or single quotes. If no name is given, ProAsm will be used.

Example:

```
SREC    2,64,$f20000,"foobar"
```

If several directives, controlling the output file format, appear in the source code, only the last one is used.

## 1.63 pro.guide/Sections

Sections

.....

When building AmigaDOS executables, it is often desirable to divide a program up into a few separate sections or hunks. Each section is a contiguous block of memory which has certain attributes. All of the code or data within a particular section is guaranteed to remain in the order in which you defined it, but different sections can be loaded anywhere in memory by AmigaDOS. This is known as scatter-loading. The following directives are used to define sections in ProAsm:

- Begin section: SECTION name[,type[,memtype[,typecode]]][,  
reloctype ]

This directive ends any previous section and starts adding code and data to a new section. The name is a string up to 256 characters long. It must be quoted if it contains special characters such as whitespace or commas. It can be anything you want, but code sections are usually called code or text, while data and bss sections are usually called data or `__MERGED`. Section names are case-sensitive.

The type may be CODE, DATA, or BSS. (They do not need to be written in uppercase, however.) Any data defined in a BSS section, such as data produced by DC statements, is lost: only its size is remembered, and AmigaDOS initializes the entire section to zero when the program is loaded. The default section type is CODE.

If the memtype is present, it specifies that the section must be loaded in a certain type of memory. It can be either CHIP, FAST, PUBLIC, MEMF, or ADVISORY. If memtype is MEMF, then the additional typecode argument must be supplied, which is a numeric value to be passed to the operating system's memory allocator (AllocMem()) when loading this section. This is usually some combination of the MEMF\_ values defined in the include file exec/memory.i.

Programs assembled with the MEMF option can only be loaded on Kickstart 2.0 or later, and object files which use this feature may not be compatible with some linkers. Therefore, we recommend that you do not use the MEMF option if you are only specifying chip or fast memory; use CHIP or FAST instead.

The memtype ADVISORY is only supported by AmigaDOS 3.0 (V39) or later. A hunk marked with ADVISORY will be ignored by the loader if its type is not understood. When ignored, the marked hunk is treated like a debug hunk. Under AmigaDOS versions prior to 3.0 executables marked with ADVISORY will fail to load with a bad hunk type error.

You can combine the type and the memtype by appending an underscore (\_) and the first letter of the memtype to the type. For example, CODE\_C is the same as CODE,CHIP.

The reloctype, if specified, can be RELOC32 or RELOC32SHORT. RELOC32 is the default setting, and generates a standard relocation hunk (hunk\_reloc) in the output. RELOC32SHORT generates a hunk\_reloc32short chunk if possible to save some space in the output file and make loading slightly faster. The hunk\_reloc32short chunk is functionally identical to hunk\_reloc. It is only more compact and applicable only to smaller programs. Also, this chunk type is only supported under AmigaDOS 2.0 or later, so it should not be used in programs that must be able to run under earlier versions of AmigaDOS.

If a previous section was defined earlier with the same name as

this section, ProAsm continues adding code and data to that section instead of creating a new section. The type and memtype may not conflict with the previous definition. If they are not specified at all, they are assumed to be the same. Case is always significant when comparing section names.

If any code or data is defined before the first SECTION directive (or one of its equivalents below), this data will be placed in a default code section, as if CODE had been included at the very beginning of the file.

Some examples:

```
SECTION "foo",code,public,reloc32short
; same as SECTION "foo",code,,reloc32short
...
SECTION "bar",data,chip
...
SECTION "foofoo",data,memf,(MEMF_REVERSE|MEMF_PUBLIC)
...
SECTION "foobar",bss
...
```

- Begin CODE section: CODE [memtype[,typecode]][,reloctype]
- : CODE name[,memtype][,reloctype]
- Begin DATA section: DATA [memtype[,typecode]][,reloctype]
- : DATA name[,memtype[,typecode]][,reloctype]
- Begin BSS section: BSS [memtype[,typecode]][,reloctype]
- : BSS name[,memtype[,typecode]][,reloctype]
- Begin CODE section: CSEG [memtype[,typecode]][,reloctype]
- : CSEG name[,memtype[,typecode]][,reloctype]
- Begin DATA section: DSEG [memtype[,typecode]][,reloctype]
- : DSEG name[,memtype[,typecode]][,reloctype]

These are short forms of the SECTION directive, which begin new, facultatively unnamed, sections of type CODE, DATA, and BSS, respectively, optionally with a specified memtype. Since the name is optional, names that are similar to memtype keywords are invalid (CHIP, FAST, PUBLIC, MEMF, and ADVISORY).

If MEMF is given as memtype an additional argument must be supplied with the directive which is a numeric value to be passed to the operating system's memory allocator (AllocMem()) when loading this section. This is usually some combination of the MEMF\_ values defined in the include file exec/memory.i.

The reloctype, if specified, can be RELOC32 or RELOC32SHORT. RELOC32 is the default setting, and generates a standard relocation hunks (hunk\_reloc) in the output. RELOC32SHORT generates a hunk\_reloc32short chunk if possible to save some space in the output file and make loading slightly faster. The hunk\_reloc32short chunk is functionally identical to hunk\_reloc. It is only more compact and applicable only to smaller programs. Also, this chunk type is only supported under AmigaDOS 2.0 or later, so it should not be used in programs that must be able to

run under earlier versions of AmigaDOS.

If a previous section was defined earlier with the same name (or no name) as this section, ProAsm continues adding code and data to that section instead of creating a new section. The type and memtype may not conflict with the previous definition. If they are not specified at all, then they are assumed to be the same. Case is always significant when comparing section names.

Refer to SECTION, for detailed information about defining sections.

Some Examples:

```
CODE    "Function",,reloc32short
; same as CODE "Function",,reloc32short
; and SECTION "Function",code,,reloc32short
...
DATA    "Images",chip
...
CODE
...
BSS
...
BSS     memf,MEMF_CLEAR
...
```

- Revert to the previous section: SECTION \_\_OLDSECTION

This directive reverts to the section that was in effect before the most recent SECTION (or equivalent) directive. This can be used in include files, for example, to temporarily switch to a different section and then later switch back to the section that was in use by the "parent" source file. This feature does not stack, however; additional SECTION \_\_OLDSECTION directives merely toggle between the two most recent sections.

The keyword \_\_OLDSECTION is case-insensitive.

Example:

```
moveq   #0,d0
SECTION 'lolo',code           ; Section: lolo
nop
SECTION __OldSection         ; Section: default (CODE),
rts                               ; where moveq is
SECTION __OldSection         ; Section: lolo
rts
END
```

I implemented it for an easy code/data handling for something like 'runtime generated' structures:

```
MyImage MACRO
    section "GUI_Structures_Images",data,chip
\1:
```

```

dc.l    \2,\3
dc.b    \4,0
even
incbin  "dh1:asm/images/\5"
...
section __OldSection
ENDM

```

Now within the code you may have a more structured ``layout``:

```

*
* Handle Image #1
*
HandleMyImage1:
    jsr    xxx(a6)
    ...
    rts

MyImage imagel,$14,0,"Add Font",FontImage.iff

```

At the absolute minimum, a program really only needs one section containing all the program's code and data. However, it is often considered good practice to divide a program into at least two separate sections: a CODE section containing instructions and a DATA section containing the program's data. While AmigaDOS does not currently require this, some other operating systems, such as Unix, do and AmigaDOS may will in the future. In addition to the code and data sections, a "bss" section is often used to hold variables that do not need to be initialized to explicit values on startup, conserving space in the executable file.

If your program contains graphics, sampled sounds, copper lists, or other data that must be accessed directly by the Amiga's custom chips, remember to place this data into a CHIP section. Otherwise, the program will not work on any machine with fast memory. It is permissible to put all your code and data into one CHIP section. However, this may make your program run more slowly, since machine code runs faster from FAST memory.

## 1.64 pro.guide/SMALLOBJ

- Merge all hunks into one: SMALLOBJ
- Disable merging: NORMOBJ

The SMALLOBJ directive tells the assembler to merge all hunks into one single CODE hunk. The default is to keep individual chunks separate in the output. NORMOBJ can be used to counteract SMALLOBJ.

- Merge all code hunks into one: SMALLCODE

- Merge all data hunks into one: SMALLDATA
  - Merge all bss hunks into one: SMALLBSS
- These directives force the assembler to merge all hunks of the specified type (code, data or bss) into one single hunk. The default is to keep individual chunks separate in the output. There is no negative form to empower merging.

## 1.65 pro.guide/Debugging Information

### Debugging Information

.....

AmigaDOS executables and object modules have the capability to store debug information of various types, which are not necessary for actually running the program, but can be used by debuggers to make debugging the program easier. You can instruct ProAsm to attach debug information to its output files with the DEBUG directive.

- Output debugging information: DEBUG [keyword[,keyword...]]
- This directive instructs ProAsm to output debugging information, and specifies exactly what kind of debugging information to generate.

The keywords are used to specify the kind of information to be produced.

The following modes specify the debugging output format. If none of them is used, it defaults to LINE or the last specified:

#### LINE

produces line number information (SAS/C compatible debug hunk format).

#### HCLN

produces compressed line number information (SAS/C compatible debug hunk format).

#### FULL

is an alias for LINE,ALL.

#### OFF

turns off the debugging information generation.

The types below are used to specify the type of debugging information that is to be produced. If none is used, it defaults to CODE,DATA or the last specification:

#### CODE

generates debugging information for the M68000 instruction set.

#### NOCODE

is the complement of the CODE keyword.

**DATA**

generates debugging information for the data producing directives.

**NODATA**

is the complement of the DATA keyword.

**ALL**

generates debugging information for all included files.

**ADDSYM**

adds symbol information to all hunks of an executable or a linkable output file, see ADDSYM.

**NOSYM**

is the complement of the ADDSYM keyword, see NOSYM.

If you just specify DEBUG with no operand, it defaults to DEBUG LINE, CODE, DATA or the last set specification.

The DEBUG directive will generate a warning if used for neither an executable nor a linkable output file.

```
DEBUG   HCLN
DEBUG   LINE, CODE, NODATA, ALL
DEBUG
```

- Enable debugging information: OPT DEBUG
  - Disable debugging information: OPT NODEBUG  
These directives enable or disable debugging information without changing the options specified with DEBUG. You can use these directives to output debugging information selectively for only certain parts of your program.
  
  - Enable compressed debugging information: OPT HCLN
  - Disable compressed debugging information: OPT NOHCLN  
These directives enable or disable compressed debugging information. The options specified with the DEBUG directive may be changed. This is a more efficient way of encoding debugging information in a file since it reduces the size of the output file.
  
  - You can use these options to output debugging information selectively for only certain parts of your program.
  
  - Generate symbol information: ADDSYM
  - : OPT D+
  - Disable symbol information: NOSYM
-

- : OPT D-  
The ADDSYM directive tells the assembler to add symbol information to all hunks of an executable or a linkable output file. Only global labels representing a relocatable location within the hunk will be added to the symbol information.

The ADDSYM directive will be ignored if neither an executable nor a linkable output file is generated.

The produced symbol information can be used by any symbolic debugger which supports the AmigaDOS symbol hunk format. Note that this option mutually excludes the x+ option.

- Generate symbol information for exported symbols only: OPT XDEBUG
- : OPT X+
- Suppress symbol information (default): OPT NOXDEBUG
- : OPT X-  
These options are equivalent to the above, except that they only control symbol information generation for exported (XDEF) symbols (see XDEF). It is only useful if linkable code is produced. For executables, symbol information for all symbols will be generated, as if ADDSYM was used.

- Generate symbol information for current section: SECSYM  
This directive is a special form of the above mentioned ADDSYM directive. The SECSYM forces the assembler to add symbol information to the executable or linkable output file only for the current section (while ADDSYM adds symbol information to all hunks). Only global labels representing a relocatable location within the hunk will be added to the symbol information.

Using this directive you can choose the sections from whom symbol information should occur in the output file. The SECSYM is normally used in the following way:

```
SECTION "code",CODE      ; add symbol information only
SECSYM                  ; for the code section
...
SECTION "foo",DATA
...
```

The SECSYM directive will be ignored if neither an executable nor a linkable output file is generated.

- Generate symbol information for single symbols: SELSYM symbol [, symbol [, ...]]  
The SELSYM directive is another special variant of the ADDSYM directive. Using this directive you can force the assembler to add the symbol information of the symbol given as arguments to an



executable or a linkable output file. Only global labels representing a relocatable location within the hunk can be added to the symbol information.

Example:

```

        SELSYM foo,bar    ; generate symbol information
        ...              ; only for the foo and bar symbols
foo:
foobar:
bar:
foofoo:

```

The SELSYM directive will be ignored if neither an executable nor a linkable output file is generated.

## 1.66 pro.guide/Object Modules

### Creating Linkable Object Modules

---

Object modules are usually divided into sections just like executables are. However, in this case, the names assigned to sections assume more importance, because the linker always merges sections with the same name into a single section in the final executable.

- Specify program unit name: IDNT string
- : IDENTIFY string
 

This directive will set the module name (hunk\_unit) to the given string. The name can be up to 128 characters long and should be enclosed in double or single quotes if it contains any whitespaces. The directive is only useful if a linkable output is produced; else it is ignored. Only the last of these directives appearing in the source will be used.

## 1.67 pro.guide/Defining and Referencing External Symbols

- Reference an external symbol: XREF symbol[,symbol[,...]]
 

The XREF directive is used to specify a list of symbols that is referenced within this object module but is defined in another object module (via XDEF). If this directive is used while no linkable or preassembled output is being generated, a warning will be reported and the directive will be ignored.

Example:

```

XREF GetThis,GetThat,DoThis,DoThat
XREF start
...
jsr    GetThis
jsr    DoThat

```

- Declare a symbol as externally visible: XDEF symbol[,symbol[, ...]]

For each symbol defined in the symbol list following the XDEF directive, the assembler generates an external symbol definition, making it possible for other object modules to reference these symbols. A warning will be issued if this directive is use while no linkable or preassembled output is generated.

Example:

```

XDEF GetThis,GetThat,DoThis,DoThat
XDEF start

start:
    ...
GetThis:
    ...
GetThat:
    ...
DoThis:
    ...
DoThat:
    ...

```

- Defining and referencing external symbols: PUBLIC symbol[,symbol[,...]]  
The PUBLIC directive is treated as either XREF or XDEF directive, depending on whether the symbol in question has been defined in the current object module or not.

Symbols defined in the object module will be exported by the assembler, making it possible for other object modules to reference these symbols. The other symbols that are not defined in the object module will be referenced by the assembler as external symbols.

If this directive is used while no linkable or preassembled output is being generated, a warning will be reported and the directive will be ignored.

Example:

```

PUBLIC GetThis,DoThat,Start
    ...
Start: jsr    GetThis
      jsr    DoThat

```

- Reference all unknown symbols as external symbols: AUTOXREF
-

The AUTOXREF directive forces the assembler to treat all undefined symbols as external symbols.

If this directive is used while no linkable or preassembled output is being generated, a warning will be issued and the directive will be ignored.

For example:

```
AUTOXREF
      ...
Start: jsr   GetThis
      jsr   DoThat
```

GetThis and DoThat are not defined within the object module and therefore they will be referenced by the assembler as external symbols. The same result can be achieved in the example above when the AUTOXREF is replaced by XREF GetThis,DoThat.

If case insensitivity is in effect, symbol names that are written to the AmigaDOS object file are written in all-uppercase.

## 1.68 pro.guide/Output File Attributes

Output File Attributes

-----

See 'The AmigaDOS Manual', 3rd Edition, Bantam Computer Books, for more information about AmigaDOS file attributes.

Protection Flags

.....

- Set pure bit on the output file: PURE
  - If this directive appears in the assembly source file, the assembler's output file will be marked pure (the file's p protection bit is set). This will allow the file to be made resident with the AmigaDOS Resident command.

Note that ProAsm does not check to see if your code really is pure. It is your responsibility to make sure the program does not use any statically allocated data.

PURE is just a special form of the FILEPROTECT directive.

- Set output file protection bits: FILEPROTECT +flags
  - : FILEPROTECT -flags
  - : FILEPROTECT =flags
-

The FILEPROTECT directive sets the AmigaDOS file protection flags for the output file. Which bits are set, cleared, or left alone depends on the first character in the operand:

- + Set the specified flag bits, leaving all others unchanged.
- Clear the specified flag bits, leaving all others unchanged.
- = Set the specified flag bits and clear all others

Here is a list of the supported flags and their meanings under AmigaDOS.

- r Read permission
- w Write permission
- e Execute permission
- d Delete permission
- a Archived
- p Pure (can also be set with the PURE directive)
- s Script
- h Hidden

For example, this line causes the pure and archive bits to be set on the output file:

```
FILEPROTECT +pa
```

Comment

.....

- Set the output file comment: FILENOTE comment  
This directive allows you to set the AmigaDOS comment attached to the output file, which can be up to 79 characters long. (This string is displayed when the file is listed with the AmigaDOS List command, for example.) If the comment includes spaces, tabs, or semicolons, it must be enclosed in single or double quotes. If no comment argument is given, any existing filenote will be deleted
-

from the output file.

Example:

```
FILENOTE "my first program"
```

If no FILENOTE directive appears in the source file, the comment on the output file is left unchanged (or the comment is set to an empty string, if the output file did not exist before ProAsm was run).

## 1.69 pro.guide/Auxiliary Output Files

### Auxiliary Output Files

=====

Besides its normal output of assembled code, ProAsm can also generate two types of auxiliary output files which reflect the results of the assembly process: error files, equate files, source listings, and cross-reference listings.

An error file is a file listing all errors ProAsm encountered during assembly. Many text editors have the capability of automatically invoking an assembler on source code being edited and reading the resulting error file. The user can then step through each error in turn, and the editor automatically positions the cursor at the place in the source code where the error was found, allowing the user to correct it easily. See your editor's reference manual for information about using features like this.

An equate file contains all symbols and labels defined in the source code.

A listing file can be generated, which will contain a copy of the original source code re-formatted in a way that clearly indicates how ProAsm interpreted the source code. For example, macros and include files are usually expanded in the listing, making it easier to determine their behavior and debug them.

A cross-reference listing file shows the usage of symbols, of labels with the number of references made to them and to the label's offset, and of absolute symbols with their values. The file gives also additional information on the hunk structure of the produced executable or linkable object file.

- Set the name of the error file: ERRFILE filename

This directive sets the name of the file to which ProAsm writes error and warning messages. If the assembler encounters more than one ERRFILE directive, only the first one is used and a warning message will be issued for all other occurrences. The filename specified by this directive can be overridden with the `-e` option on the command line (see Invoking).

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces.

Note that no error file is generated unless at least one error or one warning message is reported.

The messages written to the error file are of a shorter form than the ones displayed in the shell (CLI), to allow easier processing by other software (e.g. by a text editor's ARexx interface).

The following example shows different errors and their depiction in the error file:

```
foo:
foo:                                ; the same symbol 'foo' is defined twice
      mova.l d2,a0                  ; movea.l would be the correct ←
      instruction
```

The two errors in the code lines above are displayed in the error file like this:

```
Pass 1
2      : symbol defined twice
3      : unrecognized instruction 'mova'
```

- Set the name of the equate file: EQUFILE filename

This directive sets the name of the file to which the assembler writes the equate listing.

If the assembler encounters more than one EQUFILE directive, only the last one is used. The filename specified by this directive can be overridden with the -q option on the command line (see Invoking).

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces.

All symbols and labels defined in the source code (this includes also external source files loaded into the program source) will be written to the equate file.

The following example shows the equate file (shortened) of the perfmon.s example source code:

```
**
** ProAsm Equate Listing
**
** created on: 07.10.92
** for:      dhl:sources/perfmon.s
**

...
progbase      equ $00000000
clistartup    equ $00000028
wbstartup     equ $0000003A
main          equ $000001A4
DoQuit        equ $00000352
```

```

DoAbout                equ $00000358
mulu32                 equ $00000370
divu32                 equ $0000038E
PrintTitle             equ $000003B8
smc_PrintText          equ $000003C0
smc_afterPrintText1643 equ $000003E8
DoRawFmt               equ $000003E8
InstallDaemon          equ $00000402
DaemonCode              equ $00000464
waitloop               equ $000004EA
de_closedevice         equ $00000512
de_rempport            equ $0000051E
de_freesignal          equ $0000051E
de_exit                equ $0000052C
de_signal              equ $0000052E
DaemonName              equ $0000053E
DaemonNameReal         equ $00000552
Switch                 equ $0000056A
Launch                 equ $000005A4
...
end

```

- Set the name of the list file: LISTFILE filename

This directive sets the name of the file to which ProAsm writes the listing. If ProAsm encounters more than one LISTFILE directives, only the last one is used. Note that no listing file is generated unless listing is actually turned on with the LIST directive at some point in the program. The filename specified by this directive can be overridden with the `-p` option on the command line (see Invoking). If no listing file is specified, the listing will be sent to PRT: by default.

- Set the name of the cross-reference file: CREFFILE filename[,expression]
- This directive sets the name of the file to which the assembler writes the cross reference listing.

If the assembler encounters more than one CREFFILE directive, only the last one is used. The filename specified by this directive can be overridden with the `-r` option on the command line (see Invoking).

The filename must be in normal AmigaDOS format and must be enclosed in double or single quotes if it contains any whitespaces.

An expression can optionally follow the filename (separated by a comma), that specifies the lowest number of references for the symbols which you want to have in the cross reference listing.

The following example shows the cross reference listing (shortened) for the `perfmon.s` example source code:

## 1.70 pro.guide/OPT NOLIST

- Enable listing: LIST
  - : OPT LIST
  - Disable listing: NOLIST
  - : OPT NOLIST
- The LIST directive enables listing generation, while NOLIST disables it. By default listings are not generated.
- Pairs of NOLIST followed by LIST directives can be nested. For example, if listing is enabled and three NOLIST directives are encountered, then listing will be disabled until after three LIST directives.
- 
- List macro definitions: OPT MD
  - Suppress macro definitions in listing: OPT NOMD
- This option includes or suppresses the macro definition text in the listing. By default, macro definitions are included in the listing.
- 
- List expanded macros: OPT MEX
  - : OPT M+
  - Leave macros collapsed in listing: OPT NOMEX
  - : OPT M-
- This option includes or suppresses the complete macro text for all macro expansions in the listing. All lines in the expanded macros are preceded by a plus sign (+) in the listing. By default, the macros are not expanded in the listing.
- 
- Generate symbol listing in listing file: LISTSYMS expression
  - : OPT SYMTAB
  - : OPT S+
  - Suppress symbol listing (default): OPT NOSYMTAB
  - : OPT S-
- This option creates or suppresses the printing of a symbol listing at the end of the list file. Only the last option of this type is taken into account.

The LISTSYMS directive supports an expression as additional argument. This expression will be used to confine the listing to only those symbols that have been referenced at least that many times in the program code.

---



- List conditional assembly directives (default): OPT CL
- Suppress listing of conditional assembly directives: OPT NOCL  
This option enables or disables the printing of conditional assembly directives in the listing file. Refer to

Conditional Assembly  
for more information on conditional assembly.

- List false branches of conditionals: LFCOND
- Suppress false branches of conditional in listing: SFCOND  
These directives are used to control the inclusion in the listing file of conditional expressions that evaluated false. By default, only the conditional expressions that evaluated true (i.e. the ones that actually got assembled) are included in the listing.

Example:

```
LFCOND
foo      EQU      1
IFNE     foo
...
ELSE
...
; this block will be written to the listing
ENDC
```

- Send special strings to the listing file: LISTCHAR expression[,expression...]  
The given stream of expressions will be broken down into an ASCII string as with DC.B, and sent to the listing file instead of the output file. This can be used, for example, to send control codes to a printer or console window to which the listing is being sent.

Example:

```
LIST      ; Enables the listing
LISTFILE "CON:0/0/640/200/Listing File"
          ; Sends the listing to a console window
LISTCHAR $9b,"l","m"
          ; Sets the typeface to bold
...
```

## 1.71 pro.guide/LLEN

- Set line length: LLEN line-length  
The LLEN directive sets the line length in characters of the assembly listing to line-length, which may be from 38 to 255. By default, the line length is set to 132 characters per line. For

example, this statement sets the line length to 80 characters:

```
    LLEN 80
```

- Define page length: PLEN page-length

The PLEN directive sets the page length for the assembly listing to page-length, which may be from 12 to 255. A value of zero turns paging off. By default, the page length is set to 60 lines. For example, this line instructs the assembler to assume pages of 88 lines:

```
    PLEN 88
```

- Insert blank lines in listing: SPC lines

This directive is used to waste paper. The integer expression in the operand field specifies the number of blank lines that are to be put in the listing. It won't space past the top of a new page.

Example:

```
    SPC 15 ; output 15 blank lines in the listing
```

- Control listing format: FORMAT option[,option[,...]]

This directive may be used to alter the default listing format by controlling various options. All options are specified by one digit followed by a plus (+) or minus (-) sign. n+ will enable the format option and n- disables it. Multiple options can be set by separating them with a comma (,).

0+

0-

This option can be used to control the inclusion of line numbers in the listing. By default, the line numbers are written to the listing (0+).

1+

1-

This option tells the assembler to include (1+) or suppress the section number and program counter offset in the listing.

2+

2-

This option enables (2+) or disables the writing of the produced code as hexdump to the listing.

2:expression

Specify the allowed number of bytes in the hexdump field. The range of a valid expression is from 0-22. An odd expression will be aligned to the next word boundary. An

expression of zero disables the hexdump. The assembler defaults to a field size of 2 bytes.

Example:

```
FORMAT 2:8 ; specifies a hexdump field of 8 bytes
```

4+

4-

This option includes (4+) or suppresses the complete macro text for all macro expansions in the listing. By default, the macros are not expanded in the listing. It is identical to the OPT M option.

5+

5-

This option includes (5+) or suppresses the contents of an included file (via INCLUDE) in the listing.

7+

7-

This option forces the assembler to include (7+) or suppress all titles and subtitles (defined using TTL and SUBTTL) in the listing.

- Next page: PAGE

- : PAGEUP

Unless paging has been inhibited (page size is zero) or listing is disabled, this directive causes the listing to move to the top of the next page.

- Disable paging: NOPAGE

This directive turns off the output of page headers and form-feed characters. This is equivalent to PLEN 0.

- Set page title: TTL title

- : TITLE title

The TTL directive sets the page title to title. The string will be printed in the third line of the page header at the top of each page in the listing until a new title is set or paging is disabled. The title string may be up to 256 characters long. If it contains spaces, tabs or semicolons, it must be enclosed in single or double quotes. If no title string is defined, the line is either left blank or the name of the current include file is written.

Example:

```
TTL 'listing of my first program'
```

- Set page subtitle: SUBTTL subtitle

The SUBTTL directive sets the page subtitle to subtitle. The supplied subtitle will be printed below the listing title (in the fourth line respectively the third line in the listing header if no title is given). until a new subtitle is specified or paging is disabled. The string may be up to 256 characters long. If it contains any whitespaces or semicolons, it must be enclosed within single or double quotes. To turn off the printing of the subtitle in the listing a null string can be used as subtitle argument. By default, no subtitle string is defined.

Consider for example:

```
TTL      "Example for TTL and SUBTTL"

SUBTTL  "subtitle for page 1"      ; pre-define subtitle for first page
SUBTTL  "subtitle for page 2"      ; pre-define subtitle for 2nd page
SUBTTL  "subtitle for all the other pages"

LIST                                ; start listing...
...
```

## 1.72 pro.guide/OPT

- control assembler options: OPT option[,option[,...]]
  - : OPTION option[,option[,...]]
- These directives may be used to alter the assemblers default behavior by controlling various options.

Most options are specified by one or two letters followed by a plus (+) or minus (-) sign. However, some options have two forms: the base form which enables the option, and the option name preceded by NO which will disable the option. As an example, the following lines will enable and disable, respectively, the printing of macro definitions in the listing file:

```
OPT      MD
OPT      NOMD
```

Multiple options can be set by separating them with a comma (,). Unless otherwise noted, all options may be used as many times as you wish. All options are case-insensitive.

The options themselves are described in the appropriate places in other parts of this manual. For quick reference, here is a list of the supported options, each with a reference to the place in the manual where it is described:

A  
Automatic PC-relative assembly.  
See  
OPT AUTOPC

.

## ABL

Set default address register base displacement size to longword.

See

Set default behaviour

.

## ABW

Set default address register base displacement size to word.

See

Set default behaviour

.

## AUTOPC

Enable automatic PC-relative assembly.

See OPT AUTOPC.

## BDL

Set default base displacement size to longword.

See

Set default behaviour

.

## BDW

Set default base displacement size to word.

See

Set default behaviour

.

## BRL

Set default size for branch instructions to longword.

See

Set default behaviour

.

## BRW

Set default size for branch instructions to word.

See

Set default behaviour

.

## BRB

Set default size for branch instructions to byte.

See

Set default behaviour

.

## BRS

Set default size for branch instructions to byte.

See

Set default behaviour

.

## C

Case sensitivity control.

See

Case Sensitivity

.

CASE  
Case sensitivity.  
See  
Case Sensitivity  
.

CHKBIT  
Report error if bit number not in bounds.  
See  
OPT CHKBIT  
.

CHKIMM  
Check absolute addresses for missing #.  
See  
OPT CHKIMM  
.

CHKPC  
Enable position-independent code check.  
See  
Position-independent code  
.

CL  
List conditional assembly directives in the listing file.  
See  
OPT CL  
.

D  
Control output of a symbol table for debugging.  
See  
ADDSYM  
.

DEBUG  
Enable debugging information.  
See  
OPT DEBUG  
.

E  
Control display of error messages.  
See  
OPT E  
.

ESSn{ub}  
Select symbol search algorithm.  
See  
OPT ESSn  
.

F

---

Control filename display during assembly.  
See           OPT F  
              .

FPSP40  
Enable MC68040 software-supported instructions.  
See           OPT FPSP40  
              .

FPSP60  
Enable MC68060 software-supported instructions.  
See           OPT FPSP60  
              .

GENSYM  
Generate a preassembled file.  
See           PREASM  
              .

HCLN  
Enable compressed debugging information.  
See           OPT HCLN  
              .

I  
Check for immediate operands accidentally written in absolute  
addressing mode.  
See           OPT CHKIMM  
              .

INCONCE{UB}  
Ignore or allow duplicate includes of the same file.  
See           OPT Y  
              .

L  
Specify creation of linkable or executable output.  
See           LINKABLE  
              .

LIST  
Enable listing.  
See           LIST  
              .

LOCALDOT  
Use the period (.) character to introduce local labels.  
See

---

Local Label Introducer

.

LOCALU

Use the underscore (\_) character to introduce local labels.

See

Local Label Introducer

.

M

Include or suppress macro expansions in the listing file.

See

OPT M

.

MD

List macro definitions.

See

OPT MD

.

MEX

Include macro expansions in the listing file.

See

OPT MEX

.

NOAUTOPC

Disable automatic PC-relative assembly.

See OPT NOAUTOPC.

NOCASE

No case sensitivity.

See

Case Sensitivity

.

NOCHKBIT

Do not check bit number.

See

OPT NOCHKBIT

.

NOCHKBIT

Do not report error if bit number not in bounds.

See

OPT NOCHKBIT

.

NOCHKPC

Disable position-independent code check.

See

OPT NOCHKPC

.

NOCL

---



Do not list conditional assembly directives in the listing file.  
See  
    OPT NOCL  
    .

NODEBUG  
Disable debugging information.  
See  
    OPT NODEBUG  
    .

NOFPSP40  
Disable MC68040 software-supported instructions.  
See  
    OPT NOFPSP40  
    .

NOFPSP60  
Disable MC68060 software-supported instructions.  
See  
    OPT NOFPSP60  
    .

NOHCLN  
Disable compressed debugging information.  
See  
    OPT NOHCLN  
    .

NOINCONCE  
Do not ignore or allow duplicate includes of the same file.  
See  
    OPT NOINCONCE  
    .

NOLIST  
Disable listing.  
See  
    OPT NOLIST  
    .

NOMD  
Suppress macro definitions in listing.  
See  
    OPT NOMD  
    .

NOMEX  
Suppress macro expansions in the listing file.  
See  
    OPT NOMEX  
    .

NOOPTIMLIB  
Disable the use of the optimizer library.  
See  
    OPT NOOPTIMLIB

---

.

NOSP60  
Forbid MC68060 integer instruction emulation.  
See  
    OPT NOSP60  
.

NOSUPER  
Enable warnings on supervisor-mode-only instructions.  
See  
    OPT NOSUPER  
.

NOSYMTAB  
Suppress symbol listings in the list file.  
See  
    OPT NOSYMTAB  
.

NOTYPE  
Disable type checking.  
See  
    OPT NOTYPE  
    NOWARN  
Disable assembler warning messages.  
See  
    OPT NOWARN  
.

NOXDEBUG  
Suppress debug symbol table in the output file for exported symbols only.  
See  
    OPT NOXDEBUG  
.

NOXPK  
Disable XPK support.  
See  
    OPT NOXPK  
.

O  
Optimization options.  
See  
    Optimization  
.

ODL  
Set default size for outer displacement to longword.  
See  
    Set default behaviour  
.

ODW  
Set default size for outer displacement to word.

---

See  
Set default behaviour  
.

OPTIMLIB  
Enable the use of the optimizer library.  
See  
OPT OPTIMLIB  
.

OW  
Optimization warnings.  
See  
OPT OW  
.

P  
Check position-independent code.  
See  
Position-independent code  
.

P=processor  
Specify processor type.  
See  
OPT P=  
.

PCBL  
Set default size for program counter relative base displacement to longword.  
See  
Set default behaviour  
.

PCBW  
Set default size for program counter relative base displacement to word.  
See  
Set default behaviour  
.

Q  
Advanced optimization options.  
See  
OPT Q  
.

QW  
Advanced optimization warnings.  
See  
OPT QW  
S  
Generate or suppress symbol listings in the list file.  
See  
Symbol listing  
.

---

SP60  
Enable MC68060 integer instruction emulation.  
See  
    OPT SP60  
    .

SUPER  
Disable warnings on supervisor-mode-only instructions.  
See  
    OPT SUPER  
    .

SW  
Enable or disable warnings on supervisor-mode-only instructions.  
See  
    OPT SW  
    .

SYMTAB  
Generate symbol listings in the list file.  
See  
    Symbol listing  
    .

T  
Enable or disable type checking.  
See  
    Type Checking  
    .

TYPE  
Enable type checking.  
See  
    OPT TYPE  
    .

U  
Specify whether the underscore (\_) or period (.) character is used to introduce local labels.  
See  
    Local Label Introducer  
    .

W  
Enable or disable assembler warning messages.  
See  
    OPT W  
    .

WARN  
Enable assembler warning messages.  
See  
    OPT WARN  
    .

---

**WARNOBIT**

Report warning if bit number not in bounds.

See

OPT WARNOBIT

.

**X**

Create or suppress debug symbol table in the output file for exported symbols only.

See

OPT X

.

**XDEBUG**

Create debug symbol table in the output file for exported symbols only.

See

OPT XDEBUG

.

**XPB**

Enable XPB support.

See

OPT XPB

.

**Y**

Ignore or allow duplicate includes of the same file.

See

OPT Y

.

**Z**

Show original or preprocessed source lines with error and warning messages.

See

OPT Z

.

## 1.73 pro.guide/OPT XPB

- Enable XPB support (default): OPT XPB
- Disable support: OPT NOXPB

If the XPB support is enabled, any packed file (with XPB) will be automatically unpacked by the assembler. Note that encryption sublibraries are not supported.

The XPB package is not required by ProAsm to run. It enlarges only capacity of ProAsm if installed.

But should you force the XPB support to be enabled (by using the OPT XPB option) and ProAsm cannot open the required xpbmaster.library library, an error will be reported.

By default, the assembler enables the XPK support if the `xpkmaster.library` can be opened successfully. No message will be displayed if it failed.

For more information about the XPK Data Compression Package, refer to the package itself that is freely distributed and can be found on Aminet and many public domain libraries. There is also a developer package for all programmers that can also be found on Aminet.

- Temporarily store states: OPT STO
- Recall stored states: OPT RCL
- Reset states: OPT RESET

OPT STO and OPT RCL can be used to force the assembler to temporarily store and recall the states of almost all options, modes, and settings of ProAsm.

The possibility to store and recall these states can be used to make routines, that are included into the source code, more modular. The states can be stored at the beginning of a routine, then changed to the routines needs, and finally restored at the end of the routine to their previous settings.

Consider the following example:

A routine that is included into the source code if used, but it should not get in conflict with the current settings and states.

```

        IFND     RESET_FPU_R      ; This IFND mechanism forbids
RESET_FPU_R EQU      1          ; this include file to be loaded twice

        OPT     STO               ; store current states,
        OPT     O+,OW-,Q+,QW-    ; enable optimization
        MC68882                ; and select MC68882

ResetFPU:
        ...
        rts

        OPT     RCL               ; restore temporarily saved states
                                     ; => previous used states
        ENDC

```

The use of OPT STO and OPT RCL cannot be nested. Only the last stored states can be recalled. If OPT RCL is used when no settings were stored previously, the option is ignored.

OPT RESET causes the assembler to initialize most of the options, modes, and states to their default.

The states of the following directives and options are stored, recalled, and resetted by using the described options (Note that only the names of the directives and options are used below, without any further declarations and without their negative respectively positive form):

- Select symbol search algorithm: OPT ESS1+
- : OPT ESS2+
- : OPT ESS3+
- Unselect symbol search algorithm: OPT ESS1-
- : OPT ESS2-
- : OPT ESS3-

Next to ProAsm's default symbol search algorithm, three additional, enhanced algorithms can be selected.

They do not only differ in the resulting search time, but also in the memory usage. OPT ESS1+ uses 131072 bytes, OPT ESS2+ 32768 bytes, and OPT ESS3+ 524288 bytes! (The last one is probably not usable on machines with less than 2MB of memory).

The memory usage an algorithm uses does still not guarantee a faster search. For example, source code produced by any reassembler contains tons of symbols that look quite similar (i.e. Lxx00002 etc.). In such cases the OPT ESS1+ option is usually better than the other two algorithms.

If you often use a reassembler, you probably have to experiment which one of these algorithms brings you the most advantage in assembling the produced source codes.

If not enough memory is available for the selected algorithm, ProAsm drops back to its default algorithm.

These options should only be used before any symbols are defined or an option must be at beginning error will be reported. By default, none of these additional algorithm is selected.

## 1.74 pro.guide/ESCAPESTR

- Enable escape sequences: ESCAPESTR
- : ESCAPESTR ON
- Disable escape sequences: ESCAPESTR OFF

The ESCAPESTR directive can be used to control the allowance of escape characters inside quoted strings of data generating directives (such as DC, CSTRING and many more).

Escape character in strings are represented by a backslash character ( \ ) and a second character.

The legal escape sequences and their meaning are listed below:

- \0  
Zero byte (DB 0)
- \b  
Backspace (DB 8) [ANSI C]

```

\c      Control sequence introducer (DB $9b)

\e      Escape (DB 27)

\f      Form feed (DB 12) [ANSI C]

\n      Newline (DB 10) [ANSI C]

\r      Carriage return (DB 13) [ANSI C]

\t      Horizontal tab (DB 9) [ANSI C]

\v      Vertical tab (DB 11) [ANSI C]

\xnn    Insert a two digit hex value (DB $nn)

\       Backslash (DB "\") [ANSI C] The escape sequences marked with
        [ANSI C] are defined by the ANSI C standard.

```

Even though you need two keystrokes to write these sequences into your source code, they occupy exactly one byte in the produced object code.

For example:

```

        ESCAPESTR
        DC.B    "Hello\nworld!\0"
;       => DC.B "Hello", $a, "world!", 0

        CSTRING "\c1mA\c0m\B\n"      ; make 'A' bold
;       => CSTRING $9b"1mA", $9b, "0m\B", $a

```

#### - Achieve C symbol format: CSYMFMT

This directive forces the assembler to prefix all symbols with an underscore (`_`) that are written either as external references or as external definitions to the object file. No underscore character is added to a symbol if it starts with an atSign character (`@`).

The advantage of using this directive is that the generated object code can be made linking compatible with common C compilers without the need of manually add an underscore as prefix to each symbol.

The CSYMFMT directive is ignored if no linkable output is



generated.

## 1.75 pro.guide/VERBOSE

- Report object and timing information: VERBOSE
  - Report timing information: TIMES
  - Suppress all additional information (default): QUIET
- These directives can be used to control the information output that the assembler reports when it finishes the last pass.

Using the TIMES directive ProAsm displays the needed time for the current task and the resulting "lines per minute" (LPM) ratio. Also the number of defined and used macros are displayed.

In addition VERBOSE displays also hunk information and a list of unused registers.

By default, QUIET, no additional information is reported.

- Set assembler task priority: ASMPRI value
- Since the Amiga is a multitasking machine, it uses priority numbers assigned to each task to determine the relative importance they should be serviced. This directive can be used to set ProAsm's task priority during assembly.

The range of the priority are the integers from -128 to 127. The CLI and the most tasks have a priority of 0, and it is suggested that you do not set the priority higher than 10. A priority higher than zero means that the assembler will not be slowed down as much by other tasks. Setting the priority lower than 0 will prevent ProAsm from slowing down other programs with normal priority.

Example:

```
ASMPRI 3 ; Raise assembler priority level to 3
```

- Define comment block: COMMENT delimiter comment delimiter
- This directive can be used to define easily a comment that spawns one or more lines.

The first character found is taken as delimiter except if it is a whitespace character. The text following this delimiter becomes the comment block until the next occurrence of the delimiter. Should the delimiter character be used inside the comment block, then double the delimiter character to tell the assembler that this is not the end-delimiter.

During assembly the comment blocks are ignored completely.

---

The advantage of using the COMMENT directive is that you do not need to comment each line using the semicolon (;) or the asterisk (\*).

For example:

```
COMMENT |
This is a comment block
spawned over as many lines as required.
A delimiter character used inside the comment block
must be doubled (||) to tell the assembler that
this is not the end-delimiter.
|
```

- Print remaining line: PRINTX remaining line...  
This directive can be used to output the remainder of the line to the standard output.

For example:

```
IFD    __DEMO__
PRINTX ** demo version generated.
ELSE
PRINTX ** full version generated.
ENDC
```

## 1.76 pro.guide/RCRESET

- Reset return code: RCRESET  
This directive sets the \_\_RCODE symbol to zero. The \_\_RCODE symbol contains the return code ProAsm returns to AmigaDOS if the source file ended (see \_\_RCODE).
- Set return code: RCSET expression  
The RCSET directive sets the \_\_RCODE symbol to the desired expression. This symbol contains the return code ProAsm returns to AmigaDOS if the source file ended (see \_\_RCODE).

Setting the return code can be useful if ProAsm is used inside batch files.

- Enable additional error information: ERRFLAG  
This directive instructs ProAsm to display additional information to the error messages and warning reports.

These information can be used to determine the state the assembler was in when the error or warning occurred. This information may

be irrelevant for the user, but can be important to us if internal error messages (e.g. INTERNAL: ...) have occurred. This directive was especially implemented for beta testing reasons.

The flags will be printed after the error descriptions, enclosed in square brackets ( [flags] ).

The flags have following meaning:  
error occurred...

- C  
during conditional assembly.
  - I  
inside an included source code.
  - M  
inside a macro body.
  - R  
inside REPT/ENDR loop.
  - r  
inside REPEAT/UNTILcc loop.  
with the following flag set:
  - e0  
Standard symbol search algorithm used.
  - e1  
OPT ESS1+ option selected.
  - e2  
OPT ESS2+ option selected.
  - e3  
OPT ESS3+ option selected.
  - N  
Both syntax modes accepted (RELAX).
  - o0  
Executable output format selected.
  - o1  
Linkable output format selected.
  - o2  
Binary output format selected.
  - o3  
Motorola S-record as output format selected.
  - o4  
Preasm as output format selected.
-

## 1.77 pro.guide/LOCKSYM

- Lock symbols: LOCKSYM symbol[,symbol[...]]
  - Unlock symbols: UNLOCKSYM symbol[,symbol[...]]
- The LOCKSYM directive is used to specify a list of symbols as "locked", meaning that these symbols cannot be referenced anymore. Any references are reported by an error message.

The UNLOCKSYM directive on the other hand removes the "lock" mark from the symbols of the given symbol list.

This mechanism of locking symbols allows you to check symbol references. If, for example, symbols were just defined for clearness and should not be referenced in the program code, they can be locked using the LOCKSYM directive:

```

LOCKSYM write_to_StdOut

PrintText:
    ...
write_to_StdOut:                ; this symbol might be inserted
    move.l  DosBase(pc),a6      ; for commenting a code fragment
    jsr    __LVOOutput(a6)
    move.l  d0,d1
    jmp    __LVOWrite(a6)

```

## 1.78 labseg

- Begin new labseg scope: LABSEG name
- When ProAsm sees a LABSEG directive, it takes the current symbol and macro tables, stores them away somewhere for safe keeping, then creating new empty ones. Only the special internal symbols (\_\_RS, etc.) remain.

The name is a string up to 256 characters long. It must be quoted if it contains special characters such as whitespaces or semicolons (;). LABSEG names are case-insensitive and are used to identify the new created tables.

Symbols within a LABSEG scope (from the LABSEG directive to the next one or to the end of the source code) can have the same names as symbols outside this scope, since they are stored in different symbol tables. The same is also valid for any macros. You have to use the EQUX directive to import symbols from other LABSEG scopes. See below for the EQUX description for more details.

The immediate usefulness of this directive is that you can encapsulate a piece of code that performs some specific function, and ensure that it does not make any (forbidden) references to anything outside that LABSEG scope.

With the LABSEG directive you have also the possibility to include source codes of other programs without renaming all the duplicate symbols these sources may have.

If a LABSEG scope was defined earlier with the same name as the new scope, ProAsm continues adding the symbols and macros to the first scope instead of creating new tables. Any symbols and macros defined before the first LABSEG directive will be placed in the default tables. The default tables are unnamed, as if a LABSEG directive had been included at the very beginning of the source file with an empty string as name ("").

For example:

```

eval    EQUX    "calculator",eval    ; import a symbol
...
bsr     eval

LABSEG  "calculator"          ; create new tables
INCLUDE "sources:otherprojects/calculator.s"
LABSEG  __OldLabseg          ; drop back to previous used ←
tables

```

Another example:

```

...
rts

LABSEG  "foo"                ; create new tables named foo
foo:    ...
LABSEG  "bar"                ; create new tables named bar
bar:    ...
LABSEG  "foo"                ; use tables labeled foo
foofoo: ...
LABSEG  ""                   ; drop back to initial symbol
; macro tables

```

- Revert to the previous LABSEG scope: LABSEG \_\_OLDLABSEG  
This directive reverts to the LABSEG scope that was in effect before the most recent LABSEG directive.

This can be used in include files, for example, to temporarily switch to a different section and then later switch back to the LABSEG scope that was in use by the "parent" source file. This feature does not stack, however; additional LABSEG \_\_OLDLABSEG directives merely toggle between the two most recent scopes.

The keyword \_\_OLDLABSEG is case-insensitive.

Example:

```

foo     EQUX    "foo",bar     ; import symbol bar from LABSEG foo
dots
LABSEG  "foo"                ; create new tables named foo
bar:    dots
LABSEG  __OldLabseg         ; toggle back to previous scope

```

```

bar:    dots
        bra    foo           ; jump to symbol bar within LABSEG foo

```

- Import a symbol from another LABSEG scope: `symbol EQUX scope,xsymbol`  
This directive allows you to import a symbol from another LABSEG scope.

The symbol in question is the symbol `xsymbol` from the LABSEG scope `scope`, and it will be added to the current symbol table as `symbol`.

Consider the following example:

```

        LABSEG "mainscope"
foo:    ...

        LABSEG "subscope_1"
foo2 EQUX "mainscope",foo ; import symbol foo named as foo2
foo:    ...
        bsr    foo2       ; go to foo from the mainscope LABSEG scope
        bra    foo        ; jump to foo of this scope

```

It works similar to the `EQU` directive, except that you have to name the LABSEG scope, the symbol in question (`xsymbol`) is taken from. Note that symbols can only be imported and never exported.

## 1.79 pro.guide/Special Symbols

### Special Symbols

\*\*\*\*\*

ProAsm defines a variety of built-in symbols which are automatically substituted with certain values when you use them. Some of these are useful at any point in the program, while others are useful only in certain places like inside macro bodies. You may use these symbols in any combination of uppercase and lowercase: for example, `__DATE`, `__date`, and `__Date` are equivalent. These symbols cannot be redefined with `EQU`, `SET`, etc. If you try to, ProAsm reports a reserved symbol error.

Besides the special symbols listed here, there are a few others which are only useful inside macro bodies; they are listed in

### Symbols and Keywords for Macros

.

- ProAsm version number: `__PRO`  
This symbol is replaced by the version number of ProAsm, in binary coded decimal. Bits 8-15 contain the version number and bits 0-7 the revision number.

```
$0082 means version 0, revision 82
$0102 means version 1, revision 02
```

Most of the modern assemblers support a similar type of identification to make it possible for the source code to determine which assembler is currently being used. For example, the following source code ensures that ProAsm 1.74 or later is being used to assemble the source code:

```
IFND    __PRO
FAIL    You need ProAsm to assemble this source code.
ENDC

IFGT    __PRO,$0174
FAIL    ProAsm version 1.74 or higher required.
ENDC
```

- RS structure offset: `__RS`
- SO structure offset: `__SO`
- FO frame offset: `__FO`

These symbols always represent the current values of the structure offset maintained by the RS... and SO... directives, and the frame offset maintained by the FO... directives, respectively. See

Structure Offsets  
, for information on these directives.

Note that `__RS` and `__SO` represent the value that will be assigned to the label in the next RS... or SO... directive, respectively, while `__FO` represents the value that was assigned to the label in the previous FO... directive. This is because the RS... and SO... directives are post-incrementing, while the FO... directives are pre-decrementing.

- Number of registers moved in last MOVEM: `_MOVEMREGS`
  - Number of bytes moved in last MOVEM: `_MOVEMBYTES`
  - Register list moved in last MOVEM: `_MOVEMLIST`
- The symbol `_MOVEMREGS` always contains the number of registers moved in the last movem instruction seen by the assembler.

`_MOVEMBYTES` contains the number of bytes moved by that movem. `_MOVEMBYTES` is always twice `_MOVEMREGS` if the last movem was a word (16-bit) instruction, and four times `_MOVEMREGS` if the last movem was a longword (32-bit) instruction.

The symbol `_MOVEMLIST` contains the register list moved in the last movem instruction seen by the assembler, as if it had been defined with REG.

These symbols are useful to an assembly language routine for accessing parameters stored "C-style" on the stack, when the

function uses `movem` to save and restore registers:

```
myroutine:
    movem.l d2-d5/a2,-(sp)          ; Save registers
    move.l  _MOVEBYTES(sp),d0      ; First parameter
    ...

    movem.l (sp)+,_MOVEMLIST      ; Restore registers
    rts
```

This way, you do not have to change the offset for each instruction that accesses these parameters whenever you change the number of saved registers.

Notice that these three symbols are prefixed with only one underscore (`_`), unlike most of the other built-in symbols, which are prefixed with two underscores (`__`). This is for compatibility with existing assemblers.

- Current base address: `__BASE`

This symbol always contains the current base address, if any is defined. For example, after the statement `BASE base_addr, __BASE` will be the same as the symbol `base_addr`. If no base is defined, `__BASE` is zero. See

```
    BASE
    , for information about the BASE
directive.
```

- Command-line variable `n`: `__Vn`

These symbols are replaced with the values you specify in the `-Vn=value` command-line options of ProAsm, where `n` is between 0 and 9, as described in Invoking. If you do not specify a value for one of these symbols on the command line, value is set to zero by default.

- Host processor: `__PR`

ProAsm replaces this symbol with a number indicating which M68000 family processor ProAsm is currently executing on. It will contain one of the following values:

0	68000
1	68010
2	68020
3	68030



4	68040
6	68060
13	68EC030 (a 68030 without an MMU)
14	68LC040 (a 68040 without an MMU) or 68EC040 (a 68040 without an MMU or FPU)

Note: This symbol represents the processor ProAsm is executing on (the host processor), not the processor ProAsm is writing code for (the target processor). This directive has nothing to do with the MC... directives, which select the target processor.

- Host coprocessors: `__CP`

This symbol is replaced with a number indicating what coprocessors, if any, are installed in the computer ProAsm is running on. Any of the following bits may be set in the number:

Bit 0

Either a 68881 or a 68882 math coprocessor is installed. (The built-in FPU on the 68040 does not cause this bit to be set.)

Bit 1

A 68882 math coprocessor is installed (not a 68881 or the 68040 FPU).

Bit 2

A 68851 MMU is installed. (The built-in MMU on the 68030 or 68040 does not cause this bit to be set.)

The actual values you may get for the various M68000 series processors are broken down in the following table:

68000	0
68010	0
68020	0 No coprocessors installed.
	1 68881
	3 68882
	4 68851
	5 68881 and 68851

---

```

7      68882 and 68851

68030
0      No external coprocessors installed.

1      68881

3      68882

68040
0

```

- Current date: DD-*MMM*-*YYYY*: *\_\_DATE*
- Current date: DD.*MM*.*YY*: *\_\_DATE2*
- Current date: *MM/DD/YY*: *\_\_DATE3*
- Current time: *HH:MM:SS*: *\_\_TIME*
- Current day of the week: *\_\_DAY*

As their names indicate, these symbols are automatically replaced with the current date or time (i.e. when ProAsm is assembling the current program), expressed as an ASCII string which you can use in a DC.B directive.

The date can be formatted in three different ways, depending on which symbol you use:

#### *\_\_DATE*

The date is formatted as DD-*MMM*-*YYYY*, where DD is the two-digit day of the month (between 01 and 31), *MMM* is the month as an abbreviated name (Jan, Feb, etc.), and *YYYY* is the year, expressed as a four-digit number (i.e. 1992).

#### *\_\_DATE2*

The date is formatted as DD.*MM*.*YY* (European format), where DD is the day, *MM* is the month (from 01 to 12), and *YY* is the last two digits of the year (i.e. 92 for 1992).

#### *\_\_DATE3*

The date is formatted as *MM/DD/YY* (U.S. format)--exactly like *\_\_DATE2*, except the month and day are swapped, and the periods are replaced with slashes.

*\_\_TIME* is substituted with the current time, in the standard *HH:MM:SS* format.

*\_\_DAY* is replaced with the current day of the week (i.e. Monday).

Here is a code fragment that demonstrates the use of these symbols:

```

DC.B      'Assembled on ',__DAY,' ',__DATE,' at ',__TIME,0
; => e.g. Assembled on Monday 16-Aug-1993 at 19:51:08

```

- Current line number: `__LINENUM`  
This symbol always contains the current line number of the program text.
  
  - Built-in Symbol: `__RCODE`  
If ProAsm has detected any errors during assembly at this point, `__RCODE` is set to 10; otherwise it is set to zero. In effect, it is the return code ProAsm would return to AmigaDOS if the source file ended at the point of the `__RCODE`. You can change the return code ProAsm returns (and thus the contents of `__RCODE`) with the `RC...` directives (see `RCRESET`, and `RCSET`).
  
  - Output object type: `__LK`  
This symbol is replaced with a number describing the type of object file to be produced from this assembly. To set the output file format, see Output File Format.  
  
0  
    No object file is being produced  
  
2  
    Raw binary  
  
3  
    AmigaDOS linkable object module  
  
4  
    AmigaDOS executable  
  
5  
    Preasm file  
  
6  
    Motorola S-record
  
  - Not-a-number: `__NAN`
  - Signaling not-a-number: `__SNAN`
  - Infinity: `__INFINITY`  
`__NAN` (not-a-number) represents the result of operations that have no mathematical interpretation, such as infinity divided by infinity. Any computation for which a NAN is provided as input will generate a NAN as output: NANs propagate through floating-point computations like a virus. This symbol allows you
-

now to use NaNs, for example, to invalidate floating-point variables so that any computations that accidentally use them will produce clearly invalid results.

\_\_SNAN represents a signaling not-a-number (SNAN). When an SNAN is used in a computation, the processor generates a floating-point exception.

The symbol \_\_INFINITY represents a positive infinite floating-point number. Infinities represent real values that exceeded the overflow threshold.

These symbols can only be used with floating-point instructions or directives.

Example

```
fmove.x #__INFINITY,fp0 ; load FP0 with positive infinity
fmove.d #-__INFINITY,fp1 ; and FP1 with negative infinity
```

Refer to the 'MC68881/MC68882 Floating-Point Coprocessor User's Manual' or any other manual that describes the MC68881/MC68882 coprocessors for detailed information about the data formats.

- Number of arguments to macro: NARG

This symbol is replaced by the number of arguments the macro was invoked with, as illustrated by this code:

```
foo      MACRO
         dc.l    NARG
         ENDM

         foo                ; => 0
         foo      a         ; => 1
         foo      a,b       ; => 2
```

This feature is usually used to create macros that can operate on different numbers of arguments, performing slightly different functions as appropriate. Therefore, NARG is usually used with IF directives.

- Number of real arguments to macro: RARG

RARG is quite similar to the NARG symbol, except that NARG contains the number of arguments passed to the macro and RARG the number of arguments that actually contain anything ("real" arguments).

```
myMacro      1,2,3,4,5      ;NARG=5, RARG=5
myMacro      1,2,,4,5      ;NARG=5, RARG=4
myMacro      1,2,,,,5      ;NARG=6, RARG=3
myMacro.b    1,2,3,4,5      ;NARG=6, RARG=6
myMacro.w    1,2,,4,5      ;NARG=6, RARG=5
```

- Current macro number: `_MCOUNT`  
This symbol can always be used to determine the number of macros currently used. (How many times macros have been expanded.)

## 1.80 pro.guide/Support Libraries

### Support Libraries

\*\*\*\*\*

The support libraries are shared Amiga libraries that can be used to increase the power and flexibility of ProAsm. They are not required by ProAsm or any of its associated utilities to run. Install the libraries to enlarge the capacity of ProAsm and de-install them to remove these additional capacities.

### Installation and De-installation

=====

The installation and de-installation process is quite as easy as flipping a power switch.

To use the features of one or more libraries, install them by copying the library to the search path of ProAsm. ProAsm searches its libraries in the `libs: assign`, the `libs:proasm` directory, and in ProAsm's home directory.

To de-install a library, just remove it from the ProAsm's library search pathes. Instead of removing a library we recommend to rename it (e.g. add an underscore (`_`) to its name). This way that the libraries vanish from your harddisk or disk.

### The `proasmlang.library`

=====

ProAsm fully supports the AmigaDOS 2.1 localization library mechanism for the error texts. Some locale catalogs are included with this release.

To use localization install this library and the desired catalog file. When installed correctly, all during an assembly process reported errors will be issued in the selected language.

### The `proasmoptim.library`

=====

---

=====

The proasmoptim.library looks at the generated code while assembly is in progress and makes suggestions about possible optimizations.

The current library reports only the recognized optimizations and does not lay its hands on the produced code. A future version may have the possibility to optimize the code, but for the actual version all optimizations have to be made by hand.

See also see OPT OPTIMLIB.

## 1.81 pro.guide/Errors

### Errors

=====

This appendix lists in alphabetical order the error messages ProAsm generates, and describes their cause and how to fix the problem.

.W or .L as index size expected

The size specifier of an index register in an indirect addressing mode with index must be either word or long word. (See

Addressing Modes

).

absolute expression must evaluate

The expression supplied with the RORG directive is too small. The program counter cannot be set to the new location, because it would overwrite already generated code. As shown in the following example:

```

...
movem.l (a7)+,d1-a6
moveq   #0,d0
rts

RORG   4           ; => error
...

```

To ensure a correct argument for the RORG directive, the expression argument - current program counter must always be positive.

absolute not allowed

The operation will not allow the absolute addressing mode.

abuse of textual symbol (equr/equstr)

You tried to use a textual symbol defined with the EQUR or EQUSTR directives (see

EQUR

) in a situation in which they are not allowed,

such as within a \\*LEFT(), \\*RIGHT(), or \\*MID() directive (see

Substituting subsections of strings

). To fix this, simply "wrap"  
the textual symbol name in a \\*STRING() directive (see

Substituting textual symbols in symbol names  
)

additional LABSEG on pass 2

An additional LABSEG directive was encountered in pass 2, but not in pass 1. Possibly caused by a misuse of the IF1 or IF2 directive.

additional symbol found

additional symbol on pass 2

A symbol was defined in pass 2, but not in pass 1. Possibly caused by a misuse of the IF1 or IF2 directive.

addressing mode not allowed

The specified addressing mode is not allowed for this instruction. Refer to Motorola's Programmer's Reference Manual (see

Bibliography  
) for detailed information about the instruction and its addressing modes.

addressing mode not recognized

ProAsm could not recognize the addressing mode used for an assembly language instruction. Make sure the instruction's operands conform to one of the valid addressing modes in

Addressing Modes  
.

address register expected

An address register direct was expected as effective address.

bad address size, .W or .L expected

As valid sizes for absolute addressing modes only word and longword sizes are accepted. If no size is given, the assembler defaults to longword size. (See

Addressing Modes  
, for more  
details about the absolute addressing modes.)

Example:

```
move.l (4).W,a6
lea $dff000.l,a6
```

bad arguments

The supplied argument for a directive is not valid. To fix this, refer to the description of the directive that caused this error message. Example:

```
FILEPROTECT +PX ; => error, X is an invalid argument
```

bad character in expression

An unexpected character was encountered in the expression.

bad line length

---

This error should not occur. If it does, please report it.

#### bad macro name

The specified macro name contains an invalid character. A macro name is like an ordinary symbol name, with the exception that the point (.) is not accepted as a valid character.

#### bad operation size for FPU instruction

The specified operation size for the floating-point instruction is not valid. Refer to the 'M68000 Family Programmer's Reference Manual' or the 'MC68881/MC68882 Floating-Point Coprocessor User's Manual', for the correct operation sizes.

For example:

```
fmovem.l (a7)+,fp0-fp7 ; => error
```

```
fmovem.x (a7)+,fp0-fp7 ; => ok
```

#### bad operation size for register

The specified operation size for the instruction is not valid for the used register. Usually, valid operation sizes for registers are only byte, word, or longword.

Refer to the 'M68000 Family Programmer's Reference Manual', for the correct operation sizes.

#### bad register

The instruction that caused this error expected another register.

```
MC68030
pflush a0,0 ; => error, either sfc, dfc,
... ; or a data register expected
```

#### bad scale factor

Only the numbers 1,2,4, or 8 are allowed for the scaling factor. (See

```
Addressing Modes
).
```

#### bad size, .B or .W expected

#### bad size, .B, .W or .L expected

#### bad size, .W or .L expected

#### bad size, byte expected

#### bad size, word expected

#### bad size, long expected

These errors occur if the specified operation size is not valid for the instruction. These error messages display also all valid size specifiers for that specific instruction.

For example:

```
move.d d0,d7
; => error, bad size, .B, .W or .L expected
```

```
add.b d0,a2
; => error, bad size, .W or .L expected
```

#### bit field offset/width out of range



The specified field offset or field width were out of range. The value of the offset field must be in the range of 0-31, and the value of the width field in the range of 1-32 (a width field of zero specifies a width of 32).

bit number must be from 0-7

bit number must be from 0-31

These error messages only occur when OPT CHKBIT was selected. OPT CHKBIT forces the assembler to be more strict when checking the immediate bit number for the bit manipulating instructions (bclr, bchg, bset, and btst). The accepted bit number depends on the destination addressing modes specified. If the destination is a data register, the bit numbering is from 0 to 31. For memory location, the bit numbers are from 0 to 7.

Generally this additional check is not needed, since the processor uses the bit number modulo 8 respectively 32. But it may be useful if you want to generate a "correct" code. (See OPT CHKBIT, for the descriptions of OPT CHKBIT, and refer to 'M68000 Family Programmer's Reference Manual' for detailed information on the bit manipulating instructions.)

Example:

```
OPT      CHKBIT
btst     #10,foo(a5)
```

branch out of range (by number bytes)

A branch was specified outside of the range that can be reached by two bytes (-32768-32767).

byte displacement out of range (by number bytes)

The byte displacement of an "address register indirect with index (8 bit displacement)" or "program counter with index (8 bit displacement)" addressing mode is out of range.

calculator buffer overflow

This error should not occur. If it does, please report it.

cannot nest MACRO definitions

The assembler has found a macro definition within another macro definition (a MACRO directive was found inside a macro definition). (See

```
Macros
, for more details about defining macros.)
```

closing brace '}' expected

The assembler expected a closing brace } but did not find one. Braces are used by a bitfield selection ({offset:width}. You should check the bitfield selection perhaps it was improperly declared.

colon expected

A colon (:) is used within a bitfield selection ({offset:width}, to separate the higher and lower long word of a quad word (highlong:lowlong), and to represent a pair of registers as used by certain instructions. These instructions are:  
cas2 Dc1:Dc2,Du1,Du2,(Rn1):(Rn2)

```

divs.l <ea>,Dr:Dq
divsl.l <ea>,Dr:Dq
divu.l <ea>,Dr:Dq
divul.l <ea>,Dr:Dq
muls.l <ea>,Dh:Dl
mulu.l <ea>,Dh:Dl
fsincos <ea>,Fpc:FPs
fsincos Fpm,Fpc:FPs
(Dc1, Dc2 ,Dul ,Du2, Dr, Dq, Dh, Dl any data registers; Rn1, Rn2
any data or address registers; Fpc, FPs, Fpm any floating-point
data registers). Refer to 'M68000 Family Programmer's Reference
Manual' for detailed information about these instructions.

```

Examples:

```

bfclr    (a0){3:16}                ; bitfield selection
pmove    #$007e0004:$80200000,crp ; quadword
divu.l   foo(pc),d0:d1             ; 64-bit dividend in d0:d1

```

comma expected

The operand field of an instruction or directive may contain multiple operands. These operands must be separated by commas.

```

move.l   (a0) (a2)
; the assembler expects a comma between the two addressing modes

```

Whitespaces may be used within the operands field, but only within quoted strings:

```

move.l   #"opt ",d0
dc.b     "Hello world!"

```

could not examine file 'filename'

The assembler cannot get the information concerning the file filename. Check the spelling of filename and that the supplied pathname is correct.

could not open asl.library or req.library.

ProAsm uses the ASL file requester if you have Kickstart version 2.04 or later, or the REQ file requester on an earlier version of Kickstart. None of these libraries could be found. Install one of these libraries (according to your Kickstart version), or try not to run ProAsm with the -F/FILEREQ command line option set.

could not open file

could not open file 'filename' (DOS error code)

The assembler cannot open the specified file. Check the spelling of filename and that the supplied pathname is correct. The DOS error code may give you more detailed information why the assembler could not open the file (see

```

    AmigaDOS Error Codes
).

```

could not open xpkmaster.library

The xpkmaster.library could not be found. The XPK Data Compression Package is probably not installed to your system. To fix the problem, you can either remove the option (OPT XPK) or install the XPK software package to your system. (Refer to

```

OPT XPK
  for further details about the XPK package.)

```

could not read file (DOS error code)

While reading the main source file, an AmigaDOS error occurred. Refer to the AmigaDOS error codes listed below for detailed information ( AmigaDOS Error Codes ).

data register expected

An operand that should be a data register is not a data register.

For example:

```

moveq #0,(a0) ; => error

```

data too large

A given operand is larger than it is allowed to be. To fix this, refer to the description of the directive that caused this error message to get details about operand ranges etc.

For example:

```

DS.B -2,0 ; => error

```

If an instruction causes this error message, refer to the 'M68000 Family Programmer's Reference Manual' for detailed information about this instruction.

divide by zero

During the evaluation of an expression the assembler encountered a division by zero. Check the concerning expression for any division by zero, you may also have to look at the symbol values.

For example:

```

foo EQU 0
...
move.l #2*(bar/foo),d0

```

DX only in executable object files allowed

The use of the DX directive is only in executable object files allowed. In other object files the use of DS is recommended. (See

```

  Declaring Data Blocks
  , for more detailed information about
  these directives).

```

DX section cannot contain data or code

The DX directive must be at the end of a section (unless within a BSS section) and no other code or data generating directive is allowed after a DX directive.

ENDR/UNTILcc changed by mistake

The REPT - ENDR loop and the REPEAT - UNTILcc loop are two different kind of loops. This error is caused, because either their loop introducers (REPT, REPEAT) or their loop terminators (ENDR, UNTILcc) are changed by mistake. (See

```

  Repeating Text
  ,

```

for more information about this directives.)

error while writing source file 'filename' (DOS error code)

An AmigaDOS error occurred while writing the source file. Refer to the AmigaDOS error codes listed below for detailed information

```
(
    AmigaDOS Error Codes
).
```

fatal error: unable to open window

ProAsm is unable to open an output window. Try to use other window dimensions if any were given. Another way to force ProAsm to send all its output to a window is to redirect it to a CON-window:

```
pro >CON:0/0/640/200/ProAsm_output mySource.s
```

This version has the disadvantage, that it does not wait for a keystroke after the assembly job has finished.

file load error (DOS error code)

An AmigaDOS error occurred while loading a file. (See

```
    AmigaDOS Error Codes
, for a list of the AmigaDOS error codes.)
```

fatally out of memory!

There is not enough memory available for the assembler to complete its task. This error occurs only if there is not enough memory to start the assembly or if the memory for the object code could not be allocated.

floating-point number value error

This error occurs if a floating-point constant has not the correct format. ProAsm accepts two notations for floating-point numbers: the pure fractional number and the engineering (exponential) notation. (See Floating-point Constants, for more details about the floating-point constants and their formats.)

For example:

```
foo    FEQU    .E1        ; => error
       fmove.x #1.1.2,fp0 ; => error
```

flow of assembly changed

This error should not occur except by the abuse of the IF1 or IF2 directive. Check all the IF1 and IF2 conditional statements in your source code. None of them should contain code- or data-generating instructions or directives, or macro definitions.

This example below causes the assembler to report this error message:

```
       IF1
foo    MACRO
       ...
       ENDM
       ENDC

       IF2
foo    MACRO
```

```

...
ENDM
ENDC

```

This construct is not valid, because the assembler finds one macro name with two different definitions.

#### forward reference not allowed (OPT A+)

A forward reference was found while the automatic PC-relative addressing mode was enabled. When this mode is turned on (by using the OPT A+ or OPT AUTOPC options, see OPT AUTOPC), the assembler transforms all absolute long addressing modes into PC-relative addressing modes. This error is only reported if the assembler transformed a forward referenced absolute long addressing mode only in the first pass, since in the second pass this reference may be out of range for the PC-relative addressing mode. To bypass this transformation you can append the .L size specifier to the absolute long addressing mode. (See OPT AUTOPC, and

```

    Addressing Modes
    , for more details.)

```

#### FPU data register expected

An operand that should be a floating-point data register is not a floating-point data register. For example:

```

    fadd.x fp0,(a0) ; a FPU data register is expected
                   ; as destination operand

```

#### FPU register expected

An operand that should be a floating-point register is not a floating-point register.

For example:

```

    lreg    EQU    FP0
    lregn   EQU    1
    ...
    fmove   (a0),lregn
           ; probably symbol names changed by mistake

```

#### garbage following instruction

The assembler issues this error when something unexpected follows an instruction or operand. For example:

```

    SECTION __OldSection, ; => error

```

#### illegal address mode for CPU type

This error occurs if you try to use an addressing mode that is only valid on 68020 and higher processors while in MC68000 or MC68010 mode. If you want the code to run on any M68000 processor, fix the instruction to use a simpler addressing mode (see

```

    Addressing Modes
    ). Otherwise, set the processor type appropriately
with one of the MC... directives (see
    Processor Options
    ).

```

#### illegal arithmetic on xref symbol

External symbol operations are limited in their combinations.

This error is reported if an illegal arithmetic operation is performed on an external symbol. Refer to the table in the section Operators (Operators) for an overview on all allowed operations and combinations with external symbols.

#### illegal BSR.S

The destination of the BSR.S is the following instruction, which is an illegal branch offset for the short (1-byte) form of BSR. To branch to the following instruction, you must use BSR.W. For example:

```
        bsr.s  foo
foo:    ...
```

#### illegal directive inside INCEQU file

Files included using the INCEQU directive have restrictions on their content, because they are read and processed in the first pass only. Only symbol and macro definitions should be used in a file loaded by the INCEQU directive. Code or data generating directives and temporary symbols are not allowed in such a file.

#### illegal instruction for processor options

The selected processor type does not support the opcode specified. (See Processor Type.)

#### illegal operator

Check the operand field for a bad operator. (See Operators, for a list of all operators that ProAsm recognizes.)

#### illegal public symbol (xdef) definition

You try to declare a symbol as externally visible (using XDEF) that is neither a label nor a numeric symbol. Check all symbols in the operand field of the particular XDEF directive for their types. Not allowed as arguments of the XDEF directive are symbols of the following types: any local symbols, register lists masks, textual symbols (EQR), external references (XREF), floating-point constants, and undefined symbols.

#### illegal symbol name

A symbol with an illegal name was encountered. Check the spelling of the symbol in question (you may refer to Labels for more details about symbols and their formats).

#### illegal type combination

An attempt was made to combine quarrelsome symbol types and/or constants in an arithmetical expression. Consider the following example:

```
        foo      FEQU      2
        bar      EQU       8

        move.l   #foo*bar,d0    ; => error
```

Check all symbols and constants in the expression for their correct type.

#### illegal use of external reference (xref)

An operation will not allow an external symbol to be used as operand. Refer to the description of the particular directive or

instruction for more information about its usage.

#### illegal use of inside 'inceu' defined symbol

Files included using the INCEQU directive have restrictions on their content, so have symbols defined in these files. These restrictions come from the concept of the INCEQU directive. Files included into the main source code using INCEQU are read and processed in the first pass only. If you try to change an inside 'inceu' defined symbol, this symbol cannot be set back to the initial value during the next pass. (See  
     INCEQU  
   ).

Use the INCLUDE directive if you have to redefine symbols defined in external files.

#### illegal value

The expression evaluator could not find a valid value: a symbol, constant, or an expression. An invalid prefix to a number or a bad symbol name in an operand may also generate this error message. For example:

```
move.l #14+),d0      ; => error
move.l ##14,d0      ; => error
move.l #{2+3}*4,d0  ; => error
```

#### immediate value expected

Some instructions of the M68000 family require immediate data as source operand. The value is specified by the expression following the pound sign #. Either the pound sign is missing or the wrong instruction is used. The following examples generate this message:

```
subq.l 3,d0
trap #8
andi %1101011,d2
```

#### invalid argument for this keyword

An argument of a keyword is invalid. Refer to the description about this keyword to get more details about what type of argument is requested.

Consider the following example:

```
foo MACRO
dc.b \*STRING(\1),0 ; textual symbol as argument expected
EVEN
ENDM

foo 1                ; => error
```

#### invalid default mode

A DEFAULT directive was encountered with an invalid mode argument that cannot be combined with the given type argument.

Refer to the description of the DEFAULT directive for more detailed information about the default types and their accepted modes.

#### invalid k-factor

The k-factor of the FMOVE.P instruction or the SETKFACTOR directive is out of range. The range of a k-factor must be from -64 to +63. (See

```
    SETKFACTOR
    ).
```

#### invalid number

An invalid digit was encountered in a number. For example, using a 2 or 3 in a binary number:

```
    move.l  %#1010201,d0
```

#### invalid option

The options specified does not exist. (See

```
    OPT
    , for an overview
```

of all supported options.)

#### invalid parameter

A directive was encountered with one or more invalid parameters in its operand field. See the description of the directive that generated this message for full details about the accepted parameters.

#### invalid parameter line, consult ProAsm manual for more details.

The parameter line has not the requested format, or one or more unknown parameters have been encountered. Start ProAsm with the question mark (?) as only argument to get a small usage information displayed, or refer to Invoking for a complete list of all supported command line parameters and the accepted command line format.

#### invalid section memory requirement

Only CHIP, FAST, PUBLIC, MEMF, and ADVISORY are allowed as valid section memory requirements. (Refer to SECTION for more detailed information about these memory requirements.)

#### invalid section type

Only CODE, DATA, and BSS are allowed as valid section types. (Refer to SECTION for more details.)

#### invalid size

An instruction has either no size at all or it does not support the used size. Example:

```
    add.p   #12,d0   ; invalid size packed for the add
                    ; instruction
```

#### label from other section

This error occurs if you try to make PC relative references to another section. To fix this error avoid PC relative references, try to use absolute addressing modes

Consider the following example:

```
SECTION "bar",code
jsr    foo(pc) ; relative jump => error,
...           ; use jsr foo instead.

bra    foo     ; relative jump => error,
```



```
...                ; use jmp foo instead.
```

```
SECTION "foobar",code
foo:
```

#### label value changed between passes

This indicates an assembler error where the label does not have the same value on pass 2 as it does on pass 1.

This error may also be caused by a misuse of the IF1 or IF2 directive (see IF1).

#### line malformed

The line that caused this error message has not the prescribed form. Each line has the following general layout:

```
label mnemonic operands comment
```

(For more details about the line format see  
Source Line Format  
.)

#### linker format restriction

This error occurs only if the linkable object file format was changed to `_RELATIVE` (see  
DEFAULT  
) and a 32-bit data section  
relative relocation (`hunk_dreloc32`) should be generated.

Since the linker reference mode `_RELATIVE` is implemented only for compatibility with some older (non-standard) linkers, this error message should not be a problem at all. These linkers do not support the `hunk_dreloc32` hunk format.

If you are using either `ALINK` (the Amiga standard linker) or `BLINK` (the replacement linker from 'The Software Distillery') the linker reference mode should be set to `_BASERELATIVE` (with `DEFAULT` `_LINKREF,_BASERELATIVE`).

Note that ProAsm defaults to `_BASERELATIVE`.

#### local label not allowed

The directive that caused this error does not allow local labels as arguments. Refer to the description of this directive for more detailed information.

For example:

```
linkable
xdef .foo,1$ ; local symbols cannot be exported.
...
.foo:
1$:
```

#### locked symbol used

A symbol that was marked previously as "locked" using the `LOCKSYM` directive is used within an expression or addressing mode. Check why the symbol is "locked", or unlock it using the `UNLOCKSYM`

directive (see UNLOCKSVM).

macro defined twice

The macro label name is multiply defined in the same program. You will need to change at least one of them to a unique name.

MC68040 software-supported instruction used

A floating-point instruction is used that has to be emulated by Motorola's floating-point software package on the MC68040 processors to ensure complete compatibility to the floating-point coprocessors.

You can either enable the software-supported instructions using the OPT FPSP40 option (see OPT FPSP40) or remove this instruction and replace it by a subroutine (or whatever).

Note that by default all MC68040 software-supported instructions are enabled.

MC68060 software-supported instruction used

An integer or floating-point instruction is used that has to be emulated by Motorola's software package on the MC68060 processors to ensure complete compatibility to older processor models and the floating-point coprocessors.

You can use the OPT SP60 option (see OPT SP60) to enable the MC68060 integer instruction emulation, or the OPT FPSP60 option (see OPT FPSP60) to enable the floating-point software-supported instructions.

Note that by default all MC68060 software-supported instructions are enabled.

misplaced ELSE/ELSEIF encountered

An ELSE or ELSEIF directive was encountered when not currently in conditional assembly.

misplaced ENDIF/ENDC encountered

An ENDC or ENDIF directive was encountered when not currently in conditional assembly.

missing close bracket

An unmatched open bracket (()) was found in an expression.

missing ENDIF/ENDC

An IF directive is encountered without a matching ENDIF/ENDC directive. This error commonly occurs if an END directive within a matching IF - ENDC block was found.

For example:

```

foo      EQU      0
      ...
      IFEQ      foo
      ...
      END              ; => error
      ENDC

```

(See  
     Conditional Assembly  
     , for more information about conditional  
 assembly directives and their use.)

#### missing quote

An unmatched quote (single or double quote) was found in a string. Make sure the string is correctly enclosed in quotes. You may not mix single and double quotes.

Remember you must use the sequence " or "" to represent a single respectively a double quote inside a string. If escape sequences are allowed (see

    ESCAPESTR  
     ), a quote can also be represented by  
 either \' for a single or \" for a double quote.

#### Examples:

```
dc.b    "'Ouch!'",0      ; => 'Ouch!'
dc.b    '''Ouch!''',0   ; => 'Ouch!'

ESCAPESTR
dc.b    "'\Ouch!\'",0   ; => 'Ouch!'
dc.b    '\Ouch!\'',0   ; => 'Ouch!'
```

#### MMU register expected

An operand that should be a MMU register is not. For example:

```
lreg    EQU    TC
lregn   EQU    1
...
pmove   (a0),lregn ; probably symbol names changed by mistake
```

#### more than 2147483647 IFs nested!

Conditional assembly directives can "only" be nested up to 2147483647 times. (See

    Conditional Assembly  
     , for more information  
 about the conditional assembly directives.)

#### negative value expected

An operand to a NB, NW, or NL directive was positive. Change the value or use a normal DC directive instead. (See

```
    Initialized Data with Restricted Range  

      .)
```

#### no code or data allowed ahead of the HEADER directive

The HEADER directive must be at the beginning of a source code. No code or data generating directives are allowed ahead of the HEADER directive. To fix this error, move the directive to the top of the source code or replace it by the INCLUDE directive.

#### no label on line

No label was specified for the MACRO directive. The given label represents the macro body later in the code:

```
CALL_   MACRO
        jsr    \1
```

```

                ENDM

                dots
                CALL_ PrintTitle
                dots
    (More about macros can be found on
     Macros
     .)

```

no MACRO definitions within REPT/REPEAT loops

Macro definitions within repeat loops are not allowed, since the same macro would be defined more than once. Just move the macro definition outside the repeat loop.

no relocation in binary/s-record file

An absolute long addressing mode or a reference that needs a relocation was encountered (see Relocatable Symbols).

To avoid relocation within binary/s-record files, you can use PC-relative addressing modes (see

```

    Addressing Modes
    ) or the ORG

```

```

directive (see
    Absolute Assembly
    ).

```

See also the description of the MEA pseudo-opcode, that can make program counter relative assembly easier (see Move Effective Address). If you do not want to use this pseudo-opcode for certain problems, you have also the possibility to split up the instruction that generated the error message in two or more instructions. Look at the following example:

```

    BINARY

    move.l #foo,d0          ; => error

    split it up to:
    pea   foo(pc)
    move.l (a7)+,d0

```

no size expected

An unsized instruction was assembled with a size specifier. For example:

```

    nop.l    ; => error

```

not allowed in binary/s-record file

The directive that caused this error is not allowed if a binary or s-record output file is generated. Refer to the description of the particular directive for more detailed information.

not a valid effective address

A MOVEM or FMOVEM instruction was encountered with an invalid effective address. Refer to the description of the instruction to get details about the allowed addressing modes.

A misuse of the given register lists may also generate this error message. Consider for example:

```

foo      REG      d0-d7/a2
bar      REG      d0-a6

```

```

movem.l bar,foo(a5)

```

not a valid FPU control register list

The control register list in a FMOVEM instruction or FREG directive is malformed. A register list must contain one or more FPU control registers (fpcr, fpiar, fpsr). The registers must be separated by a slash (/) or a minus sign (-).

not a valid FPU register list

The register list in a FMOVEM instruction or FREG directive is malformed. A register list must contain one or more FPU data registers (fp0-fp7). The registers must be separated by a slash (/) or a minus sign (-).

not a valid register list

The register list in a MOVEM instruction or REG directive is malformed. A register list must contain one or more registers. The registers must be separated by a slash (/) or a minus sign (-). Consider the following example:

```

foo      REG      d0/d2-d4/d7-a2/a4-a5

```

The symbol foo is a register list mask for the registers d0,d2 through d4, d7 through a2, a4, and a5.

no value given

No term given, or the expression evaluator could not find a valid term. A symbol, constant, or expression might be invalid.

no XPK archives supported

You tried to load an XPK archive file using a file including directive (such as INCLUDE). ProAsm accepts only normal source files or XPK packed files to be included into the program source.

object filename already specified

More than one OUTPUT or OBJFILE directive was present in the source code, specifying conflicting filenames. Find the conflicting directives and eliminate one of them, or change the filenames to match.

object size differs between passes

The object size does not have the same value after pass 2 as it had after pass 1.

This error may be caused by a misuse of the IF1 or IF2 directive (see IF1).

odd address

Instructions of the M68000 Family must always be on a word (even) address. Use the alignment directives (see Alignment Padding) to

ensure word boundary. For example:

```

DC.B      "Hello world!",0 ; 13 byte long string...
EVEN      ; the EVEN directive ensures word alignment
moveq    #0,d0

```

## opening brace '{' expected

The assembler expected an opening brace } but did not find one. Bitfield manipulating instructions expect bitfield selections straight after the last operand. This message may be generated if too many operands were given for such an instruction. For example:

```
bftst    d0,d11:6      ; too many operands for bftst
```

## operand too large, byte expected

The operand, in most cases an immediate addressing mode, is too large to fit in the expected byte size. For example

```
move.b   #300,d0      ; value not within a signed 8 bit range
```

## operand too large, word expected

The operand, in most cases an immediate addressing mode, is too large to fit in the expected word size. For example

```
move.w   #100000,d0   ; value not within a signed 16 bit range
```

## option must be at beginning

The used option must be placed at the beginning of the source code. This error occurs if a symbol concerning option (such as CASEON, OPT C, OPT ESS1 etc.) is used after a symbol is defined. Just move this option to a place before the first symbol is defined:

```

; begin of source code
foo:    ...
        OPT      C-          ; invalid location in the source code

; begin of source code
        OPT      C-          ; correct location in the source code
foo:    ...
```

## out of memory

There is not enough memory available for the assembler to complete its task. To get more memory you probably have to quit other running applications, etc. If you still have not enough memory, cut the program into smaller portions, assemble the portions of the program separately and then bind them together using a linker.

## overflow

This error is reported if an overflow occurs while evaluating an expression. In other words, the result of the evaluated expression is larger than \$ffffffff, +\$7fffffff, or -\$80000000.

For example:

```
move.l   #$7fffffff+1,d0 ; => overflow
```

## PC value changed between passes!

This error message occurs rarely and indicates an inconsistency; mostly by abusing the IF1 and IF2 directives. No code or data generating instruction or directive should be inside a IF1/IF2 condition statement (see IF1). Consider the following example:

```
IF2
DC.L "this text is placed very unhappily"
ENDC
```

It is also unwise to make constructions as shown in the following

example:

```
IF1
DS.B 8,0
ENDC

IF2
DS.B 8,$20
ENDC
```

No error message will be reported by the assembler, since the program counter does not differ. However, such hacks may complicate the task of finding any bugs in the code.

positive value expected

An operand to a PB, PW, or PL directive was negative. Change the value or use a normal DC directive instead. (See

```
    Initialized Data with Restricted Range
    ).
```

preasm file expected

You tried to load a file using the MACLIB directive that is not a preassembled file. The MACLIB directive is used to import symbol and macro definitions only from preassembled files. (See

```
    MACLIB
```

for more information about this directive.)

premature end byte reached

An end of file (EOF) was encountered while the assembler expected more source code. This error occurs also if a null byte was found in the source file. Examples:

```
addq.l #2,<NULL>d0
move.l #12<EOF>
```

probably immediate addressing mode

An absolute addressing mode as source operand was given where it was not supposed to be. This error will only reported when the OPT I+ option is selected. There is probably a pound sign # missing. You can force the assembler to use absolute addresses by appending one of the absolute address size specifiers .L or .W.

Example:

```
OPT      I+
move.l  4,a6          ; no error will be reported
add.l   $100,d0       ; immediate addressing mode was probably meant
move.l  $f80000.L,d0  ; no error will be reported
```

quite a heap of code (PC overflow!)

The code you tried to generate was greater than 4294967295 bytes. (Since the Amiga is a 32 bit machine you cannot access more than 4 GBytes of memory.) You probably used one or more DS directives (or its synonyms) with too large arguments.

Consider for example:

```
...
rts
```

```
ds.b $7fffffff,0 ; => error
```

register or immediate value expected

An operand that should be a register or immediate value is not.

For example:

```
MC68851
```

```
pflushs (a0),#0 ; => error
```

In the example above, the first operand should be either a data register, a sfc, or a dfc register, instead of the address register indirect addressing mode ((a0)).

If an instruction causes this error message, refer to the 'M68000 Family Programmer's Reference Manual' for detailed information.

relative not allowed

The operation requires that the operand is not relative (not a label).

relocation not allowed (OPT P+/A+/T+, DSEG)

An instruction or directive was encountered that produced a relocation (absolute addressing modes, etc.).

Check the use of the OPT P+, OPT A+, or OPT T+ options and/or the DSEG directive. These options and this directive dictate the assembler a certain behaviour when a relocation is encountered.

Remove these options and directive or avoid relocation. To avoid relocation you can use PC-relative addressing modes (see

```
Addressing Modes
), or the ORG directive if you want to produce
absolute code (see
Absolute Assembly
).
```

See also the description of the MEA pseudo-opcode that can make program counter relative assembly easier (see Move Effective Address). If you do not want to use this pseudo-opcode for certain problems, you also have the possibility to split up the instruction that generated the error message into two or more instructions. Look at the following example:

```
BINARY
```

```
move.l #foo,d0 ; => error
```

split it up to:

```
pea foo(pc)
move.l (a7)+,d0
```

reserved register

A processor register name was used as a symbol name. (Refer to

```
Registers
, for an overview of all supported processor registers.)
```



**reserved symbol**

You tried to redefine a reserved symbol such as `__DATE`. (See

Special Symbols  
.)

**short branch out of range (by number bytes)**

A short branch was specified outside of the range that can be reached by one byte (-128-127). Use a branch or jump instruction which can access a larger range.

**source and object files are the same**

The name of the specified output file is identical to the name of the processed source file. This is to prevent the source file from been overwritten.

**source expired prematurely**

An END directive or a zero byte (end of source code) was found during conditional assembly within a macro body or a repeat loop. In most cases an ENDC, ENDM, ENDR, or an UNTILcc directive is missing.

**spurious APOPM encountered**

An APOPM pseudo opcode was encountered without matching APUSHM.

**spurious ENDORG encountered**

An ENDORG directive is encountered when not currently in absolute assembly mode (see  
Absolute Assembly  
).

**spurious ENDR or UNTILcc encountered**

An ENDR or UNTILcc directive is encountered outside a repeat loop (see  
Repeating Text  
).

**spurious REXIT encountered**

A REXIT directive is encountered outside a repeat loop (see  
Repeating Text  
).

**string expected**

An operand that should be a string is not a string. Refer to the description of the directive that caused this error for more information.

**string too long**

This error occurs if a given string is longer as expected by the directive. See the description of the particular directive for more information.

**symbol defined twice**

You tried to use the same symbol name in two different places in the same program. You will need to change at least one of them to a unique name.

---

If you find yourself accidentally reusing symbols often, you may want to use local symbols more often (see Local Labels). Using local symbols as much as possible can greatly reduce the number of symbols you have to keep track of throughout your program.

symbol expected

No symbol was specified for a directive that must have one. Refer to the description of the directive that caused this error message for more information.

symbol not defined

A symbol is used that is not defined in the source code.

symbol name too long

The symbol name was longer than 256 characters.

too many close brackets

Too many close brackets (}) are found. Either more open brackets have to be inserted or the superfluous close brackets have to be removed to get properly balanced brackets.

too many brackets

The assembler supports brackets to be nested up to 255 levels. This error occurs if more brackets are nested within an expression, than accepted by the assembler.

too many sections (max 256)

The assemblers allows the definition of maximal 256 different sections. This error occurs if you try to define more sections as allowed. (See SECTION, for more information about defining sections.)

unable to define preassembled symbol 'symbol name'

The symbol 'symbol name' in the preassembled file may be already defined or has an illegal name. (Refer to Labels for more details about symbols and their formats.)

Check your source code for any, already defined, symbols that are named as symbol name.

unable to define symbol

This error is reported when a symbol is not correctly defined via command line parameters (-S/DEFINE and -T/STRING, Invoking)

Check the spelling of the symbol in question (you may refer to Labels for more details about symbols and their formats), and ensure the correct use of the command line parameters.

unable to import symbol

An attempt was made to import a symbol (using the XREF directive) that is already defined or has an illegal name. (You may refer to Labels for more details about symbols and their formats.)

unable to open list file 'filename' (DOS error code)

An AmigaDOS error occurred while trying to open the listing file. Refer to the AmigaDOS error codes listed below for detailed

---

information.

#### undefined symbol

You tried to use a symbol that has not been defined in the program. Check if the symbol is not misspelled in either its definition or its reference.

One common cause of this error is referencing a symbol with different case than it is defined while case sensitivity is turned on, as this example demonstrates:

```

Foo:
    ...
    bsr    foo    ; Error: Undefined symbol

```

To fix this, turn case sensitivity off (see Case Sensitivity) or change the symbol names to match exactly.

Another common cause of this error is trying to reference a local label "across" a global label. This often happens just after inserting a new global label in the middle of existing code. In the following example, an entrypoint bar has just been inserted into the routine foo:

```

foo:
    ...
.loop:
    ...
bar:
    ...
    bra    .loop    ; Error: Undefined symbol

```

To fix this, turn the global label (bar in this example) into a local label, or turn the local label (.loop) into a global label.

#### unexpected ENDM or MEXIT encountered

An ENDM or MEXIT directive was encountered when not inside a macro definition.

#### unknown keyword for this directive

A directive with an unknown keyword was encountered. See the description of the directive that generated this error message for information about the allowed keywords.

#### unrecognized default mode

This error is reported if the mode argument of the DEBUG directive is not recognized by the assembler. Check the spelling of the mode argument. (See also

```

    DEFAULT
    , for more information about
the DEFAULT directive and its arguments.)

```

#### unrecognized default type

This error is reported if the type argument of the DEBUG directive is not recognized by the assembler. Check the spelling of the

---

type argument. (See also  
    DEFAULT  
    , for more information about  
the DEFAULT directive and its arguments.)

unrecognized instruction

unrecognized instruction 'instruction'

The assembler did not recognize the instruction mnemonic. Check the spelling of the instruction or the macro.

unresolved symbol 'symbol' found

This error is reported if circular references are found while multi-pass optimization is enabled.

unsupported preasm file entry encountered

An unsupported entry is encountered while processing the preasm file. This error only occurs if the included preasm file is crippled.

Try the same assembly job with a new generated preasm file. (See PREASM, for more information about this directive.)

unsupported preasm file version

You tried to load an unsupported version of a preasm file.

Try the same assembly job with a new generated preasm file. (See PREASM, for more information about this directive.)

user error

The FAIL directive was assembled (see FAIL, for more details).

value between 1 and 8 expected

An operand has to be within the range of 1-8. Refer to the description of the particular directive for more information. If an instruction causes this error message, refer to the 'M68000 Family Programmer's Reference Manual' for detailed information about this instruction.

value must be zero

An operand that should result in a value of zero is not. Refer to the description of the directive that caused this error for more information.

For example:

```
...  
DX.B 6,4 ; => error, DX.B 6,0 expected.
```

value not in bounds

An operand is not within valid bounds as dictated by the directive. Refer to the description of the particular directive for information about the range of accepted values.

word displacement out of range (by number bytes)

A word displacement, as for address register indirect with displacement and program counter indirect with displacement addressing modes, is not within the range of -32768-32767 by number of bytes. (See

Addressing Modes  
 , for more information  
 about word displacements and addressing modes.)

XPK error, XPK specific error message

This error is displayed when the XPK software package returns an error to ProAsm. Refer to the XPK software documentation for more information about errors and their causes.

\*\* error occurred trying to create error file 'filename' (DOS error code)  
 An AmigaDOS error occurred while creating the error file filename.  
 (See

AmigaDOS Error Codes  
 , for a list of the AmigaDOS error codes.)

\*\* Panic! premature end byte reached

An unexpected null byte (end of file) is encountered while the assembler expected more source code. Please check your source code for any misplaced null bytes or crippled instructions at the end of the source code.

INTERNAL: ...

These are internal error messages which should never occur. If the assembler does report one of them, please report it. Please provide as many details as possible about the situation in which the error occurred; this will greatly aid correcting the problem (see ERRFLAG, to get more information about the error).

## 1.82 pro.guide/Warnings

Warnings

=====

This appendix lists in alphabetical order the warning messages generated by ProAsm, and describes their cause and meaning. Although you can often safely ignore warning messages, sometimes they indicate a problem; in general you should at least examine each warning once and make sure you understand it. Fixing the code, so the warning is not generated at all is usually better, though.

ADDA/SUBA #0,An instruction skipped

This warning is generated when the assembler skipped an adda or suba instruction with an immediate source operand of zero. This optimization is only made if the Q4 option is turned on (see

Optimization  
 ).

These instructions can be skipped unproblematically, since they do not affect the condition codes and an addition or subtraction of zero has no effect to the address register.

absolute long address mode made short

The assembler optimized an absolute long address to its short form (absolute word address). Only absolute addresses are optimized.

This optimization is only made if the O3 option is turned on (see

Optimization  
).

absolute short address mode possible

The assembler encountered a forward referenced absolute long address that can be shortened to absolute word. You may optimize it yourself to save two bytes. This optimization possibility is only reported if the O18 option is turned on (see

Optimization  
).

APUSHM and APOPM directives converted to (two) NOPs each

The assembler encountered a pair of APUSHM and APOPM pseudo-opcodes with empty register lists and replaced each of them with two nop instructions.

(See Stack Manipulation, for the descriptions of the APUSHM and APOPM pseudo-opcodes.)

ASL/ROXL #1,Dx converted to ADD/ADDX Dx,Dx

This warning is issued when the assembler could convert a `asl.y #1,dx` instruction to `add.y dx,dx`, or a `roxl.y #1,dx` instruction to `addx.y dx,dx`. This optimization is only made if the Q2 option is enabled (see

Optimization  
).

AUTOXREF only in linkable files allowed

You used the AUTOXREF directive while generating non-linkable output, like an executable. (See

Defining and Referencing External Symbols  
, for the description of  
the AUTOXREF directive.)

base displacement shortened

The assembler reports this message when a base displacement within the range of -32768-32767 is optimized from long to word size. This optimization is only made if the O13 option is enabled (see

Optimization  
).

To suppress this optimization you may add a longword size specifier (.L) to the base displacement.

bit number should be from 0-7

bit number must be from 0-31

These warning messages only occur when OPT WARNBIT was selected. OPT WARNBIT forces the assembler to be more strict when checking the immediate bit number for the bit manipulating instructions (`bclr`, `bchg`, `bset`, and `btst`). The accepted bit number depends

on the destination addressing modes specified. If the destination is a data register, the bit numbering is from 0 to 31. And for memory location, the bit numbers are from 0 to 7.

Generally this additional check is not needed, since the processor uses the bit number modulo 8, respectively 32. But it may be useful if you want to be informed where immediate bit numbers are "out of range."

Example:

```
OPT      WARNBIT
btst    #10,foo(a5)
```

(See OPT WARNBIT, for the descriptions of this option, and refer to 'M68000 Family Programmer's Reference Manual' for detailed information on the bit manipulating instructions.)

branch made short

A branch was optimized to its short form (see Optimization, for information about the O1 option).

CLR.L converted to MOVEQ

The assembler converted a clr.l dn instruction to moveq #0,dn, where dn is a valid data register. (See Optimization, for more information on the Q1 option.)

CMPI.x #0,<EA> converted to TST.x <EA>

If the O9 option is turned on (OPT O9+), the assembler converts cmp, cmpi, and cmpa instructions with an immediate source operand of zero to a tst instruction. (See Optimization.)

code ahead of the ORG directive

The assembler found code or data ahead of the ORG directive. Using the ORG directive while generating executable or linkable object code is not recommended (see Absolute Assembly). However, this warning message can be ignored when the use of absolute code within executable or linkable code is desired.

code/data is not allowed in preasm files, ignored

Since a preasm file contains only the symbol and macro table, all code and data generating directives are useless and therefore ignored by the assembler. Only their sizes are taken into account to update the program counter. (See PREASM, for the description of this directive.)

DX statement in empty hunk found

This warning message is reported when a DX directive is found within an empty code or data hunk. A code or data hunk that contains only DX directives, is in its function equivalent to a

bss hunk. You may use a bss hunk instead. (See 'The AmigaDOS Manual' for more information about AmigaDOS hunk formats.)

EOR(I).x #-1,<EA> converted to NOT.x <EA>

The assembler converted an eor or an eori instruction with an immediate source operand of zero to a not instruction. This conversion is only made if the Q12 option is turned on (see

Optimization  
) .

error file already specified, directive ignored

This warning is reported if more than one ERRFILE is present in the source code, specifying conflicting filenames for the error file. (See The Error File, for more information about the use of this directive.)

external references (xref) only in linkable files allowed

You used the XREF directive while generating non-linkable output, like an executable. If you actually try to use the symbol defined as XREF, an error will occur. (See

Defining and Referencing External Symbols  
, for the description of  
the XREF directive.)

f-line exception will be generated!

This warning messages is issued when the assembler encounters a ptest that specifies an address translation cache search with an address register operand while the MC68030 processor is selected. The MC68030 processor takes an F-line unimplemented instruction exception when this ptest instruction is executed. (Refer to the 'M68000 Programmer's Reference Manual' for more details about the ptest instruction on a MC68030 processor.)

For example, the following code fragment generates this warning:

```
MC68030
ptestr d0,4(a2),#0,a0 ; => warning
; level 0 specifies searching the address
; translation cache only.
```

instruction has no effect on a MC68EC030

The MC68EC030 executes all instructions just as the MC68030 does, except that the MC68EC030 does not execute all memory management unit instructions. The pflush and pflushn instructions have no effect when executed. (Refer to the 'M68000 Family Programmer's Reference Manual', for complete details on the MC68EC030 instruction set.)

instruction made quick

This warning is reported when the assembler optimized an instruction to its 'quick' form. Instructions with quick variants are: add (addq) and sub (subq). (See

Optimization  
, for more  
details O5 option.)



**invalid LINK displacement**

The assembler found a positive or odd link displacement. Example:  
link #16,a5

**long MOVEA/ADDA/SUBA converted to word**

A movea.l, adda.l, or suba.l instruction with an immediate value within the range of -32768-32767 is shortened to word operation size.

This optimization is only made if the O8 option is enabled (see

Optimization  
).

**Motorola reserved Constant ROM offset used**

The fmovecr instruction, that causes this warning, uses a Constant ROM offset that is reserved by Motorola. (Refer to the 'M68000 Family Programmer's Reference Manual' for a list of valid offsets and their meanings.)

The use of reserved constants is not recommended, since they are useful only to the on-chip microcode routines and may be different on various mask sets of the floating-point coprocessor.

**MOVEA #0,An converted to SUBA An,An**

This warning message is reported when the assembler optimized a movea instruction with an immediate value of zero to suba an,an.

This optimization is only made if the Q3 option is enabled (see

Optimization  
).

**MOVE.L converted to MOVEQ**

This warning is reported when the assembler optimized a move.l instruction to its 'quick' form (moveq).

This optimization is only performed if the O4 option is turned on. (See

Optimization  
, for more details O4 option.)

**MOVE.L #n,Dx converted to MOVEQ #m,Dx & NOT**

The assembler optimized a move.l #n,dx instruction to a shorter form by splitting it in a combination of a moveq #m,dx and a not instruction.

This optimization is only made if the Q5 option is enabled (see

Optimization  
).

**MOVE.L #n,Dx converted to MOVEQ #-128,Dx & SUBQ.L #n+128,Dx**

This warning is issued when the assembler optimized a move.l #n,dx instruction to a shorter form by splitting it in a combination of a moveq #-128,dx and a subq.l #n+128,dx instruction.

This optimization is only made if the Q7 option is enabled (see

Optimization  
).

MOVE.L #n,Dx converted to MOVEQ #m,Dx & SWAP Dx

The assembler optimized a move.l #n,dx instruction to a shorter form by splitting it in a combination of a moveq #m,dx and a swap dx instruction.

This optimization is only made if the Q8 option is turned on (see

Optimization  
).

MOVE.L #n,Dx converted to MOVEQ #m,Dx & ADD.B Dx,Dx

This warning message is reported when a move.l #n,dx instruction was optimized to a shorter form by splitting it into a combination of a moveq #m,dx and a add.b dx,dx instruction.

This optimization is only made if the Q9 option is enabled (see

Optimization  
).

MOVE.B #-1,<EA> converted to ST.B <EA> (CCR changed!)

The assembler optimized a move.b #-1,<EA> instruction to a st.b <EA> instruction. This optimization is only made if the Q11 option is turned on (see

Optimization  
).

Caution, the move.b #-1,<ea> instruction affects the condition code flags, while the st.b <ea> instruction does not. (Refer to the 'M68000 Family Programmer's Reference Manual' for more details about these instructions.) To avoid any problems you should ensure that no condition code afflicted instructions (such as bcc, dbcc, and scc) follow directly the optimized instruction.

no code allowed in BSS sections

A M68000 instruction or any other code or data generating directive was found inside a BSS section. The BSS section specifies a block of uninitialized workspace that is allocated by the AmigaDOS loader, therefore no specific data is stored in a BSS section. This warning is only reported once, even if there is more code or data generating directives inside a BSS section. The assembler ignores these directives and just uses their size to reserve the appropriate space. To reserve space inside a BSS section correctly, the following directives (or their synonyms) should be used:

DS.size expression[,0]

DX.size expression[,0]

DC.size 0

SB 0, SW 0, SL 0

PB 0, PW 0, PL 0

UB 0, UW 0, UL 0

(For detailed information about these directives see

```
    Initialized Data,  
        Initialized Data with Restricted Range  
        and
```

```
        Declaring Data Blocks  
    .)
```

no external references (xref) in preasm file, ignored

An external reference (xref) is encountered while a preasm file is generated. Such references are only allowed in linkable files. Since a preasm file contains only the symbol and macro table, all external references will be useless, and therefore ignored by the assembler. (See PREASM, for the description of this directive.)

no section statements in preasm files, ignored

A preasm file contains only symbol and macro definition in a preassembled form. No code and data is written to this file, therefore any section directives have no effect, and are ignored by the assembler. (See PREASM, for the description of this directive.)

object filename already specified

More than one OUTPUT or OBJFILE directive is present in the source code, specifying conflicting filenames. Find the conflicting directives and eliminate one of them, or change the filenames to match.

odd address or offset detected

This warning is issued if the assembler encounters a CCNOP directive that needs an odd pad length to obtain the desired alignment. CCNOP pads the program counter to the desired boundary by inserting nop's. If the pad length is odd, a null-byte has to be inserted first by CCNOP. (See CCNOP).

offset removed

An address register indirect addressing mode with a displacement of zero is optimized to the address register indirect addressing mode.

This optimization is only made if the O2 option is enabled (see

```
    Optimization  
    ).
```

other case sensitivity or significance in preasm file

The symbols stored within the preasm file have another case dependency than you currently use. This can have unwished sideeffects: symbols that are expected to be case-sensitive can be converted to case-insensitive and vice versa. Note that case-sensitive symbols are stored in upper case in the preasm file. (See

```
    Case Sensitivity  
    , and PREASM.)
```

outer displacement shortened

The assembler reports this message if an outer displacement within the range of -32768-32767 is optimized from long to word size.

---

This optimization is only made if the O14 option is enabled (see Optimization ).

To suppress this optimization you may add a longword size specifier (.L) to the outer displacement.

public symbols only in linkable files allowed

A public symbol definition is encountered but no linkable file is generated. Refer to the description of the PUBLIC directive for more details ( Defining and Referencing External Symbols ).

Read-Modify-Write cycle opcode

This warning is reported if a cas, cas2, or tas instruction was encountered. Since the system DMA can conflict with this instruction's special indivisible read-modify-write cycle. Use the READMODWRITE directive to control this warning (see

READMODWRITE ).

relative possible

This warning is reported when the assembler encountered a forward referencing absolute long addressing mode that can be made PC-relative.

You may optimize it yourself to save two bytes. This optimization possibility is only reported if the O16 option is turned on (see

Optimization ).

relocation converted to relative

This warning message is issued if a backward referencing absolute long addressing mode (within the range of -32768-0) is optimized to PC-relative addressing mode. This optimization is only made if the O7 option is enabled (see

Optimization ).

To suppress this optimization you may add a longword size specifier (.L) to the absolute long address.

relocs won't be written in binary file!

A binary file is a raw binary image without any structures attached to it, like hunks, etc.. Consequently any reloc information cannot be stored in such a file. Note that relocation may be needed by your code. A binary file generation that produced this warning message may not work properly.

To avoid this warning try to use relative addressing modes for instructions. For relocatable data structures you may use offset oriented data.

---

#### short branch converted to NOP

The destination of a short branch was the following instruction, which is an illegal branch offset. The branch was replaced by a nop instruction.

#### short branch possible

The assembler encountered a forward referenced branch instruction that can be shortened. You may optimize it yourself to save two bytes. This optimization possibility is only reported if the O6 option is turned on (see Optimization).

#### sign-extended operand

A moveq instruction is encountered with a positive immediate value within the range of 128-255. The moveq instruction expects an 8-bit (-128-127) immediate value that, during execution, is sign-extended to long (32-bit). Thus any 8-bit number greater than 127 will automatically become negative when sign-extended to long. To suppress this warning you have to add a longword size specifier (.L).

(Refer to the 'M68000 Programmer's Reference Manual' for the description of the moveq instruction.)

#### size should be word

A branch instruction with a longword operation size is encountered while assembling code for the MC68000 (or MC68010) processor.

It is recommended to change the operation size to word (or to remove the size specifier), since long branches are not supported by the MC68000 and MC68010 processors. This can avoid conflicts when you once decide to use also code for later processors (MC68020 or higher).

#### supervisor only opcode

This warning is reported if a privileged instruction was encountered. Use the SUPER directive (or its synonyms) to control this warning (see SUPER).

#### this section will be empty

This warning message informs you that the assembler encountered an empty section (no code and data). An empty section is a code, data or bss section with no size. In all cases such a section is useless and can be removed from the object file.

#### unable to load the proasmoptim.library - option ignored

The proasmoptim.library could not be found. The library is probably not correctly installed to your system. (See The proasmoptim.library, for more information. The option (OPT OPTIMLIB) is ignored.)

#### user warning

The WARN directive was assembled. (See WARN, for details.)

---

XDEF is not allowed in preasm files, ignored

A XDEF directive is encountered while a preasm file is generated. Such external symbol definitions are only allowed in linkable files. Since a preasm file contains only the symbol and macro table, all external symbol definitions will be useless, and therefore ignored by the assembler. (See PREASM, for the description of this directive.)

XDEF only in linkable files allowed

A XDEF directive is encountered while generating non-linkable output such as an executable. (See XDEF.)

zero offset

The assembler encountered a displacement of zero of an address register indirect with displacement addressing mode that can be removed. You may optimize it yourself to save two bytes. This optimization possibility is only reported if the O17 option is turned on (see Optimization).

zero space defined

A DS or DX directive is encountered that reserved no memory space. For example:

```
DS.L 0,4
```

You may have made a typing error. Note that DS.L 0 and DS.W 0 with no second argument can be used to align data (see

```
Alignment Padding
).
```

## 1.83 pro.guide/AmigaDOS Error Codes

AmigaDOS Error Codes

=====

This appendix lists the possible AmigaDOS errors. For detailed information about the AmigaDOS and the AmigaDOS errors, refer to 'The AmigaDOS Manual', from Commodore-Amiga Inc.

103

Not enough memory

Not enough memory in your Amiga to carry out the operation.

104

Process table full

The limit of the maximal number of possible processes is reached.

114

Bad template

Incorrect command line.

115

---

Bad number  
The program was expecting a numeric argument.

116  
Required argument missing  
Invalid command line, an argument that was required was not given.

117  
Argument after "=" missing  
Invalid command line.

118  
Too many arguments  
Invalid command line, too many arguments given.

119  
Unmatched quotes  
Invalid command line.

120  
Argument line invalid or too long  
Invalid command line.

121  
File is not an executable object  
Misspelled command name, or file may not be a loadable program or script file.

122  
Invalid resident library  
You are trying to use commands with a previous version of AmigaDOS, for example, Version 2.0 commands with Version 1.3.

202  
Object is in use  
The specified file or directory is already being used by other applications. If an application is reading a file, no other program can write to it and vice versa.

203  
Object already exists  
The specified name already belongs to another file or directory.

204  
Directory not found  
AmigaDOS cannot find the directory you specified. You may have made a typing or spelling error.

205  
Object not found  
AmigaDOS cannot find the file or directory you specified. You may have made a typing or spelling error.

206  
Invalid window description  
This error occurs when specifying a window size for the Output Window, a Shell, ED or ICONX window. You may have made the window

---

too big or too small, or you may have omitted an argument. This error also occurs with the NEWSHELL command, if you supply a device name that is not a window.

209

Packet request type unknown

You have asked a device handler to attempt an operation it cannot do. For example, the console handler cannot rename anything.

210

Object name invalid

There is an invalid character in the filename or the filename is too long. Remember, filenames cannot be longer than 30 characters and cannot contain control characters.

211

Invalid object lock

You have used something that is not a valid lock.

212

Object not of required type

You may have specified a filename for an operation that requires a directory name or vice versa.

213

Disk not validated

If you have just inserted a disk, the disk validation process may still be in progress. It is also possible that the disk is corrupt.

214

Disk is write-protected

The plastic tab of the disk is in the write-protect position.

215

Rename across devices attempted

The RENAME command only changes a filename on the same volume. You can use RENAME to move a file from one directory to another, but you cannot move files from one volume to another.

216

Directory not empty

This error occurs if you attempt to delete a directory that contains files or sub-directories.

217

Too many levels

You have exceeded the limit of 15 soft links.

218

Device (or volume) not mounted

If the device is a floppy disk, it has not been inserted in a drive. If it is another type of device, it has not been mounted with the MOUNT command. It is also possible that you have made a typing error when specifying the device name.

219



Seek error  
You have attempted to call Seek() with invalid arguments.

220  
Comment is too long  
Your filenote has exceeded the maximum number of characters (79).

212  
Disk is full  
There is not enough room left on the disk to perform the requested operation.

222  
Object is protected from deletion  
The D (deletable) protection bit of the file or directory is clear.

223  
File is write protected  
The W (writable) protection bit of the file is clear.

224  
File is read protected  
The R (readable) protection bit of the file is clear.

225  
Not a valid DOS disk  
The disk in the drive is not an AmigaDOS disk, it has not been formatted, or it is corrupt.

226  
No disk in drive  
The disk is not properly inserted in the specified drive.

232  
No more entries in directory  
This indicates that the AmigaDOS call ExNext() has no more entries in the directory you are examining.

233  
Object is soft link  
You tried to perform an operation on a soft link that should only be performed on a file or directory.

303  
Buffer overflow  
User or internal buffer overflow.

304  
Break  
A break character was received.

305  
Not executable  
The r (readable) protection bit of the specified file is clear.

---

## 1.84 pro.guide/Instruction Set Summary

Instruction Set Summary  
=====

This node (Page) intentionally left blank.

## 1.85 pro.guide/ProOpts Directives Summary

ProOpts Directives Summary  
=====

This node (Page) intentionally left blank.

## 1.86 pro.guide/Bibliography

Bibliography  
=====

The following manual covers the complete M68000 family (including the MC68881, MC68882, and MC68851 coprocessors) instruction set, and is recommended to those, who wish to learn the 680x0 assembly language:

- \* Motorola Inc., 'M68000 Family Programmer's Reference Manual',  
Motorola Inc.

The following manual is an essential reference tool for all Amiga programmers, who want to take full advantage of the Amiga's impressive capabilities:

- \* Commodore-Amiga Inc., 'AMIGA ROM Kernel Reference Manual:  
Includes and Autodocs', third edition, Addison-Wesley Publishing  
Co.

The AmigaDOS manual describes this powerful operating system and is a comprehensive reference to the commands, functions, and innermost workings of AmigaDOS:

- \* Commodore-Amiga Inc., 'The AmigaDOS Manual', third edition,  
Bantam Books.

The user's manuals listed below contain the information on the specific microprocessors.

---

- \* Motorola Inc., 'M68000 8-/16-/32-Bit Microprocessor User's Manual', seventh edition, Prentice Hall.
- \* Motorola Inc., 'MC68020 32-Bit Microprocessor User's Manual', third edition, Prentice Hall.
- \* Motorola Inc., 'MC68030 Enhanced 32-Bit Microprocessor User's Manual', third edition, Prentice Hall.
- \* Motorola Inc., 'MC68EC030 32-Bit Embedded Controller User's Manual', Motorola Inc.
- \* Motorola Inc., 'M68040/M68EC040/M68LC040 Microprocessors User's Manual', Motorola Inc.
- \* Motorola Inc., 'M68060/M68EC060/M68LC060 Microprocessors User's Manual', Motorola Inc.
- \* Motorola Inc., 'MC68881/MC68882 Floating-Point Coprocessor User's Manual', second edition, Prentice Hall.
- \* Motorola Inc., 'MC68851 Paged Memory Management Unit User's Manual', Prentice Hall.

The Motorola literature can be obtained from your local Motorola Sales Office or Distributor. Local Sales offices and Distributors are listed in the back of any Motorola Master Selection Guide. If you can not find a local representative, some Motorola Literature Distributions Center addresses are listed below:

USA:

Motorola Literature Distribution  
P.O.Box 20912  
Phoenix, Arizona 85036

Europe:

Motorola Ltd.  
European Literature Center  
88 Tanners Drive  
Blakelands  
Milton Keynes, MK14 5BP  
England

Japan:

Nippon Motorola Ltd.  
4-32-1 Nishi-Gotanda  
Shinagawa-ku  
Tokyo 141 Japan

Asia-Pacific:

Motorola Semiconductors H.K. Ltd  
Silicon Harbour Center  
No. 2 Dai King Street  
Tai Po Industrial Estate  
Tai Po  
N.T.  
Hong Kong

---

## 1.87 pro.guide/Directive Index

Index of Assembler Directives and Special Symbols

=====

\* \*

\*  
=  
==  
\#  
\(n)  
\\*B  
\\*BIN  
\\*BINOF  
\\*D  
\\*D(expression)  
\\*H  
\\*H(expression)  
\\*HEX  
\\*HEXOF  
\\*L  
\\*LEFT  
\\*LOWER  
\\*LOWER(string)  
\\*M  
\\*MID  
\\*O  
\\*OCT  
\\*OCTOF

---

```
\*R
\*RIGHT
\*S
\*STRING
\*STRING(textual symbol)
\*STRLEN
\*UPPER
\*UPPER(string)
\*V
\*V(expression)
\*VAL
\*VALOF
\0
\@
\n
_LINENUM
_MCOUNT
_MOVEBYTES
_MOVEMLIST
_MOVEMREGS
__BASE
__CP
__DATE
__DATE2
__DATE3
__DAY
__FO
__INFINITY
```

---

---

\_\_LK  
\_\_NAN  
\_\_OLDLABSEG  
\_\_OLDSECTION  
\_\_PR  
\_\_PRO  
\_\_RCODE  
\_\_RS  
\_\_SNAN  
\_\_SO  
\_\_TIME  
\_\_Vn  
\* A \*

ADDSYM  
ALIGN  
ALIGN.<size>  
ALIGNDX  
ALIGNDX.<size>  
ALIGNFO  
ALIGNFO.<size>  
ALIGNRS  
ALIGNRS.<size>  
ALIGNSO  
ALIGNSO.<size>  
APOPM  
APUSHM  
ASEG  
ASM  
ASMPRI

---

AUTOXREF

\* B \*

BASE

BASEREG

BINARY

BINARYONLY

BINRYONLY

BSS

\* C \*

CASEOFF

CASEON

CCNOP

CLRFO

CLRSO

CNOP

CODE

COMMENT

CREFFILE

CSEG

CSYMFMT

\* D \*

DATA

DB

DC.B

DC.L

DC.W

DEBUG

DEFAULT

DEFINE

DL

DS.W 0

DSEG

DW

\* E \*

ELSE

ELSEIF

END

ENDASM

ENDC

ENDIF

ENDM

ENDMAC

ENDORG

ENDR

ENDSRC

EQU

EQUATE

EQUFILE

EQR

EQURL

EUSTR

EQUX

ERRFILE

ERRFLAG

ESCAPESTR

EVEN

EXE

---



EXECUTABLE

EXEOBJ

\* F \*

FAIL

FAILAT

FEQU

FEQUR

FILENOTE

FILEPROTECT

FORMAT

FOVAL

FREG

FSET

FSETR

FSETRL

\* H \*

HEADER

\* I \*

IBYTES

IDENTIFY

IDNT

IF1

IF2

IFC

IFCC

IFCS

IFD

IFEQ

IFEQ

IFGE

IFGE

IFGT

IFGT

IFHI

IFHI

IFHS

IFLE

IFLE

IFLO

IFLS

IFLS

IFLT

IFLT

IFMI

IFMI

IFNC

IFND

IFNE

IFNE

IFNU

IFPL

IFPL

IFU

IFVC

IFVS

IIF

INCBIN

---

INCDIR  
INCEQU  
INCLUDE  
INCPATH  
\* L \*  
  
LABSEG  
LABSEG \_\_\_OLDLABSEG  
  
LFCOND  
LINKABLE  
  
LINKOBJ  
  
LIST  
  
LISTCHAR  
LISTFILE  
  
LISTSYMS  
  
LLEN  
  
LOCKSYM  
\* M \*  
  
MACLIB  
  
MACRO  
  
MC68000  
MC68008  
MC68010  
MC68020  
MC68030  
MC68040  
MC68060  
MC680X0  
MC68851  
MC68881

---

MC68882

MC68EC020

MC68EC030

MCRELAX

MEA

MEXIT

MULTIPASS

\* N \*

NARG

NB

NEWSYNTAX

NL

NOBASE

NOLIST

NOOBJ

NOOPTIM

NOPAGE

NORMOBJ

NOSYM

NW

\* O \*

OBJ

OBJFILE

ODD

ODD2ERROR

ODD2OK

ODDERROR

ODDOK

OLDSYNTAX

---

OPT  
OPT A+  
OPT A-  
OPT ABL  
OPT ABW  
OPT AUTOPC  
OPT BDL  
OPT BDW  
OPT BRB  
OPT BRL  
OPT BRS  
OPT BRW  
OPT C+  
OPT C-  
OPT Cn+  
OPT Cn-  
OPT CASE  
OPT CHKBIT  
OPT CHKIMM  
OPT CHKPC  
OPT CL  
OPT D+  
OPT D-  
OPT DEBUG  
OPT E+  
OPT E-  
OPT ESS1+  
OPT ESS1-

---

---

OPT ESS2+  
OPT ESS2-  
OPT ESS3+  
OPT ESS3-  
OPT F+  
OPT F-  
OPT FPSP40  
OPT FPSP60  
OPT GENSYM  
OPT HCLN  
OPT I+  
OPT I-  
OPT INCONCE  
OPT L+  
OPT L-  
OPT LIST  
OPT LOCALDOT  
OPT LOCALU  
OPT M+  
OPT M-  
OPT MD  
OPT MEX  
OPT NOAUTOPC  
OPT NOCASE  
OPT NOCHKBIT  
OPT NOCHKIMM  
OPT NOCHKPC  
OPT NOCL  
OPT NODEBUG

---

---

OPT NOFPSP40

OPT NOFPSP60

OPT NOHCLN

OPT NOINCONCE

OPT NOLIST

OPT NOMD

OPT NOMEX

OPT NOOPTIMLIB

OPT NOSP60

OPT NOSUPER

OPT NOSYMTAB

OPT NOTYPE

OPT NOWARN

OPT NOXDEBUG

OPT NOXPK

OPT O+

OPT O-

OPT On+

OPT On-

OPT ODL

OPT ODW

OPT OPTIMLIB

OPT OW+

OPT OW-

OPT OWn+

OPT OWn-

OPT P+

OPT P-

OPT P=

---

---

OPT PCBL  
OPT PCBW  
OPT Q+  
OPT Q-  
OPT Qn+  
OPT Qn-  
OPT QW+  
OPT QW-  
OPT QWn+  
OPT QWn-  
OPT RCL  
OPT RESET  
OPT S+  
OPT S-  
OPT SP60  
OPT STO  
OPT SUPER  
OPT SW+  
OPT SW-  
OPT SYMTAB  
OPT T+  
OPT T-  
OPT TYPE  
OPT U+  
OPT U-  
OPT U1+  
OPT U1-  
OPT U2+

---



---

OPT U2-  
OPT W+  
OPT W-  
OPT WARNBIT  
OPT WARN  
OPT X+  
OPT X-  
OPT XDEBUG  
OPT XPK  
OPT Y+  
OPT Y-  
OPT Z+  
OPT Z-  
OPTIMIZE  
OPTIMOFF  
OPTIMON  
OPTION  
ORG  
OUTPUT  
\* P \*  
  
PAGE  
PAGEUP  
PB  
PFLUSHA  
PFLUSHA30  
PFLUSHA40  
PL  
PLEN  
POP .

---

POPM

PREASM  
PRINTX

PUBLIC

PURE

PUSH

PUSHM

PW  
\* Q \*

QUAD

QUIET  
\* R \*

RARG

RCRESET

RCSET

READMODWRITE

REG

RELAX

REPEAT

REPT

REXIT

RORG

RSRESET

RSVAL  
\* S \*

SB

SECSYM

SECTION

SECTION \_\_OLDSECTION

---

---

SELSYM  
SET  
SETKFACTOR  
SETR  
SETREG  
SETRL  
SETSTR  
SETVAL  
SFCOND  
SL  
SMALLBSS  
SMALLCODE  
SMALLDATA  
SMALLOBJ  
SOVAL  
SPC  
SREC  
SUBTTL  
SUPER  
SW  
\* T \*  
TIMES  
TITLE  
TTL  
\* U \*  
UB  
UL  
UNLOCKSVM

---

UNTIL

UW

\* V \*

VERBOSE

\* W \*

WARN

\* X \*

XDEF

XREF

## 1.88 pro.guide/Concept Index

Concept Index

=====

This node (page) intentionally left blank.

---