



Apple®

Apple Numerics Manual



Second Edition



Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California New York

Don Mills, Ontario Wokingham, England Amsterdam Bonn

Sydney Singapore Tokyo San Juan

🍏 APPLE COMPUTER, INC.
Copyright © 1988 by Apple
Computer, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

Apple, the Apple logo, Apple IIGS, LaserWriter, Macintosh, ProDOS, and SANE are registered trademarks of Apple Computer, Inc.

Aztec C is a trademark of Manx Software Systems.

ITC Avant Garde Gothic, ITC Garamond, and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation.

Lightspeed Pascal is a trademark of THINK Technologies, Inc.

Mac C is a trademark of Consulair Corp.

Microsoft is a registered trademark of Microsoft Corporation.

POSTSCRIPT is a registered trademark of Adobe Systems Incorporated.

Turbo Pascal is a registered trademark of Borland International.

Varietyper is a registered trademark, and VT600 is a trademark, of AM International, Inc.

VAX is a trademark of Digital Equipment Corporation.

Simultaneously published in the United States and Canada.

ISBN 0-201-17738-2
ABCDEFGHIJ-DO-898
First printing, May 1988

WARRANTY INFORMATION

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

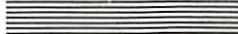
IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.



Contents



Figures and tables xv

Foreword About Standard Numerics xxi

Preface About This Book xxv

About the second edition xxv

Parts of the manual xxvi

Special notation xxvi

 Computer voice xxvi

 Hexadecimal numbers xxvi

Part I The Standard Apple Numerics Environment 1

Chapter 1 About IEEE Standard Arithmetic 3

Starting to use IEEE standard arithmetic 4

Well-behaved arithmetic 4

 Extended precision and range 4

 Accurate results 5

 Gradual underflow 5

 Careful rounding 5

 Example: inverse operations 6

Alternatives to stopping 7

 Example: compound conditional statements 7

 Searching without stopping 8

 Example: parallel resistances 8

Advanced features 9

 Control of rounding 9

 Exception handling 10

 Elementary functions 10

Chapter 2 SANE Data Types 11

- Choosing a data type 12
- Values represented 13
- Range and precision of SANE types 14
 - Example: range of single 15
- Formats 15
 - Single 16
 - Double 17
 - Comp 17
 - Extended 18

Chapter 3 Conversions in SANE 19

- Conversions between extended and single or double 21
 - Conversions from extended to single or double 21
 - Example: Is x negligible? 21
- Conversions to comp and other integral formats 22
- Conversions between binary and decimal 22
 - Example: binary approximation of decimal fractions 23
 - Accuracy of decimal-to-binary conversions 23
 - Reading and writing decimal data 24
 - Conversions from decimal strings to SANE types 25
 - Decform records and conversions from SANE types to decimal strings 26
 - The decimal record type 27
 - Conversions from decimal records to SANE types 28
 - Conversions from SANE types to decimal records 28
 - Unusual cases for decimal records 29
- Conversions between decimal formats 29
 - Conversion from decimal strings to decimal records 30
 - Conversion from decimal records to decimal strings 31
 - Floating-style decimal output 31
 - Fixed-style decimal output 32
 - Unusual cases for decimal strings 32
 - Alignment and field width 33
 - Example: decimal records 33

Chapter 4 Expression Evaluation in SANE 35

- Extended-precision expression evaluation 36
 - Example: expression evaluation 36
- Using extended temporaries 37
 - Example: extended temporaries 37
- Expression evaluation and the IEEE Standard 38

Chapter 5 Infinities, NaNs, and Denormalized Numbers 39

- Infinities 40
- NaNs 41
- Denormalized numbers 42
 - Example: gradual underflow 42
 - Why gradual underflow? 43
- Sign of zero 43
- Inquiries: class and sign 44

Chapter 6 Arithmetic Operations, Comparisons, and Auxiliary Procedures 45

- Arithmetic operations 46
 - Remainder 46
 - Remainder example 1 47
 - Remainder example 2 47
 - Round-to-integer 47
- SANE comparisons 47
 - Comparisons with NaNs and Infinities 48
 - The Relation function 49
- Auxiliary procedures 49
 - Sign manipulation 49
 - Nextafter functions 50
 - Special cases for Nextafter functions 50
 - Binary scaling and logarithmic functions 50
 - Special cases for Logb 50

Chapter 7 Controlling the SANE Environment 51

- Rounding direction 52
 - Example: rounding upward 53
- Rounding precision 53
- Exception flags and halts 54
 - Types of exceptions 55
 - Invalid operation 55
 - Underflow 56
 - Divide-by-zero 56
 - Overflow 56
 - Inexact 56
- Managing environmental settings 57
 - Example: setting rounding direction 57
 - Example: setting environment 58
 - Example: setting exceptions 58

Chapter 8 Elementary Functions in SANE 61

- Logarithmic functions 62
 - Special cases for logarithmic functions 62
- Exponential functions 63
 - Example: using Exp1 63
 - Special cases for Exp2, Exp, and Exp1 63
 - Special cases for XpwrI 64
 - Special cases for XpwrY 64
- Financial functions 64
 - Compound 64
 - Annuity 65
 - Special cases for Compound 65
 - Special cases for Annuity 65
- Trigonometric functions 66
 - Accuracy of pi 66
 - Special cases for Cos and Sin 66
 - Special cases for Tan 67
 - Special cases for ArcTan 67
- Random number generator 67

Chapter 9 Other Elementary Functions 69

- Exception handling 70
- Functions 70
 - Secant 70
 - CoSecant 70
 - CoTangent 71
 - ArcSin 71
 - ArcCos 71
 - Constants for Sinh and Cosh 72
 - Sinh 72
 - Cosh 73
 - Tanh 73
 - ArcSinh 74
 - ArcCosh 74
 - ArcTanh 74

Chapter 10 More Examples Using SANE 75

- Continued fraction 76
- Area of a triangle 77
 - Heron's formula 77
 - Improved formula 78

Part II The 65C816 and 6502 Assembly-Language SANE Engines 79

Chapter 11 65C816 SANE Basics and Data Types 81

- Operation forms 83
 - Arithmetic and auxiliary operations 83
 - Conversions 84
 - Comparisons 84
 - Other operations 84
- External access 85
- Calling sequence 86
 - The opword 88
 - Example 88
 - Assembly-language macros 88
 - Example 1 89
 - Example 2 89
 - Example 3 89
- 65C816 SANE data types 90

Chapter 12 65C816 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries 91

- Add, Subtract, Multiply, and Divide 93
 - Example 93
- Square Root 93
 - Example 93
- Round-to-Integer and Truncate-to-Integer 93
- Remainder 94
 - Example 94
- Logb and Scalb 94
 - Example 94
- Negate, Absolute Value, and CopySign 95
 - Example 95
- Nextafter 95
 - Example 95
- Comparisons 96
 - Example 1 97
 - Example 2 97
- Inquiries 97
 - Example 98

Chapter 13	Conversions in 65C816 SANE 99
	Conversions between binary formats 100
	Conversions to extended 100
	Example 100
	Conversions from extended 101
	Example 101
	Binary-decimal conversions 101
	Binary to decimal 102
	Example 102
	Fixed-format overflow 102
	Decimal to binary 102
	Example 102
	Techniques for maximum accuracy 103
Chapter 14	Controlling the 65C816 SANE Environment 105
	The Environment word 106
	Example 107
	Get-Environment and Set-Environment 108
	Example 108
	Test-Exception and Set-Exception 109
	Example 109
	Procedure-Entry and Procedure-Exit 110
	Example 110
Chapter 15	Halts in 65C816 SANE 111
	Conditions for a halt 112
	The halt mechanism 112
	Halt status information 113
	Halt vector operations 115
	Using the halt mechanism 116
	Halt example for the 65C816 116
	Halt example for the 6502 117
Chapter 16	Elementary Functions in 65C816 SANE 119
	One-argument functions 120
	Example 121
	Two-argument functions 122
	Example 122
	Three-argument functions 123
	Example 123
Chapter 17	65C816 SANE Scanners and Formatter 125
	Numeric scanners 126
	Numeric formatter 127

Chapter 18	Examples: Using the 65C816 and 6502 SANE Engines 129
	65C816 examples 130
	65C816 example: polynomial evaluation 130
	65C816 example: scanning and formatting 132
	6502 examples 133
	6502 example: polynomial evaluation 133
	6502 example: scanning and formatting 135
Part III	The MC68000 Assembly-Language SANE Engine 137
Chapter 19	MC68000 SANE Basics and Data Types 139
	Operation forms 140
	Arithmetic and auxiliary operations 141
	Conversions 141
	Comparisons 142
	Other operations 142
	External access 142
	Calling sequence 143
	The opword 143
	Assembly-language macros 144
	Example 1 144
	Example 2 144
	Example 3 144
	MC68000 SANE data types 145
Chapter 20	MC68000 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries 147
	Add, Subtract, Multiply, and Divide 148
	Example 148
	Square Root 149
	Example 149
	Round-to-Integer and Truncate-to-Integer 149
	Remainder 149
	Example 149
	Logb and Scalb 150
	Example 150
	Negate, Absolute Value, and CopySign 150
	Example 150
	Nextafter 151
	Example 151
	Comparisons 151
	Example 1 152
	Example 2 152
	Inquiries 153
	Example 153

Chapter 21	Conversions in MC68000 SANE	155
	Conversions between binary formats	156
	Conversions to extended	156
	Example	156
	Conversions from extended	157
	Example	157
	Binary-decimal conversions	157
	Binary to decimal	158
	Example	158
	Fixed-format overflow	158
	Decimal to binary	158
	Example	158
	Techniques for maximum accuracy	159
Chapter 22	Controlling the MC68000 SANE Environment	161
	The Environment word	162
	Example	163
	Get-Environment and Set-Environment	164
	Example	164
	Test-Exception and Set-Exception	164
	Example	165
	Procedure-Entry and Procedure-Exit	165
	Example	165
Chapter 23	Halts in MC68000 SANE	167
	Conditions for a halt	168
	The halt mechanism	168
	Using the halt mechanism	170
Chapter 24	Elementary Functions in MC68000 SANE	171
	One-argument functions	172
	Example	173
	Two-argument functions	173
	Example	173
	Three-argument functions	174
	Example	174
Chapter 25	MC68000 SANE Scanners and Formatter	175
	Numeric scanners	176
	Numeric formatter	177

Chapter 26	Examples: Using the MC68000 SANE Engine 179
	Example: polynomial evaluation 180
	Example: language interface 181
	Example: scanning and formatting 181
Part IV	Using the MC68881 SANE Engine 183
Chapter 27	About the MC68881 and SANE 185
	SANE implementations on the Macintosh 186
	SANE software for the MC68881 186
	All calls to software packages 187
	Fundamental operations on the MC68881 188
	Transcendental operations on the MC68881 189
	Calls to SANE and calls to the MC68881 190
	MC68881 data types 190
	MC68881 floating-point registers 191
	96-bit extended format 191
	Comparison of extended formats 192
	Conversions between extended formats 193
	Using the comp format with the MC68881 193
	SANE macros for the MC68881 194
Chapter 28	Functions of the MC68881 and SANE Software 195
	Functions that are the same on both 196
	Functions that are similar 197
	Functions only the SANE software has 198
	Functions only the MC68881 has 198
	Accuracy of the MC68881's elementary functions 199
Chapter 29	Controlling the MC68881 Environment 201
	The MC68881's environment registers 202
	The Exception Enable and Mode Control bytes 203
	The Exception Status and Accrued Exception bytes 205
Chapter 30	The MC68881 Trap Mechanism 207
	Halts and traps 208
	MC68881 exception handling 208
Chapter 31	Examples: Using the MC68881 SANE Engine 211
	Example: polynomial evaluation 212
	Example: language interface 213
	Example: scanning and formatting 213

Appendixes 217

Appendix A SANE in High-Level Languages 219

- SANE programming 220
- SANE library 221
- Pascal SANE extensions 221
 - Data types 221
 - Constants 222
 - Expressions 222
 - Comparisons 222
 - Functions and procedures 223
 - Input/output 223
 - Numeric environment 223
 - Pascal SANE library 224
- C SANE extensions 233
 - Data types 233
 - Constants 233
 - Expressions 233
 - Comparisons 234
 - Functions 234
 - Input/output 234
 - Numeric environment 234
 - C SANE library 235

Appendix B The SANE Engines: Availability 240

- SANE for the 6502 240
- SANE for the 65C816 240
- SANE for the MC68000 241
- SANE for the MC68020 and MC68881 241

Appendix C Porting Programs to SANE 242

- Semantics of arithmetic evaluation 242
- Mixed formats 243
- Floating-point precision 243
 - Wider precision 243
 - Double rounding 244
 - Computed error bounds 244
- The rules of evaluation 244
- The invalid-operation flag 245

Appendix D 65C816 and 6502 SANE Quick Reference Guide 246

- Formats of SANE types 246
 - Single: 32 bits 247
 - Double: 64 bits 248
 - Comp: 64 bits 248
 - Extended: 80 bits 249
- Operations 249
 - Abbreviations and symbols 249
 - Operands 250
 - Data types 251
 - 65C816 and 6502 processor registers 251
 - Exceptions 252
 - Environment and halts 252
 - Arithmetic operations (entry points FP816, FP6502) 253
 - Auxiliary routines (entry points FP816, FP6502) 254
 - Conversions (entry points FP816, FP6502) 255
 - Comparisons (entry points FP816, FP6502) 256
 - Inquiries: class and sign (entry points FP816, FP6502) 257
 - Environmental control (entry points FP816, FP6502) 258
 - Halt control (entry points FP816, FP6502) 259
 - Elementary functions (entry points Elems816, Elems6502) 259
 - Scanners and formatter (entry points DecStr816, DecStr6502) 260
- The Environment word 261

Appendix E MC68000 SANE Quick Reference Guide 263

Formats of SANE types	263
Single: 32 bits	264
Double: 64 bits	265
Comp: 64 bits	265
Extended: 80 bits	266
Operations	266
Abbreviations and symbols	266
Operands	267
Data types	267
MC68000 processor registers	268
Exceptions	268
Environment and halts	268
Arithmetic operations (entry point FP68K)	269
Auxiliary routines (entry point FP68K)	270
Conversions (entry point FP68K)	271
Comparisons (entry point FP68K)	272
Inquiries: class and sign (entry point FP68K)	273
Environmental control (entry point FP68K)	274
Halt control (entry point FP68K)	274
Elementary functions (entry point Elems68K)	275
Scanners and formatter (entry point DecStr68K)	275
The Environment word	276
Glossary	279
Bibliography	281
Index	287

Figures and tables

Part I The Standard Apple Numerics Environment 1

Chapter 1 About IEEE Standard Arithmetic 3

Figure 1-1 Parallel resistances 8

Chapter 2 SANE Data Types 11

Figure 2-1 Single format 16
Figure 2-2 Double format 17
Figure 2-3 Comp format 17
Figure 2-4 80-bit extended format 18
Figure 2-5 96-bit extended format 18
Table 2-1 Names of data types 13
Table 2-2 SANE types 14
Table 2-3 Symbols used in format diagrams 16
Table 2-4 Values of single-format numbers (32 bits) 16
Table 2-5 Values of double-format numbers (64 bits) 17
Table 2-6 Values of comp-format numbers (64 bits) 17
Table 2-7 Values of extended-format numbers 18

Chapter 3 Conversions in SANE 19

Figure 3-1 Conversions to and from extended format 20
Figure 3-2 Conversions to and from decimal formats 20
Figure 3-3 Conversion cycle with first-time error 24
Figure 3-4 Conversion cycle with correct result 25
Table 3-1 Approximations of fractions 23
Table 3-2 Syntax for string conversions 26
Table 3-3 Examples of conversions to decimal records 30
Table 3-4 Format of decimal output string in floating style 31
Table 3-5 Format of decimal output string in fixed style 32
Table 3-6 Examples of conversions to decimal strings 34

Chapter 5 Infinities, NaNs, and Denormalized Numbers 39

Figure 5-1 Denormalized single-precision numbers
on the number line 43
Table 5-1 SANE NaN codes 41
Table 5-2 Example of gradual underflow 42

Chapter 6 Arithmetic Operations, Comparisons, and Auxiliary Procedures 45

Figure 6-1	Integer-division algorithm	46
Table 6-1	Arithmetic operations in Pascal	46
Table 6-2	Comparisons involving NaNs	48

Chapter 10 More Examples Using SANE 75

Figure 10-1	Graph of continued fraction $cf(x) = rf(x)$	76
Table 10-1	Area using Heron's formula	78

Part II The 65C816 and 6502 Assembly-Language SANE Engines 79

Chapter 11 65C816 SANE Basics and Data Types 81

Figure 11-1	SANE operands on the 65C816 stack	87
Figure 11-2	SANE operands on the 6502 stack	87
Figure 11-3	Memory format of a variable of type single	90
Table 11-1	65C816 SANE data types	90
Table 11-2	Bits in a variable of type single	90

Chapter 12 65C816 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries 91

Table 12-1	Results of comparisons	96
Table 12-2	Operand classes	97

Chapter 13 Conversions in 65C816 SANE 99

Table 13-1	Conversions to extended format	100
Table 13-2	Conversions from extended format	101

Chapter 14 Controlling the 65C816 SANE Environment 105

Figure 14-1	The Environment word for the 6502 and 65C816	106
Table 14-1	Bits in the Environment word for the 6502 and 65C816	107
Table 14-2	Bits in the Exception word	109

Chapter 15 Halts in 65C816 SANE 111

Figure 15-1	65C816 SANE direct-page contents upon halt	113
Figure 15-2	6502 SANE status record upon halt	114
Figure 15-3	Data returned in X and Y registers	115

Part III The MC68000 Assembly-Language SANE Engine 137

Chapter 19 MC68000 SANE Basics and Data Types 139

Figure 19-1	SANE operands on the MC68000 stack	143
Figure 19-2	Memory format of a variable of type single	145
Table 19-1	MC68000 SANE data types	145
Table 19-2	Bits in a variable of type single	145

Chapter 20 MC68000 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries 147

Table 20-1	Results of comparisons	152
Table 20-2	Operand classes	153

Chapter 21 Conversions in MC68000 SANE 155

Table 21-1	Conversions to extended format	156
Table 21-2	Conversions from extended format	157

Chapter 22 Controlling the MC68000 SANE Environment 161

Figure 22-1	The Environment word for the MC68000	162
Table 22-1	Bits in the Environment word for the MC68000	163
Table 22-2	Bits in the Exception word	164

Chapter 23 Halts in MC68000 SANE 167

Figure 23-1	Stack frame for halt	169
-------------	----------------------	-----

Part IV Using the MC68881 SANE Engine 183

Chapter 27 About the MC68881 and SANE 185

Figure 27-1	Application calling packages for all arithmetic	187
Figure 27-2	Application calling the MC68881 for fundamental operations	188
Figure 27-3	Application calling the MC68881 for all floating-point arithmetic	189
Figure 27-4	96-bit extended format	191
Figure 27-5	Comparison of extended formats	192
Table 27-1	Values of extended-format numbers	191

Chapter 29 Controlling the MC68881 Environment 201

Figure 29-1	The MC68881's Floating-Point Control register (FPCR) 202
Figure 29-2	The MC68881's Floating-Point Status register (FPSR) 202
Figure 29-3	The MC68881's Exception Enable and Mode Control bytes 203
Figure 29-4	The MC68881's Exception Status and Accrued Exception bytes 205
Table 29-1	Bits in the MC68881's Exception Enable and Mode Control bytes 204
Table 29-2	Bits in the MC68881's Exception Status and Accrued Exception bytes 206

Appendixes 217

Appendix A SANE in High-Level Languages 219

Table A-1	Pascal data types 221
Table A-2	C data types 233

Appendix D 65C816 and 6502 SANE Quick Reference Guide 246

Figure D-1	Memory format of a variable of type single 246
Figure D-2	Single format 247
Figure D-3	Double format 248
Figure D-4	Comp format 248
Figure D-5	Extended format 249
Figure D-6	SANE operands on the 65C816 stack 250
Figure D-7	SANE operands on the 6502 stack 250
Figure D-8	Data returned in X and Y registers 258
Figure D-9	The Environment word for the 65C816 and 6502 261
Table D-1	Format diagram symbols 247
Table D-2	Values of single-format numbers (32 bits) 247
Table D-3	Values of double-format numbers (64 bits) 248
Table D-4	Values of comp-format numbers (64 bits) 248
Table D-5	Values of extended-format numbers (80 bits) 249
Table D-6	Operands 250
Table D-7	Data types 251
Table D-8	65C816 and 6502 processor registers 251
Table D-9	Exceptions 252
Table D-10	Environment and halts 252
Table D-11	Arithmetic operations (entry points FP816, FP6502) 253

Table D-12	Auxiliary routines (entry points FP816, FP6502) 254
Table D-13	Binary-to-binary conversions (entry points FP816, FP6502) 255
Table D-14	Binary-to-decimal conversions (entry points FP816, FP6502) 255
Table D-15	Decimal-to-binary conversions (entry points FP816, FP6502) 255
Table D-16	Relation information 256
Table D-17	Comparisons (entry points FP816, FP6502) 256
Table D-18	Class information 257
Table D-19	Sign information 257
Table D-20	Classify (entry points FP816, FP6502) 257
Table D-21	Environmental control (entry points FP816, FP6502) 258
Table D-22	Halt control (entry points FP816, FP6502) 259
Table D-23	Elementary functions (entry points Elems816, Elems6502) 259
Table D-24	Scanners and formatter (entry points DecStr816, DecStr6502) 260
Table D-25	Bits in the Environment word for the 65C816 and 6502 262

Appendix E MC68000 SANE Quick Reference Guide 263

Figure E-1	Memory format of a variable of type single 263
Figure E-2	Single format 264
Figure E-3	Double format 265
Figure E-4	Comp format 265
Figure E-5	Extended format 266
Figure E-6	SANE operands on the MC68000 stack 267
Figure E-7	The Environment word for the MC68000 276
Table E-1	Format diagram symbols 264
Table E-2	Values of single-format numbers (32 bits) 264
Table E-3	Values of double-format numbers (64 bits) 265
Table E-4	Values of comp-format numbers (64 bits) 265
Table E-5	Values of extended-format numbers (80 bits) 266
Table E-6	Operands 267
Table E-7	Data types 267
Table E-8	MC68000 processor registers 268
Table E-9	Exceptions 268
Table E-10	Environment and halts 268
Table E-11	Arithmetic operations (entry point FP68K) 269
Table E-12	Auxiliary routines (entry point FP68K) 270
Table E-13	Binary-to-binary conversions (entry point FP68K) 271

Table E-14	Binary-to-decimal conversions (entry point FP68K) 271
Table E-15	Decimal-to-binary conversions (entry point FP68K) 271
Table E-16	Relation information 272
Table E-17	Comparisons (entry point FP68K) 272
Table E-18	Class information 273
Table E-19	Sign information 273
Table E-20	Classify (entry point FP68K) 273
Table E-21	Environmental control (entry point FP68K) 274
Table E-22	Halt control (entry point FP68K) 274
Table E-23	Elementary functions (entry point Elems68K) 275
Table E-24	Scanners and formatter (entry point DecStr68K) 275
Table E-25	Bits in the Environment word for the MC68000 277



About Standard Numerics

Part I of this book is mainly for people who perform scientific, statistical, or engineering computations on Apple® computers. The rest is mainly for producers of software, especially of language processors, that people will use on Apple computers to perform computations in those fields and in finance and business too. Moreover, if the first edition was any indication, people who have nothing to do with Apple computers may well buy this book just to learn a little about an arcane subject, floating-point arithmetic on computers, and will wish they had an Apple.

Computer arithmetic has two properties that add to its mystery:

- What you see is often not what you get, and
- What you get is sometimes not what you wanted.

Floating-point arithmetic, the kind computers use for protracted work with approximate data, is intrinsically approximate because the alternative, exact arithmetic, could take longer than most people are willing to wait—perhaps forever. Approximate results are customarily displayed or printed to show only as many of their leading digits as matter instead of all digits; what you see need not be exactly what you've got. To complicate matters, whatever digits you see are *decimal* digits, the kind you saw first in school and the kind used in hand-held calculators. Nowadays almost no computers perform their arithmetic with decimal digits; most of them use *binary*, which is mathematically better than decimal where they differ, but different nonetheless. So, unless you have a small integer, what you see is rarely just what you have.

In the mid-1960's, computer architects discovered shortcuts that made arithmetic run faster at the cost of what they reckoned to be a slight increase in the level of rounding error; they thought you could not object to slight alterations in the rightmost digits of numbers since you could not see those digits anyway. They had the best intentions, but they accomplished the opposite of what they intended. Computer throughputs were not improved perceptibly by those shortcuts, but a few programs that had previously been trusted unreservedly turned treacherous, failing in mysterious ways on extremely rare occasions.

For instance, a very Important Bunch of Machines launched in 1964 were found to have two anomalies in their double-precision arithmetic (though not in single): First, multiplying a number Z by 1.0 would lop off Z 's last digit. Second, the difference between two nearly equal numbers, whose digits mostly canceled, could be computed wrong by a factor almost as big as 16 instead of being computed exactly as is normal. The anomalies introduced a kind of noise in the feedback loops by which some programs had compensated for their own rounding errors, so those programs lost their high accuracies. These anomalies were not "bugs"; they were "features" designed into the arithmetic by designers who thought nobody would care. Customers did care; the arithmetic was redesigned and repairs were retrofitted in 1967.

Not all Capriciously Designed Computer arithmetics have been repaired. One family of computers has enjoyed notoriety for two decades by allowing programs to generate tiny "partially underflowed" numbers. When one of these creatures turns up as the value of T in an otherwise innocuous statement like

```
if T = 0.0 then Q := 0.0 else Q := 702345.6/(T + 0.00189/T);
```

it causes the computer to stop execution and emit a message alleging "Division by Zero." The machine's schizophrenic attitude toward zero comes about because the test for $T = 0.0$ is carried out by the adder, which examines at least 13 of T 's leading bits, whereas the divider and multiplier examine only 12 to recognize zero. Doing so saved less than a dollar's worth of transistors and maybe a picosecond of time, but at the cost of some disagreement about whether a very tiny number T is zero or not. Fortunately, the divider agrees with the multiplier about whether T is zero, so programmers could prevent spurious divisions by zero by slightly altering the foregoing statement as follows:

```
if 1.0 * T = 0.0 then Q := 0.0 else Q := 702345.6/(T + 0.00189/T);
```

Unfortunately, the Same Computer designer responsible for "partial underflow" designed another machine that can generate "partially overflowed" numbers T for which this statement malfunctions. On that machine, Q would be computed unexceptionably except that the product $1.0 * T$ causes the machine to stop and emit a message alleging "Overflow." How should a programmer rewrite that innocuous statement so that it will work correctly on both machines? We should be thankful that such a task is not encountered every day.

Anomalies related to roundoff are extremely difficult to diagnose. For instance, the machine on which $1.0 * T$ can overflow also divides in a peculiar way that causes quotients like $240.0/80.0$, which ought to produce small integers, sometimes to produce nonintegers instead, sometimes slightly too big, sometimes slightly too small. The same machine multiplies in a peculiar way, and it subtracts in a peculiar way that can get the difference wrong by almost a factor of 2 when it ought to be exact because of cancellation.

Another peculiar kind of subtraction, but different, afflicts the machines that are schizophrenic about zero. Sets of three values X , Y , and Z abound for which the statement

```
if (X = Y) and ((X - Z) > (Y - Z)) then writeln('Strange!');
```

will print “Strange!” on those machines. And many machines will print “Strange!” for unlucky values X and Y in the statement

```
if (X - Y = 0.0) and (X > Y) then writeln('Strange!');
```

because of underflow.

These strange things cannot happen on current Apple computers.

I do not wish to suggest that all but Apple computers have had quirky arithmetics. A few other computer companies, some Highly Prestigious, have Demonstrated Exemplary Concern for arithmetic integrity over many years. Had their concern been shared more widely, numerical computation would now be easier to understand. Instead, because so many computers in the 1960's and 1970's possessed so many different arithmetic anomalies, computational lore has become encumbered with a vast body of superstition purporting to cope with them. One such superstitious rule is “*Never* ask whether floating-point numbers are exactly equal.”

Presumably the reasonable thing to do instead is to ask whether the numbers differ by less than some tolerance; and this *is* truly reasonable provided you know what tolerance to choose. But the word *never* is what turns the rule from reasonable into mere superstition. Even if every floating-point comparison in your program involved a tolerance, you would wish to predict which path execution would follow from various input data, and whether the different comparisons were mutually consistent. For instance, the predicates $X < Y - \text{TOL}$ and $Y - \text{TOL} > X$ seem equivalent to the naked eye, but computers exist (*not* made by Apple!) on which one can be true and the other false for certain values of the variables. To ask “Which?” violates the superstitious rule.

There have been several attempts to avoid superstition by devising mathematical rules called *axioms* that would be valid for all commercially significant computers and from which a programmer might hope to be able to deduce whether his program will function correctly on all those computers. Unfortunately, such attempts cannot succeed without failing! The paradox arises because any such rules, to be valid universally, have to encompass so wide a range of anomalies as to constitute the specifications for a hypothetical computer far worse arithmetically than any ever actually built. In consequence, many computations provably impossible on that hypothetical computer would be quite feasible on almost every actual computer. For instance, the axioms must imply limits to the accuracy with which differential equations can be solved, integrals evaluated, infinite series summed, and areas of triangles calculated; but these limits are routinely surpassed nowadays by programs that run on most commercially significant computers, although some computers may require programs that are so special that they would be useless on any other machine.

Arithmetic anarchy is where we seemed headed until a decade ago when work began upon IEEE Standard 754 for binary floating-point arithmetic. Apple's mathematicians and engineers helped from the very beginning. The resulting family of coherent designs for computer arithmetic has been adopted more widely, and by more computer manufacturers, than any other single design. Besides the undoubted benefits that flow from any standard, the principal benefit derived from the IEEE standard in particular is this:

Program importability: Almost any application of floating-point arithmetic, designed to work on a few different families of computers in existence before the IEEE Standard and programmed in a higher-level language, will, after recompilation, work at least about as well on an Apple computer or on any other machine that conforms to IEEE Standard 754 as on any nonconforming computer with comparable capacity (memory, speed, and word size).

The Standard Apple Numerics Environment (SANE) is the most thorough implementation of IEEE Standard 754 to date. The fanatical attention to detail that permeates SANE's implementation largely relieves Apple computer users from having to know any more about those details than they like. If you come to an Apple computer from some other computer that you were fond of, you will find the Apple computer's arithmetic at least about as good, and quite likely rather better. An Apple computer can be set up to mimic the worthwhile characteristics of almost any reasonable past computer arithmetic, so existing libraries of numerical software do not have to be discarded if they can be recompiled. SANE also offers features that are unique to the IEEE Standard, new capabilities that previous generations of computer users could only yearn for; but to learn what they are, you will have to read this book.

As one of the designers of IEEE Standard 754, I can only stand in awe of the efforts that Apple has expended to implement that standard faithfully both in hardware and in software, including language processors, so that users of Apple computers will actually reap tangible benefits from the Standard. And I thank Apple for letting me explain in this foreword why we needed that standard.

Professor W. Kahan
Mathematics Department and
Electrical Engineering and
Computer Science Department
University of California at Berkeley
December 16, 1987



Preface

About This Book

This book is the reference manual for the Standard Apple® Numerics Environment (SANE®). Apple supports SANE on all its current microcomputers and intends to support SANE on future products. The core features of SANE are not exclusive to Apple; rather they are taken from IEEE Standard 754 for binary floating-point arithmetic (see the bibliography at the end of this manual).

In one sense, SANE is an abstraction: a definition of an environment for computer numerics, independent of a specific computer. To have an instance of SANE, you need a language in which to describe operations and an implementation unit—a SANE engine—to carry them out. This manual is divided into four major parts: one part describes the SANE definition, and other parts describe the different implementations of SANE.

About the second edition

This edition is expanded from the first edition in three areas:

- additional explanatory material in Part I
- information about SANE for the 65C816 added to Part II
- a new Part IV with information about using the MC68881 floating-point coprocessor as a SANE engine

The organization of this book is also different from that of the first edition. To make it easier to distinguish the parts, all the chapters are now numbered consecutively and all have different titles. The quick reference guides for the different microprocessors are together at the end of the book.

Parts of the manual

Part I describes the features that are shared by all implementations of SANE; it includes examples that show how to use SANE effectively. There are different SANE implementations for the microprocessors used in different Apple computers.

Whereas high-level languages insulate users from the differences, assembly-language programmers need to know about them. Part II explains the use of assembly-language SANE engines for the 65C816 and for the 6502, Part III does the same for the MC68000, and Part IV discusses the MC68881 coprocessor as a SANE engine.

The facilities of SANE can be provided to users of virtually any high-level programming language, as well as to assembly-language programmers. Appendix A describes Apple's SANE extensions to the high-level languages Pascal and C, as implemented in Apple's Macintosh® Programmer's Workshop (MPW) and Apple IIGS® Programmer's Workshop (APW). Appendix B gives information about gaining access to the SANE engines on different kinds of Apple computers. Appendix C gives a few suggestions for programmers who are porting programs from other computers to run in SANE.

Appendix D is a quick reference guide for the 65C816 and 6502; Appendix E is a similar guide for the MC68000.

Special notation

Throughout this manual, numbers in brackets are references to the annotated bibliography at the end of the manual. Words printed in **boldface** type are defined in the glossary at the end of the manual. Algebraic expressions appear in ordinary type, with variables in *italics*.

❖ *Note:* Information that is incidental to the main text, such as language-specific notes, appears in flagged paragraphs like this one.

Important

Warnings and especially important information appear in boxes like this.

Computer voice

Program fragments—actual computer code, whether in assembly language or in Pascal or other high-level languages—appear in a monospaced type named Courier. Literal names of functions, where they differ from the English names, also appear in Courier. Here is a sample of Courier:

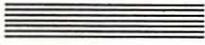
```
if x = 0 or y/x < 3 then writeln ('Eureka!');
```

Hexadecimal numbers

Hexadecimal numbers are flagged with a leading dollar sign, like this: \$FF00.



Apple[®]

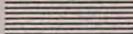


Apple Numerics Manual

Second Edition



Part I



The Standard Apple Numerics Environment

Part I is a general description of SANE. Chapter 1 discusses some of the ways the Standard Apple Numerics Environment (SANE®) makes life easier for programmers. The chapters that follow describe the basic features that are shared by all implementations of SANE:

- the SANE data types, including extended
- conversions between the different types
- operations supported by SANE, including basic arithmetic, auxiliary procedures, and elementary functions
- special values NaN (Not-a-Number) and Infinity
- environment options, such as rounding direction and exception handling

Although Part I uses Pascal in its examples, the facilities of SANE can be provided to users of virtually any high-level programming language, as well as to assembly-language programmers. Appendix A describes Apple's SANE extensions to the high-level languages Pascal and C.



Chapter 1



About IEEE Standard Arithmetic

Example: inverse operations

Suppose your program performs operations that are mutually inverse; that is, operations $y = f(x)$, $x = g(y)$ such that $g(f(x)) = x$. There are many such operations, such as

$$y = \log(x), x = \exp(y)$$

$$y = 375x, x = y/375$$

The computed values $F(x)$ and $G(y)$ will sometimes differ from $f(x)$ and $g(y)$. Even so, if both functions are continuous and well-behaved, and if you always round $F(x)$ and $G(y)$ to the nearest value, you might expect your computer arithmetic to return x when it performs the cycle of inverse operations, $G(F(x))$. It is difficult to predict when this relation will hold for computer numbers. Experience with other computers says it is too much to expect, but SANE very often returns the correct inverse value.

There are two reasons for SANE's good behavior with respect to inverse operations. One is that SANE normally uses the extended data type for intermediate values. When you store the result in a narrower format, SANE rounds the result to the nearest value, often rounding away the errors. Another reason is that SANE rounds so carefully. Even with all operations in, say, single precision, SANE evaluates the expression $3 \times (1/3)$ to 1.0 exactly; some other computers don't. If you find that surprising, you might enjoy running the following example on a computer that doesn't use IEEE arithmetic and on an Apple® computer using SANE. SANE's default rounding gives good results: the Apple computer prints 'No failures'. The program will fail on a computer that doesn't have IEEE arithmetic—in particular, IEEE arithmetic's treatment of halfway cases of rounding to nearest.

```
PROGRAM invop;

VAR
  x, y, a, b: single;
  ix, iy: integer;
  nofail: boolean;

BEGIN
  nofail := true;
  FOR ix := 1 TO 12 DO
    IF ix<>7 AND ix<>11 THEN { so ix is a sum of two powers of 2 }
      FOR iy := 1 TO 50 DO
        BEGIN
          x := ix;
          y := iy;
          a := y/x;
          b := x*a;
          IF b<>y THEN
            BEGIN
              nofail := false;
              writeln('It failed for x =', ix, ' y =', iy)
            END;
          END;
        END;
      IF nofail THEN writeln('No failures');
    END.
END.
```

- ❖ *Note:* This example deliberately avoids the use of extended in order to demonstrate one effect of careful rounding. Declaring the temporary variable *a* to be extended—normally good programming practice with SANE—removes the necessity for restricting *ix* to sums of two powers of 2.

Alternatives to stopping

There are limits to everything; when you exceed them, something exceptional happens. The exceptional events are

- invalid operation
- underflow
- overflow
- division by zero
- inexact result

Many computers either stop on these exceptions or simply ignore them. IEEE Standard arithmetic gives programmers the choice of continuing, stopping, or executing special code.

IEEE arithmetic includes special values NaN (Not-a-Number) and Infinity. When a program encounters an invalid operation, overflow, or division by zero, the arithmetic returns the appropriate NaN or Infinity so that the program can continue. For detailed descriptions of NaN and Infinity, please see Chapter 5, “Infinities, NaNs, and Denormalized Numbers.”

IEEE Standard arithmetic allows (and SANE provides) the option to stop computation when these situations arise, but there are good reasons why you might prefer not to have to stop. The following examples illustrate some of them.

Example: compound conditional statements

Suppose a programmer wants to write a simple statement to perform two tests, one of which can cause an invalid operation such as $0/0$. In Pascal, the statement might look like this:

```
if x = 0 or y/x < 3 then writeln ('Eureka!');
```

When *x* and *y* are both equal to 0, the programmer intends this statement to print “Eureka!” With a Pascal compiler that supports SANE, the statement will produce the desired result. To obtain the desired result on all computers, the programmer would have to be careful and write something more cumbersome. By allowing y/x when *x* and *y* are zero, SANE lets the programmer write simpler code.

This program fragment demonstrates the principal service performed by NaNs: permitting deferred judgments about variables whose values may be unavailable (that is, uninitialized) or the result of invalid operations. Instead of having the computer stop a computation as soon as a NaN appears, you might prefer to have it continue in the hope that whatever caused the NaN will turn out to be irrelevant to the solution.

❖ *Pascal note:* Apple's MPW Pascal compilers include a short-circuit option (\$SC+) that causes the program not to evaluate the second part of a compound conditional if the result value is already determined. Code compiled with that option avoids evaluating 0/0 in the fragment above, but not in the following similar one:

```
if y/x < 3 or x = 0 then writeln ('Eureka!');
```

Searching without stopping

Suppose your program has to search through a database for something like a maximum value that has to be calculated. The search loop might call a subroutine to perform some calculation on the data in each record and return a value for the program to test or compare. For some records, data might be nonexistent or invalid. On many machines, that would cause the program to stop. To avoid having the program stop during the search, you'd have to add tests for all the exceptional cases. With SANE, the subroutine doesn't stop for nonexistent or invalid data; it simply returns a NaN.

This is another example of the way arithmetic that includes NaN allows the program to ignore irrelevancies, even when they cause invalid operations. Using arithmetic without NaNs, you would have to anticipate all exceptional cases and add code to the program to handle every one of them in advance. With NaNs at your disposal, you can handle all exceptional cases after they have occurred.

Example: parallel resistances

Like NaNs, Infinities enable the program to handle cases that otherwise would require special programming to keep from stopping. Here is an example where arithmetic with Infinities is entirely reasonable.

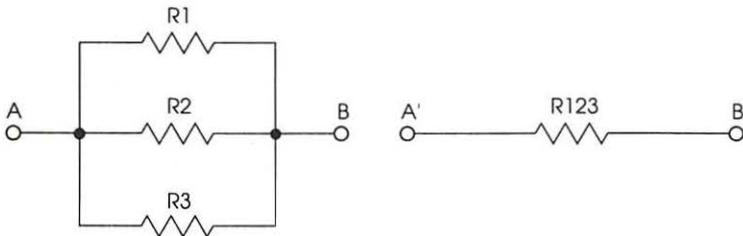


Figure 1-1
Parallel resistances

When three electrical resistances R1, R2, and R3 are connected in parallel as shown in Figure 1-1, their effective resistance is the same as a single resistance whose value R123 is given by this formula:

$$R_{123} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

The formula gives correct results for positive resistance values between 0 (corresponding to a short circuit) and infinity (corresponding to an open circuit) inclusive. On computers that don't allow division by zero, the programmer would have to add tests designed to filter out the cases with resistance values of zero. (Negative values can cause trouble for this formula, regardless of the style of the arithmetic, but that reflects their troublesome nature in circuits, where they can cause instability.)

Arithmetic with Infinities usually gives reasonable results for expressions in which each independent variable appears only once.

Advanced features

SANE also includes more advanced features, such as control of rounding direction and precision, tools for handling exceptional cases, and a set of elementary (transcendental) functions suitable for use as core routines in mathematical functions. These features are only introduced here; for complete descriptions, please see Chapter 7, "Controlling the SANE Environment," and Chapter 8, "Elementary Functions in SANE."

Control of rounding

Rounding is normally carried out to the nearest value, but IEEE Standard arithmetic gives the programmer complete control of rounding precision and direction (see the section "Rounding Direction" in Chapter 7).

Sometimes you may want to know that roundoff has not invalidated a computation. One way to do that would be to force the rounding direction so that you can be sure your results are higher than the exact answer. IEEE arithmetic gives you a means of doing that. Fully developed, this strategy is called *interval arithmetic*. See Kahan [22].

Exception handling

There are three ways for a program to deal with exceptions:

- continue operation
- stop on exceptions, if you think they're causing trouble
- include code to do something special when exceptions happen

The features of IEEE arithmetic enable programs to deal with the exceptions in reasonable ways, as this book explains. There are the special values NaN and Infinity so a program can continue operation: see the sections “Infinities” and “NaNs” in Chapter 5. There are also flags, which a program can test to detect exceptional events, and halts, which transfer control to code for handling special cases: see the section “Exception Flags and Halts” in Chapter 7.

Elementary functions

SANE includes high-precision elementary functions that are consistent with the IEEE Standard and that can be used as building blocks in numerical functions. The elementary functions include the usual logarithmic and exponential functions, plus $\ln(1 + x)$ and $e^x - 1$; financial functions for compound interest and annuity calculations; trigonometric functions; and a random number generator.



Chapter 2



SANE Data Types

SANE provides three **application types** (single, double, and comp) and the **arithmetic type** (extended). Single, double, and extended store floating-point values, and comp stores integral values.

The **extended type** is called the arithmetic type because, to make expression evaluation simpler and more accurate, SANE performs all arithmetic operations in extended precision and delivers arithmetic results to the extended type. The application types **single**, **double**, and **comp** can be thought of as space-saving storage types for the extended-precision arithmetic. (This manual uses the term *extended precision* to denote both the extended precision and the extended range of the extended type.)

The IEEE Standard gives exact specifications for the types single and double, but for the extended type, specifies only lower bounds for precision and exponent range. SANE implementations supported by hardware floating point may adopt the hardware's extended format, which may differ from the 80-bit format described here. You should store external data in one of the application types rather than in the extended type, not only for economy but also because the extended format may vary among different implementations of IEEE arithmetic.

All values representable in single, double, and comp (as well as 16-bit and 32-bit integers) can be represented exactly in extended. Thus, values can be moved from any of these types to the extended type and back without any loss of information.

Choosing a data type

Typically, picking a data type requires that you determine the trade-offs between

- fixed-point or floating-point form
- precision
- range
- memory usage
- speed

The precision, range, and memory usage for each SANE data type are shown in Table 2-2. Effects of the data types on performance (speed) are different for different implementations of SANE. (See Chapter 3, "Conversions in SANE," for information on aspects of conversion relating to precision.)

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by representing money values as integral numbers of cents or mills, which can be stored exactly in the storage format of the comp (for *computational*) type. The sum, difference, or product of any two comp values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in mills. In addition, comp values (such as the results of accounting computations) can be mixed with extended values in floating-point computations (such as compound interest).

Arithmetic with comp-type variables, like all SANE arithmetic, is done internally using extended-precision arithmetic. There is no loss of precision, as conversion from comp to extended is always exact. Space can be saved by storing numbers in the comp format, which is 20 percent shorter than the 80-bit extended format. Non-accounting applications will normally be better served by the floating-point data formats.

❖ *Language note:* In SANE Pascal and C, floating-point constants are stored in extended format.

Table 2-1
Names of data types

SANE type	Pascal type
IEEE single	Real
IEEE double	Double*
SANE comp	Comp*
IEEE extended	Extended*

* SANE extensions to the standard Pascal language

Values represented

The floating-point storage formats (single, double, and extended) provide binary encodings of a sign (+ or -), an **exponent**, and a **significand**. A represented number has the value

$$\pm \text{significand} \times 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

❖ *Note:* This definition applies to both normalized and denormalized numbers. For a discussion of denormalized numbers, please refer to Chapter 5, “Infinities, NaNs, and Denormalized Numbers.”

Range and precision of SANE types

Table 2-2 describes the range and precision of the numeric data types supported by SANE. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table 2-2
SANE types

	Application data types			Arithmetic data type
	Single	Double	Comp	Extended*
Size (bytes:bits)	4:32	8:64	8:64	10:80
Range of binary exponents				
Minimum	-126	-1022	—	-16383
Maximum	127	1023	—	16383
Significant precision				
Bits	24	53	63	64
Decimal digits	7-8	15-16	18-19	19-20
Decimal range approximate				
Maximum positive	3.4E+38	1.7E+308	≈9.2E18	1.1E+4932
Minimum positive norm	1.2E-38	2.3E-308		1.7E-4932
Minimum positive denorm [†]	1.5E-45	5.0E-324		1.9E-4951
Maximum negative denorm [†]	-1.5E-45	-5.0E-324		-1.9E-4951
Maximum negative norm	-1.2E-38	-2.3E-308		-1.7E-4932
Minimum negative	-3.4E+38	-1.7E+308	≈-9.2E18	-1.1E+4932
Infinities[†]	Yes	Yes	No	Yes
NaNs[†]	Yes	Yes	Yes	Yes

* A SANE implementation may have an extended type whose size, precision, or range exceeds these specifications.

† Denorms (denormalized numbers), NaNs (Not-a-Number), and Infinities are defined in Chapter 5.

Whenever possible, SANE stores results in a **normalized number** form, where the high-order bit in the significand is 1 (that is, $1 \leq \text{significand} < 2$). Keeping numbers normalized assures uniqueness of representation and affords maximum precision for a given significand width.

Example: range of single

In single format, the largest representable number is made up as follows:

$$\begin{aligned} \text{significand} &= 2 - 2^{-23} \\ &= 1.111111111111111111111111111111_2 \end{aligned}$$

$$\text{exponent} = 127$$

$$\begin{aligned} \text{value} &= (2 - 2^{-23}) \times 2^{127} \\ &\approx 3.403 \times 10^{38} \end{aligned}$$

The smallest positive normalized number representable in single format is made up as follows:

$$\begin{aligned} \text{significand} &= 1 \\ &= 1.0000000000000000000000000000_2 \end{aligned}$$

$$\text{exponent} = -126$$

$$\begin{aligned} \text{value} &= 1 \times 2^{-126} \\ &\approx 1.175 \times 10^{-38} \end{aligned}$$

For denormalized numbers (see Chapter 5), the smallest positive value representable in single is made up as follows:

$$\begin{aligned} \text{significand} &= 2^{-23} \\ &= 0.0000000000000000000000000001_2 \end{aligned}$$

$$\text{exponent} = -126$$

$$\begin{aligned} \text{value} &= 2^{-23} \times 2^{-126} \\ &\approx 1.401 \times 10^{-45} \end{aligned}$$

Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Each of the diagrams on the following pages is followed by a table that gives the rules for evaluating the number. In each field of each diagram, the leftmost bit is the msb (most significant bit) and the rightmost is the lsb (least significant bit). Symbols used in the diagrams are defined in Table 2-3.

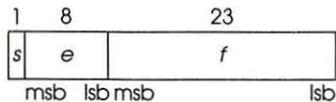
Table 2-3

Symbols used in format diagrams

Symbol	Description
v	Value of number
s	Sign bit
e	Biased exponent
i	Explicit one's bit (extended type only)
f	Fraction
d	Nonsign bits (comp type only)

Single

The 32-bit single format is divided into three fields as shown in Figure 2-1.

**Figure 2-1**

Single format

The value v of the number is determined by these fields as shown in Table 2-4. See Chapter 5 for information about the contents of the f field for NaNs.

Table 2-4

Values of single-format numbers (32 bits)

Biased exponent e	Fraction f	Value v	Class of v
$0 < e < 255$	(any)	$v = (-1)^s \times 2^{(e-127)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-126)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 255$	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 255$	$f \neq 0$	v is a NaN	NaN

Double

The 64-bit double format is divided into three fields as shown in Figure 2-2.

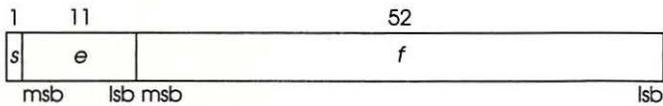


Figure 2-2
Double format

The value v of the number is determined by these fields as shown in Table 2-5.

Table 2-5
Values of double-format numbers (64 bits)

Biased exponent e	Fraction f	Value v	Class of v
$0 < e < 2047$	(any)	$v = (-1)^s \times 2^{(e-1023)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-1022)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 2047$	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 2047$	$f \neq 0$	v is a NaN	NaN

Comp

The 64-bit comp format is divided into two fields as shown in Figure 2-3.

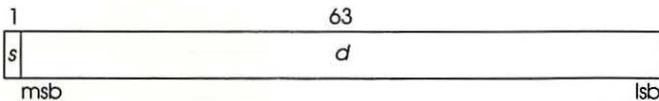


Figure 2-3
Comp format

The value v of the number is determined by these fields as shown in Table 2-6.

Table 2-6
Values of comp-format numbers (64 bits)

Sign bit s	Nonsign bits d	Value v
$s = 1$	$d = 0$	v is the unique comp NaN.
$s = 1$	$d \neq 0$	v is the two's-complement value of the 64-bit representation.
$s = 0$	(any)	v is the two's-complement value of the 64-bit representation.

Extended

The SANE software packages use an 80-bit extended format. The 80-bit extended format is made up of four fields as shown in Figure 2-4.

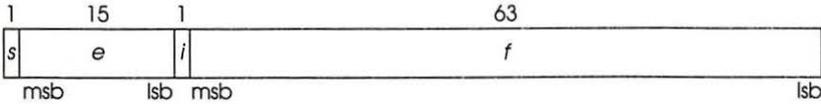


Figure 2-4
80-bit extended format

The MC68881 floating-point coprocessor uses a 96-bit extended format made up of five fields as shown in Figure 2-5. (The 96-bit format is used because the MC68020 accesses data more efficiently on longword boundaries.) Note that the *s*, *e*, *i*, and *f* fields in the 96-bit format are the same as those in the 80-bit format; the shaded field is unused. (For more information about the differences between the software SANE packages and the MC68881 SANE engine, refer to Part IV.)

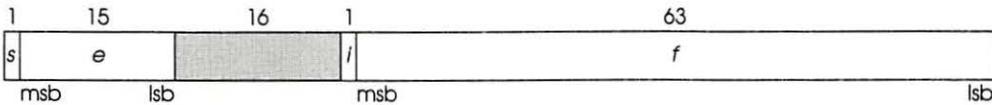


Figure 2-5
96-bit extended format

Table 2-7 shows how the value *v* of the number is determined by the fields shown in Figures 2-4 and 2-5.

Table 2-7
Values of extended-format numbers

Biased exponent <i>e</i>	Integer <i>i</i>	Fraction <i>f</i>	Value <i>v</i>	Class of <i>v</i>
$0 \leq e \leq 32766$	1	(any)	$v = (-1)^s \times 2^{(e-16383)} \times (1.f)$	Normalized
$0 \leq e \leq 32766$	0	$f \neq 0$	$v = (-1)^s \times 2^{(e-16383)} \times (0.f)$	Denormalized
$0 \leq e \leq 32766$	0	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 32767$	(any)	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 32767$	(any)	$f \neq 0$	v is a NaN	NaN



Chapter 3



Conversions in SANE

SANE provides conversions between the extended type and each of the other SANE types (single, double, and comp). A particular SANE implementation also provides conversions between extended and those numeric types supported in its particular larger environment, as illustrated in Figure 3-1. For example, a Pascal implementation includes conversions between extended and the Pascal integer type.

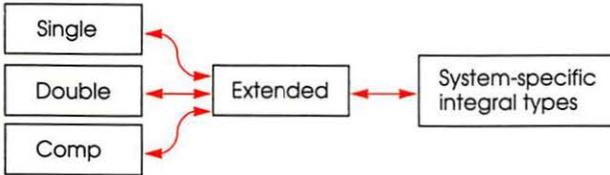


Figure 3-1
Conversions to and from extended format

SANE implementations also provide either conversions between decimal strings and SANE types, or conversions between a decimal record type and SANE types, or both. Conversions between decimal records and decimal strings may also be included.

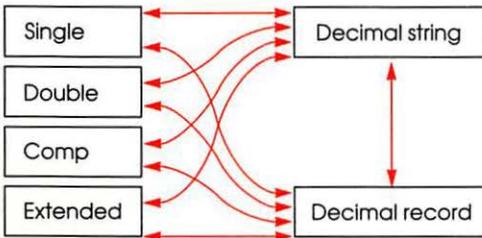


Figure 3-2
Conversions to and from decimal formats

❖ *Language note:* This chapter, along with others throughout Part I, uses statements in Pascal to illustrate the features of SANE. These statements do not define an interface to SANE for other languages; for that, please refer to Appendix A, “SANE in High-Level Languages,” the assembly-language sections of this book, and language reference manuals.

Conversions between extended and single or double

A conversion to extended is always exact. A conversion from extended to single or double moves a value to a storage type with less range and precision, and sets the overflow, underflow, and inexact exception flags as appropriate. (See Chapter 7, “Controlling the SANE Environment,” for a discussion of exception flags.)

❖ *Pascal note:* With Apple’s SANE extensions (described in Appendix A), Pascal converts between extended format and single or double formats in the following cases:

- during expression evaluation
- on assignments to or from real or double
- on parameter passing by value
- by explicit conversion functions

Conversions from extended to single or double

```
FUNCTION Num2Real( x: extended ): real;
```

```
FUNCTION Num2Double( x: extended ): double;
```

These conversion functions are useful when you want to coerce a result to single or double format without using a separate assignment statement. For example, you could use a conversion function to test a single-format variable against an expression, as shown in the example that follows.

Example: Is x negligible?

Suppose you need to find out whether some single-precision value x is negligible compared with some other single-precision value y . You might perform a calculation to determine this, such as

```
if ((y + x) = y) then Xneglig := true;
```

Because SANE implementations evaluate $(y + x)$ in extended format and perform the comparison in extended, the test won’t work the way you might expect it to. The statement above gives `Xneglig` the value `true` only for the far tinier values of x that are negligible with respect to extended precision. To determine whether x is negligible merely with respect to single precision, you could round $(y + x)$ to single, like this:

```
if (Num2Real(y + x) = y) then Xneglig := true;
```

Conversions to comp and other integral formats

```
FUNCTION Num2Integer( x: extended ): integer;
```

```
FUNCTION Num2Longint( x: extended ): longint;
```

```
FUNCTION Num2Comp( x: extended ): comp;
```

The SANE routines convert numbers to integral formats by first rounding to an integral value (honoring the current rounding direction) and then, if possible, delivering this value to the destination format. If the source operand of a conversion from extended to comp is a NaN, an Infinity, or out-of-range for the comp format, then the result is the comp NaN. If the source is an Infinity or is out-of-range, the invalid exception is signaled.

For conversion to a system-specific integer type (that doesn't have an appropriate representation for the exceptional result), if the source operand is a NaN, an Infinity, or is out-of-range for that type, then invalid is signaled.

- ❖ *Note:* The result values of conversions of NaNs, Infinities, and out-of-range values to two's-complement integer types may be different in SANE implementations for different microprocessors. Please refer to the section on conversions in the part of this book for your machine's microprocessor.

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (to nearest), conversions to comp or to a system-specific integer type will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Rounding is discussed in detail in Chapter 7, "Controlling the SANE Environment.")

- ❖ *Pascal note:* Programs can use assignments to convert floating-point variables to comp format, but not to integer. For that, you must use the explicit functions Round, Trunc, Num2Integer, and Num2Longint. Note that Round is not sensitive to the setting of rounding direction, but always performs Pascal-type rounding to nearest (rounding away from zero on halfway cases).
- ❖ *Language note:* In general, where languages define the rounding behavior for conversion to integer format or to integral value, SANE languages conform.

Conversions between binary and decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate.

Example: binary approximation of decimal fractions

Some fractions that have exact decimal representations can also be represented exactly in binary; for example, $1/2$ (0.5 exactly in decimal) can be represented exactly in binary as 0.1. Other fractions with exact representations in decimal have repeating digits in binary, as shown in Table 3-1. For example, $1/10$, or decimal 0.1 exactly, is 0.000110011... in binary. Its closest representation in single format is closer to 0.100000001_{10} than to 0.100000000_{10} .

❖ *Note:* Errors of this kind are unavoidable in any computer approximation of real numbers. Because of these errors, sums of fractions are often slightly incorrect. For example, $4/3 - 5/6$ is not computed exactly equal to $1/2$ on any computer that rounds correctly in either binary or decimal floating-point arithmetic.

Table 3-1
Approximations of fractions

Fraction	Decimal approximation*	Binary approximation†
$1/10$	0.1000000000‡	0.000110011001100110011001101
$1/2$	0.5000000000‡	0.100000000000000000000000‡
$4/3$	1.333333333	1.010101010101010101010101011
$5/6$	0.8333333333	0.1101010101010101010101010101
$4/3 - 5/6$	0.4999999997	0.100000000000000000000000001

* 10 significant digits, rounded to nearest

† single format, rounded to nearest

‡ exact value

Accuracy of decimal-to-binary conversions

As binary storage formats generally provide only close approximations to decimal values, it is important that conversions between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best—that is, correctly rounded—binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-to-decimal conversions should obtain this best value to reduce accumulated errors.

Conversion routines in SANE implementations meet or exceed the stringent error bounds specified by the IEEE Standard. This means that, even though in extreme cases the conversions do not deliver the correctly rounded results, the results they do deliver are very nearly as good as the correctly rounded results. (The IEEE Standard does not specify error bounds for conversions involving values beyond the double format. See the IEEE Standard [21] for a more detailed description of error bounds.)

Reading and writing decimal data

Suppose a program converts values from decimal to binary and back repeatedly. Such conversion cycles would occur, for example, in repeated execution of a program that updates a decimal file on a binary computer. Each time the program runs, it deliberately changes only a handful of values, but *all* the values get converted from decimal to binary and back again. A conversion strategy used by some computers just drops extra digits. Such rounding by truncation can cause severe downward drift when applied repeatedly. Using SANE arithmetic with rounding to nearest, the values don't drift when you run the program repeatedly. That is, even though the conversions may change a few values the first time you run the program, there won't be any further changes on subsequent conversions.

Figure 3-3 is a graphical model of such a conversion cycle with rounding to nearest, where the vertical marks represent decimal and binary computer numbers on the number line. The one-way arrow shows a decimal-to-binary conversion that does not get converted back to the original decimal value; the two-way arrow shows subsequent conversions returning the same value. In all cases, repeated conversions after the first give the same binary value; the error doesn't go on increasing.

Numbers with similar precision

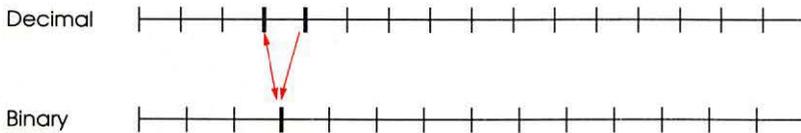


Figure 3-3
Conversion cycle with first-time error

What's more, if the binary format has enough extra precision beyond that of the decimal format, SANE returns the original value the first time. The two-way arrow in Figure 3-4 shows a conversion cycle with different degrees of precision; in this situation, the nearest decimal value to the binary result is always the original decimal value.

You might ask, "How much precision is that?" Or, more precisely, "What is the largest number of decimal digits for which you can be certain that conversion from decimal to binary and back will return the exact original value?" The answer is different for different formats: for single, it is 6; for double, 15.

Numbers with different precision

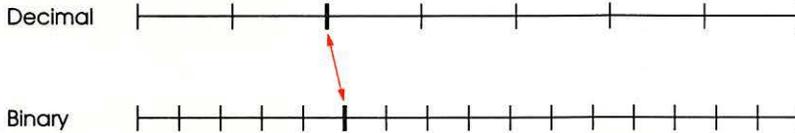


Figure 3-4

Conversion cycle with correct result

A similar question might arise for outputs: you might want to know how many decimal digits to print, assuming someone would use the printed value as input. The question then becomes, “How many decimal digits should you print to be confident that conversion from binary to decimal and back will return the exact original value?” For single, the answer is 9; for double, 17.

❖ *Note:* These values bracket the ones given in Table 2-2.

Conversions from decimal strings to SANE types

```
DecStr = string[DecStrLen];
```

```
FUNCTION Str2Num(s: DecStr): extended;
```

Languages may provide routines to convert numeric decimal strings to the SANE data types. The accepted syntax is formally defined, using Backus-Naur form, in Table 3-2. Examples of acceptable input are

```
123      123.4E-12      -123.      .456      3e9      -0
-INF     Inf      NAN(12)      -NaN()      nan
```

The 12 in `NAN(12)` is a NaN code (see Chapter 5 for a description of NaN codes).

❖ *Language note:* Conversions take place in the following cases:

- use of decimal constants in source
- input of decimal strings (by procedures such as `read` in Pascal)
- calls to explicit routines (such as `Str2Num` in Pascal)

Table 3-2
Syntax for string conversions

Object	Definition
<decimal number>	::= [[space tab]] <left decimal>
<left decimal>	::= [+ -] <unsigned decimal>
<unsigned decimal>	::= <finite number> <Infinity> <NAN>
<finite number>	::= <significand> [<exponent>]
<significand>	::= <integer> <mixed>
<integer>	::= <digits> [.]
<digits>	::= {0 1 2 3 4 5 6 7 8 9}
<mixed>	::= [<digits>] . <digits>
<exponent>	::= E [+ -] <digits>
<Infinity>	::= INF
<NAN>	::= NAN[[<digits>]]

Note: In the table, square brackets enclose optional items, braces (curly brackets) enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the *E* marking the exponent field, may be either uppercase or lowercase.

❖ *Pascal note:* With Apple's SANE extensions (described in Appendix A, "SANE in High-Level Languages"), Pascal accepts <Infinity> and <NAN> as decimal string input for `read` (see Table 3-2). In source code files, `INF` is a predefined constant and `NAN` is a function in the SANE library.

Decform records and conversions from SANE types to decimal strings

```
DecForm = RECORD
    style: (FloatDecimal, FixedDecimal);
    digits: integer
END;

PROCEDURE Num2Str(f: DecForm; x: extended; VAR s: DecStr);
    {s <-- x according to format f}
```

Languages may provide routines to convert SANE data types to numeric decimal strings. Each conversion to a decimal string is controlled by a **decform record** like that defined above in Pascal. A decform record contains two fields:

- `style`, a 16-bit word
- `digits`, a 16-bit integer

The value of `style` equals 0 for floating and 1 for fixed. The value of `digits` is the number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style. (The value of `digits` may be negative if the style is fixed.) Decimal strings resulting from these conversions are always acceptable input for conversions from decimal strings to SANE types. For more information about the use of the `deform` record, see the sections “Conversions From SANE Types to Decimal Records” and “Conversion From Decimal Records to Decimal Strings,” later in this chapter.

❖ *Note:* Formatting details, such as the representations of 0 and 1 in the 16-bit `style` word, are implementation dependent. Please refer to Chapter 17, “65C816 SANE Scanners and Formatter,” for details about the 65C816 and 6502 implementations and to Chapter 25, “MC68000 SANE Scanners and Formatter,” for details about the 68000 implementation.

The decimal record type

```
Decimal = RECORD
    sgn: 0..1;
    exp: integer;
    sig: string[SigDigLen]
END;
```

The decimal record type provides an intermediate unpacked form for programmers who wish to do their own parsing of numeric input or formatting of numeric output. The decimal record format has three fields:

- `sgn`, a 16-bit word
- `exp`, a 16-bit integer
- `sig`, a string

The value represented is

$$(-1)^{sgn} \times sig \times 10^{exp}$$

when the length of `sig` is 18 or less (see the note that follows). String `sig` contains the significand as a decimal integer in the form of a Pascal string, that is, with the string length in the zeroth byte (`sig[0]`) and the initial character of the string in the first byte (`sig[1]`). The value of `sgn` is 0 for plus and 1 for minus. For example, if `sgn` equals 1, `exp` equals -3, and `sig` equals '85' (string length `sig[0]` equals 2, not shown), then the number represented is -0.085.

❖ *Note:* The maximum length of the string `sig` is implementation dependent; some implementations allow additional information in characters beyond the 18th. Also, the representations of 0 and 1 in the 16-bit word `sgn` are implementation dependent. Please refer to Chapter 17 for details about the 65C816 and 6502 implementations and to Chapter 25 for details about the MC68000 implementation.

Conversions from decimal records to SANE types

FUNCTION Dec2Num(d: Decimal): extended;

Conversions from the decimal record type handle any `sig` digit-string of length 18 or less (with an implicit decimal point at the right end). The following special cases apply:

- If `sig[1]` is '0' (zero), the decimal record is converted to zero. For example, a decimal record with `sig = '0913'` is converted to zero.
- If `sig[1]` is 'N', the decimal record is converted to a NaN. Except when the destination is of type `comp` (which has a unique NaN), the succeeding characters of `sig` are interpreted as a hex representation of the result's significand: if fewer than four characters follow the `N`, then they are right justified in the high-order 15 bits of the field `f` illustrated in the section "Formats" in Chapter 2, "SANE Data Types"; if four or more characters follow the `N`, then they are left justified in the result's significand; if no characters, or only zeros, follow the `N`, then the result NaN code is set to `nanzero` (that is, the number \$15 is stored, right-justified, in the high-order 15 bits of the `f` field; see Chapter 5 for a description of NaN codes).
- If `sig[1]` is 'I' and the destination is not of `comp` type, the decimal record is converted to an Infinity. If the destination is of `comp` type, the decimal record is converted to a NaN and `invalid` is signaled.
- Other special cases produce undefined results.

Conversions from SANE types to decimal records

PROCEDURE Num2Dec(f: DecForm; x: extended; VAR d: Decimal)

Each conversion to a decimal record `d` is controlled by a `decform` record `f`. (`DecForm` is defined in a previous section.) All implementations allow at least 18 digits to be returned in `sig`. The implied decimal point is at the right end of `sig`, with `exp` set accordingly.

Zeros, Infinities, and NaNs are converted to decimal records with `sig` parts '0', 'I', and strings beginning with 'N', respectively, whereas `exp` is undefined. For NaNs, 'N' may be followed by a hex representation of the input significand. The third and fourth hex digits following the `N` give the NaN code. For example, 'N4021000000000000' has NaN code \$21.

Unusual cases for decimal records

When the number of digits specified in a `decform` record exceeds an implementation maximum (which is at least 18), the result is undefined.

A number may be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for `sig` is 18, then 10^{15} (which requires 16 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than two digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the SANE implementation) one of two results is returned: an implementation may return the most significant digits of the number in `sig` and set `exp` so that the decimal record contains a valid floating-style approximation of the number; alternatively, an implementation may simply set `sig` to the string '?'. Note that in any implementation, the following test (using Pascal syntax) determines whether a nonzero finite number is too large for the chosen fixed style.

```
VAR
  d: Decimal;
  f: DecForm;
  TooBig: Boolean;

TooBig := (-d.exp <> f.digits) or (d.sig[1] = '?');
```

For fixed-point formatting, SANE treats a negative value for `digits` as a specification for rounding to the left of the decimal; for example, `digits = -2` means round to hundreds. For floating-point formatting, a negative value for `digits` gives unspecified results.

Conversions between decimal formats

A SANE implementation may provide a scanner for converting from decimal strings to decimal records and a formatter for converting from decimal records to decimal strings.

Conversion from decimal strings to decimal records

```
PROCEDURE Str2Dec( s: DecStr; var Index: integer; var d: Decimal;
var ValidPrefix: boolean );
```

The SANE scanner is designed for use either with fixed strings or with strings being received interactively character by character. On input, the integer argument `Index` gives the starting position in the string; on output, its value is one greater than the position of the last character in the numeric substring just parsed. The scanner parses the longest possible numeric substring; if no numeric substring is recognized, then the value of `Index` remains unchanged. The scanner returns a Boolean argument `ValidPrefix` indicating that the input string, beginning at the input index, is a valid numeric string or a prefix of a valid numeric string. The accepted input for this conversion is the same as for conversions from decimal strings to SANE types; see the earlier section “Conversions From Decimal Strings to SANE Types.” Output is the same as for conversions from SANE types to decimal records; see the earlier section “Conversions From SANE Types to Decimal Records.” The scanner signals no exceptions. It faithfully converts all values, within the extended range, that are representable in the decimal record format.

Table 3-3
Examples of conversions to decimal records

Input string	Index		Output value	Valid-prefix
	In	Out		
'12'	1	3	12	TRUE
'12E'	1	3	12	TRUE
'12E-'	1	3	12	TRUE
'12E-3'	1	6	12E-3	TRUE
'12E-X'	1	3	12	FALSE
'12E-3X'	1	6	12E-3	FALSE
'x12E-3'	2	7	12E-3	TRUE
'IN'	1	1	NAN	TRUE
'INF'	1	4	INF	TRUE

To convert floating-point strings embedded in text, parse to the beginning of a floating-point string (`[+ | -] digit`) and pass the current scan location as the index into the text. The conversion routine will return the value scanned and a new value of the index for continued parsing.

You may need to distinguish those numeric ASCII strings that represent values of an integer format. You can do this by scanning the source, looking for integer syntax. You can handle integers yourself and send to the SANE scanner any strings with floating-point syntax (that is, containing `.`, `E`, or `e`). You may want to pass along to the scanner any strings that cause integer overflow.

The SANE scanner can be used to process not only static strings but also strings received character by character. Here is a sample algorithm in Pascal:

```
{ Initialize string.}
ScanStr := '';

{Loop until string is not a valid prefix.}
REPEAT
  {
    ...Code to get next character and append to string goes here...
  }
  {Scan string.}
  Index := 1;
  Str2Dec(ScanStr, Index, DecRec, ValidPrefix);
UNTIL ValidPrefix = false;

{Convert from decimal to SANE-type result.}
Result := Dec2Num(DecRec);
```

Conversion from decimal records to decimal strings

```
PROCEDURE Dec2Str( f: DecForm; d: Decimal; var s: DecStr );
```

The SANE formatter is controlled by a decform record *f*, as shown earlier in the section “Decform Records and Conversions From SANE Types to Decimal Strings.” Input *d* is a decimal record (with fields *sgn*, *exp*, and *sig*), the same as for conversions from decimal records to SANE types. The formatter is always exact and signals no exception.

Floating-style decimal output

If the *style* field of the decform record equals 0 (in Pascal, *f.style* = *FloatDecimal*), the output string is formatted in floating style, with the *digits* field specifying the number of significant digits required. Output in floating style is represented in the following format; Table 3-4 defines its components.

$[- |]m[. nnn]e[+ | -]dddd$

Table 3-4
Format of decimal output string in floating style

Component	Description
minus sign (-) or space	According as <i>sgn</i> is 1 or 0
<i>m</i>	Single digit, 0 only if value represented is 0
point (.)	Present if <i>digits</i> > 1
<i>nnn</i>	Digit-string, present if <i>digits</i> > 1
<i>e</i>	The letter <i>e</i>
plus sign (+) or minus sign (-)	According as $exp \geq 0$ or $exp < 0$
<i>dddd</i>	One to four exponent digits

Fixed-style decimal output

If the `style` field of the `decform` record equals 1 (in Pascal, `f.style = FixedDecimal`), the output string is formatted in fixed style, with the `digits` field specifying the number of digits to follow the decimal point. All output in fixed style is represented in the following format; Table 3-5 defines its components.

`[-]mmm[. nnn]`

Table 3-5

Format of decimal output string in fixed style

Component	Description
minus sign (-)	Present if <code>sgn = 1</code>
<code>mmm</code>	Digit-string: at least one digit but no superfluous leading zeros
point (.)	Present if <code>digits > 0</code>
<code>nnn</code>	Digit-string of length equal to <code>digits</code> , present if <code>digits > 0</code>

Note that if `sgn` equals 0, then floating-style output begins with a space, but fixed-style output does not.

Unusual cases for decimal strings

Negative values for `digits` are treated as 0 for fixed formatting, but give unspecified results in floating format.

The formatter never returns fewer significant digits than are contained in `sig`. However, if the `decform` record calls for more significant digits than are contained in `sig`, then the formatter pads with zeros as needed.

If more than 80 characters are required to honor `digits`, then the formatter returns the string `'?'`.

NaNs are formatted as `NAN (ddd)`, where `ddd` is a three-decimal-digit code telling the origin of the NaN; Infinities are formatted as `INF`. A leading sign or space is included according to the style convention.

Alignment and field width

With floating style, numbers formatted using the same value for `digits` have aligning decimal points and *e*'s. To assure also that numbers have the same width, pad the exponent-digits field with spaces to a width of 4. (Extended values may require four exponent digits.) For example, if `digits = 12`, then pad `12 + 8 - length(s)` spaces on the right of the result string `s`. The value 8 accounts for the sign, point, *e*, exponent sign, and four exponent digits. Note that this scheme gives the correct field width for NaNs and Infinities too.

With fixed style, numbers formatted using the same value for `digits` have aligning decimal points if enough leading spaces are added to the result string `s` to attain a fixed width, which must be no narrower than the widest `s`.

Example: decimal records

Suppose you have an accounting program that computes exact values using binary numbers of pennies and prints outputs in dollars and cents. If you simply divide the number of pennies by 100 to get dollars, you incur errors due to the fact that hundredths are not exact in binary. One way to print out exact values in dollars is to convert the number of pennies to a decimal record, perform the division by adjusting the exponent, and print the result.

```
VAR
    df:          DecForm;
    pennies:     extended;
    dPennies:    decimal;          { decimal value for pennies }
    dollars:     DecStr;          { string to print as $$$.$¢¢ }

BEGIN
    df.style := FixedDecimal;
    df.digits := 0;              { start with 0 digits after dec. pt. }

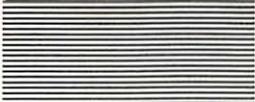
    Num2Dec(df, pennies, dPennies); { decimal pennies }
    dPennies.exp := dPennies.exp - 2; { divide by 100 }

    df.digits := 2;              { request 2 digits after dec. pt. }
    Dec2Str(df, dPennies, dollars); { dollar string to print }
END;
```

Table 3-6

Examples of conversions to decimal strings

Style	digits	sgn	exp	sig	Result string s
Float	3	0	-2	'123'	' 1.23e+0'
Float	3	1	-4	'123'	'-1.23e-2'
Float	1	0	200	'123'	' 1.23e+202'
Float	5	1	1000	'123'	'-1.2300e+1002'
Float	1	0	-30	'4'	' 4e-30'
Float	1	1	0	'0'	'-0e+0'
Float	30	0	0	'1'	' 1.000000000000000000000000000000e+0'
Float	76	0	0	'1'	'?'
Float	76	1	0	'1'	'?'
Float	5	0	-98	'N0024'	' NAN (036) '
Float	2	1	103	'N0015'	'-NAN (021) '
Float	2	0	0	'I'	' INF '
Float	2	1	-217	'I'	'-INF '
Fixed	3	0	-3	'12345'	'12.345'
Fixed	3	1	-3	'12345'	'-12.345'
Fixed	5	0	-3	'12345'	'12.34500'
Fixed	3	1	-5	'1234567'	'-12.34567'
Fixed	0	0	0	'12345'	'12345'
Fixed	0	1	3	'12345'	'-12345000'
Fixed	-2	0	2	'12345'	'1234500'
Fixed	-2	1	1	'12345'	'-123450'
Fixed	3	0	63	'0'	'0.000'
Fixed	-3	1	0	'0'	'-0'
Fixed	5	0	74	'1'	'?'
Fixed	4	1	74	'1'	'?'
Fixed	5	0	-98	'N0024'	'NAN (036) '
Fixed	2	1	103	'N0015'	'-NAN (021) '
Fixed	2	0	0	'I'	' INF '
Fixed	2	1	-217	'I'	'-INF '



Chapter 4



Expression Evaluation in SANE

SANE arithmetic is extended-based. Arithmetic operations produce results with extended precision and extended range. For minimal loss of accuracy in more complicated computations, you should use extended temporary variables to store intermediate results.

Extended-precision expression evaluation

To obtain the full benefits of SANE, floating-point expressions should be evaluated using extended format. High-level languages that support SANE evaluate all noninteger numeric expressions to extended precision, regardless of the types of the operands. If you are a Pascal or C programmer, refer to Appendix A, "SANE in High-Level Languages."

Example: expression evaluation

If C is of type `comp` and `MaxComp` is the largest `comp` value, then the right side of

```
C := (MaxComp + MaxComp) / 2;
```

would be evaluated in extended to the exact result $C = \text{MaxComp}$, even though the intermediate result `MaxComp + MaxComp` exceeds the largest possible `comp` value.

If your program can generate intermediate values that are fractional or out-of-range, you may need to check the `inexact` and `overflow` flags to determine whether the final results are free of error. For example, the expression

```
MaxComp * 5 - MaxComp * 4
```

sets the `inexact` flag and returns an even result (even though `MaxComp` is odd) because `MaxComp * 5` is `inexact` even in extended format.

Using extended temporaries

An algorithm that works well if carried out using substantially more precision than that of the given data and desired solution may fail ignominiously if the precision of the arithmetic is only slightly greater than the data and solution. Compilers that support SANE use extended format for temporary variables, thereby avoiding this kind of problem. Programmers can further reduce the effects of round-off error, overflow, and underflow on the final results by declaring their temporary variables as extended.

Example: extended temporaries

To compute the single-precision sum

$$S = X[1] \times Y[1] + X[2] \times Y[2] + \dots + X[N] \times Y[N]$$

where X and Y are arrays of type single, declare an extended variable xs and compute the extended-format sum as shown in the following example:

```
VAR
  X: ARRAY [1..N] OF real;
  Y: ARRAY [1..N] OF real;
  s: real;
  xs: extended;

BEGIN;
  xs:= 0;
  FOR i:= 1 TO N DO xs:= xs+X[i]*Y[i]; {extended-precision arithmetic}
  s:= xs;                               {deliver final result to single.}
END;
```

Even when input and output values have only single precision, it may be difficult to prove that single-precision arithmetic is sufficient for a given calculation. Using extended-precision arithmetic for intermediate values often improves the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the extended type for intermediate results may yield correct final results in the single type in cases when using the single type for intermediate results would cause an overflow or a catastrophic underflow. Extended-precision arithmetic is also useful for calculations involving double or comp variables: see “Example: Expression Evaluation.”

- ❖ *Note:* Compilers that support SANE use extended format for intermediate values. To obtain the identical values when debugging, you must do the same; that is, make sure any temporary variables you introduce are in extended format.

Expression evaluation and the IEEE Standard

The IEEE Standard encourages extended-precision expression evaluation. On rare occasions, extended evaluation produces results slightly different from those produced by other IEEE implementations that lack extended evaluation. Thus, in a single-only IEEE implementation,

$z := x + y;$

with x , y , and z all single, is evaluated in one single-precision operation, with at most one rounding error. Under extended evaluation, however, the addition $x + y$ is performed in extended; then the result is coerced to the single precision of z , with at most two rounding errors. Both implementations conform to the standard.

Programmers using SANE can obtain the effect of an IEEE implementation having only single-precision or double-precision numbers by using rounding precision control, as described in Chapter 7, “Controlling the SANE Environment.”

❖ *Language note:* SANE Pascal and C compilers generate floating-point constants in extended format.



Chapter 5



Infinites, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point systems, IEEE floating-point arithmetic also supports Infinities, NaNs, and denormalized numbers.

Many programs deliver correct results despite the transient appearance of NaN or Infinity. You can use SANE without knowing anything about NaNs and Infinities unless you perform a calculation that would cause a malfunction on a machine that doesn't have them. When that happens, NaNs and Infinities and the flags associated with them help you diagnose the malfunction and deal with it appropriately.

Infinities

An **Infinity** is a special bit pattern that can arise in one of two ways:

1. When a SANE operation should produce a mathematical infinity (such as $1/0$), the result is an Infinity.
2. When a SANE operation attempts to produce a number with magnitude too great for the number's intended floating-point storage format, the result may be a value with the largest possible magnitude or it may be an Infinity (depending on the current rounding direction).

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The Infinities, one positive (abbreviated +INF) and one negative (-INF), generally behave as suggested by the theory of limits. For example:

- Adding 1 to +INF yields +INF.
- Dividing -1 by +0 yields -INF.
- Dividing 1 by -INF yields -0.

Each of the storage types single, double, and extended provides unique representations for +INF and -INF. The comp type has no representations for Infinities. (An Infinity moved to the comp type becomes the comp NaN.)

- ❖ *Pascal note:* For input and output, Infinities are written as INF and -INF. In source code, INF is a predefined constant.

NaNs

When a SANE operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN** (Not-a-Number). For example, 0 divided by 0, +INF added to -INF, and $\text{sqrt}(-1)$ yield NaNs. A NaN can occur in any of the SANE storage types (single, double, extended, and comp); but, generally, system-specific integer types have no representation for NaNs. NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN (and has the same NaN code). If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs**, the usual kind produced by floating-point operations; and **signaling NaNs**.

When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and a quiet NaN is the delivered result. Signaling NaNs are not created by any SANE operations. The most significant bit of the fraction field f (illustrated in the section “Formats” in Chapter 2, “SANE Data Types”) distinguishes quiet and signaling. It is set for quiet NaNs and clear for signaling NaNs.

A NaN in a floating-point format may have an associated NaN code that indicates the NaN’s origin. These codes are listed in Table 5-1. The NaN code is the 8th through 15th most significant bits of the fraction field f (illustrated in the section “Formats” in Chapter 2). The comp NaN is unique and has no NaN code.

❖ *Pascal note:* For input and output, NaNs are written as `NAN(ddd)`, where `ddd` is the decimal representation of the NaN code. In source code, `NaN` is the function

```
FUNCTION NAN( i: integer ): extended;  
where the input argument i gives the NaN code.
```

Table 5-1
SANE NaN codes

Name	Dec	Hex	Meaning
NANSQRT	1	\$01	Invalid square root, such as $\text{sqrt}(-1)$
NANADD	2	\$02	Invalid addition, such as $(+INF) - (+INF)$
NANDIV	4	\$04	Invalid division, such as $0/0$
NANMUL	8	\$08	Invalid multiplication, such as $0 \times INF$
NANREM	9	\$09	Invalid remainder or mod such as $x \text{ rem } 0$
NANASCBIN	17	\$11	Attempt to convert invalid ASCII string
NANCOMP	20	\$14	Result of converting comp NaN to floating
NANZERO	21	\$15	Attempt to create a NaN with a zero code
NANTRIG	33	\$21	Invalid argument to trig routine
NANINVTRIG	34	\$22	Invalid argument to inverse trig routine
NANLOG	36	\$24	Invalid argument to log routine
NANPOWER	37	\$25	Invalid argument to x^i or x^y routine
NANFINAN	38	\$26	Invalid argument to financial function

Note: All NaNs created by the MC68881 coprocessor have NaN code 255 (\$FF). Codes of NaNs passed to the MC68881 are preserved.

Denormalized numbers

Whenever possible, floating-point numbers are **normalized** to keep the leading significant bit 1: normalization maximizes the resolution of the storage type and ensures that representations are unique. When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a denormalized representation. A **denormalized number** is a nonzero number that is not normalized and whose exponent is the minimum exponent for the storage type.

❖ *Note:* Some references use the term *subnormal* instead of *denormalized*.

Example: gradual underflow

Table 5-2 shows how a single-precision value A_0 becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called **gradual underflow**. In the table, values $A_1 \dots A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number in the single type.

Table 5-2
Example of gradual underflow

Variable or operation	Value	Comment
A_0	1.100 1100 1100 1100 1100 1101 * 2^{-126}	$\approx 0.1_{10} * 2^{-122}$
$A_1 = A_0/2$	0.110 0110 0110 0110 0110 0110 * 2^{-126}	Inexact*
$A_2 = A_1/2$	0.011 0011 0011 0011 0011 0011 * 2^{-126}	Exact result
$A_3 = A_2/2$	0.001 1001 1001 1001 1001 1010 * 2^{-126}	Inexact*
	The process continues
$A_{22} = A_{21}/2$	0.000 0000 0000 0000 0000 0011 * 2^{-126}	Exact result
$A_{23} = A_{22}/2$	0.000 0000 0000 0000 0000 0010 * 2^{-126}	Inexact*
$A_{24} = A_{23}/2$	0.000 0000 0000 0000 0000 0001 * 2^{-126}	Exact result
$A_{25} = A_{24}/2$	0.0	Inexact*

* Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

Figure 5-1 illustrates the relative magnitudes of normalized and denormalized numbers in single precision. It shows two versions of the number line: first without denormalized numbers, then with denormalized numbers in place. The spacing of the vertical marks indicates the relative density of numbers in each part of the number line; the denormalized numbers have the same density as the normalized numbers in the smallest **binade**.

❖ *Note:* The figure shows only the *relative* density of the numbers; in reality, the density is immensely greater than it is possible to show in such a figure. For example, there are 2^{23} (8,388,608) single-precision numbers x in the interval $2^{-126} \leq x < 2^{-125}$.

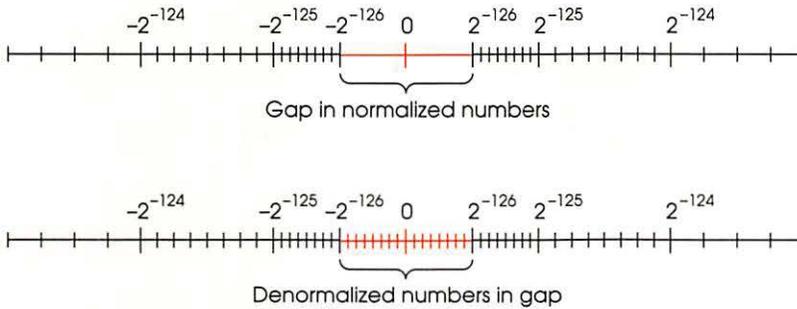


Figure 5-1
Denormalized single-precision numbers on the number line

Why gradual underflow?

The use of denormalized numbers makes the following statement true for all real numbers:

$$x - y = 0 \text{ if and only if } x = y$$

This statement is not true for most older systems of computer arithmetic, because they exclude denormalized numbers. For those systems, the smallest nonzero number is a normalized number with the minimum exponent; when the result of an operation is smaller than that smallest normalized number, the system delivers zero as the result. For such **flush-to-zero** systems, if $x \neq y$ but $x - y$ is smaller than the smallest normalized number, then $x - y$ is computed as 0. Systems using denormalized numbers do not have this defect: the value of $x - y$, although denormalized, is correctly nonzero.

Another advantage of gradual underflow is that error analysis involving small values is much easier without the gap around zero shown in Figure 5-1. See Demmel [14].

Sign of zero

Each floating-point format (single, double, and extended) has two representations for zero: $+0$ and -0 . The two zeros compare as equal: $+0 = -0$; however, their behaviors in the arithmetic are slightly different.

Ordinarily, the sign of zero doesn't matter except (possibly) for a function discontinuous at zero. Though the two forms are numerically equal, they are not identical; a program can distinguish $+0$ from -0 by operations such as division by zero or performing the SANE CopySign or SignNum function.

❖ *Language note:* Some languages define their own sign inquiry functions that ignore the sign of zero.

The sign of zero obeys the usual sign laws for multiplication and division. For example, $(+0) \times (-1) = -0$ and $1/(-0) = -\text{INF}$. Because extreme negative underflows yield -0 , expressions like $1/x^3$ produce the correct sign for Infinity when x is **tiny** and negative. Addition and subtraction produce -0 only in these cases:

- $(-0) - (+0)$ yields -0
- $(-0) + (-0)$ yields -0

When rounding downward, with x finite,

- $x - x$ yields -0
- $x + (-x)$ yields -0

The square root of -0 is -0 .

The sign of zero is important in complex arithmetic. See Kahan [25].

Inquiries: class and sign

```
NumClass = (SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum );  
  
FUNCTION ClassReal( x: real ): NumClass;  
FUNCTION ClassDouble( x: double ): NumClass;  
FUNCTION ClassComp( x: comp ): NumClass;  
FUNCTION ClassExtended( x: extended ): NumClass;  
  
FUNCTION SignNum( x: extended ): integer;  
    {0 if sign bit clear, 1 if sign bit set}
```

Each valid representation in a SANE data type (single, double, comp, or extended) belongs to exactly one of these classes:

- signaling NaN
- quiet NaN
- Infinity
- zero
- normalized
- denormalized

SANE implementations provide the user with the facility to determine easily the class and sign of any valid representation.



Chapter 6



Arithmetic Operations, Comparisons, and Auxiliary Procedures

This chapter describes the arithmetic operations, the comparisons, and the auxiliary procedures: functions for sign manipulation, obtaining Nextafter value, and binary scaling and logarithm.

Arithmetic operations

SANE provides the arithmetic operations for the SANE data types, as shown for Pascal in Table 6-1.

Table 6-1
Arithmetic operations in Pascal

Operation	Pascal
Add	+
Subtract	-
Multiply	*
Divide	/
Square root	Sqrt ()
Remainder	Remainder ()
Round-to-Integer	Rint ()

Apple's Pascal and C language processors, and others that follow Apple's guidelines, automatically use SANE extended-precision arithmetic for the normal in-line operators (+, -, *, /). All the arithmetic operations produce the best possible result: the mathematically exact result, coerced to the precision and range of the extended type. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE Standard (see Chapter 7, "Controlling the SANE Environment").

Remainder

FUNCTION Remainder(x, y: extended; Var quo: integer): extended;

Generally, Remainder (and modulo) functions are defined by the expression

$$x \text{ rem } y = x - y \times n$$

where n is some integral approximation to the quotient x/y . This expression can be found even in the conventional integer-division algorithm, shown in Figure 6-1.

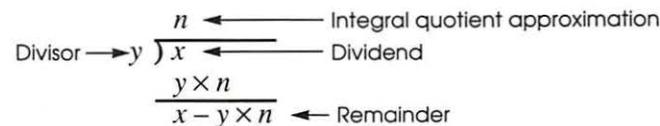


Figure 6-1
Integer-division algorithm

SANE supports the Remainder function specified in the IEEE Standard: When $y \neq 0$, the remainder $r = x \text{ rem } y$ is defined (regardless of the rounding direction) by the mathematical relation $r = x - y \times n$, where n is the integral value nearest the exact value x/y ; whenever $|n - x/y| = 1/2$, n is even. The remainder is always exact. If the value of r is 0, the sign of r is that of x .

❖ *Pascal note:* The integer variable `quo` receives the seven low-order bits of $|n|$ (negated if n is negative) as a value between -127 and 127 . It is useful for programming functions, such as the trigonometric functions, that require argument reduction.

Remainder example 1

Find $5 \text{ rem } 3$. Here $x = 5$ and $y = 3$. Because $1 < 5/3 < 2$ and because $5/3 = 1.66666\dots$ is closer to 2 than to 1, `quo` is taken to be 2, so

$$5 \text{ rem } 3 = r = 5 - 3 \times 2 = -1$$

Remainder example 2

Find $43.75 \text{ rem } 2.5$. Because $17 < 43.75/2.5 < 18$ and because $43.75/2.5 = 17.5$ is equally close to both 17 and 18, `quo` is taken to be the even quotient, 18. Hence,

$$43.75 \text{ rem } 2.5 = r = 43.75 - 2.5 \times 18 = -1.25$$

The IEEE Remainder function differs from other commonly used remainder and modulo functions. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. Other remainder functions can be constructed from the IEEE Remainder function by appropriately adding or subtracting y .

Round-to-integer

```
FUNCTION Rint ( x: extended ): extended;
```

This function rounds an input argument according to the current rounding direction to an integral value and delivers the result to the extended format. For example, 12345678.875 rounds to 12345678.0 or 12345679.0 . (The rounding direction, which can be set by the user, is explained fully in Chapter 7, "Controlling the SANE Environment.")

Note that, in each floating-point format, all values of sufficiently great magnitude are integral. For example, in single, all numbers whose magnitudes are at least 2^{23} are integral.

SANE comparisons

SANE supports the usual numeric comparisons: less, less-or-equal, greater, greater-or-equal, equal, and not-equal. For real numbers, these comparisons behave according to the familiar ordering of real numbers.

Comparisons with NaNs and Infinities

SANE comparisons handle NaNs and Infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any SANE values a and b , exactly one of the following is true:

- $a < b$
- $a > b$
- $a = b$
- a and b are unordered

Determination is made by the following rule: If x or y is a NaN, then x and y are unordered; otherwise, x and y are less, equal, or greater according to the ordering of the real numbers, with the understanding that $+0 = -0$ and $-\infty < \text{each real number} < +\infty$. (Note that a NaN always compares unordered—even with itself.)

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the Pascal expression $x \leq y$ is true if x is less than y or if x equals y , and is false if x is greater than y or if x and y are unordered. Note that the SANE not-equal relation means less, greater, or unordered—even if not-equal is written $<>$, as in Pascal.

Some relational operators in high-level language comparisons contain the predicate less or greater, but not unordered. In Pascal those relational operators are $<$, $<=$, $>$, and $>=$ (but not $=$ and $<>$). For those relations, comparisons signal invalid if the operands are unordered; that is, if either operand is a NaN. For the operators equal and nonequal, comparisons with NaN aren't misleading: thus, when x or y is a NaN, the relation $x = y$ returns FALSE, which is not misleading. Likewise, when x or y is a NaN, $x <> y$ returns TRUE, again not misleading. On the other hand, when x or y is a NaN, $x < y$ being false might tempt you to conclude that $x \geq y$, so SANE signals invalid to help you avoid the pitfall. Table 6-2 shows the results of such comparisons in Pascal; other SANE-supporting high-level languages behave similarly. This issue is also discussed in the section "The Invalid-Operation Flag" in Appendix C, "Porting Programs to SANE."

A comparison with a signaling NaN as an operand always signals invalid, just as in arithmetic operations.

Table 6-2
Comparisons involving NaNs

Relational operation*	Signal
$x < y$	Invalid
$x \leq y$	Invalid
$x \geq y$	Invalid
$x > y$	Invalid
$x = y$	(None)
$x <> y$	(None)

* where x or y is a quiet NaN

The Relation function

```
RelOp = ( GreaterThan, LessThan, EqualTo, Unordered );
```

```
FUNCTION Relation( x, y: extended ): RelOp;
```

High-level languages supporting SANE supplement the usual comparison operators with a function that takes two numeric arguments and returns the appropriate relation (less, equal, greater, or unordered). Programs can use the result of this function in expressions to test for combinations not supported by the comparison operators, such as “less-than or unordered.”

The Relation function signals invalid only if an operand is a signaling NaN.

Auxiliary procedures

SANE includes the following special routines that are recommended in an appendix to the IEEE Standard as aids to programming:

- Neg, make negative
- Abs, absolute value
- CopySign, copy the sign
- Nextafter functions
- Scalb, binary scaling
- Logb, binary exponent

Sign manipulation

The sign manipulation operations change only the sign of their argument. The Negate function reverses the sign of its argument. The Absolute Value function makes the sign of its argument positive.

- ❖ *Pascal note:* Pascal provides negation by means of the unary `-` operator and obtains absolute values by means of the function `Abs`.

```
FUNCTION CopySign( x, y: extended ): extended;
```

`CopySign` takes two arguments and copies the sign of its first argument onto the absolute value of its second argument.

- ❖ *Note:* The order of the operands in the SANE `CopySign` function is reversed from that suggested in IEEE Standard 754.

A SANE implementation may treat these operations as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal the invalid-operation exception.

Nextafter functions

```
FUNCTION NextReal( x, y: real ): real;  
FUNCTION NextDouble( x, y: double ): double;  
FUNCTION NextExtended( x, y: extended ): extended;
```

The floating-point values representable in single, double, and extended formats constitute a finite set of real numbers. The Nextafter functions (one for each of these formats) generate the next representable neighbor in the proper format, starting with an initial value x and in the direction from x toward y . As elsewhere, the names of the functions may vary with the implementation.

Special cases for Nextafter functions

If the initial value and the direction value are equal, then the result is the initial value.

If the initial value is finite but the next representable number is infinite, then overflow and inexact are signaled.

If the next representable number lies strictly between $-M$ and $+M$, where M is the smallest positive normalized number for that format, and if the arguments are not equal, then the Nextafter function returns the next representable denormalized number and signals underflow and inexact.

Binary scaling and logarithmic functions

```
FUNCTION Scalb( n: integer; x: extended ): extended;  
FUNCTION Logb( x: extended ): extended;
```

The Scalb and Logb functions are provided for manipulating binary exponents.

Scalb(n, x) efficiently scales a given number x by a given integer power n of 2, returning $x \times 2^n$. Using Scalb is more efficient than a straightforward computation of $x \times 2^n$.

Logb returns the binary exponent of its input argument as a signed integral value.

When the input argument is a denormalized number, the exponent is determined as if the input argument had first been normalized.

You can use Scalb and Logb to find the value of the mantissa of a number, like this:

```
m := Abs(Scalb(-Num2Integer(Logb(x)), x));
```

❖ *C note:* The C mathematical library has functions named `frexp` and `ldexp` that are similar in spirit to Logb and Scalb, though different in details.

❖ *Note:* Logb in SANE follows the recommendation in IEEE Standard 854. It differs from IEEE Standard 754 in its handling of denormalized numbers.

Special cases for Logb

If x is an Infinity, Logb(x) returns +INF.

If $x = 0$, Logb(x) returns -INF and signals divide-by-zero.



Chapter 7



Controlling the SANE Environment

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

Rounding direction

```
RoundDir = (ToNearest, Upward, Downward, TowardZero);
```

```
PROCEDURE SetRound(r: RoundDir);
```

```
FUNCTION GetRound: RoundDir;
```

The available rounding directions are

- to-nearest
- upward
- downward
- toward-zero

The rounding direction affects all conversions except conversions between decimal records and decimal strings and all arithmetic operations except remainder. Except for conversions between binary and decimal (described in Chapter 3, “Conversions in SANE”), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

❖ *Note:* Transcendental functions are not arithmetic operations and do not produce the correctly rounded value described here.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type or when the round-to-integral-value operation is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half **unit in the last place**), then the Infinity with the corresponding sign is delivered.

The other rounding directions are upward, downward, and toward-zero. When rounding upward, the result is the format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integral-value operation. (See also the sections on expressions in “Pascal SANE Extensions” and “C SANE Extensions” in Appendix A.)

Example: rounding upward

One reason to change the rounding direction would be to put bounds on errors (at least for the rational operations and square root). Suppose you want to evaluate an expression like

$$x = (a \times b + c \times d) / (f + g)$$

where a , b , c , d , f , and g are positive.

To make sure that the result is always larger than the exact value, you can change the expression such that all roundings cause errors in the same direction. The following code fragment changes the rounding direction to compute an upper bound for the expression, then restores the previous rounding.

```
VAR
  r: RoundDir;           {local storage for rounding direction}
  xUp: extended;
  . . .

r := GetRound;          {save rounding direction}
SetRound(Downward);    {downward rounding for denominator}
xUp := f+g;
SetRound(Upward);      {upward rounding for expression}
xUp := (a*b+c*d)/xUp;
SetRound(r)             {restore previous rounding direction}
. . .
```

Rounding precision

```
RoundPre = (ExtPrecision, DblPrecision, RealPrecision);
PROCEDURE SetPrecision(p: RoundPre);
FUNCTION GetPrecision: RoundPre;
```

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or to double. If the SANE user sets rounding precision to single (or to double), then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and to extended formats are affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or to double does not significantly enhance performance, and in some SANE implementations may hinder performance.

Exception flags and halts

TYPE

```
Exception = integer;
```

CONST

```
Invalid = 1;  
Underflow = 2;  
Overflow = 4;  
DivByZero = 8;  
Inexact = 16;
```

```
PROCEDURE SetException(e: Exception; b: boolean);
```

```
FUNCTION TestException(e: Exception): boolean;
```

```
PROCEDURE SetHalt(e: Exception; b: boolean);
```

```
FUNCTION TestHalt(e: Exception): boolean;
```

- ❖ *Note:* The values of the exception constants and the type definition for `Exception` vary among different implementations of SANE, but code that uses the functions, procedures, and symbolic constant names to access exceptions and halts should port across the different implementations. Please refer to other parts of this manual for implementation-dependent information.

Exceptions are signaled when detected; if the corresponding halt is enabled, the SANE engine transfers control to a user-specified location. (A high-level language may not pass on to its user the facility to set this location, but instead may stop the user's program.) The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings).

```
{If halt vector is to be made available to Pascal users:}
```

```
FUNCTION GetHaltVector: longint;
```

```
PROCEDURE SetHaltVector(v: longint);
```

A control mechanism such as this can also be provided by hardware—for example, the Motorola MC68881 floating-point coprocessor. On machines with hardware exception trapping, programs should use the hardware mechanism instead of the software-supported mechanism described here. For information about the halt (trap) mechanism on the MC68881, please refer to Chapter 30, “The MC68881 Trap Mechanism,” and to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.

Types of exceptions

SANE supports five exception flags with corresponding halt settings:

- invalid operation (often called simply *invalid*)
- underflow
- overflow
- divide-by-zero
- inexact

Invalid operation

The invalid-operation exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions for the different operations are these:

- addition or subtraction: magnitude subtraction of Infinities, for example, $(+INF) + (-INF)$
 - multiplication: $0 \times INF$
 - division: $0/0$ or INF/INF
 - remainder: $x \text{ rem } y$, where y is zero or x is infinite
 - square root: if the operand is less than zero
 - conversion: to the comp format or to a system-specific integer format when excessive magnitude, Infinity, or NaN precludes a faithful representation in that format (see Chapter 3, “Conversions in SANE,” for details)
 - comparison: with predicates involving less-than or greater-than, but not unordered, when at least one operand is a NaN
 - any operation on a signaling NaN except the following: class and sign inquiries and, on some implementations, sign manipulations (Negate, Absolute Value, and CopySign)
- ❖ *Note:* Compilers for high-level languages may move extended-format numbers either by extended-to-extended conversions, which detect signaling NaNs, or by bit copies, which don't. Thus, some compiler-generated moves cause signaling NaNs to raise the invalid exception earlier than expected.

Underflow

The (unhalted) underflow exception is signaled when a floating-point result is both tiny and inexact (and therefore is perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if its magnitude is smaller than its format's smallest positive normalized number.

- ❖ *Note:* Different SANE engines may test for a tiny result either before or after rounding the result to its destination format. If the underflow halt is set, the halt occurs either when the result is tiny and inexact or when the result is simply tiny; see “Example: Gradual Underflow” in Chapter 5. For details about the 65C816 and 6502 SANE engines, refer to Chapter 15. For details about the MC68000 SANE engine, refer to Chapter 23. For details about the MC68881 SANE engine, refer to Chapter 30 and to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.

Divide-by-zero

The divide-by-zero exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result; for example, `Logb(0)` returns `-INF` and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

Overflow

The overflow exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

Inexact

The inexact exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result. Thus, inexact is always signaled in conjunction with overflow or underflow. Valid operations on Infinities are always exact and therefore signal no exceptions. Invalid operations on Infinities are described at the beginning of this section.

Managing environmental settings

Environmental settings include the rounding direction, rounding precision, exception flags, and halt settings. These settings are global and can be explicitly changed by the program. Thus, all routines inherit these settings and are capable of changing them. Conventionally, routines that change these settings first save them, then restore when finished.

```
Environment = integer;
```

```
PROCEDURE SetEnvironment (e: Environment);
PROCEDURE GetEnvironment (var e: Environment);
PROCEDURE ProcEntry (var e: Environment);
PROCEDURE ProcExit (e: Environment);
```

❖ *Note:* The type definition of the Environment word can be different for different SANE implementations, but code that uses the procedures to access the environment should port across the different implementations. On a machine with an MC68881, the SANE environment is stored in the MC68881's control and status registers; see Chapter 29, "Controlling the MC68881 Environment."

Example: setting rounding direction

A subroutine that includes the following statements uses to-nearest rounding while not affecting its caller's rounding direction.

```
VAR
  r: RoundDir;           {local storage for rounding direction}

BEGIN
  r := GetRound;         {save caller's rounding direction}
  SetRound(ToNearest);  {set to-nearest rounding}

  { subroutine's operations here }

  SetRound(r)           {restore caller's rounding direction}
END;
```

Notice that if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient procedures for managing the environment as a whole: the Procedure-Entry and Procedure-Exit procedures.

The Procedure-Entry procedure returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example: setting environment

A subroutine that includes the following statements runs under the default environment while not affecting its caller's environment.

```
VAR
  e: Environment;           {local storage for environment}

BEGIN
  ProcEntry(e);             {save caller's environment and set
                             default environment}
  { subroutine's operations here }

  SetEnvironment(e)        {restore caller's environment}
END;
```

Example: setting exceptions

The Procedure-Exit procedure facilitates writing subroutines that appear to their callers to be atomic operations (such as addition, square root, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which may be irrelevant or misleading. The Procedure-Exit procedure, which takes a saved environment as argument, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (Note that if enabled, halts could occur at this step.)

Thus, exceptions signaled between the Procedure-Entry and Procedure-Exit procedures are hidden from the calling program unless the exceptions remain raised when the Procedure-Exit procedure is called.

The following function signals underflow if its result is denormalized, overflow if its result is Infinity, and inexact always, but hides spurious exceptions occurring from internal computations:

```
FUNCTION NumFcn: extended;

VAR
  e:      Environment;      {local storage for environment}
  c:      NumClass;        {for class inquiry}

BEGIN
  ProcEntry(e);            {save caller's environment and set
                           default environment - now halts are
                           disabled}
  . . .                   {internal computation}

  NumFcn := Result;       {result to be returned}
  c := ClassExtended(Result); {class inquiry}

  SetException(Invalid + Underflow + Overflow + DivByZero, false);
                           {clear exceptions}
  SetException(Inexact, true); {signal inexact}

  IF c = Infinite THEN
    SetException(Overflow, true)
  ELSE IF c = DenormalNum THEN
    SetException(Underflow, true);
  ProcExit(e)              {restore caller's environment, including
                           any halt enables, and then signal
                           exceptions from subroutine}
END                          {NumFcn} ;
```



Chapter 8



Elementary Functions in SANE

SANE provides several basic mathematical functions, including logarithms, exponentials, two important financial functions, trigonometric functions, and a random number generator.

All the elementary functions except the random number generator handle NaNs, overflow, and underflow appropriately. Some elementary functions may conservatively signal inexact when determining exactness would be too costly.

Logarithmic functions

```
FUNCTION Log2 (x: extended) : extended;  
FUNCTION Ln (x: extended) : extended;  
FUNCTION Ln1 (x: extended) : extended;
```

SANE provides three logarithmic functions:

- `Log2 (x)` : base-2 logarithm
- `Ln (x)` : base-e or natural logarithm
- `Ln1 (x)` : base-e logarithm of 1 plus argument

`Ln1 (x)` accurately computes $\ln(1 + x)$. If the input argument x is small, then the computation of `Ln1 (x)` is more accurate than the straightforward computation of $\ln(1 + x)$ by adding x to 1 and taking the natural logarithm of the rounded sum.

Special cases for logarithmic functions

If $x = +\text{INF}$, then `Log2 (x)`, `Ln (x)`, and `Ln1 (x)` return $+\text{INF}$. No exception is signaled.

If $x = 0$, then `Log2 (x)` and `Ln (x)` return $-\text{INF}$ and signal divide-by-zero. Similarly, if $x = -1$, then `Ln1 (x)` returns $-\text{INF}$ and signals divide-by-zero.

If $x < 0$, then `Log2 (x)` and `Ln (x)` return NaN and signal invalid. Similarly, if $x < -1$, then `Ln1 (x)` returns NaN and signals invalid.

Exponential functions

```
FUNCTION Exp2 (x: extended): extended;  
FUNCTION Exp (x: extended): extended;  
FUNCTION Exp1 (x: extended): extended;  
FUNCTION XpwrI (x: extended; i: integer): extended;  
FUNCTION XpwrY (x, y: extended): extended;
```

SANE provides five exponential functions:

- `Exp2 (x)`: the base-2 exponential 2^x
- `Exp (x)`: the base-e or natural exponential e^x
- `Exp1 (x)`: the base-e exponential minus 1
- `XpwrI (x, i)`: the integer exponential x^i
- `XpwrY (x, y)`: the general exponential x^y

The function `Exp1 (x)` accurately computes $e^x - 1$. If the input argument x is small, then the computation of `Exp1 (x)` is more accurate than the straightforward computation of $e^x - 1$ by exponentiation and subtraction.

Example: using Exp1

Financial applications often compute a value r for internal rate of return given the compounded value Q defined as

$$Q(r, n) = (1 + r)^n$$

For very large values of n , the following obvious expression for r gives errors:

```
r := Exp (Ln (Q) * (1/n)) - 1;
```

A better expression for large n uses `Exp1`, like this:

```
r := Exp1 (Ln (Q) * (1/n));
```

Special cases for Exp2, Exp, and Exp1

If $x = +\text{INF}$, then `Exp2 (x)`, `Exp (x)`, and `Exp1 (x)` return $+\text{INF}$ and do not signal an exception.

If $x = -\text{INF}$, then `Exp2 (x)` and `Exp (x)` return 0 and `Exp1 (x)` returns -1 . The functions do not signal an exception in this case.

Special cases for XpwrI

If the integer exponent i equals 0 and x is not a NaN, then $XpwrI(x, i)$ returns 1. Note that $XpwrI(x, 0)$ returns 1 even if x is 0 or Infinity.

If x is +0 and i is negative, then $XpwrI(x, i)$ returns +INF and signals divide-by-zero.

If x is -0 and i is negative, then $XpwrI(x, i)$ returns +INF if i is even, or -INF if i is odd. The function signals divide-by-zero in either case.

Special cases for XpwrY

If x is +0 and y is negative, then the general exponential $XpwrY(x, y)$ returns +INF and signals divide-by-zero.

If x is -0 and y is integral and negative, then $XpwrY(x, y)$ returns +INF if y is even, or -INF if y is odd.

The function signals divide-by-zero in either case. The function $XpwrY(x, y)$ returns a NaN and signals invalid if any of the following is true:

- Both x and y equal 0.
- x is \pm INF and y equals 0.
- x equals 1 and y is \pm INF.
- x is -0 or less than 0 and y is nonintegral.

Financial functions

SANE provides two functions, Compound and Annuity, that can be used to solve various financial, or time-value-of-money, problems.

Compound

FUNCTION Compound(r , n : extended): extended;

The Compound function computes

$$\text{compound}(r, n) = (1 + r)^n$$

where r is the interest rate and n is the number (perhaps nonintegral) of periods.

When the rate r is small, compound gives a more accurate computation than does the straightforward computation of $(1 + r)^n$ by addition and exponentiation.

The Compound function is directly applicable to computation of present and future values:

$$PV = FV \times (1 + r)^{-n} = \frac{FV}{\text{compound}(r, n)}$$

$$FV = PV \times (1 + r)^n = PV \times \text{compound}(r, n)$$

Annuity

FUNCTION Annuity(*r*, *n*: extended): extended;

The Annuity function computes

$$\text{annuity}(r, n) = \frac{1 - (1 + r)^{-n}}{r} \text{ when } r \neq 0$$

where *r* is the interest rate and *n* is the number of periods. Annuity is more accurate than the straightforward computation of the expression above using basic arithmetic operations and exponentiation. The Annuity function is directly applicable to the computation of present and future values of ordinary annuities:

$$PV = PMT \times \frac{1 - (1 + r)^{-n}}{r}$$

$$= PMT \times \text{annuity}(r, n)$$

$$FV = PMT \times \frac{(1 + r)^n - 1}{r}$$

$$= PMT \times (1 + r)^n \times \frac{1 - (1 + r)^{-n}}{r}$$

$$= PMT \times \text{compound}(r, n) \times \text{annuity}(r, n)$$

where *PMT* is the amount of one periodic payment.

Special cases for Compound

If *r* = 0 and *n* is infinite, or if *r* < -1, then `Compound(r, n)` returns a NaN and signals invalid. (The `Compound` function is intended for financial contexts where *r* < -1 is regarded as a mistake.)

If *r* = -1 and *n* < 0, then `Compound(r, n)` returns +INF and signals divide-by-zero.

Special cases for Annuity

If *r* = 0, then `Annuity(r, n)` computes the sum of 1 + 1 + ... + 1 over *n* periods, and therefore returns the value *n* and signals no exceptions (the value *n* corresponds to the limit as *r* approaches 0).

If *r* < -1, then `Annuity(r, n)` returns a NaN and signals invalid, because the function is intended for financial contexts where *r* < -1 is regarded as a mistake.

If *r* = -1 and *n* > 0, then `Annuity(r, n)` returns +INF and signals divide-by-zero.

Trigonometric functions

```
FUNCTION Cos(x: extended): extended;  
FUNCTION Sin(x: extended): extended;  
FUNCTION Tan(x: extended): extended;  
FUNCTION ArcTan(x: extended): extended;
```

SANE provides the basic trigonometric functions:

- `Cos(x)` computes the cosine of x .
- `Sin(x)` computes the sine of x .
- `Tan(x)` computes the tangent of x .
- `ArcTan(x)` computes the arctangent of x .

The remaining trigonometric functions can be easily and efficiently computed from the elementary functions provided (see Chapter 9, “Other Elementary Functions”).

The arguments for `Cos(x)`, `Sin(x)`, and `Tan(x)` and the results of `ArcTan(x)` are expressed in radians. The Cosine, Sine, and Tangent functions use an argument reduction based on the Remainder function (see Chapter 6, “Basic Operations, Comparisons, and Auxiliary Procedures”) and `pi`, where `pi` is the nearest extended-precision approximation of π . The Cosine, Sine, and Tangent functions are periodic with respect to this `pi`, so their periods are slightly different from their mathematical counterparts and diverge from their counterparts when their arguments become very large. Number results from `ArcTan(x)` lie in the interval $-\pi/2$ to $\pi/2$.

Accuracy of pi

Programmers sometimes think they are having trouble because the value they’re using for π is not accurate enough. Actually, the trouble is often caused by rounding in earlier steps.

Even though SANE’s `pi` is not exactly equal to π , it doesn’t usually cause errors, because other sources of error are greater. For example, when large arguments have errors in the last digit, the trigonometric functions magnify the errors. Even if the trigonometric functions were computed absolutely accurately, this problem would still occur.

Special cases for Cos and Sin

If x is infinite, then `Cos(x)` and `Sin(x)` return a NaN and signal invalid.

Special cases for Tan

If x is $\pm\pi/2$, then $\text{Tan}(x)$ returns $\pm\text{INF}$.

❖ *Note:* The MC68881 coprocessor returns large magnitudes, but not $\pm\text{INF}$, for the tangent of $\pm\pi/2$. For more information about the MC68881, please refer to Part IV of this book.

If x is infinite, then $\text{Tan}(x)$ returns NaN and signals invalid.

Special cases for ArcTan

If $x = \pm\text{INF}$, then $\text{ArcTan}(x)$ returns $\pm\pi/2$.

Random number generator

FUNCTION RandomX (var x : extended): extended;

SANE provides a pseudorandom number generator, $\text{RandomX}(x)$. The RandomX function has one argument, passed by address. A sequence of pseudorandom integral values x in the range

$$1 \leq x \leq 2^{31} - 2$$

can be generated by initializing an extended variable x to an integral value (the seed) in the above range and making repeated calls to $\text{RandomX}(x)$; each call delivers in x the next random number in the sequence.

RandomX uses the iteration formula

$$x \leftarrow (7^5 \times x) \bmod (2^{31} - 1)$$

If seed values of x are nonintegral or outside the range

$$1 \leq x \leq 2^{31} - 2$$

then results are unspecified. A pseudorandom rectangular distribution on the interval (0,1) can be obtained by dividing the results from RandomX by

$$2^{31} - 1 = \text{Scalb}(31,1) - 1$$

❖ *Pascal note:* Pascal's RandomX function returns the next random number as the value of the function as well as in x .



Chapter 9



Other Elementary Functions

The Standard Apple Numerics Environment (SANE) provides several transcendental functions; from these, you can construct other high-quality functions. This chapter gives several examples of such functions. These robust, accurate functions are based on algorithms developed by Professor William Kahan of the University of California at Berkeley.

Exception handling

Unlike the SANE elementary functions, these functions do not provide complete handling of special cases and exceptions. The most troublesome exceptions can be correctly handled if you

- begin each function with a call to the Procedure-Entry procedure
- clear the spurious exceptions indicated in the comments
- end each function with a call to the Procedure-Exit procedure (see Chapter 7, “Controlling the SANE Environment”)

Functions

All variables in the Pascal code in the following sections are extended. The constant C is 2^{-33} or `Scalb(-33, 1)`. C was chosen to be nearly the largest value for which $1 - C^2$ rounds to 1.

Secant

```
FUNCTION Secant (x: extended): extended;  
  
    BEGIN  
        Secant := 1/Cos(x)  
    END;
```

CoSecant

```
FUNCTION CoSecant (x: extended): extended;  
  
    BEGIN  
        CoSecant := 1/Sin(x)  
    END;
```

CoTangent

```
FUNCTION CoTangent(x: extended): extended;
```

```
BEGIN
  CoTangent := 1/Tan(x)
END;
```

ArcSin

```
FUNCTION ArcSin(x: extended): extended;
```

```
VAR
  Y: extended;

BEGIN
  Y := Abs(x);
  IF Y>c THEN
    BEGIN
      IF Y>0.5 THEN
        BEGIN
          Y := 1-Y;
          Y := 2*Y-Y*Y
        END
      ELSE
        Y := 1-Y*Y;
        ArcSin := ArcTan(x/sqrt(Y))
        { spurious divide-by-zero may arise }
      END
    ELSE
      ArcSin := x
    END;
END;
```

ArcCos

```
FUNCTION ArcCos(x: extended): extended;
```

```
BEGIN
  ArcCos := 2*ArcTan(sqrt((1-x)/(1+x)))
  { spurious divide-by-zero may arise }
END;
```

Constants for Sinh and Cosh

For values of x larger than about 10^5 , intermediate values of $\text{Sinh}(x)$ and $\text{Cosh}(x)$ overflow even though $\text{Sinh}(x)$ and $\text{Cosh}(x)$ do not. To prevent overflows, these formulas use the following constants (in addition to C , defined earlier).

```
gamma := -Ln(c)/2;
lambda := Ln(NextExtended(+INF, 0));
lambda := (Ln(2)+lambda)-lambda; {value for ln(2) with trailing zeros}
mu := exp(lambda)/2; {compensates for lost digits of lambda}
```

Sinh

```
FUNCTION Sinh(x: extended): extended;
```

```
VAR
  Y: extended;

BEGIN
  Y := Abs(x);
  IF Y < c THEN
    Sinh := x
  ELSE
    BEGIN
      IF Y < gamma THEN
        BEGIN
          Y := Exp1(Y);
          Y := 0.5*(Y+Y/(1+Y))
        END
      ELSE
        Y := mu*exp(Y-lambda);
        Sinh := CopySign(x, Y)
      END;
    END;
  END;
```

Cosh

```
FUNCTION Cosh(x: extended): extended;

VAR
  Y: extended;

BEGIN
  Y := Abs(x);
  IF Y>gamma THEN
    Cosh := Sinh(Y)
  ELSE
    BEGIN
      Y := 0.5*exp(Y);
      Cosh := Y+0.25/Y
    END;
  END;
END;
```

Tanh

```
FUNCTION Tanh(x: extended): extended;

VAR
  Y: extended;

BEGIN
  Y := Abs(x);
  IF Y>c THEN
    BEGIN
      Y := Exp1(-2*Y);
      Y := -Y/(2+Y)
    END;
  Tanh := CopySign(x, Y)
END;
```

ArcSinh

```
FUNCTION ArcSinh(x: extended): extended;

VAR
  Y: extended;

BEGIN
  Y := Abs(x);
  IF Y>c THEN Y := Ln1(Y+Y/(1/Y+sqrt(1+1/(Y*Y))));
  { spurious underflow may arise }
  ArcSinh := CopySign(x, Y)
END;
```

ArcCosh

```
FUNCTION ArcCosh(x: extended): extended;

VAR
  Y: extended;

BEGIN
  Y := sqrt(x-1);
  ArcCosh := Ln1(Y*(Y+sqrt(x+1)))
END;
```

ArcTanh

```
FUNCTION ArcTanh(x: extended): extended;

VAR
  Y: extended;

BEGIN
  Y := Abs(x);
  IF Y>c THEN Y := 0.5*Ln1(2*(Y/(1-Y)));
  ArcTanh := CopySign(x, Y)
END;
```



Chapter 10



More Examples Using SANE

Like the examples in Chapter 1, the examples in this chapter illustrate the ways in which SANE arithmetic makes programming easier.

Continued fraction

Consider a typical continued fraction $cf(x)$.

$$cf(x) = 4 - \frac{3}{x - 2 - \frac{1}{x - 7 + \frac{10}{x - 2 - \frac{2}{x - 3}}}}$$

An algebraically equivalent expression is $rf(x)$.

$$rf(x) = \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))}$$

Both expressions represent the same rational function, one whose graph is smooth and unexceptional, as shown in Figure 10-1.

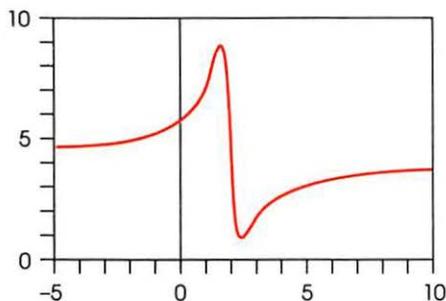


Figure 10-1

Graph of continued fraction $cf(x) = rf(x)$

Although the two functions $rf(x)$ and $cf(x)$ are equal, they are not computationally equivalent. For instance, consider $rf(x)$ at the following values of x :

- $x = 1, rf(1) = 7$
- $x = 2, rf(2) = 4$
- $x = 3, rf(3) = 8/5$
- $x = 4, rf(4) = 5/2$

Whereas $rf(x)$ is perfectly well behaved, those values of x lead to division by zero when computing $cf(x)$ and cause many computers to stop. SANE arithmetic, where division by zero produces an Infinity, has no difficulty in computing $cf(x)$ for those values.

On the other hand, simply computing $rf(x)$ instead of $cf(x)$ can also cause problems. If the absolute value of x is so big that x^4 causes an overflow, then $cf(x)$ approaches $cf(\infty) = 4$ but computing $rf(x)$ encounters (overflow)/(overflow), which yields something else. SANE returns NaN for such cases; some other machines get (maximum value)/(maximum value) = 1. Also, at arguments x between 1.6 and 2.4 the formula $rf(x)$ suffers from round-off much more than $cf(x)$ does. For those reasons, computing $cf(x)$ is preferable to computing $rf(x)$ if division by zero works the way it does in SANE; that is, if it produces Infinity instead of stopping computation.

In general, division by zero is an exceptional event not merely because it is rare, but because different applications require different consequences. If you are not satisfied with the consequences supplied by default in SANE, you can choose other consequences by making the program test for NaNs and Infinities (or for the flags that signal their creation).

Rather than sprinkle tests throughout the program in an attempt to keep exceptions from occurring, you may prefer to put one or two tests near the end of the code to detect the (rare) occurrence of an exception and modify the results appropriately. That is more economical than testing every divisor for zero when zero divisors are rare.

Area of a triangle

Here is a familiar and straightforward task that fails when subtraction is aberrant: Compute the area $A(x,y,z)$ of a triangle given the lengths x, y, z of its sides. The formula given here performs this calculation almost as accurately as floating-point multiplication, division, and square root are performed by the computer it runs on, provided the computer doesn't drop digits prematurely during subtraction. The formula works correctly, and provably so, on a wide range of machines, including all implementations of SANE.

Heron's formula

The classical formula, attributed to Heron of Alexandria, is

$$A(x,y,z) = \sqrt{s(s-x)(s-y)(s-z)}$$

where $s = (x + y + z)/2$.

For needle-shaped triangles, that formula gives incorrect results on computers *even when every arithmetic operation is correctly rounded*. For example, Table 12-1 shows an extreme case with results rounded to five decimal digits. With the values shown, rounding $(x + (y + z))/2$ must give either 100.01 or 100.02. Substituting those values for s in Heron's formula yields either 0.0 or 1.5813 instead of the correct value 1.000025.

Evidently, Heron's formula would be a very bad way to calculate ratios of areas of nearly congruent needle-shaped triangles.

Table 10-1
Area using Heron's formula

	Correct	Rounding downward	Rounding upward
x	100.01	100.01	100.01
y	99.995	99.995	99.995
z	0.025	0.025	0.025
$(x + (y + z))/2$	100.015	100.01	100.02
A	1.000025	0.0000	1.5813

Improved formula

A good procedure, numerically stable on machines that don't truncate prematurely during subtraction, is the following:

1. Sort x, y, z , so that $x \geq y \geq z$.
2. Test for $z \geq x - y$ to see whether the triangle exists.
3. Compute A by the formula

$$A := \sqrt{((x + (y + z))(z - (x - y))(z + (x - y))(x + (y - z)))/4}$$

Warning

This formula works correctly only if you don't remove parentheses.

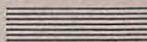
The success of this formula depends upon the following easily proved theorem.

Theorem: If p and q are represented exactly in the same conventional floating-point format, and if $1/2 \leq p/q \leq 2$, then $p - q$ too is representable exactly in the same format (unless $p - q$ suffers underflow, something that can't happen in IEEE arithmetic).

The theorem merely confirms that subtraction is exact when massive cancellation occurs. That is why each factor inside the square-root expression is computed correctly to within a unit or two in its last digit kept, and A is not much worse, on computers that subtract the way SANE does. On machines that flush tiny results to zero, this formula for A fails because $(p - q)$ can underflow.



Part II



The 65C816 and 6502 Assembly-Language SANE Engines

The software packages described in Part II of this manual provide the features of the Standard Apple Numerics Environment (SANE) to assembly-language programmers using Apple's 65C816-based and 6502-based systems. SANE—described in detail in Part I of this manual—fully supports the IEEE Standard 754 for binary floating-point arithmetic, and augments the Standard to provide greater utility for applications in accounting, finance, science, and engineering. The IEEE Standard and SANE offer a combination of quality, predictability, and portability heretofore unknown for numerical software.

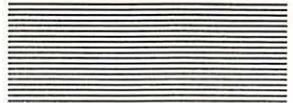
Part II describes two different SANE engines, one for the 65C816 microprocessor and one for the 6502. The emphasis is on the newer 65C816 version; in each instance where the 6502 version is different, a 6502 note makes the difference explicit.

❖ *6502 note:* Notes like this contain information specific to the 6502 SANE engine.

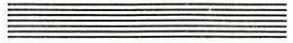
Functionally equivalent assembly-language SANE engines are available for Apple's 68000-based systems and for Apple's 68020-based systems with the 68881 floating-point coprocessor. Thus, numerical algorithms coded in assembly language for an Apple 65C816-based or 6502-based system can be readily recoded for an Apple 68000-based or 68020-based system. Apple has defined macros for accessing the different engines to make it easier to port algorithms from one system to another. Part III of this manual describes the 68000 SANE engine; Part IV describes the SANE engine for the 68020 with the 68881 coprocessor.

Part II of this manual describes the use of the 65C816 and 6502 assembly-language SANE engines, but does not describe SANE itself. For example, Part II explains how to call the SANE Remainder function from assembly language but does not discuss what this function does. See Part I for information about the semantics of SANE.

❖ *6502 note:* See Appendix B for information about obtaining the 6502 assembly-language SANE engine.



Chapter 11



65C816 SANE Basics and Data Types

Programs using either 65C816-based or 6502-based SANE engines use the same convention for making most calls: first push the parameters on the stack, then invoke the macro for the desired operation. For example, a typical assembly-language call to the 65C816 SANE engine looks like this:

```
PUSHLONG  A_ADR      ; push address of (single-format) A
PUSHLONG  B_ADR      ; push address of (extended-format) B
FSUBS     ; Subtract with source operand in
           ; single format
```

❖ *6502 note:* The same call to the 6502 SANE engine looks like this:

```
PUSH      A_ADR      ; push address of (single-format) A
PUSH      B_ADR      ; push address of (extended-format) B
FSUBS     ; Subtract with source operand in
           ; single format
```

This example is typical of SANE engine calls, most of which pass operands by pushing the addresses of the operands onto the stack prior to invoking the operation. The form of the operation in the example ($B \leftarrow B - A$, where A is a numeric type and B is extended) is similar to the forms for most SANE operations.

In both examples, `FSUBS` is an assembly-language macro defined in the macro file provided with the development software for the microprocessor you are using, either the 65C816 or the 6502. The macros expand into instructions that specify the operation and transfer control to the SANE engine; see the section “Calling Sequence” later in this chapter.

❖ *Note about macros:* The macro names used in this and succeeding chapters are those provided with the Apple IIGS[®] Programmer's Workshop (APW). For more information about availability of SANE software and macros, please refer to Appendix B.

`PUSHLONG` is the 65C816 assembly-language macro used for pushing a 4-byte address onto the stack.

❖ *6502 note:* `PUSH` is the 6502 assembly-language macro used for pushing a 2-byte address onto the stack. It is in the macro file provided with the 6502 SANE software.

The SANE engine for the 65C816 occupies three sections of code named `FP816`, `Elems816`, and `DecStr816`. Arithmetic operations, comparisons, conversions, environmental control, and halt control are in `FP816`. The elementary functions are in `Elems816` and the SANE scanners and formatter are in `DecStr816`. Chapters 12 through 15 describe the functions of `FP816`. Chapters 16 and 17 describe the functions of `Elems816` and `DecStr816`, respectively.

❖ *6502 note:* In the 6502 SANE engine, these routines are in three files named `FP6502`, `Elems6502`, and `DecStr6502`. Access to all three is similar; details are given with the corresponding routines for the 65C816 engine.

Operation forms

The example at the beginning of the chapter illustrates the form of a SANE binary operation. Forms for other SANE operations are described in this section. Examples and further details are given in subsequent chapters.

Arithmetic and auxiliary operations

Most numeric operations are either unary (one operand), like square root and negation, or binary (two operands), like addition and multiplication.

The assembly-language SANE engines provide unary operations in a one-address form:

$DST \leftarrow \langle op \rangle DST$ Example: $B \leftarrow \text{sqrt}(B)$

The operation $\langle op \rangle$ is applied to (or operates on) the operand DST and the result is returned to DST , overwriting the previous value. DST stands for *destination operand*.

The SANE engines provide binary operations in a two-address form:

$DST \leftarrow DST \langle op \rangle SRC$ Example: $B \leftarrow B/A$

The operation $\langle op \rangle$ is applied to the operands DST and SRC and the result is returned to DST , overwriting the previous value. SRC stands for *source operand*.

To store the result of an operation (unary or binary), the location of the operand DST must be known to the SANE engine, so DST is passed by address. In general, all operands, both source and destination, are passed by address. The only exceptions are operands in the 16-bit integer format, which are passed by value for certain operations.

For most operations the storage format for a source operand (SRC) can be the 16-bit integer format, the 32-bit longint (long integer) format, or one of the SANE numeric formats (single, double, extended, or comp). The destination operand (DST) must be in the extended format.

The Nextafter functions have the two-address form:

$DST \leftarrow DST \langle op \rangle SRC$

They differ from the conventions above in that SRC and DST are both single, both double, or both extended.

Conversions

The 65C816 and 6502 SANE engines provide conversions between the extended format and other SANE formats, between extended and integers, and between extended and decimal records. Conversions between binary formats (single, double, extended, comp, and integers) and conversions from decimal to binary have the form

$$DST \leftarrow SRC$$

Conversions from binary to decimal have the form

$$DST \leftarrow SRC \text{ according to } SRC2$$

where *SRC2* is a decimal record specifying the decimal format for the conversion of *SRC* to *DST*.

Comparisons

Comparisons have the form

$$\langle \text{relation} \rangle \leftarrow SRC \text{ compared with } DST$$

where *DST* is extended and *SRC* is single, double, comp, extended, integer, or longint and where $\langle \text{relation} \rangle$ is less, equal, greater, or unordered according as

$$SRC \langle \text{relation} \rangle DST$$

Here the result $\langle \text{relation} \rangle$ is returned in the CPU's registers, rather than in a memory location. The details are given in the section "Comparisons" in Chapter 12, "65C816 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries."

Other operations

The 65C816 and 6502 SANE engines provide inquiries for determining the class and sign of an operand and operations for accessing the floating-point Environment word and the halt address. Forms for these operations vary and are given as the operations are introduced.

External access

On Apple 65C816-based computers such as the Apple IIGS, the SANE engine is part of the Apple IIGS Toolbox and programs invoke it through the Tool Dispatcher.

- ❖ *Note:* Your program must start certain tool sets before initializing the SANE Tool Set. Those tool sets are the Tool Locator, the Memory Manager, and the Miscellaneous Tool Set. For information about those tool sets, refer to Chapter 2 of the *Apple IIGS Toolbox Reference*.

Invocations to the 65C816 SANE engine end with the following instructions:

```
LDX      #ToolSetNum + FuncNum*256
JSL      $E10000
```

The value loaded into the X register contains both the tool set number for SANE and the SANE function number. The function number specifies a SANE function, which corresponds to one of the three operational divisions of the SANE engine: FP816, Elems816, and DecStr816.

- ❖ *Note:* The tool set number for 65C816 SANE is \$0A; the function numbers for the three operational divisions are \$09 for FP816, \$0A for Elems816, and \$0B for DecStr816.
- ❖ *6502 note:* Each division of the 6502 SANE engine is accessed by a JSR instruction to an entry point in memory. The entry points are labeled FP6502, Elems6502, and DecStr6502.

A program that uses the SANE engine should initialize the SANE Environment word. If the program handles floating-point exceptions, it must call SANE to enable the appropriate halt and set the halt vector. See Chapter 14, “Controlling the 65C816 SANE Environment,” and Chapter 15, “Halts in 65C816 SANE,” for information about the Environment word and the halt vector.

A program that uses the 65C816 SANE engine must provide a 256-byte direct page. To do this, the program first calls the Memory Manager to reserve the space and obtain its address. Then the program calls the routine `SANEStartUp` and passes it the address of the direct-page space. (The call to `SANEStartUp` is made through the Tool Dispatcher; the function number is \$02.) The SANE engine requires that parts of the direct page remain unchanged between calls, so the program should not store anything in the space.

- ❖ *6502 note:* FP6502 uses 52 bytes of the zero page for temporary storage. FP6502 does not preserve the pre-call contents of those 52 bytes. Note that FP6502’s use of the zero page is temporary, so the calling program need not preserve the contents of those 52 bytes between calls to FP6502. DecStr6502 uses at most 30 bytes of the same part of zero page; Elems6502 uses none.

The A, X, and Y registers and CPU status flags are not preserved by the SANE engine. All FP816 operations return 0 in the A register and set the carry bit to 0. Elems816 returns through FP816, so it also returns 0 in A and carry, as does DecStr816. FP816 operations return information in the X and Y registers and in the status flags. Operations by Elems816 and DecStr816 leave the contents of the X and Y registers unspecified.

- ❖ *6502 note:* Calls to the 6502 SANE engine leave the contents of the A register unspecified.

Each time it is called, the SANE engine removes its input arguments from the stack and returns no results on the stack. Temporary stack growth during calls to FP816 and Elems816 does not exceed 50 bytes; calls to DecStr816 do not add to the stack size. On exit, decimal mode is clear.

- ❖ *6502 note:* Elems6502 uses at most 50 bytes of the stack. FP6502 uses at most 20 bytes of the stack. DecStr6502 requires no stack growth.

Calling sequence

A typical call to the 65C816 SANE engine consists of a sequence of `PUSHLONG` macros to push the operands and a `PUSHWORD` macro for the opword, followed by the invocation of the Tool Dispatcher at location `$E10000`. `PUSHLONG` pushes a 4-byte address onto the stack: first the high word, then the low word.

```
PUSHLONG <source address>
PUSHLONG <destination address>
PUSHWORD <opword>
LDX      <SANE toolset number> + 256*<function number>
JSL      $E10000
```

- ❖ *Operand note:* For most SANE operations, the source operand is passed by address; for a few operations, the source operand is passed by value.
- ❖ *Macro note:* Macros such as `PUSHLONG` and `PUSHWORD` are normally part of the standard macros provided in an assembly-language development system.

Other calls may have more or fewer operands to push onto the stack. All source operands are pushed before the destination operand, as shown in Figures 11-1 and 11-2.

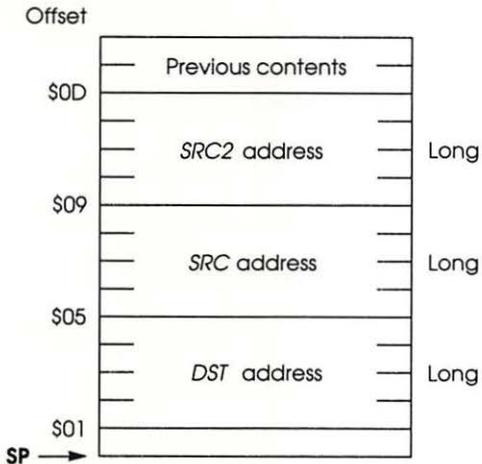


Figure 11-1
SANE operands on the 65C816 stack

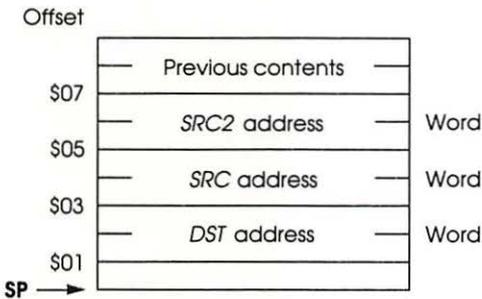


Figure 11-2
SANE operands on the 6502 stack

- ❖ *6502 note:* A typical call to the 6502 SANE engine consists of a similar sequence of 6502 assembly-language instructions and macros, where `PUSH` pushes a 2-byte address onto the stack: first the high byte, then the low byte. In this case, `PUSH` is a macro in the SANE library. There is no Tool Dispatcher; instead, the call passes control directly to the entry point of one of the three operational divisions: `FP6502`, `Elms6502`, and `DecStr6502`.

```

PUSH    <source address>|<source value>
PUSH    <destination address>
PUSH    <opword>
JSR     <entry point>

```

The opword

The opword contains an operand format code in its high-order byte and an operation code in its low-order byte.

The operand format code specifies the format (extended, double, single, integer, longint, or comp) of one of the operands. The operand format code typically gives the format for the source operand (*SRC*). At most one operand format need be specified, because other operands are always extended (or, in the case of Nextafter, because both operands are of the same format).

The operation code specifies the operation to be performed by the SANE engine.

Operation macro names, opwords, and operand format codes are listed in Appendix D, “65C816 and 6502 SANE Quick Reference Guide.”

Example

The format code for single is \$0200. The operation code for divide is \$0006. Hence, the opword \$0206 indicates divide by a value of type single.

Assembly-language macros

Assembly-language development systems such as APW provide mnemonic macros that invoke the Tool Dispatcher with the correct function number for each SANE operation. Each such macro combines a `PUSHWORD` instruction (with the appropriate opword) and a Tool Dispatcher call. For the macro names, see Appendix D.

Using the macros, each call to the SANE engine consists of a `PUSHLONG` for the address of each operand followed by a macro whose name is a mnemonic for the operation and the operand type.

❖ *6502 note:* The 6502 SANE macros (see Appendix D) combine `PUSH <opword>` and `JSR FP6502` to provide mnemonics for calls to the 6502 SANE engine.

Example 1

Add a single-format operand A to an extended-format operand B .

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FADDS     ; Floating-point ADD Single: B <-- B + A
```

❖ *6502 note:* Using macros, calls to the 6502 SANE engine look much the same as calls to the 65C816 SANE engine. Example 1 looks like this:

```
PUSH      A_ADR      ; push address of A
PUSH      B_ADR      ; push address of B
FADDS     ; Floating-point ADD Single: B <-- B + A
```

Example 2

Compute ($B \leftarrow \text{sqrt}(A)$), where A and B are extended. The value of A should be preserved.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FX2X     ; Floating-point eXtended to eXtended:
          ; B <-- A
PUSHLONG  B_ADR      ; push address of B
FSQRTX   ; Floating SQUare RooT eXtended:
          ; B <-- sqrt(B)
```

Example 3

Compute ($C \leftarrow A - B$), where A , B , and C are in the double format. Because destinations are extended, a temporary extended variable T is required.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  T_ADR      ; push address of 10 byte temporary
FD2X     ; Fl-pt convert Double to eXtended:
          ; T <-- A
PUSHLONG  B_ADR      ; push address of B
PUSHLONG  T_ADR      ; push address of temporary
FSUBD    ; Fl-pt SUBtract Double:
          ; T <-- T - B
PUSHLONG  T_ADR      ; push address of temporary
PUSHLONG  C_ADR      ; push address of C
FX2D     ; Fl-pt convert eXtended to Double:
          ; C <-- T
```

65C816 SANE data types

Both the 65C816 SANE engine and the 6502 SANE engine fully support the SANE data types and the integer types shown in Table 11-1.

Table 11-1
65C816 SANE data types

Name	Description
Single	32-bit floating-point
Double	64-bit floating-point
Comp	64-bit integer
Extended	80-bit floating-point
Integer	16-bit two's-complement integer
Longint	32-bit two's-complement integer

Both the 65C816 and the 6502 engines use the convention that least significant bytes are stored in low memory. For example, consider a variable of type single as shown in Table 11-2.

Table 11-2
Bits in a variable of type single

Name	Description
s	Sign
$e_0 \dots e_7$	Exponent (msb ... lsb)
$f_0 \dots f_{22}$	Significand fraction (msb ... lsb)

Figure 11-3 shows the logical structure of this 4-byte variable. If this variable is assigned the address \$1000, its bits are distributed to the bytes in locations \$1000 to \$1003 as shown in the figure. The SANE engines for the 6502 and the 65C816 store the other SANE formats in memory in a similar fashion. Please refer to Chapter 2, "SANE Data Types," for the specifications of those data formats.

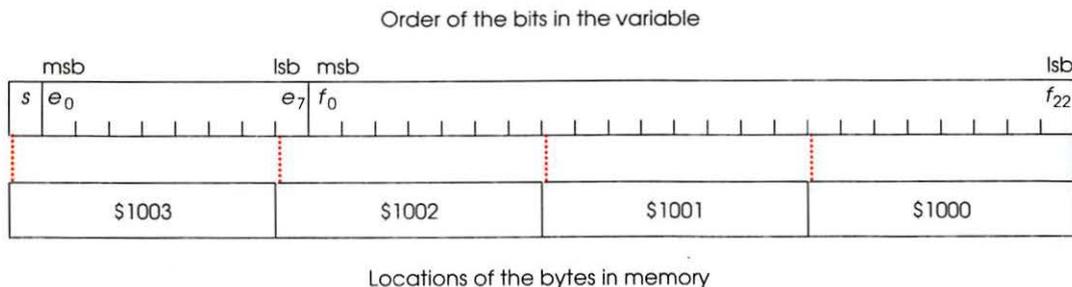


Figure 11-3
Memory format of a variable of type single



Chapter 12



65C816 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries

The operations covered in this chapter follow the access schemes for assembly-language macros described in Chapter 11, “65C816 SANE Basics and Data Types.”

Unary operations follow the one-address form:

$$DST \leftarrow \langle op \rangle DST$$

In the 65C816 SANE engine, they use the calling sequence

```
PUSHLONG  <DST address>
<callname>
```

❖ *6502 note:* In the 6502 SANE engine, unary operations use the calling sequence

```
PUSH      <DST address>
<callname>
```

Binary operations follow the two-address form:

$$DST \leftarrow DST \langle op \rangle SRC$$

In the 65C816 SANE engine, there are two calling sequences for binary operations. The following one is for operations with source operands passed by address:

```
PUSHLONG  <SRC address>
PUSHLONG  <DST address>
<callname>
```

The 65C816 SANE engine uses the following calling sequence for operations with source operands passed by value:

```
PUSHWORD  <SRC value>
PUSHLONG  <DST address>
<callname>
```

❖ *6502 note:* In the 6502 SANE engine, binary operations use the calling sequence

```
PUSH      <SRC address>|<SRC value>
PUSH      <DST address>
<callname>
```

The destination operand (*DST*) for these operations is passed by address and is usually in the extended format. Generally, the source operand (*SRC*) is passed by address and may be single, double, comp, extended, (16-bit) integer, or (32-bit) longint. Some operations are distinguished by passing the source operand by value, by requiring some specific type for *SRC*, by using a nonextended destination, or by returning auxiliary information in the X and Y registers and in the processor status bits. In this section, operations so distinguished are noted. The examples employ the macro names listed in Appendix D, “65C816 and 6502 SANE Quick Reference Guide.”

❖ *6502 note:* Unless specifically for the 6502, the examples that follow are written for the 65C816. Calls to the 6502 SANE engine are similar, but the macro they use for pushing 2-byte addresses onto the stack is named `PUSH` instead of `PUSHLONG`.

Add, Subtract, Multiply, and Divide

These are binary operations and follow the two-address form.

Example

$B \leftarrow B/A$, where A is double and B is extended.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FDIVD     ; divide with source operand in
           ; double format
```

Square Root

Square Root is a unary operation and follows the one-address form.

Example

$B \leftarrow \text{sqrt}(B)$, where B is extended.

```
PUSHLONG  B_ADR      ; push address of B
FSQRTX    ; square root (operand is always
           ; extended format)
```

Round-to-Integer and Truncate-to-Integer

These are unary operations and follow the one-address form.

The Round-to-Integer operation `FRINTX` rounds (according to the current rounding direction) to an integral value in the extended format. The Truncate-to-Integer operation `FTINTX` rounds toward zero (regardless of the current rounding direction) to an integral value in the extended format. The calling sequence is the usual one for unary operations, illustrated in the previous section for Square Root.

Remainder

This is a binary operation and follows the two-address form.

Remainder returns auxiliary information. The seven low-order bits of the magnitude of the integer quotient n are returned in the X register. The N status bit is set if and only if n is negative. The Y register receives \$80 if n is negative and 0 otherwise.

Example

$B \leftarrow B \text{ rem } A$, where A is single and B is extended.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FREMS                    ; remainder with source operand in
                        ; single format
```

Logb and Scalb

Logb is a unary operation and follows the one-address form.

Scalb is a binary operation and follows the two-address form. Its form is unusual in that its source operand is a 16-bit integer passed by value.

Example

$B \leftarrow B \times 2^{130}$, where B is extended.

```
PUSHWORD  #$0082      ; (82 hex = 130 decimal)
PUSHLONG  B_ADR      ; push address of B
FSCALBX                    ; scalb
```

❖ *6502 note:* In the 6502 SANE engine, calls Scalb and Logb push the source onto the stack a byte at a time and use a macro named PUSH in place of PUSHLONG. The code looks like this:

```
LDA        #0          ; push high byte
PHA                          ; of source
LDA        #82         ; push low byte of source
PHA                          ; (82 hex = 130 decimal)
PUSH      B_ADR      ; push address of B
FSCALBX                    ; scalb
```

Negate, Absolute Value, and CopySign

Negate and Absolute Value are unary operations and follow the one-address form. CopySign is a binary operation and follows the two-address form. The SANE engine treats these operations as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal invalid.

❖ *Note:* The order of the operands in the SANE CopySign function is reversed from that suggested in IEEE Standard 754.

Example

Copy the sign of a comp number *A* into the sign of an extended number *B*.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FCPYSGNC  ; copy-sign with source operand in
           ; comp format
```

Nextafter

The Nextafter operations are binary and use the two-address form; they require both source and destination operands to be of the same floating-point type (single, double, or extended).

Example

$B \leftarrow \text{Nextafter}(B)$ in the direction of *A*, where *A* and *B* are double (so *next-after* means *next-double-after*).

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FNEXTD    ; next-after in double format
```

Comparisons

The SANE engine provides two comparison operations: `FCPX` (which signals invalid if its operands compare unordered) and `FCMP` (which does not). Each compares a source operand (which may be single, double, comp, extended, integer, or longint) with a destination operand (which must be extended). The result of a comparison is the relation (less, greater, equal, or unordered) for which

SRC <relation> *DST*

is true. The result is delivered in the Z, N, and V status bits and redundantly in the low bytes of the X and Y registers, as shown in Table 12-1. Note that the X and Y registers hold the same low byte value unless the relation is equal: this is a by-product of an implementation optimization.

Table 12-1
Results of comparisons

Result	Status bit			X register*	Y register*
	Z	N	V		
Greater	0	0	1	\$40	\$40
Less	0	1	0	\$80	\$80
Equal	1	0	0	\$02	\$00
Unordered	0	0	0	\$01	\$01

* 1-byte value, in the low byte of register on the 65C816

The IEEE Standard specifies that a relational operator that involves less or greater but not unordered should signal invalid if the operands are unordered. These relational operators are implemented by choosing the comparison that signals invalid appropriately.

❖ *Note:* The comparison macros are listed in Appendix D, “65C816 and 6502 SANE Quick Reference Guide.”

Example 1

Test $A \leq B$, where A is single and B is extended; if true, branch to LOC; signal if unordered.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FCPXS                    ; compare using source of type single,
                        ; signal invalid if unordered
FBLE      LOC        ; branch if A <= B
```

Example 2

Test A not-equal B , where A is double and B is extended; if true, branch to LOC. (Note that not-equal is equivalent to less, greater, or unordered, and invalid should not be signaled on unordered.)

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FCMPD                    ; compare using source of type double,
                        ; do not signal invalid if unordered
FBNE      LOC        ; branch if A not-equal B
```

Inquiries

The classify operation provides both class and sign inquiries. This operation takes one source operand (single, double, extended, comp, integer, or longint), which is passed by address.

The class of the operand is returned in the low byte of the X register, as shown in Table 12-2.

Table 12-2
Operand classes

Hexadecimal value	Two's-complement value	Class
\$FC	-4	Signaling NaN
\$FD	-3	Quiet NaN
\$FE	-2	Infinity
\$FF	-1	Zero
\$00	0	Normalized
\$01	1	Denormalized

The N status bit receives the sign bit of the operand. Redundantly, the low byte of the Y register is set to \$80 if the sign bit is set and \$00 otherwise.

Example

Branch to `LOC` if the single-format value `A` is an Infinity.

```
PUSHLONG      A_ADR ; push address of A
FCLASSS      ; classify single
FBINF LOC      ; branch on infinite to LOC
```

- ❖ *Note about macros:* Like the other macros in Part II, the floating-point branch-control macros `FBNE`, `FBLE`, and `FBINF` are provided with APW.



Chapter 13



Conversions in 65C816 SANE

This chapter discusses conversions between binary formats and conversions between binary and decimal formats. Conversions between decimal formats, provided by DecStr816 and DecStr6502, are discussed in Chapter 17, “65C816 SANE Scanners and Formatter.”

- ❖ *6502 note:* All the examples in this chapter are written for the 65C816. Calls to the 6502 SANE engine are similar, but the macro they use for pushing 2-byte addresses onto the stack is named `PUSH` instead of `PUSHLONG`.

Conversions between binary formats

The SANE engine provides conversions between the extended type and the SANE types single, double, and comp, as well as the 16- and 32-bit integer types.

Conversions to extended

The SANE engine provides conversions of a source with format single, double, comp, extended, integer, or longint to a destination in extended format, as shown in Table 13-1.

Table 13-1
Conversions to extended format

Function name	Operation
FS2X	extended ← single
FD2X	extended ← double
FC2X	extended ← comp
FX2X	extended ← extended
FI2X	extended ← integer
FL2X	extended ← longint

All operands, even integer ones, are passed by address. The following example illustrates the calling sequence.

Example

Convert *A* to *B*, where *A* is in comp format and *B* is in extended format.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FC2X      ; convert comp to extended
```

Conversions from extended

The SANE engine provides conversions of an extended-format source to a destination of format single, double, comp, extended, integer, or longint, as shown in Table 13-2.

Table 13-2
Conversions from extended format

Function name	Operation
FX2S	single ← extended
FX2D	double ← extended
FX2C	comp ← extended
FX2X	extended ← extended
FX2I	integer ← extended
FX2L	longint ← extended

Note that conversion to a narrower format may alter values. Contrary to the usual scheme, the destination for these conversions need not be of type extended. All operands are passed by address. The following example illustrates the calling sequence.

Example

Convert *A* to *B*, where *A* is in extended format and *B* is double.

```
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  B_ADR      ; push address of B
FX2D      ; convert extended to double
```

Binary-decimal conversions

The SANE engine provides conversions between the binary types (single, double, comp, extended, integer, and longint) and the decimal record type.

Decimal records and decform records (used to specify the form of decimal representations) are described in Chapter 3, "Conversions in SANE." The maximum length of the `sig` digit-string of a decimal record is 28.

- ❖ *Note:* The value 28 is specific to the 6502 and 65C816 implementations; algorithms you intend to port to other SANE implementations should use no more than 18 digits in `sig`.

The values of `style` fields of decform records and of `sgn` fields of decimal records are stored as 16-bit integers. The integer fields of decimal and decform records conform to the 65C816 and 6502 convention of storing the least significant byte at the lowest address.

Binary to decimal

The calling sequence for a conversion from a binary format to a decimal record passes the address of a decform record, the address of a binary source operand, and the address of a decimal-record destination.

Example

Convert a comp-format value *A* to a decimal record *D* according to the decform record *F*.

```
PUSHLONG  F_ADR      ; push address of F
PUSHLONG  A_ADR      ; push address of A
PUSHLONG  D_ADR      ; push address of D
FC2DEC    ; convert comp to decimal
```

Fixed-format overflow

If a number is too large for a chosen fixed style, the SANE engine returns the 28 most significant digits of the number in the `sig` field of the decimal record and sets the `exp` field so that the decimal record contains a valid floating-point representation of the number. (SANE implementations for the MC68000 simply set `sig` to the string '?.')

Decimal to binary

The calling sequence for a conversion from decimal to binary passes the address of a decimal-record source operand and the address of a binary destination operand.

Example

Convert the decimal record *D* to a double-format value *B*.

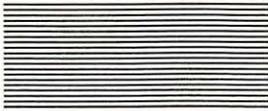
```
PUSHLONG  D_ADR      ; push address of D
PUSHLONG  B_ADR      ; push address of B
FDEC2D    ; convert decimal to double
```

Techniques for maximum accuracy

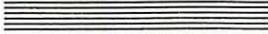
The following technique applies to the 65C816 and 6502 SANE engines; other SANE implementations require other techniques.

If you are writing a parser and must handle a number with more than 28 significant digits, follow these rules:

1. Place the implicit decimal point to the right of the 28 most significant digits.
2. If any of the discarded digits to the right of the implicit decimal point is nonzero, then
 - signal the inexact exception
 - if the number is positive and the rounding direction is upward, or if the number is negative and the rounding direction is downward, then replace the last (28th) ASCII character with its successor to guarantee a correctly rounded result. (The successor of '9' is ':'.)



Chapter 14



Controlling the 65C816 SANE Environment

The Environment word

The floating-point environment is encoded in the 16-bit integer format as shown in Figure 14-1. Table 14-1 gives the hexadecimal value of each of the environment flags. Rounding direction and precision are stored as 2-bit encoded values; exception and halt-enabled flags are set as individual bits. Note that the default environment is represented by the integer value zero.

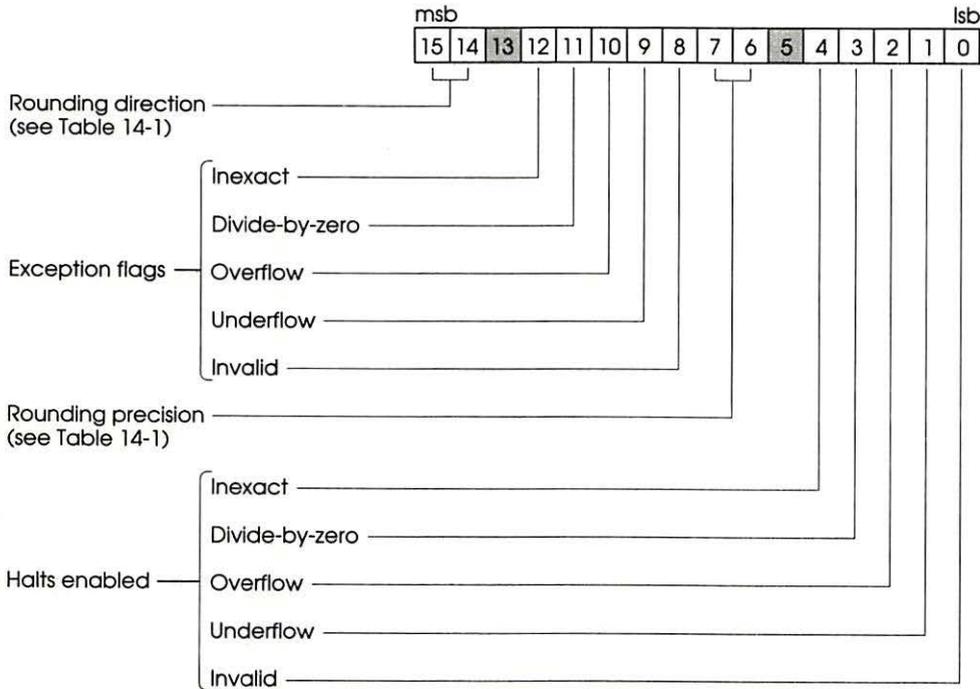


Figure 14-1
The Environment word for the 6502 and 65C816

Table 14-1

Bits in the Environment word for the 6502 and 65C816

Group name	Mask bits	Mask value	Description
Rounding direction (Bit group \$C000)	15 14 0 0 0 1 1 0 1 1	\$0000 \$4000 \$8000 \$C000	To-nearest Upward Downward Toward-zero
Exception flags (Bit group \$1F00)	12 11 10 9 8 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$1000 \$0800 \$0400 \$0200 \$0100	Inexact Divide-by-zero Overflow Underflow Invalid
Rounding precision (Bit group \$00C0)	7 6 0 0 0 1 1 0 1 1	\$0000 \$0040 \$0080 \$00C0	Extended Double Single (Undefined)
Halts enabled (Bit group \$001F)	4 3 2 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$0010 \$0008 \$0004 \$0002 \$0001	Inexact Divide-by-zero Overflow Underflow Invalid

* *Note:* Bits 5 and 13 are not used.

Example

With rounding toward-zero, inexact and underflow exception flags raised, extended rounding precision, and halt on invalid, overflow, and division-by-zero, the most significant byte of the Environment word has the value \$D2 and the least significant byte has the value \$0D.

You gain access to the environment settings through the procedures Get-Environment, Set-Environment, Test-Exception, Set-Exception, Procedure-Entry, and Procedure-Exit.

Get-Environment and Set-Environment

Get-Environment takes no input operand. The Environment word is returned in the X register. The most significant byte of the environment is also returned in the low byte of the Y register.

- ❖ *6502 note:* In the 6502 SANE engine, Get-Environment returns the least significant byte of the Environment word in the X register and the most significant byte in the Y register.

Set-Environment has one input operand: a 16-bit integer, passed by value, that is interpreted as an Environment word.

- ❖ *Note:* Setting the Environment word does not cause halts.

Example

Set rounding direction to downward.

```
FGETENV          ; get environment
TXA              ; A <-- environment word
AND      #$3FFF  ; clear C0 bits
ORA      #$8000  ; set bits for round downward
PHA              ; push environment word
FSETENV          ; set environment
```

- ❖ *6502 note:* The following code shows how to change the Environment word in the 6502 SANE engine:

```
FGETENV          ; get environment
TYA              ; A <-- msbyte
AND      #03F    ; clear C0 bits
ORA      #080    ; set bit for round downward
PHA              ; push msbyte
TXA              ; A <-- lsbyte
PHA              ; push lsbyte
FSETENV          ; set environment
```

Test-Exception and Set-Exception

Test-Exception has one integer operand, passed by value, which is regarded as a sum of the hex values of the individual bits, as shown in Table 14-2.

Table 14-2
Bits in the Exception word

Bit value	Description
\$01	Invalid
\$02	Underflow
\$04	Overflow
\$08	Divide-by-zero
\$10	Inexact

If an exception flag is set for any of the corresponding bits set in the operand, then Test-Exception clears the Z flag in the Processor Status register; otherwise, Test-Exception sets the Z flag.

Example

Branch to XLOC if invalid or overflow is set.

```
PUSHWORD #5 ; invalid + overflow
FTESTXCP ; test exception
BNE XLOC ; branch if Z is clear
```

❖ *6502 note:* The following code shows how to check the exception flags in the 6502 SANE engine:

```
LDA #00 ; A <-- 0
PHA ; push msbyte
LDA #05 ; A <-- invalid + overflow
PHA ; push lsbyte
FTESTXCP ; test exception
BNE XLOC ; branch if Z is clear
```

Set-Exception takes one integer operand, passed by value, which encodes a set of exceptions in the manner described above for Test-Exception. Set-Exception stimulates the exceptions indicated in the operand; that is, the command not only turns on the bits in the Environment word, it also causes a halt if any corresponding halt bit is set.

Procedure-Entry and Procedure-Exit

Procedure-Entry saves the current SANE Environment word at the address passed as the operand, and sets the operative environment to the default settings.

Procedure-Exit saves the exception flags (temporarily), sets the Environment word to the value passed as the operand, and then stimulates the saved exceptions (that is, turns on the bits in the Environment word and causes a halt if any corresponding halt bit is set).

Example

Here is a procedure that appears to its callers as an atomic operation:

```
ATOMICPROC  PUSHLONG  ENV_ADR  ; push address to store
                                     ; environment
            FPROCENTRY ; procedure entry

            ; ...body of routine...

            PUSHWORD  ENV        ; push saved environment
            FPROCEXIT ; procedure exit
            RTS

ENV         ds         2          ; storage for saved
                                     ; environment
ENV_ADR     dc         i4'ENV'    ; address of ENV
```

❖ *6502 note:* The following code shows how to use these calls in a 6502 program:

```
ATOMICPROC  PUSH  E_ADR  ; push address for storing
                                     ; environment
            FPROCENTRY ; procedure entry

            ...body of routine...

            PUSH  E        ; push saved environment
            FPROCEXIT ; procedure exit
            RTS

E           .WORD        ; storage for saved environment
E_ADR      .WORD  E      ; address of E
```



Chapter 15



Halts in 65C816 SANE

The 65C816 SANE engine lets the application transfer program control when selected floating-point exceptions occur. Because this facility is used to implement halts in high-level languages, we refer to it as a halt mechanism. The assembly-language programmer can write a halt handler routine to cause special actions for floating-point exceptions. (The 65C816 SANE halt mechanism differs from the traps that are an optional part of the IEEE Standard.)

Conditions for a halt

Any floating-point exception triggers a halt if the corresponding halt is enabled. The halt for a particular exception is enabled when two conditions are met:

- The halt (trap) vector, which can be set by using the operation `SetHaltVector`, is not zero.
- The halt-enable bit corresponding to that exception is set.

The halt mechanism

If the halt for a given exception is enabled, the 65C816 SANE engine does the following things when that exception occurs.

1. The engine returns the same result to the destination address that it would return if the halt were not enabled. (However, the engine does not set floating-point exception flags for the current operation and does not return results to the X, Y, and P registers.)
2. The engine leaves the caller's return address on the top of the stack.
3. The engine leaves halt status information in its direct page, as shown in Figure 15-1.
- ❖ *6502 note:* The 6502 SANE engine leaves its halt status information in a record in memory and sets the X and Y registers to the least and most significant bytes, respectively, of the address of the record. Refer to Figure 15-2.
4. The engine transfers control by means of a JSL (long jump-to-subroutine) instruction to the location given by the halt vector.

Important

Halts occur only on calls to FP816. Elems816 stimulates halts only through a `ProcExit` call to FP816. DecStr816 makes no calls to FP816 and never stimulates halts.

Halt status information

Figure 15-1 shows the contents of the 65C816 SANE engine's direct page immediately after a halt occurs. In addition to halt status information, the direct page contains the input parameters and SANE opword.

For one-argument calls, the *C* address is the address of *DST*. For two-argument calls, the *C* address is the address of *DST* and *B* is the address of *SRC*. For binary-to-decimal conversion, the *A* address is the address of the decimal record, *B* is the address of the binary value, and *C* is the address of the decimal record.

Offset	Field	Size
\$22	Pending Y-hi	Byte
\$21	Pending X-hi, Y-lo	Byte
\$20	Pending X-lo	Byte
\$1E	Pending exceptions	Word
\$1C	Environment word	Word
\$18	Halt vector	Long
\$14	A address	Long
\$10	B address	Long
\$0C	C address	Long
\$0A	Opword	Word
\$08	Caller's data bank	Word
\$06	Caller's direct page	Word
\$00	Return address	3 bytes
\$00	Return address	3 bytes

Figure 15-1
65C816 SANE direct-page contents upon halt

Important

Future implementations of SANE for the 65C816 may not store the floating-point environment and halt vector in the direct page. You may forfeit upward compatibility if you access these variables directly. Always access these variables only by means of SANE calls.

- ❖ *Note:* Scanners, which have four operands, do not cause halts, so halt handlers never deal with direct-page information for scanners.

When a halt occurs, the SANE engine uses the halt vector to transfer control to the application's halt handler. When the halt handler receives control, the 65C816's A register contains the information in the pending-exceptions field. The halt handler can continue execution as if no halt had occurred by executing an RTL instruction. When the SANE engine regains control, it uses the contents of the A register, not the value in the pending-exceptions field, to set the final floating-point exceptions. Before it returns, the application's halt handler must load the A register with the contents of the pending-exceptions field to ensure that the exceptions from the current call are handled correctly.

Important

The value in *pending exceptions* is a sum of the five exception constants, represented as an integer from 0 to 31. Unpredictable results occur if the A register contains a value out of this range when your application exits from the halt handler.

- ❖ *6502 note:* Figure 15-2 shows the 6502 SANE halt status record. In 6502 SANE, exceptions from current operation are encoded like the exception flags in the most significant byte of the floating-point environment (see Chapter 7, "Controlling the SANE Environment"). The pending-X and pending-Y bytes contain what the X and Y registers would have contained on a normal (halts not enabled) exit. Those bytes can be used by a halt handler to set the relevant bits of the 6502 status byte, as well as the X and Y registers, to their normal exit state. (The example in the section "Using the Halt Mechanism" illustrates this use of pending X and pending Y.)

Offset		
\$08	Pending exceptions	Byte
	Opword	Word
\$06	Pending Y	Byte
\$05	Pending X	Byte
\$04	Environment word	Word
\$02	Halt vector	Word
\$00		

Figure 15-2
6502 SANE status record upon halt

Halt vector operations

The SANE engine has two calls for manipulating the halt vector: SetHaltVector and GetHaltVector.

The SetHaltVector routine sets the halt vector to the 4-byte vector passed by value on the stack. There are no other operands.

- ❖ *65C816 note:* Addresses in the 65C816 are 3 bytes long; Apple software for the 65C816 passes addresses as 4 bytes by pushing or pulling two words on the stack.
- ❖ *6502 note:* The SetHaltVector routine has one input operand: a 16-bit integer, passed by value, that is interpreted as the halt vector.

The GetHaltVector routine returns a 3-byte halt vector in the X and Y registers. X contains the low 2 bytes (first and second) of the halt vector, and Y contains the second and third bytes; the second byte of the halt vector occurs in both the X and the Y registers.

- ❖ *6502 note:* The GetHaltVector routine takes no input operand and returns the least and most significant bytes of a 2-byte halt vector in the X and Y registers, respectively.

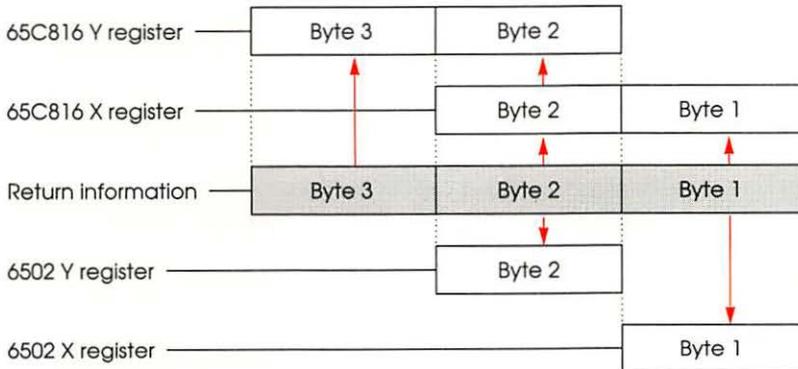


Figure 15-3
Data returned in X and Y registers

Using the halt mechanism

These examples illustrate the use of the halt mechanism. To use the halt mechanism, the program must first set the halt vector to the starting address of a halt handler routine. The halt handler shown in these examples returns control to the program directly following the call that caused the halt as if no halt had occurred.

Halt example for the 65C816

The opcodes represented by question marks are in the application's code; the first line of the halt handler is labeled HH.

```
; Set halt vector.
```

```
    PUSHLONG    #HH          ; push address of halt handler
    FSETHV      ; set halt vector
    - - -
    F???        ; SANE call causing a halt
    ???        ; halt handler returns control to
    - - -      ; here
```

```
HH      START
```

```
; Insert any specific halt-handling code here.
```

```
; Finally, ensure that the exceptions are handled correctly (Remember that
; the contents of the A register will be used to set the final exceptions.)
```

```
    LDA        30           ; pending exceptions offset from
    RTL                           ; beginning of SANE direct page

    END
```

Halt example for the 6502

The opcodes represented by question marks are in the application's code; the first line of the halt handler is labeled HH.

```
; Set halt (trap) vector.
    PUSH      HHADR      ; push address of halt handler
    FSETHV                    ; set halt vector
    - - -
    F???                    ; call to FP6502 causing halt
    ???                    ; halt handler returns control to
                          ; here
    - - -

HH      ; halt handler routine starts here
; Store address in X and Y registers into temporary
; location.
    STX      TEMP
    STY      TEMP+1      ; temp points to status info
                          ; record

; OR exceptions from current operation into environment.
    LDY      #8
    LDA      (TEMP),Y    ; A <-- current exceptions
    LDY      #3
    ORA      (TEMP),Y    ; A <-- A OR msbyte of
                          ; environment
    STA      (TEMP),Y    ; msbyte of environment <-- A

; Operate on result info to set registers and status bits
; as though no halt occurred.
    INY                    ; Y <-- 4
    LDA      (TEMP),Y
    TAX                    ; X <-- pending X
    INY                    ; Y <-- 5
    LDA      (TEMP),Y
    STA      TEMP          ; TEMP <-- pending Y
    BIT      TEMP          ; determines V bit
    TAY                    ; Y <-- pending Y
                          ; determines N and Z bits

; Return to user operation after call to FP6502 which
; triggered halt (address already on top-of-stack).
    RTS
    - - -

HHADR   .WORD      HH      ; HHADR contains address of HH
```



Chapter 16



Elementary Functions in 65C816 SANE

The elementary functions that are specified by the Standard Apple Numerics Environment are made available to 65C816 assembly-language programs by Elems816. Elems816 also includes two functions that compute $\log_2(1 + x)$ and $2^x - 1$ accurately. Elems816 makes calls to FP816 for its basic arithmetic. The access schemes for Elems816 are similar to those for FP816 (described in Chapter 11, “65C816 SANE Basics and Data Types”). Opwords and macro names used in the examples below are listed in Appendix D.

- ❖ *6502 note:* The elementary functions for the 6502 SANE engine are in Elems6502. Like Elems816, Elems6502 makes calls to FP6502 and uses similar access schemes.

One-argument functions

The tool set includes calls for the following one-argument elementary functions:

- $\text{Log2}(x)$ computes the base-2 logarithm of x .
- $\text{Ln}(x)$ computes the natural logarithm of x .
- $\text{Ln1}(x)$ computes the natural logarithm of $(1 + x)$.
- $\text{Exp2}(x)$ computes 2^x .
- $\text{Exp}(x)$ computes e^x .
- $\text{Exp1}(x)$ computes $e^x - 1$.
- $\text{Cos}(x)$ computes the cosine of x .
- $\text{Sin}(x)$ computes the sine of x .
- $\text{Tan}(x)$ computes the tangent of x .
- $\text{Atan}(x)$ computes arctangent of x .
- $\text{RandomX}(x)$ computes a pseudorandom value with x as seed.
- $\text{Log21}(x)$ computes $\log_2(1 + x)$.
- $\text{Exp21}(x)$ computes $2^x - 1$.

These calls each have one extended argument, passed by address, and use the following one-address calling sequence to obtain $DST \leftarrow \langle op \rangle DST$.

```
PUSHLONG  <DST address>
PUSHWORD  <opword>
LDX       #ToolSetNum + FuncNum*256
JSL       $E10000
```

This calling sequence is the same as that for unary operations, such as Square Root and Negate, in the core routines described in Chapter 11, “65C816 SANE Basics and Data Types.”

❖ *6502 note:* A typical call to the 6502 SANE engine pushes a 2-byte address onto the stack: first the high byte, then the low byte. There is no Tool Dispatcher; instead, the call passes control directly to the entry point of Elems6502.

```
PUSH      <DST address>
PUSH      <opword>
JSR      ELEMS6502
```

Example

Like the core routines described in Chapter 11, the elementary functions are normally called by means of macros. For example, to obtain $B \leftarrow \sin(B)$, where B is of extended type, the call looks like this:

```
PUSHLONG  B_ADR      ; push address of B
FSINX     ; B <-- sin(B)
```

❖ *6502 note:* Using macros, calls to the 6502 SANE engine look much the same as calls to the 65C816 SANE engine. The example looks like this:

```
PUSH      B_ADR      ; push address of B
FSINX     ; B <-- sin(B)
```

Two-argument functions

The tool set includes calls for the following two-argument elementary functions:

- XPwrY(x,y) computes x^y .
- XPwrI(x,i) computes x^i .

General exponentiation (XPwrY) has two extended arguments, both passed by address. The result is returned in x .

The function uses this calling sequence for binary operations to obtain $DST \leftarrow DST^{SRC}$:

```
PUSHLONG <SRC address> ; push exponent address first
PUSHLONG <DST address> ; push base address second
PUSHWORD <opword>
LDX      #ToolSetNum + FuncNum*256
JSL      $E10000
```

Integer exponentiation (XPwrI) has two arguments. The extended argument x , passed by address, receives the result. The 16-bit integer argument i is passed by value.

The function uses this modified calling sequence for binary operations to obtain $DST \leftarrow DST^{SRC}$:

```
PUSHWORD <SRCvalue> ; push integer exponent value first
PUSHLONG <DST address> ; push base address second
PUSHWORD <opword>
LDX      #ToolSetNum + FuncNum*256
JSL      $E10000
```

Example

To obtain $B \leftarrow B^3$, where B is of extended type, the call looks like this (using a macro call):

```
PUSHWORD #3 ; push exponent by value
PUSHLONG B_ADR ; push address of B
FXPWRI ; integer exponentiation
```

❖ *6502 note:* For 6502 SANE, the example looks like this:

```
LDA      #0 ; A <-- 0
PHA      ; push msbyte of exponent
LDA      #3 ; A <-- 3
PHA      ; push lsbyte of exponent
PUSH     B_ADR ; push address of B
FXPWRI ; integer exponentiation
```

Three-argument functions

The tool set includes calls for the following three-argument elementary functions:

- Compound(r, n) computes $(1 + r)^n$.
- Annuity(r, n) computes $(1 - (1 + r)^{-n})/r$ when $r \neq 0$.

These functions use this calling sequence:

```
PUSHLONG <SR2C address>      ; push address of rate first
PUSHLONG <SRC address>       ; push address of number of
                             ; periods second
PUSHLONG <DST address>       ; push address of destination
                             ; third
PUSHWORD <opword>
LDX      #ToolSetNum + FuncNum*256
JSL     $E10000
```

The operation syntax is

$DST \leftarrow \langle op \rangle (SRC2, SRC)$

where $\langle op \rangle$ is Compound or Annuity, $SRC2$ is the rate, and SRC is the number of periods. All arguments $SRC2$, SRC , and DST must be of the extended type.

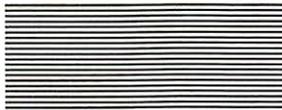
Example

To obtain $C \leftarrow (1 + R)^N$, where C , R , and N are of type extended, the call looks like this (using macros):

```
PUSHLONG R_ADR      ; push address of R
PUSHLONG N_ADR      ; push address of N
PUSHLONG C_ADR      ; push address of C
FCOMPOUND           ; compound operation
```

❖ *6502 note:* For 6502 SANE, the example looks like this:

```
PUSH      R_ADR      ; push address of R
PUSH      N_ADR      ; push address of N
PUSH      C_ADR      ; push address of C
FCOMPOUND           ; compound operation
```



Chapter 17



65C816 SANE Scanners and Formatter

The Standard Apple Numerics Environment specifies conversions between decimal strings and decimal records. In the SANE implementations for the 65C816 and the 6502, these scanning and formatting routines are contained in DecStr816 and DecStr6502. The routines in DecStr816 and DecStr6502 are designed for use with FP816 and FP6502, which provide binary-decimal conversions between decimal records and the SANE data formats. Thus, the SANE scanning routines provide the application developer the solution to the problems of scanning input strings to produce SANE-type values and of formatting SANE-type values for output.

Opwords and macro names are listed in Appendix D, “65C816 and 6502 SANE Quick Reference Guide.”

The scanning routines use an access scheme similar to that of the core routines. (See Chapter 11, “65C816 SANE Basics and Data Types.”) The scanning routines clear their arguments from the stack before returning.

Numeric scanners

The SANE numeric scanners are called Pstr2dec (*P* for the *Pascal* programming language, *str* for *string*) and Cstr2dec (*C* for the *C* programming language, *str* for *string*). Both scanners take four arguments, all passed by address:

- string to be scanned (input)
- 16-bit integer index into string (input and output)
- decimal record for result (output)
- 16-bit integer for valid-prefix indication (output)

On input, the index indicates the position in the string where scanning is to begin; on output, the index is one greater than the position of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed and returned in the decimal record. If no numeric substring is recognized, then the index remains unchanged. The valid-prefix parameter on output contains 1 (true) if the entire input string beginning at the input index is a valid numeric string or a prefix of a valid numeric string. It contains 0 (false) otherwise.

The only difference between the scanners is in the string input argument. Pstr2dec operates on a string having the string length in the zeroth byte of the string and the initial character of the string in the first byte. Cstr2dec operates on a string with the initial character of the string in the zeroth byte, no length byte, and termination of the string by a null character (ASCII code 0).

- ❖ *Note:* Because it stops scanning when it encounters a nonnumeric character, Cstr2dec can be used to scan numbers embedded in large text buffers.

The scanners use a calling sequence with four operands:

```
PUSHLONG <address of string>
PUSHLONG <address of index>
PUSHLONG <address of decimal record>
PUSHLONG <address of valid-prefix>
PUSHWORD <opword>
LDX      #ToolSetNum + FuncNum*256
JSL      $E10000
```

❖ *6502 note:* In 6502 SANE, addresses are 2 bytes and the calling sequence for the scanners looks like this:

```
PUSH     <address of string>
PUSH     <address of index>
PUSH     <address of decimal record>
PUSH     <address of valid-prefix>
PUSH     <opword>
JSR      DecStr6502
```

The next chapter gives examples of the use of the scanners.

Numeric formatter

The formatter Dec2Str takes three arguments, all passed by address:

- decform record for formatting specification (input)
- decimal record to be formatted (input)
- string for result (output)

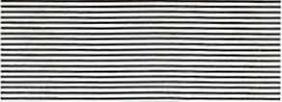
This routine returns a decimal string representing the input value from the decimal record, formatted according to input specifications passed in the decform record. The result string is a Pascal string: the zeroth byte contains the string length, and the first byte contains the first character in the string. For a full description of Dec2Str, see the section “Conversions From Decimal Records to Decimal Strings” in Chapter 3.

The formatter uses this calling sequence:

```
PUSHLONG <address of decform record>
PUSHLONG <address of decimal record>
PUSHLONG <address of string>
PUSHWORD <opword>
LDX      #ToolSetNum + FuncNum*256
JSL      $E10000
```

❖ *6502 note:* In 6502 SANE, the calling sequence for the formatter looks like this:

```
PUSH     <address of decform record>
PUSH     <address of decimal record>
PUSH     <address of string>
PUSH     <opword>
JSR      DecStr6502
```



Chapter 18



Examples: Using the 65C816 and 6502 SANE Engines

The following examples illustrate the use of the SANE engines for the 65C816 and the 6502. The names of the SANE macros used in the examples are listed in Appendix D, “65C816 and 6502 Quick Reference Guide”; macros not listed there, such as `MOVEWORD` and `PUSHLONG`, are provided by the APW development system.

65C816 examples

Each of the 65C816 examples assumes the programmer is using the Apple IIGS Toolbox and that the program starts up and shuts down the SANE tool correctly. The program must include the following steps:

1. Somewhere early in the program, call the Memory Manager to reserve 256 bytes of zero bank for use as SANE direct page. (For this example, `#SANEdirectpg` is the address of that memory.)
2. The program then makes the following call to initialize SANE:

```
PUSHWORD    #SANEdirectpg
    _SANEStartup
```

3. Near the end of the program, make the call to shut down SANE:

```
    _SANEShutdown
```

4. The program then calls the Memory Manager to release the memory that was reserved for the SANE direct page (often by releasing *all* reserved memory).

65C816 example: polynomial evaluation

This example evaluates the polynomial

$$x^3 + 2x^2 - 5$$

It illustrates the evaluation of a polynomial

$$c_0x^n + c_1x^{n-1} + \dots + c_n$$

using Horner's recurrence:

$$r \leftarrow c_0$$

$$r \leftarrow (r \times x) + c_j, \text{ for } j = 1 \text{ to } n$$

On entry, `rAdr` points to an extended result field, `cAdrBgn` points to the coefficient table of extended values, byte `nCoefs` contains the degree n (<256) of the polynomial, and `xAdr` points to an extended function argument x . The coefficient table consists of $n + 1$ extended coefficients, starting with c_0 . In this example, $n = 3$, $c_0 = 1$, $c_1 = 2$, $c_2 = 0$, and $c_3 = -5$.

Entry is by a JSR instruction to POLYEVAL. (POLYEVAL calls FP816.)

```
POLYEVAL  ENTRY
          MOVELONG  cAdrBgn,cAdr ; copy arguments
          MOVEWORD  nCoefs,NumCoefs
          PUSHLONG  cAdr          ; c0 --> r
          PUSHLONG  rAdr
          FX2X

POLYLOOP  PUSHLONG  xAdr          ; r*x --> r
          PUSHLONG  rAdr
          FMULX

          CLC
          LDA       cAdr
          ADC       #10          ; advance to next coefficient
          STA       cAdr
          LDA       cAdr+2
          ADC       #0           ; allow for possible carry
          STA       cAdr+2

          PUSHLONG  cAdr          ; r+cj --> r
          PUSHLONG  rAdr
          FADDX

          DEC       NumCoefs     ; decrement NumCoefs and
          BNE      POLYLOOP     ; branch if not 0

          RTS

cAdr      ds       4
NumCoefs  ds       1
cAdrBgn   dc       i4'Coefs'
nCoefs    dc       i'3'        ; n = 3
Coefs     dc       h'00 00 00 00 00 00 00 80 FF 3F' ; c0 = 1
          dc       h'00 00 00 00 00 00 00 80 00 40' ; c1 = 2
          dc       h'00 00 00 00 00 00 00 00 00 00' ; c2 = 0
          dc       h'00 00 00 00 00 00 00 A0 01 C0' ; c3 = -5

xAdr      dc       i4'XVal'
XVal      ds       10          ; x

rAdr      dc       i4'Result'
Result    ds       10          ; r
```

65C816 example: scanning and formatting

The following example illustrates the use of the numeric scanner and formatter. The procedure accepts as an argument an ASCII string representing a number of degrees and returns the trigonometric sine of its argument as a numeric ASCII string. Both input and output are Pascal strings; that is, byte 0 gives the length, and byte 1 contains the first character in the string. The caller of the procedure pushes the address of the input string and executes a JSR instruction to location SINE. The procedure overwrites the input string with the result, whose length may be as large as 80, and clears the stack.

```
; Symbols:
;Str:      i/o string
;Index:    16-bit integer index
;Dec:      decimal record
;ValidP:   boolean for valid prefix
;Form:     decform record
;XTemp:    extended temporary variable
;XConst:   extended constant = pi/180

SINE      ENTRY

          PULLWORD   Return      ; save return adr
          PULLLONG   Str_Adr     ; adr of Str --> Str_Adr

          LDA        #1
          STA        Index      ; 1 --> Index

          PUSHLONG   Str_Adr
          PUSHLONG   #Index
          PUSHLONG   #Dec
          PUSHLONG   #ValidP
          FPSTR2DEC          ; Str --> Dec

          PUSHLONG   #Dec
          PUSHLONG   #XTemp
          FDEC2X          ; Dec --> XTemp

          PUSHLONG   #XConst
          PUSHLONG   #XTemp
          FMULX          ; convert to radians:
                       ; XTemp * XConst --> XTemp

          PUSHLONG   #XTemp
          FSINX          ; sin(XTemp) --> XTemp

          PUSHLONG   #Form
          PUSHLONG   #XTemp
          PUSHLONG   #Dec
          FX2DEC          ; XTemp --> Dec
```

```

PUSHLONG #Form
PUSHLONG #Dec
PUSHLONG Str_Adr
FDEC2STR          ; Dec --> Str

PUSHWORD Return

RTS

Index    ds      2
ValidP   ds      2
XConst   dc      h'AE C8 E9 94 12 35 FA 8E F9 3F' ; XConst = pi/180
XTemp    ds      10
Form     dc      i'1,10'      ; fixed-point format with 10 places
Dec      ds      33          ; sign, exp, length, ASCII = (2+2+1+28)
Return   ds      2
Str_Adr  ds      4

```

6502 examples

The following examples illustrate the use of the 6502 SANE engine.

6502 example: polynomial evaluation

This example evaluates the polynomial

$$x^3 + 2x^2 - 5$$

It illustrates the evaluation of a polynomial

$$c_0x^n + c_1x^{n-1} + \dots + c_n$$

using Horner's recurrence:

$$r \leftarrow c_0$$

$$r \leftarrow (r \times x) + c_j, \text{ for } j = 1 \text{ to } n$$

On entry, rAdr points to an extended result field, cAdrBgn points to the coefficient table of extended values, byte nCoefs contains the degree n (< 256) of the polynomial, and xAdr points to an extended function argument x . The coefficient table consists of $n + 1$ extended coefficients, starting with c_0 . In this example, $n = 3$, $c_0 = 1$, $c_1 = 2$, $c_2 = 0$, and $c_3 = -5$.

Entry is by a JSR instruction to POLYEVAL. (POLYEVAL calls FP6502.)

```

POLYEVAL
    LDA    cAdrBgn                ; address of c0->cAdr
    STA    cAdr
    LDA    cAdrBgn+1
    STA    cAdr+1
    LDA    nCoefs
    STA    NumCoefs

    PUSH   cAdr                    ; c0 -> r
    PUSH   rAdr
    FX2X

POLYLOOP
    PUSH   xAdr                    ; r*x -> r
    PUSH   rAdr
    FMULX

    CLC                             ; advance to next
                                ; coefficient

    LDA    cAdr
    ADC    #10.                    ; decimal 10
    STA    cAdr
    LDA    cAdr+1
    ADC    #0
    STA    cAdr+1

    PUSH   cAdr                    ; r+cj -> r
    PUSH   rAdr
    FADDX

    DEC    NumCoefs                ; decrement NumCoefs and
    BNE    POLYLOOP                ; branch if not 0

    RTS

cAdr      .WORD    cAdrBgn
NumCoefs  .BYTE    nCoefs

cAdrBgn   .WORD    Coefs
nCoefs    .BYTE    3                ; n = 3
Coefs     .WORD    00000,00000,00000,08000,03FFF ; c0 = 1
          .WORD    00000,00000,00000,08000,04000 ; c1 = 2
          .WORD    00000,00000,00000,00000,00000 ; c2 = 0
          .WORD    00000,00000,00000,0A000,0C001 ; c3 = -5

xAdr      .WORD    XVal
XVal      .BLOCK   10.                ; x

rAdr      .WORD    Result
Result    .BLOCK   10.                ; r

```

6502 example: scanning and formatting

This example illustrates the use of the numeric scanner and formatter. The procedure accepts as argument an ASCII string representing a number of degrees and returns the trigonometric sine of its argument as a numeric ASCII string. Both input and output are Pascal strings: the zeroth byte gives the length, and the first byte contains the first character in the string. The caller of the procedure pushes the address of the input string and executes a JSR instruction to location SINE. The procedure overwrites the input string with the result, whose length may be as large as 80, and clears the stack. Sine calls FP6502, Elms6502, and DecStr6502. The Sine routine could be declared by the following Pascal statement:

```
PROCEDURE Sine (VAR s : DecStr);
```

```
; Symbols:
```

```
; s: i/o string  
; i: 16-bit integer index  
; d: decimal record  
; v: boolean for valid prefix  
; f: decform record  
; x: extended temporary  
; c: extended constant = pi/180 ;
```

```
SINE
```

```
    POP      Return          ; save return address  
    POP      sAdr           ; address of s -> sAdr  
  
    LDA      #01            ; 1 -> i  
    STA      i  
    LDA      #00  
    STA      i+1  
  
    PUSH     sAdr           ; s -> d  
    PUSH     iAdr  
    PUSH     dAdr  
    PUSH     vAdr  
    FPSTR2DEC  
  
    PUSH     dAdr           ; d -> x  
    PUSH     xAdr  
    FDEC2X  
  
    PUSH     cAdr           ; convert to radians: x*c -> x  
    PUSH     xAdr  
    FMULX
```

```

        PUSH    xAdr                ; sin(x) -> x
        FSINX

        PUSH    fAdr                ; x -> d
        PUSH    xAdr
        PUSH    dAdr
        FX2DEC

        PUSH    fAdr                ; d -> s
        PUSH    dAdr
        PUSH    sAdr
        FDEC2STR

        PUSH    Return
        RTS

iAdr    .WORD    i
i        .WORD

vAdr    .WORD    v
v        .WORD

cAdr    .WORD    c
c        .WORD    0C8AE,94E9,3512,8EFA,3FF9; c = pi/180

xAdr    .WORD    x
x        .BLOCK 10.

fAdr    .WORD    f
f

style   .WORD    1                    ; fixed-point format
digits  .WORD    10.                  ; 10 digits after point

dAdr    .WORD    d

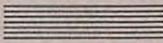
sgn      .WORD
exp      .WORD
sig      .BYTE
mark     .BLOCK 28.

Return  .WORD
sAdr    .WORD

```



Part III



The MC68000 Assembly-Language SANE Engine

The software described in Part III of this manual provides the features of the Standard Apple Numerics Environment (SANE) to assembly-language programmers using Apple's MC68000-based systems. SANE—described in detail in Part I—fully supports the IEEE Standard 754 for binary floating-point arithmetic, and augments the Standard to provide greater utility for applications in accounting, finance, science, and engineering. The IEEE Standard and SANE offer a combination of quality, predictability, and portability heretofore unknown for numerical software.

Part III of this manual describes the use of the assembly-language SANE engine for the MC68000, but does not describe SANE itself. For example, Part III explains how to call the SANE Remainder function from MC68000 assembly language, but does not discuss what this function does. See Part I for information about the semantics of SANE.

The MC68000 SANE engine is provided in all Apple MC68000-based systems; see Appendix B.



Chapter 19



MC68000 SANE Basics and Data Types

Programs using MC68000-based SANE engines use the same convention for making most calls: first push the parameters on the stack, then invoke the macro for the desired operation. The following code illustrates a typical invocation of the MC68000 SANE engine:

```
PEA    A_ADR    ; Push address of A (single format)
PEA    B_ADR    ; Push address of B (extended format)
FSUBS                ; Floating-point SUBtract Single:
                ; B <-- B - A
```

This example is typical of SANE engine calls, most of which pass operands by pushing the addresses of the operands onto the stack prior to invoking the operation. The form of the operation in the example ($B \leftarrow B - A$, where A is a numeric type and B is extended) is similar to the forms for most SANE operations. In this example, `FSUBS` is an assembly-language macro listed in Appendix E, “MC68000 SANE Quick Reference Guide.” Details of SANE engine calls are given later in this chapter, in the section “Calling Sequence.”

❖ *Note about macros:* The macro names used in this and succeeding chapters are those provided with the Macintosh® Programmer’s Workshop (MPW). For more information about the availability of SANE software and macros, please refer to Appendix B.

The SANE engine for the MC68000 occupies three software packages named `FP68K`, `Elems68K`, and `DecStr68K`. Access to all three is similar. Arithmetic operations, comparisons, conversions, environmental control, and halt control are in `FP68K`. The elementary functions are in `Elems68K`, and the SANE scanners and formatter are in `DecStr68K`. Chapters 20 through 23 describe the functions of `FP68K`. Chapters 24 and 25 describe the functions of `Elems68K` and `DecStr68K`, respectively.

Operation forms

The example above illustrates the form of an `FP68K` binary operation. Forms for other `FP68K` operations are described in this section. Examples and further details are given in subsequent chapters.

Arithmetic and auxiliary operations

Most numeric operations are either unary (one operand), like Square Root and Negate, or binary (two operands), like Add and Multiply.

The MC68000 assembly-language SANE engine, FP68K, provides unary operations in a one-address form:

$DST \leftarrow \langle op \rangle DST$ Example: $B \leftarrow \text{sqrt}(B)$

The operation $\langle op \rangle$ is applied to (or operates on) the operand DST , and the result is returned to DST , overwriting the previous value. DST stands for *destination operand*.

FP68K provides binary operations in a two-address form:

$DST \leftarrow DST \langle op \rangle SRC$ Example: $B \leftarrow B/A$

The operation $\langle op \rangle$ is applied to the operands DST and SRC , and the result is returned to DST , overwriting the previous value. SRC stands for *source operand*.

To store the result of an operation (unary or binary), the location of the operand DST must be available to FP68K, so DST is passed by address to FP68K. In general, all operands, source and destination, are passed by address to FP68K.

For most operations the storage format for a source operand (SRC) can be the 16-bit integer format, the 32-bit longint (long integer) format, or one of the SANE numeric formats (single, double, extended, or comp). To support the extended-based SANE arithmetic, a destination operand (DST) must be in the extended format.

The forms for the CopySign and Nextafter functions are unusual and are discussed in Chapter 20, "MC68000 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries."

Conversions

FP68K provides conversions between the extended format and other SANE formats, between extended and 16- or 32-bit integers, and between extended and decimal records. Conversions between binary formats (single, double, extended, comp, integer, and longint) and conversions from decimal to binary have the form

$DST \leftarrow SRC$

Conversions from binary to decimal have the form

$DST \leftarrow SRC$ according to $SRC2$

where $SRC2$ is a decform record specifying the decimal format for the conversion of SRC to DST .

Comparisons

Comparisons have the form

$\langle \text{relation} \rangle \leftarrow SRC$ compared with DST

where DST is extended and SRC is single, double, comp, extended, integer, or longint, and where $\langle \text{relation} \rangle$ is less, equal, greater, or unordered according as

$DST \langle \text{relation} \rangle SRC$

Here the result $\langle \text{relation} \rangle$ is indicated by setting the MC68000's CCR flags.

Other operations

FP68K provides inquiries for determining the class and sign of an operand and operations for accessing the floating-point Environment word and the halt address. Forms for these operations vary and are given as the operations are introduced.

External access

The SANE engine FP68K consists of position-independent code with a single entry point at its beginning. There is a static state area consisting of one word of mode bits and error flags, and a two-word halt vector.

FP68K preserves all MC68000 registers across invocations, except that the Remainder function modifies D0. FP68K modifies the MC68000's CCR flags. Except for binary-decimal conversions, it uses little more stack area than is required to save the sixteen 32-bit registers. Because the binary-decimal conversions themselves call FP68K (to perform multiplies and divides), they use about twice the stack space of the regular operations.

The access constraints described in this section also apply to Elems68K and DecStr68K, except that calls to DecStr68K do not preserve the contents of A0, A1, D0, and D1.

Calling sequence

A typical invocation of the engine consists of a sequence of PEA instructions to push operand addresses, followed by one of the macros listed in Appendix E:

```
PEA    <source address>
PEA    <destination address>
<fopmacro>
```

PEA instructions for source operands always precede those for destination operands, as shown in Figure 19-1. The macro call <fopmacro> represents a typical operation macro defined as

```
MOVE.W <opword>,-(SP)      ; Push op code.
    _FP68K
```

The macro call `_FP68K` expands to an A-line trap.

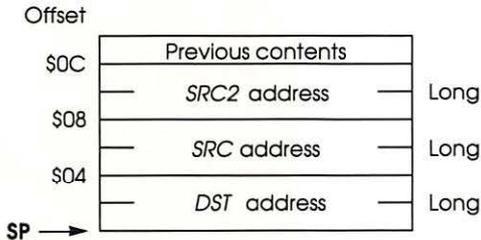


Figure 19-1
SANE operands on the MC68000 stack

The opword

The opword is the logical OR of an operand format code and an operation code.

The operand format code specifies the format (extended, double, single, integer, longint, or comp) of one of the operands. The operand format code typically gives the format for the source operand (*SRC*). At most one operand format need be specified, because other operands' formats are implied.

The operation code specifies the operation to be performed by FP68K. For example, the format code for single is \$1000. The operation code for divide is \$0006. Hence, the opword \$1006 indicates "divide by a value of type single."

Opwords, operand format codes, and operation codes are listed in Appendix E, "MC68000 SANE Quick Reference Guide."

Assembly-language macros

For most common <opword> calls to FP68K, the macros listed in Appendix E expand into the following form:

```
MOVE.W <opword>, -(SP)
      _FP68K
```

Example 1

Add a single-format operand *A* to an extended-format operand *B*.

```
PEA    A_ADR    ; Push address of A
PEA    B_ADR    ; Push address of B
FADDS                ; Floating-point ADD Single: B <-- B + A
```

Example 2

Compute $B \leftarrow \text{sqrt}(A)$, where *A* and *B* are extended. The value of *A* should be preserved.

```
PEA    A_ADR    ; Push address of A
PEA    B_ADR    ; Push address of B
FX2X                ; Floating-point eXtended to eXtended:
                    ; B <-- A
PEA    B_ADR    ; Push address of B
FSQRTX              ; Floating SQUare RooT eXtended:
                    ; B <-- sqrt(B)
```

Example 3

Compute $C \leftarrow A - B$, where *A*, *B*, and *C* are in the double format. Because destinations are extended, a temporary extended variable *T* is required.

```
PEA    A_ADR    ; Push address of A
PEA    T_ADR    ; Push address of 10-byte temporary
FD2X                ; Floating-point convert Double to eXtended:
                    ; T <-- A
PEA    B_ADR    ; Push address of B
PEA    T_ADR    ; Push address of temporary
FSUBD              ; Floating-point SUBtract Double: T <-- T - B
PEA    T_ADR    ; Push address of temporary
PEA    C_ADR    ; Push address of C
FX2D                ; Floating-point convert eXtended to Double:
                    ; C <-- T
```

MC68000 SANE data types

FP68K fully supports the SANE data types and the integer types shown in Table 19-1.

Table 19-1
MC68000 SANE data types

Name	Description
Single	32-bit floating-point
Double	64-bit floating-point
Comp	64-bit integer
Extended	80-bit floating-point
Integer	16-bit two's-complement integer
Longint	32-bit two's-complement integer

The MC68000 SANE engine uses the convention that least significant bytes are stored in high memory. For example, consider a variable of type single as shown in Table 19-2.

Table 19-2
Bits in a variable of type single

Name	Description
s	Sign
$e_0 \dots e_7$	Exponent (msb ... lsb)
$f_0 \dots f_{22}$	Significand fraction (msb ... lsb)

Figure 19-2 shows the logical structure of this 4-byte variable and the order of its bytes in memory. If this variable is assigned the address \$1000, then its bits are distributed to the locations \$1000 through \$1003 as shown.

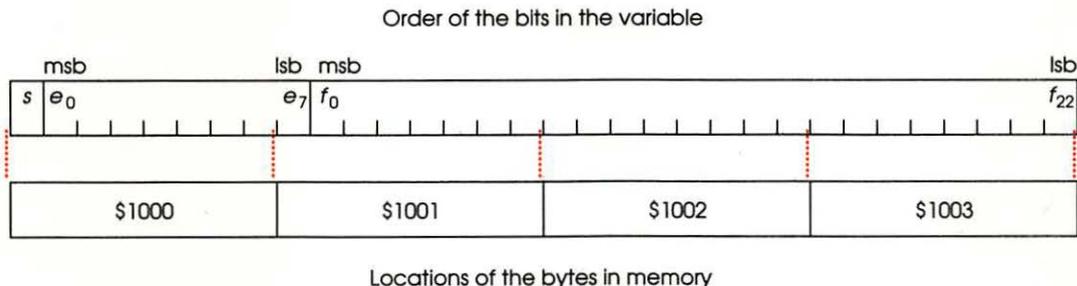


Figure 19-2
Memory format of a variable of type single

The other SANE formats are represented in memory in similar fashion. Please refer to Chapter 2, "SANE Data Types," for descriptions of the formats.



Chapter 20



MC68000 SANE Arithmetic and Auxiliary Operations, Comparisons, and Inquiries

The operations covered in this chapter follow the access schemes described in Chapter 19, “MC68000 SANE Basics and Data Types.”

Unary operations follow the one-address form:

$$DST \leftarrow \langle \text{op} \rangle DST$$

They use this calling sequence:

```
PEA    <DST address>
<fopmacro>
```

Binary operations follow the two-address form:

$$DST \leftarrow DST \langle \text{op} \rangle SRC$$

They use the following calling sequence:

```
PEA    <SRC address>
PEA    <DST address>
<fopmacro>
```

The destination operand (*DST*) for these operations is passed by address and is generally in the extended format. The source operand (*SRC*) is also passed by address and may be single, double, comp, extended, integer, or longint. Some operations are distinguished by requiring some specific type for *SRC*, by using a nonextended destination, or by returning auxiliary information in the D0 register and in the processor CCR status bits. In this section, operations so distinguished are noted. The examples employ the macros provided in the Macintosh Programmer’s Workshop.

Add, Subtract, Multiply, and Divide

These are binary operations and follow the two-address form.

Example

$B \leftarrow B/A$, where *A* is double and *B* is extended.

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FDIVD                ; divide with source operand of type double
```

Square Root

This is a unary operation and follows the one-address form.

Example

$B \leftarrow \text{sqrt}(B)$, where B is extended.

```
PEA    B_ADR    ; push address of B
FSQRTX                ; square root (operand is always extended)
```

Round-to-Integer and Truncate-to-Integer

These are unary operations and follow the one-address form.

Round-to-Integer rounds (according to the current rounding direction) to an integral value in the extended format. Truncate-to-Integer rounds toward zero (regardless of the current rounding direction) to an integral value in the extended format. The calling sequence is the usual one for unary operators, illustrated in the previous section for Square Root.

Remainder

This is a binary operation and follows the two-address form.

Remainder returns auxiliary information in D0.W: the seven low-order bits of $|n|$ (negated if n is negative). The high half of D0.L is undefined. This intrusion into the register file is extremely valuable in argument reduction—the principal use of the Remainder function. The state of D0 after an invalid remainder is undefined.

Example

$B \leftarrow B \text{ rem } A$, where A is single and B is extended.

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FREMS                ; remainder with source operand of type single
```

Logb and Scalb

Logb is a unary operation and follows the one-address form.

Scalb is a binary operation and follows the two-address form. Its source operand is a 16-bit integer.

Example

$B \leftarrow B \times 2^I$, where B is extended.

```
PEA    I_ADR    ; push address of I
PEA    B_ADR    ; push address of B
FSCALBX      ; scalb
```

Negate, Absolute Value, and CopySign

Negate and Absolute Value are unary operations and follow the one-address form.

CopySign uses the following two-address calling sequence to copy the sign of DST onto the sign of SRC :

```
PEA    <SRC address>
PEA    <DST address>
FCPYSGNX
```

❖ *Note:* The order of the operands in the SANE CopySign function is reversed from that suggested in IEEE Standard 754.

The formats of the operands of FCPYSGNX can be single, double, or extended. (For efficiency, an MC68000 assembly-language program should copy signs directly rather than calling FP68K.)

Example

Copy the sign of B (single, double, or extended) into the sign of A (single, double, or extended).

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FCPYSGNX      ; copy-sign
```

FP68K treats Negate, Absolute Value, and CopySign as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal invalid.

Nextafter

Both source and destination operands must be of the same floating-point type (single, double, or extended). The Nextafter operations use the following calling sequence:

```
PEA    <SRC address>
PEA    <DST address>
<Nextafter macro>
```

They perform $SRC \leftarrow$ next value, in the format indicated by the macro, after SRC in the direction of DST .

Important

The Nextafter operations differ from most two-address operations in that they change the SRC values rather than the DST values.

Example

$A \leftarrow$ Nextafter(A) in the direction of B , where A and B are double (so *next-after* means *next-double-after*).

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FNEXTD                ; next-after in double format
```

Comparisons

FP68K provides two comparison operations: $FCPX$ (which signals invalid if its operands compare unordered) and $FCMP$ (which does not). Each compares a source operand (which may be single, double, extended, comp, integer, or longint) with a destination operand (which must be extended). The result of a comparison is the relation (less, greater, equal, or unordered) for which

$DST <relation> SRC$

is true. The result is delivered in the X, N, Z, V, and C status bits, as shown in Table 20-1.

Table 20-1
Results of comparisons

Result	Status bit				
	X	N	Z	V	C
Greater	0	0	0	0	0
Less	1	1	0	0	1
Equal	0	0	1	0	0
Unordered	0	0	0	1	0

These status-bit encodings reflect that floating-point comparisons have four possible results, unlike the more familiar integer comparisons with three possible results. You need not learn these encodings, however; simply use the `FBxxx` series of macros for branching after `FCMP` and `FCPX`.

`FCMP` and `FCPX` are both provided to facilitate implementation of relational operators defined by higher-level languages that do not contemplate unordered comparisons. The IEEE Standard specifies that the invalid exception shall be signaled whenever necessary to alert users of such languages that an unordered comparison may have adversely affected their program's logic.

Example 1

Test $B \leq A$, where B is extended and A is single; if true, branch to `LOC`; signal if unordered.

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FCPXS                ; compare using source of type single,
                    ; signal invalid if unordered
FBLES  LOC      ; branch if B <= A
```

Example 2

Test B not-equal A , where B is extended and A is double; if true, branch to `LOC`. (Note that not-equal is equivalent to less, greater, or unordered, so invalid should not be signaled on unordered.)

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FCMPD                ; compare using source of type double,
                    ; do not signal invalid if unordered
FBNES  LOC      ; branch if B not-equal A
```

❖ *Note about macros:* Like the other macros in Part III, the floating-point branch-control macros `FBLES` and `FBNES` are provided with MPW.

Inquiries

The classify operation provides both class and sign inquiries. This operation takes one source operand (single, double, comp, or extended), which is passed by address, and places the result in a 16-bit integer destination.

The sign of the result is the sign of the source; the magnitude of the result gives the class of the operand, as shown in Table 20-2.

Table 20-2
Operand classes

Value	Class
1	Signaling NaN
2	Quiet NaN
3	Infinity
4	Zero
5	Normalized
6	Denormalized

Example

Set *C* to the sign and class of *A*.

```
PEA    A_ADR    ; push address of A
PEA    C_ADR    ; push address of result
FCLASS    ; classify single
```



Chapter 21



Conversions in MC68000 SANE

This chapter discusses conversions between binary formats and conversions between binary and decimal formats. Conversions between decimal formats provided by DecStr68K are discussed in Chapter 25, “MC68000 SANE Scanners and Formatter.”

Conversions between binary formats

FP68K provides conversions between the extended type and the SANE types single, double, and comp, as well as the 16- and 32-bit integer types.

Conversions to extended

FP68K provides conversions of a source—of type single, double, comp, extended, integer, or longint—to an extended destination.

The MC68000 SANE engine provides conversions of a source with format single, double, comp, extended, integer, or longint, to a destination in extended format, as shown in Table 21-1.

Table 21-1
Conversions to extended format

Function name	Type of conversion
FS2X	extended ← single
FD2X	extended ← double
FC2X	extended ← comp
FX2X	extended ← extended
FI2X	extended ← integer
FL2X	extended ← longint

All operands, even integer ones, are passed by address. The following example illustrates the calling sequence.

Example

Convert *A* to *B*, where *A* is of type comp and *B* is extended.

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FC2X                      ; convert comp to extended
```

Conversions from extended

The MC68000 SANE engine provides conversions of an extended source to a destination of type single, double, comp, extended, integer, or longint, as shown in Table 21-2.

Table 21-2
Conversions from extended format

Function name	Type of conversion
FX2S	single ← extended
FX2D	double ← extended
FX2C	comp ← extended
FX2X	extended ← extended
FX2I	integer ← extended
FX2L	longint ← extended

Note that conversion to a narrower format may alter values. Contrary to the usual scheme, the destination for these conversions need not be of type extended. All operands are passed by address. The following example illustrates the calling sequence.

Example

Convert *A* to *B*, where *A* is extended and *B* is double.

```
PEA    A_ADR    ; push address of A
PEA    B_ADR    ; push address of B
FX2D                      ; convert extended to double
```

Binary-decimal conversions

FP68K provides conversions between the binary types (single, double, comp, extended, integer, and longint) and the decimal record type.

Decimal records and decform records (used to specify the form of decimal representations) are described in Chapter 3, "Conversions in SANE." For FP68K, the maximum length of the `sig` digit-string of a decimal record is 20. (The value 20 is specific to this implementation; algorithms intended to port to other SANE implementations should use no more than 18 digits in `sig`.) Because decimal records contain an odd number of bytes (25), you may need to append an unused byte to preserve word alignment. The values of `style` fields of decform records and of `sgn` fields of decimal records are stored in the high-order byte of their word.

Binary to decimal

The calling sequence for a conversion from a binary format to a decimal record passes the address of a decform record, the address of a binary source operand, and the address of a decimal-record destination. The maximum number of significant digits that will be returned is 19.

Example

Convert a comp-format value *A* to a decimal record *D* according to the decform record *F*.

```
PEA    F_ADR    ; push address of F
PEA    A_ADR    ; push address of A
PEA    D_ADR    ; push address of D
FC2DEC                ; convert comp to decimal
```

Fixed-format overflow

If a number is too large for a chosen fixed style, then FP68K returns the string '?' in the *sig* field of the decimal record.

Decimal to binary

The calling sequence for a conversion from decimal to binary passes the address of a decimal-record source operand and the address of a binary destination operand.

The maximum number of significant digits in *sig* is 19. The presence of a nonzero 20th digit represents one or more additional nonzero digits after the 19th. This binary information in the 20th digit is used by FP68K only for rounding. The exponent corresponds to the 19-digit integer represented by the first 19 digits of *sig*.

Example

Convert the decimal record *D* to a double-format value *B*.

```
PEA    D_ADR    ; push address of D
PEA    B_ADR    ; push address of B
FDEC2D                ; convert decimal to double
```

Techniques for maximum accuracy

The following techniques apply to FP68K; other SANE implementations require other techniques.

For maximum accuracy, delete trailing zeros from the `sig` field of a decimal record in order to minimize the magnitude of the `exp` field. For example, for 300E-43 set `sig` to '3' and `exp` to -41.

If you are writing a parser and must handle a number with more than 19 significant digits, follow these rules:

- Place the implicit decimal point to the right of the 19 most significant digits.
- If any of the discarded digits to the right of the implicit decimal point is nonzero, then concatenate the digit '1' to `sig`.



Chapter 22



Controlling the MC68000 SANE Environment

The Environment word

The floating-point environment is encoded in the 16-bit integer format, as shown in Figure 22-1. Table 22-1 gives the hexadecimal value of each of the environment flags. Rounding direction and precision are stored as 2-bit encoded values; exception and halt-enabled flags are set as individual bits. Note that the default environment is represented by the integer value zero.

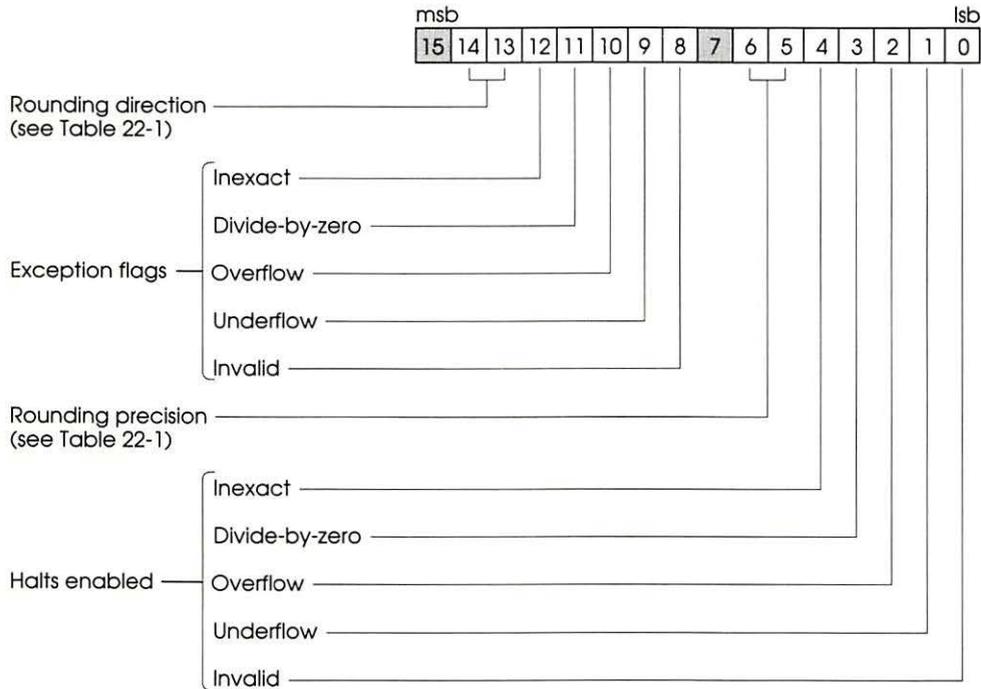


Figure 22-1
The Environment word for the MC68000

Table 22-1
Bits in the Environment word for the MC68000

Group name	Mask bits	Mask value	Description
Rounding direction (Bit group \$6000)	14 13 0 0 0 1 1 0 1 1	\$0000 \$2000 \$4000 \$6000	To-nearest Upward Downward Toward-zero
Exception flags (Bit group \$1F00)	12 11 10 9 8 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$1000 \$0800 \$0400 \$0200 \$0100	Inexact Divide-by-zero Overflow Underflow Invalid
Rounding precision (Bit group \$0060)	6 5 0 0 0 1 1 0 1 1	\$0000 \$0020 \$0040 \$0060	Extended Double Single (Undefined)
Halts enabled (Bit group \$001F)	4 3 2 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$0010 \$0008 \$0004 \$0002 \$0001	Inexact Divide-by-zero Overflow Underflow Invalid

Note: Bits 7 and 15 are not used.

Example

With rounding toward-zero, inexact and underflow exception flags raised, extended rounding precision, and halt on invalid, overflow, and division-by-zero, the most significant byte of the Environment word is \$72 and the least significant byte is \$0D.

Access to the environment settings is via the procedures Get-Environment, Set-Environment, Test-Exception, Set-Exception, Procedure-Entry, and Procedure-Exit.

Get-Environment and Set-Environment

Get-Environment takes one input operand: the address of a 16-bit integer destination. The Environment word is returned in the destination.

Set-Environment has one input operand: the address of a 16-bit integer, which is to be interpreted as an Environment word.

Example

Set rounding direction to downward.

```
PEA    A_ADR
FGETENV
LEA    A_ADR,A0      ; A0 gets address of A
MOVE.W (A0),D0      ; D0 gets environment
AND.W  #$9FFF,D0    ; clear bits 6000
OR.W   #$4000,D0    ; set rounding downward
MOVE.W D0,(A0)      ; restore A
PEA    A_ADR
FSETENV
```

Test-Exception and Set-Exception

Test-Exception takes one operand: the address of a 16-bit integer destination. On input the destination contains a bit index, as shown in Table 22-2.

Table 22-2
Bits in the Exception word

Bit index	Description
0	Invalid
1	Underflow
2	Overflow
3	Divide-by-zero
4	Inexact

If the corresponding exception flag is set, then Test-Exception returns the value 1 in the high byte of the destination; otherwise, it returns 0.

Example

Branch to XLOC if underflow is set.

```
MOVE.W    #FBUFLOW, -(SP)    ; underflow bit index
PEA       (SP)
FTESTXCP
TST.B     (SP)+               ; test byte, pop word
BNE       XLOC
```

Set-Exception takes one source operand, the address of a 16-bit integer that encodes an exception in the manner described above for Test-Exception. Set-Exception stimulates the indicated exception.

Procedure-Entry and Procedure-Exit

Procedure-Entry saves the current floating-point environment (16-bit integer) at the address passed as the sole operand, and sets the operative environment to the default state.

Procedure-Exit saves (temporarily) the exception flags, sets the environment passed as the sole operand, and then stimulates the saved exceptions.

Example

Here is a procedure that appears to its callers as an atomic operation:

```
ATOMICPROC
PEA       E_ADR    ; push address to store environment
FPROCENTRY    ; procedure entry

...body of routine goes here...

PEA       E_ADR    ; push address of stored environment
FPROCEXIT    ; procedure exit
RTS
```



Chapter 23



Halts in MC68000 SANE

The software package FP68K lets you transfer program control when selected floating-point exceptions occur. Because this facility is used to implement halts in high-level languages, it is referred to as a halt mechanism. An assembly-language program can include a halt handler routine to cause special actions for floating-point exceptions. The FP68K halt mechanism differs from the traps that are an optional part of the IEEE Standard.

Conditions for a halt

Any floating-point exception will, if the corresponding halt is enabled, trigger a halt. The halt for a particular exception is enabled when the user has set the halt-enable bit corresponding to that exception.

When enabled, the FP68K halt-on-underflow occurs when the result is both tiny and inexact.

The halt mechanism

If the halt for a given exception is enabled, FP68K does these things when that exception occurs:

1. FP68K delivers the same result to the destination address that it would return if the halt were not enabled.
2. It sets up the stack frame shown in Figure 23-1:

The first word of the record `MISC` contains in its five low-order bits the AND of the halt-enable bits with the exceptions that occurred in the operation just completing. If halts were not enabled, then (upon return from FP68K) `CCR` and `D0` would have the values given in `MISC`.

3. It passes control by a `JSR` instruction through the halt vector previously set by `FSETHV`, pushing another long word containing a return address in FP68K. If execution is to continue, the halt procedure must clear 18 bytes from the stack to remove the `opword` and the `DST`, `SRC`, `SRC2`, and `MISC` addresses.

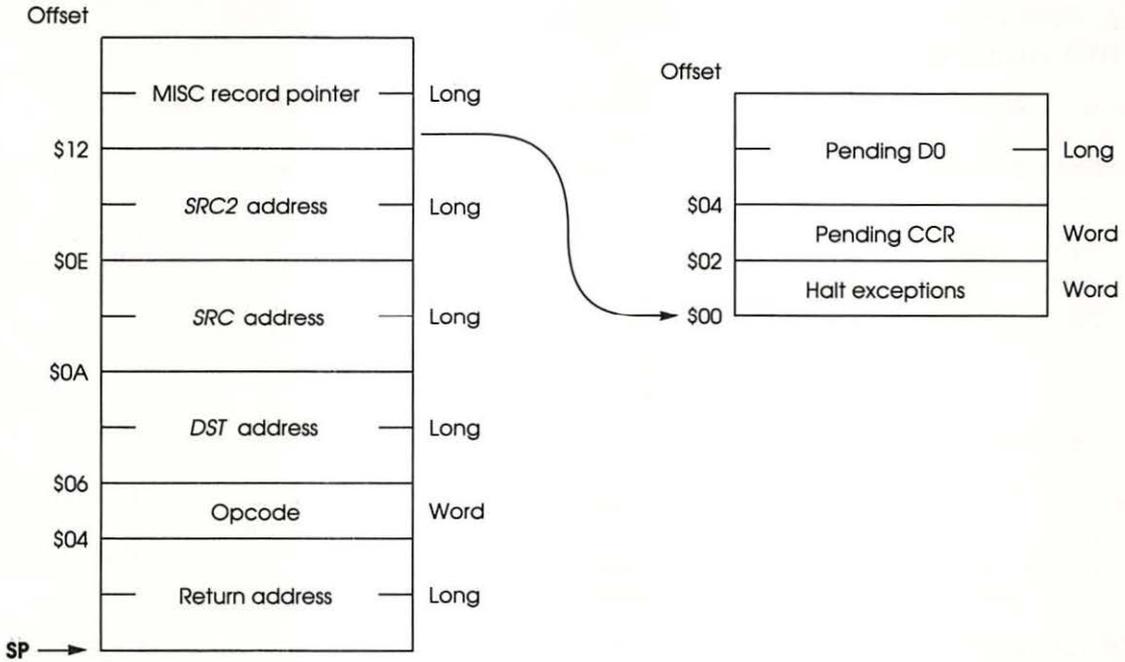


Figure 23-1
Stack frame for halt

Set-Halt-Vector has one input operand: the address of a 32-bit integer, which is interpreted as the halt vector (that is, the address to transfer control to in case a halt occurs).

Get-Halt-Vector has one input operand: the address of a 32-bit integer, which receives the halt vector.

Using the halt mechanism

This example illustrates the use of the halt mechanism. The program must set the halt vector to the starting address of a halt handler routine. This particular halt handler returns control to FP68K, which will continue as if no halt had occurred, returning to the next instruction in the program.

```
        LEA      HROUTINE,A0    ; A0 gets address of halt routine
        MOVE.L  A0,H_ADR       ; H_ADR gets same
        PEA     H_ADR
        FSETHV                      ; set halt vector to HROUTINE
;
        . . .
        PEA     ; floating-point operand here
        <FOPMACRO>                ; a floating-point call here
;
HROUTINE                      ; called by FP68K
        MOVE.L  (SP)+,A0        ; A0 saves return address in FP68K
        ADDA.W  #18,SP          ; increment stack past arguments
        JMP     (A0)            ; return to FP68K
```

The FP68K halt mechanism is designed so that a halt procedure can be written in Pascal (assuming MPW Pascal calling conventions). This is the form of a Pascal equivalent to HROUTINE:

```
type    miscrec = record
        halterrors: integer ;
        ccrpending: integer ;
        D0pending: longint ;
    end {record} ;

procedure haltroutine
    ( var misc: miscrec ;
      src2, src, dst: longint ;
      opcode: integer ) ;
begin {haltroutine}
end {haltroutine} ;
```

Like HROUTINE, haltroutine merely continues execution as if no halt had occurred.



Chapter 24



Elementary Functions in MC68000 SANE

The elementary functions that are specified by the Standard Apple Numerics Environment are made available to MC68000 assembly-language programs by Elems68K. Elems68K also includes two functions that compute $\log_2(1 + x)$ and $2^x - 1$ accurately. Elems68K makes calls to FP68K for its basic arithmetic. The access schemes for FP68K (described in Chapter 19) and Elems68K are similar. The examples that follow use macros provided with the Macintosh Programmer's Workshop. Opwords are listed in Appendix E, "MC68000 SANE Quick Reference Guide."

One-argument functions

The following SANE elementary functions have one extended argument, passed by address:

- Log2(x) computes the base-2 logarithm of x .
- Ln(x) computes the natural logarithm of x .
- Ln1(x) computes the natural logarithm of $(1 + x)$.
- Exp2(x) computes 2^x .
- Exp(x) computes e^x .
- Exp1(x) computes $e^x - 1$.
- Cos(x) computes the cosine of x .
- Sin(x) computes the sine of x .
- Tan(x) computes the tangent of x .
- Atan(x) computes arctangent of x .
- RandomX(x) computes a pseudorandom value with x as seed.
- Log21(x) computes $\log_2(1 + x)$.
- Exp21(x) computes $2^x - 1$.

The operation syntax is

$DST \leftarrow \langle op \rangle DST$

These functions use the following one-address calling sequence:

```
PEA    <DST>
<fopmacro>
```

<fopmacro> is one of the macros that generate code to push an opword and invoke Elems68K. Those macros are listed in Appendix E, "MC68000 SANE Quick Reference Guide." The calling sequence follows the FP68K access scheme for unary operations, such as Square Root and Negate.

Example

$B \leftarrow \sin(B)$, where B is of extended type.

```
PEA    B_ADR    ; push address of B
FSINX                ; B <-- sin(B)
```

Two-argument functions

The following SANE elementary functions have two-extended arguments, passed by address:

- $\text{XPwrY}(x,y)$ computes x^y .
- $\text{XPwrI}(x,i)$ computes x^i .

General exponentiation (XPwrY) has two extended arguments, both passed by address. The result is returned in x .

Integer exponentiation (XPwrI) also has two arguments. The extended argument x , passed by address, receives the result. The 16-bit integer argument i is also passed by address.

Both exponentiation functions use the following calling sequence for binary operations:

```
PEA    <SRC address>    ; push exponent address first
PEA    <DST address>    ; push base address second
<fopmacro>
```

The operations compute

$DST \leftarrow DST^{SRC}$

Example

$B \leftarrow B^K$, where the type of B is extended and K is a 16-bit integer.

```
PEA    K_ADR    ; push address of K
PEA    B_ADR    ; push address of B
FXPWRI                ; integer exponentiation
```

Three-argument functions

Compound and Annuity use the following calling sequence:

```
PEA    <SRC2 address>    ; push address of rate first
PEA    <SRC address>     ; push address of number of
                          ;   periods second
PEA    <DST address>     ; push address of destination third
<fopmacro>
```

The operations compute

$$DST \leftarrow \langle op \rangle (SRC2, SRC)$$

where $\langle op \rangle$ is compound or annuity, $SRC2$ is the rate, and SRC is the number of periods. All arguments $SRC2$, SRC , and DST must be of the extended type.

Example

$C \leftarrow (1 + R)^N$, where C , R , and N are of type extended.

```
PEA    R_ADR    ; push address of R
PEA    N_ADR    ; push address of N
PEA    C_ADR    ; push address of C
FCOMPOUND      ; compound
```



Chapter 25



MC68000 SANE Scanners and Formatter

The Standard Apple Numerics Environment specifies conversions between decimal strings and decimal records. These scanning and formatting routines are contained in DecStr68K. DecStr68K is designed for use with FP68K, which provides binary-decimal conversions between decimal records and the SANE data formats. Thus, with DecStr68K and FP68K the application developer has a solution to the problems of scanning input strings to produce SANE-type values and of formatting SANE-type values for output.

Opwords are listed in Appendix E, "MC68000 SANE Quick Reference Guide."

DecStr68K uses an access scheme similar to that of FP68K. (See Chapter 19, "MC68000 SANE Basics and Data Types.") DecStr68K removes its arguments from the stack before returning.

Numeric scanners

The DecStr68K numeric scanners are called Pstr2dec (*P* for *Pascal* programming language) and Cstr2dec (*C* for the *C* programming language). Both scanners take four arguments, all passed by address:

- string to be scanned
- 16-bit integer index into string
- decimal record for result
- 8-bit Boolean for valid-prefix indication on output

The index on input indicates where scanning is to begin, and on output is one greater than the index of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed and returned in the decimal record. If no numeric substring is recognized, then the index remains unchanged. The valid-prefix parameter on output contains 1 if the entire input string beginning at the input index is a valid numeric string or a prefix of a valid numeric string. It contains 0 otherwise.

The only difference between the scanners is in the string input argument. Pstr2dec expects a Pascal string: the string length in the zeroth byte of the string and the initial character of the string in the first byte (index = 1). Cstr2dec expects a C string: the initial character of the string in the zeroth byte (index = 0), no length byte, and termination of the string by a null character (ASCII code 0). Cstr2dec can be used to scan numbers embedded in large text buffers.

The scanners use the following calling sequence:

```
PEA          <address of string>
PEA          <address of index>
PEA          <address of decimal record>
PEA          <address of valid-prefix>
FCSTR2DEC                                ; or FPSTR2DEC
```

Numeric formatter

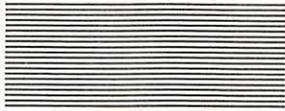
The formatter Dec2Str takes three arguments, all passed by address:

- decform record for formatting specification
- decimal record to be formatted
- string for result

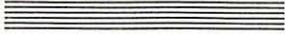
This routine returns a decimal string representing the input value from the decimal record, formatted according to input specifications passed in the decform record. The result string is a Pascal string: the zeroth byte contains the string length, and the first byte contains the first character in the string. For a full description of Dec2Str, see the section “Conversion From Decimal Records to Decimal Strings” in Chapter 3.

The formatter uses the following calling sequence:

```
PEA          <address of decform record>
PEA          <address of decimal record>
PEA          <address of string>
FDEC2STR
```



Chapter 26



Examples: Using the MC68000 SANE Engine

The following examples illustrate the use of the 68000 SANE engine. In the comments, &x means the address of x; STACK: x < y < z means x is on the top-of-stack, y is next deeper in the stack, and z is next deeper after y.

Example: polynomial evaluation

This example, taken verbatim from Elems68K source code, illustrates the evaluation of a polynomial

$$c_0x^n + c_1x^{n-1} + \dots + c_n$$

using Horner's recurrence:

$$res \leftarrow c_0$$

$$res \leftarrow res \times x + c_j, \text{ for } j = 1 \text{ to } n$$

On entry, A0 points to an extended result field, A1 points to the coefficient table of extended values, and A2 points to the extended function argument *x*. The coefficient table consists of a leading word that is a positive integer *n* giving the degree of the polynomial, and then *n* + 1 extended coefficients, starting with *c*₀. For example, for the polynomial $x^3 + 2x^2 - 5$, A1 points to

```
.WORD    3                ; n = 3
.WORD    $3FFF,$8000,$0000,$0000,$0000    ; c0 = 1
.WORD    $4000,$8000,$0000,$0000,$0000    ; c1 = 2
.WORD    $0000,$0000,$0000,$0000,$0000    ; c2 = 0
.WORD    $C001,$A000,$0000,$0000,$0000    ; c3 = -5
```

Entry is by a JSR instruction to location POLYEVAL.

POLYEVAL

```
MOVE.W   (A1)+,D0        ; n->D0, &c0->A1
PEA      (A1)            ; STACK: &c0 < &ret
PEA      (A0)            ; STACK: &res < &c0 < &ret
FX2X                                          ; c0->res, STACK: &ret
```

POLYLOOP

```
PEA      (A2)            ; STACK: &x < &ret
PEA      (A0)            ; STACK: &res < &x < &ret
FMULX                                         ; res*x->res, STACK: &ret
ADD.W    #10,A1          ; advance to next coef
PEA      (A1)            ; STACK: &cj < &ret
PEA      (A0)            ; STACK: &res < &cj < &ret
FADDX                                         ; res+cj->res, STACK: &ret
SUBQ.W   #1,D0           ; decrement loop counter
BGT.S    POLYLOOP       ; and branch if > 0
RTS
```

Example: language interface

This example illustrates the kind of code required to implement a high-level language interface to the MC68000 SANE engine. This example implements the Pascal function `Scalb`:

```
FUNCTION Scalb(n: integer; x: extended): extended;
```

The calling routine performs instructions that

- push a 4-byte pointer to a 10-byte space for the return value
- push a 2-byte value for n
- push the 4-byte address of the 10-byte value x
- execute a JSR to `Scalb`

Thus, on entry, the stack contains $\&ret < \&x < n < \&res < \dots$. The called routine clears the stack back to the result pointer.

SCALB

```
MOVEM.L    (SP)+,D0/A0    ; &ret->D0, &x->A0,
                                ; STACK: n < &res
PEA        (SP)           ; STACK: &n < n < &res
MOVEA.L    6(SP),A1       ; &res->A1
PEA        (A1)           ; STACK: &res < &n < n <
                                ; &res

MOVE.L     (A0)+, (A1)+   ; x->res (10 bytes)
MOVE.L     (A0)+, (A1)+
MOVE.W     (A0), (A1)
FSCALBX                                ; Scalb(n,x)->res,
                                ; STACK: n < &res

ADDQ.L     #2,SP          ; STACK: &res
MOVEA.L    D0,A0         ; &ret->A0
JMP        (A0)
```

Example: scanning and formatting

This example illustrates the use of the numeric scanner and formatter. It accepts as its argument an ASCII string representing a number of degrees and returns the trigonometric sine of its argument as a numeric ASCII string. Both input and output are Pascal strings: the zeroth byte gives the length; the first byte contains the first character in the string. The caller of the procedure pushes the address of the input string and executes a JSR instruction to location `SINE`. The procedure overwrites the input string with the result, whose length may be as large as 80, and clears the stack. The routine `Sine` could be declared in Pascal by the following statement:

```
PROCEDURE Sine(VAR s: DecStr);
```

```

; Offsets from A6 for i/o pointer and temporaries -
sOff    .EQU    8            ; s: i/o string
iOff    .EQU    -2           ; i: 16-bit integer index
dOff    .EQU    iOff-26     ; d: decimal record
                        ; note extra word-alignment
                        ; byte
vOff    .EQU    dOff-2      ; v: 16-bit boolean valid prefix
                        ; note extra word-alignment
                        ; byte
xOff    .EQU    vOff-10     ; x: extended
SINE    LINK    A6,#xOff    ; STACK: x < v < d < i < A6
                        ; < &ret < &s

MOVE.W   #1,iOff(A6)       ; 1 -> i
MOVE.L   sOff(A6),-(A7)    ; push &s
PEA     iOff(A6)           ; push &i
PEA     dOff(A6)           ; push &d
PEA     vOff(A6)           ; push &v
FPSTR2DEC                ; s->d

PEA     dOff(A6)           ; push &d
PEA     xOff(A6)           ; push &x
FDEC2X                ; d->x

PEA     Pi180              ; push &(pi/180)
PEA     xOff(A6)           ; push &x
FMULX                ; convert to radians:
                        ; x*(pi/180)->x

PEA     xOff(A6)           ; push &x
FSINX                ; sin(x)->x

PEA     DecForm            ; push &DecForm
PEA     xOff(A6)           ; push &x
PEA     dOff(A6)           ; push &d
FX2DEC                ; x->d

PEA     DecForm            ; push &DecForm
PEA     dOff(A6)           ; push &d
MOVE.L   sOff(A6),-(A7)    ; push &s
FDEC2STR                ; d->s

UNLK    A6                 ; STACK: &ret < &s <
MOVE.L   (SP)+,(SP)        ; STACK: &ret <
RTS

Pi180   .WORD    $3FF9,$8EFA,$3512,$94E9,$C8AE ; pi/180
DecForm .WORD    $0100,$000A ; style = fixed,
                        ; digits = 10

```



Part IV



Using the MC68881 SANE Engine

Part IV is a **delta guide** to the SANE implementation for Apple Macintosh computers that use the MC68020 microprocessor and the MC68881 coprocessor. It describes the important differences between the floating-point software packages for the MC68000 and the floating-point hardware embodied in the MC68881.

For a complete description of the operation of the MC68000 SANE packages, please refer to Part III of this book. For complete information about the Motorola MC68881, please refer to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.



Chapter 27



About the MC68881 and SANE

This chapter describes the differences between two methods of performing SANE floating-point arithmetic on Macintosh models equipped with the Motorola MC68881 floating-point coprocessor. One method involves making calls directly to the MC68881; the other involves calls to the floating-point packages, which themselves use the MC68881 for much of their internal processing.

The floating-point packages built into Macintosh models that use the MC68020 and MC68881 are functionally identical to the floating-point packages for the MC68000.

SANE implementations on the Macintosh

Across the Macintosh model line there are three different implementations of SANE:

- SANE software: floating-point packages for the MC68000
- SANE hybrid: floating-point packages for the MC68020 and MC68881
- SANE hardware: the MC68881 called directly by applications

All three implementations support Apple's approach to IEEE Standard numerics, which is characterized by the following features:

- extended format as a user type
- expression evaluation in extended format
- full conformity to the IEEE Standard 754 for floating-point arithmetic
- application access to exceptions and exception handling

On the MC68000-equipped Macintosh models, SANE is implemented as a set of software packages. Macintosh models equipped with the MC68020 and MC68881 have a different (but functionally identical) set of SANE packages that take advantage of the MC68881 floating-point processor to obtain improved performance while maintaining object-code compatibility with applications written for the earlier models. Applications that require still more speed can access the MC68881 directly, at the expense of compatibility; those applications will not run on machines that don't have the MC68881.

SANE software for the MC68881

This section describes three ways of using SANE with the MC68881:

- software package calls only
- MC68881 calls for fundamental operations; package calls for transcendental operations and a few others
- MC68881 calls for fundamental and transcendental operations; package calls for a few other operations

All calls to software packages

On Macintosh models that use the MC68020 processor and the MC68881 floating-point coprocessor, the numerics packages have been written to exploit the power of the MC68881 while maintaining complete compatibility with arithmetic on other models of the Macintosh. Programs that call the SANE packages run on both kinds of machines and obtain identical results; the only difference is that they run faster on machines that use the MC68881. Depending on the operations performed, the SANE packages that exploit the MC68881 perform floating-point operations 5 to 50 times as fast as the packages on a Macintosh Plus, with an average speed improvement of about 10 times.

Arithmetic on the MC68881 floating-point coprocessor conforms to IEEE Standard 754. For the fundamental operations—arithmetic operations (+, −, *, /, Square Root, and Remainder), comparisons, and binary-to-binary conversions—the MC68881 obtains results that are bit-for-bit identical with those obtained by the software SANE packages. For the transcendental functions, results differ in the last few bits.

The hybrid SANE packages use the MC68881 to provide fundamental operations. The hybrid packages also use those operations to provide fast transcendentals with results identical to those of the software SANE packages. Figure 27-1 illustrates this idea. (Chapter 28, “Functions of the MC68881 and SANE Software,” gives specific information about the differences between the SANE software packages and the MC68881.)

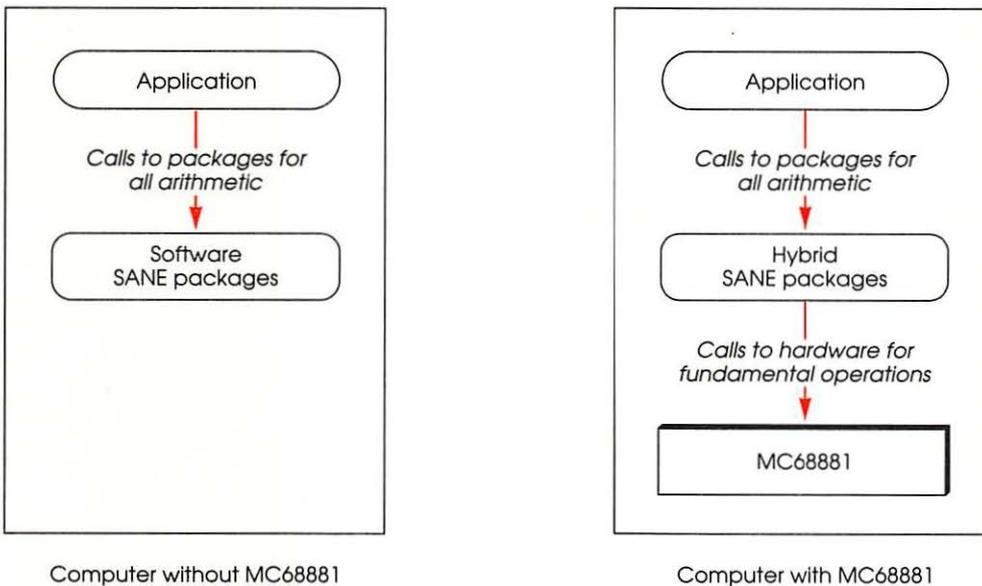


Figure 27-1
Application calling packages for all arithmetic

Fundamental operations on the MC68881

A program that doesn't need to run on all models of the Macintosh can obtain performance benefits by calling the MC68881 directly. Depending on the operations performed, such a program will perform floating-point operations 40 to 700 times as fast as on a Macintosh Plus, with an average speed improvement of about 100 times.

A program that does not require the speed of the MC68881 transcendental functions should call the MC68881 only for fundamental operations (arithmetic operations, comparisons, and conversions between binary formats) and rely on the packages for other operations. Figure 27-2 illustrates this idea.

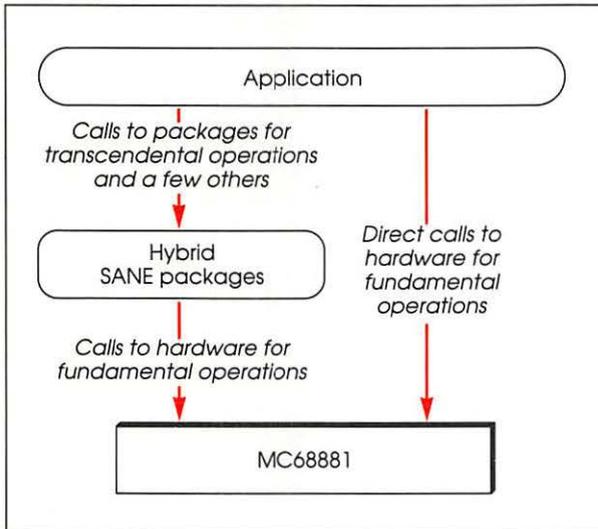


Figure 27-2
Application calling the MC68881 for fundamental operations

Transcendental operations on the MC68881

To further improve the speed of transcendental operations at the cost of a slight decrease in accuracy, programs can call the MC68881 for both fundamental and transcendental operations, as shown in Figure 27-3. Such programs still need to make calls to the packages for functions the MC68881 doesn't perform, such as conversions involving decimal records, financial functions, and certain logarithmic and exponential functions. All such functions are discussed in Chapter 28, "Functions of the MC68881 and SANE Software."

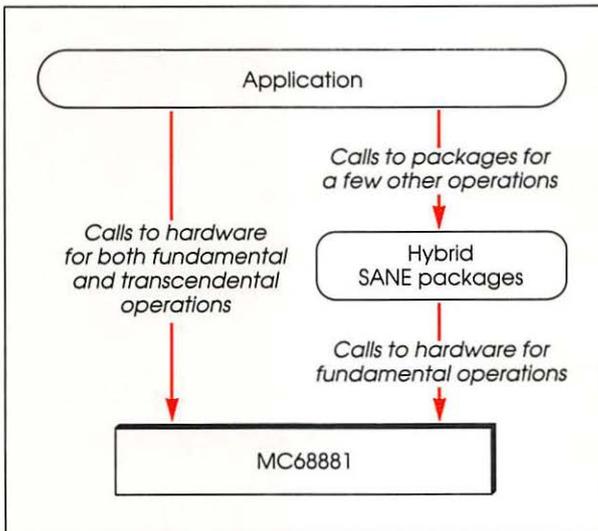


Figure 27-3
Application calling the MC68881 for all floating-point arithmetic

- ❖ *High-level language note:* The MPW Pascal and C compilers and libraries include options for specifying whether the MC68881 is to be called directly for fundamental operations and whether the MC68881 is to be called directly for transcendental functions as well. Normally, source code need not be changed. For more information, please refer to the reference manual for the language you are using.

Warning

The decision to make direct calls to the MC68881 should not be made lightly. The resulting program will not run on machines that don't have an MC68881. Furthermore, calling the MC68881 for other than the fundamental operations gives less accurate results. For more information, see the section "Accuracy of the MC68881's Elementary Functions" in Chapter 28.

Calls to SANE and calls to the MC68881

❖ *High-level language note:* This section deals with the syntax of assembly-language calls. Programmers who use high-level languages need not be concerned with the differences between calls to SANE packages and calls to the MC68881, because the compilers take care of all that.

The numerics packages for the MC68881 are called FP881 (pack4) and Elems881 (pack5). The syntax for assembly-language calls is the same as it is for calls to FP68K and Elems68K, the numerics packages for the MC68000. For example, to add $d + e$ with the result in e , with d in double format and e in extended, the package call would look like this:

```
PEA    D_ADR    ; push operand addresses
PEA    E_ADR    ;   onto the stack
FADDD                ; macro for call to add (double)
```

The syntax for calls directly to the MC68881 is quite different. Instead of pushing the operands onto the stack, you move them to and from any of eight floating-point registers, FP0 through FP7. To add $d + e$ with the result in e , the MC68881 call might look like this:

```
FMOVE.D <d>,FP2 ; move d operand to reg. 2
FADD.X  <e>,FP2 ; add (extended) in reg. 2
FMOVE.X FP2,<e> ; move result into e
```

If the result operand e were already in one of the floating-point registers, say FP4, the call would look like this:

```
FADD.D  <d>,FP4 ; add (double) with (extended) result in reg. 4
```

MC68881 data types

Like the SANE software, the MC68881 floating-point coprocessor performs arithmetic on numbers in extended format. This chapter describes the way the MC68881 handles extended-format numbers.

The MC68881 also has byte, word, and longword integer formats and a packed decimal floating-point format. For information about those formats, please consult Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.

MC68881 floating-point registers

The MC68881 coprocessor makes available eight floating-point registers, FP0 through FP7. Each of those registers is 80 bits wide and holds an extended-format number. High-level languages for the MC68881 use some or all of the floating-point registers for evaluating floating-point expressions. Programmers using those languages who want to use floating-point registers for temporary storage in assembly-language routines will need to consult language documentation to determine which ones are scratch registers and which ones must be preserved.

96-bit extended format

For data storage, the MC68881 floating-point coprocessor uses a 96-bit extended format made up of five fields, as shown in Figure 27-4. Note that the s , e , i , and f fields in the 96-bit format are the same as those in the standard SANE 80-bit format; the shaded field is unused. The 96-bit format is a multiple of 4 bytes to exploit the MC68020 microprocessor's ability to fetch data faster when the data is aligned on longword boundaries.

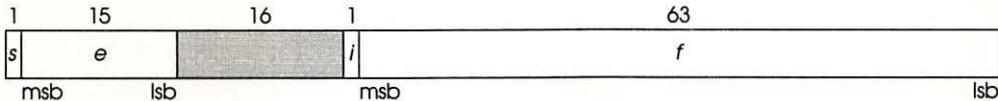


Figure 27-4
96-bit extended format

Table 27-1 shows how the value v of the number is determined by the fields shown in Figure 27-4.

Table 27-1
Values of extended-format numbers

Biased exponent e	Integer i	Fraction f	Value v	Class of v
$0 \leq e \leq 32766$	1	(any)	$v = (-1)^s \times 2^{(e-16383)} \times (1.f)$	Normalized
$0 \leq e \leq 32766$	0	$f \neq 0$	$v = (-1)^s \times 2^{(e-16383)} \times (0.f)$	Denormalized
$0 \leq e \leq 32766$	0	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 32767$	(any)	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 32767$	(any)	$f \neq 0$	v is a NaN	NaN

Important

The extended format is not a standard format. You should not use it for data stored in files used by other programs. The IEEE Standard specifies only minimum precision and range for extended format; implementations of the IEEE Standard (or SANE) may use a different extended format that meets those minimum specifications.

Comparison of extended formats

Despite the difference in lengths of the formats, numbers stored in the MC68881's 96-bit extended format have exactly the same precision as numbers in SANE's 80-bit extended format. Figure 27-5 shows the resemblance between the two extended formats. Notice that the 16-bit unused field in the 96-bit format is aligned on a word boundary. Also notice that the exponent part of both formats also occupies 16 bits on a word boundary.

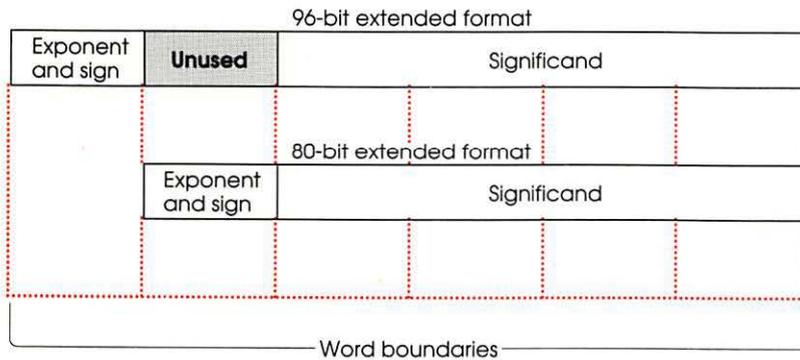


Figure 27-5
Comparison of extended formats

The MC68881 loads 96-bit extended values into its 80-bit floating-point registers by ignoring the unused 16 bits. When the MC68881 writes the contents of a floating-point register to memory, it also writes over the unused 16 bits.

Conversions between extended formats

Programs that must be compatible with MC68000-based machines should continue to use the 80-bit extended format used by the software SANE packages. Ideally, programs that make calls directly to the MC68881 should use the 96-bit extended format exclusively. It is not a good idea to mix formats in the same program because there is no way for the program to determine whether a stored value is in 80-bit format or 96-bit format.

High-level languages that support SANE use the software packages and store data in the 80-bit extended format. Many of those languages provide an option to produce code that makes use of the MC68881 and stores data in the 96-bit extended format.

For programs that call routines that use the 80-bit extended format, the SANE library for the MC68881 includes conversion routines (X96ToX80 and X80ToX96). Using these conversion routines, the programmer can define a new interface to the 80-bit routines and so make the conversions happen each time the routines are called.

For example, suppose you have a routine that uses the 80-bit format; call it `FPFunc`. You could reuse your routine under the MC68881 option by creating a 96-bit interface around it. The interface to your function could look like the following code. (Remember to name your 80-bit extended data type something distinct from the name of the 96-bit extended type.)

```
FUNCTION FPFunc (x: Extended80): Extended80; External;
FUNCTION FPFunc96 (x: Extended): Extended;
BEGIN
    FPFunc := X80ToX96(FPFunc(X96ToX80(x)));
END;
```

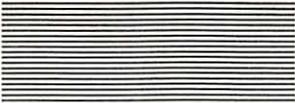
Using the comp format with the MC68881

The MC68881 doesn't support the comp format directly. High-level languages for the MC68881 machines must handle comp format conversions with software even when directed to make calls directly to the MC68881. Assembly-language programmers who want to use the comp format can either use the SANE package software to perform arithmetic with comp variables or call the conversion routines `Comp2X` and `X2Comp`, which convert back and forth between comp and extended.

SANE macros for the MC68881

MPW version 3.0 (and subsequent versions) includes new macros that replace SANE package calls with MC68881 operations using the 96-bit extended format. For operations not available on the MC68881, the macros convert 96-bit values to 80-bit extended format and make package calls. By changing their programs to use the 96-bit extended format and reassembling with the new macros, programmers who use MPW can gain a performance improvement from the MC68881.

The macros make it easy to reassemble programs for the MC68881, but they don't make the best use of the MC68881's floating-point registers. To obtain maximum performance, you may prefer to rewrite your programs for the MC68881 rather than to use the macros.



Chapter 28



Functions of the MC68881 and SANE Software

Though different in detail, the MC68881 and the SANE packages are both implementations of the IEEE Standard 754. The MC68881 and the packages perform many of the same functions, and of those, many give the same results on both. Other functions give slightly different results; a few other functions are available on only one or the other.

This chapter lists the functions in all those categories. For complete information about the MC68000 SANE packages, please refer to Part III of this book. For complete information about the Motorola MC68881, please refer to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.

Functions that are the same on both

For fundamental operations and certain other functions, the MC68881 obtains results that are bit-for-bit identical to those obtained by the SANE packages. On machines that use the MC68020 and the MC68881, the SANE packages call MC68881 operations for those functions, as explained in the previous chapter. Programs that need not be compatible with MC68000-based models of the Macintosh can obtain increased speed by calling the MC68881 directly for those functions.

❖ *Note:* The MC68881 and the SANE software packages return the same values for the operations listed here, except when the operation creates a NaN, such as square root of a negative number. Both implementations create NaNs, but the MC68881 does not support the SANE NaN codes; see the section on NaNs in Chapter 5.

The MC68881 and the SANE software packages return identical results for the following operations:

- addition
- subtraction
- multiplication
- division
- square root
- remainder (but format of *quo* is different)
- round-to-integral value
- comparison
- conversions between floating-point formats

❖ *Note:* Motorola's MC68881 manual refers to all these operations, along with the binary-to-binary conversions, as the arithmetic operations. This book calls those the fundamental operations, reserving the word *arithmetic* for the set comprising only addition, subtraction, multiplication, division, square root, and remainder.

Functions that are similar

For some operations, the MC68881 and the software packages return different results. For transcendental operations, the MC68881 obtains results that are slightly less accurate than those obtained by the software packages; for a few operations, the MC68881 obtains results that are different for cases involving 0, Infinities, and NaNs. Programs that can tolerate the diminished accuracy and that need not run on MC68000-based models of the Macintosh can obtain increased speed by calling the MC68881 for those functions instead of using the software packages.

The following operations are less accurate on the MC68881 compared with the software or differ in other ways, as noted:

- binary scale (MC68881 truncates scale factors to 14 bits)
- binary-to-decimal and decimal-to-binary conversion (MC68881's decimal format has shorter fields; it doesn't support extended format)
- base-e logarithm
- base-2 logarithm
- base-e logarithm of $1 + x$
- base-e exponential
- base-2 exponential
- base-e exponential minus 1
- sine
- cosine
- tangent
- arctangent

The following operations on the MC68881 have the same accuracy as the software, but behave differently for zero, denormalized numbers, Infinities, and NaNs:

- negation (only the MC68881 can signal exceptions and halt)
- absolute value (only the MC68881 can signal exceptions and halt)
- classify (Mode Control byte in the MC68881 classifies numbers but does not distinguish between normalized and denormalized numbers)
- round-to-integer (identical except when out-of-range: the packages give largest negative value, while the MC68881 preserves sign)
- truncate-to-integer (identical except when out-of-range: the packages give largest negative value, while the MC68881 preserves sign)
- binary logarithm (identical results except for 0 and Infinity)

Functions only the SANE software has

The SANE software packages include several functions that the MC68881 doesn't have. Those functions are

- copy sign
- next after
- integer exponentiation
- general exponentiation
- base-2 logarithm of $1 + x$
- base-2 exponential minus 1
- compound interest
- annuity factor
- random number generator
- conversions between floating-point types and decimal records
- scanning and formatting between ASCII strings and decimal records

The SANE software packages also support the comp type in arithmetic operations, conversions, classifications, and comparisons, whereas the MC68881 does not.

Functions only the MC68881 has

The MC68881 includes several functions that the SANE software packages don't have. Those functions are

- binary mantissa
- sine and cosine (in a single call)
- arcsine
- arccosine
- hyperbolic sine
- hyperbolic cosine
- hyperbolic tangent
- hyperbolic arctangent
- base-10 logarithm
- base-10 exponential
- modulo remainder (rounds toward zero)

- set condition (floating-point)
- multiply and round to single
- divide and round to single

The MC68881 also supports the single-byte signed-integer format; the software packages do not.

❖ *Note:* The transcendental operations in this list return results with accuracy similar to that of the other transcendental operations on the MC68881.

Accuracy of the MC68881's elementary functions

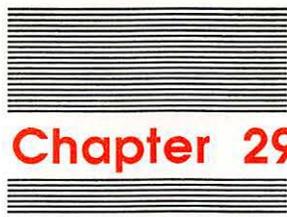
The hybrid SANE packages on Macintosh models with MC68881 coprocessors call the MC68881 for basic arithmetic and comparisons, where its results are identical to those obtained by the SANE packages for the MC68000. For the elementary functions (that is, transcendental functions), hybrid SANE performs the operations in software, using the MC68881's basic arithmetic to make them relatively fast.

The hybrid SANE packages compute the elementary functions in software both for accuracy and for compatibility: the hybrid packages for the MC68881 return the same results, bit-for-bit, as the software packages for the MC68000 do. Programs that run on either kind of Macintosh are assured of consistency of results. Furthermore, the SANE packages are better behaved than the MC68881. Even where there are errors, the SANE packages are more nearly monotonic, and more often return correct answers for inverse operations. For example, when asked to compute

$$y = \arctan (\tan (0.5))$$

the MC68881 returns an extended value for y of 0.4999999999999999800, whereas the SANE packages return 0.5000000000000000000 for y .

For the elementary functions, both the SANE packages and the MC68881 have errors in the least significant bits of the fraction part of extended-format results, but the packages' errors rarely exceed the last bit, whereas the MC68881's errors can extend to as many as the last five bits. For individual elementary functions, the MC68881 and the SANE packages return results that are practically identical when rounded to single or double precision. For complicated expressions involving elementary (transcendental) functions, the MC68881 is much more likely to return an error in a double-precision result than the SANE packages are.



Chapter 29

Controlling the MC68881 Environment

The MC68881 version of SANE differs from the MC68000 version in the way it stores the floating-point environment information.

Where the MC68000 SANE packages maintain an Environment word in low memory, the MC68881 version keeps the floating-point environment in the MC68881's status and control registers. The location of environment information is completely transparent to programs that access the environment with the SANE environment-control calls described in Chapter 7. For compatibility with programs written for machines that use the MC68000, the hybrid SANE packages for MC68881 machines use the same environment-control calls as the software SANE packages for the MC68000.

Warning

Programs that change their environment flags by directly manipulating bits in SANE's Environment word in low memory will not run correctly on models of the Macintosh that use the MC68881. This might also affect compilers that set the default environment by storing zero directly into the Environment word. As always, direct manipulation of global data in low memory is ill-advised.

The MC68881's environment registers

The MC68881 has two registers that contain (among other things) the floating-point environment information. Those registers are FPCR, the Floating-Point Control register, and FPSR, the Floating-Point Status register. Figures 29-1 and 29-2 show the bytes making up the registers. The control register FPCR contains 4 bytes, but only 2 are used: the Exception Enable byte and the Mode Control byte. The status register FPSR contains 4 bytes: the Condition Code byte, the Quotient byte, the Exception Status byte, and the Accrued Exception byte.

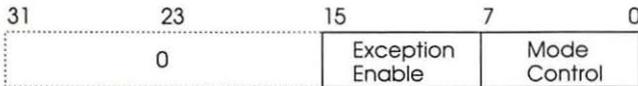


Figure 29-1
The MC68881's Floating-Point Control register (FPCR)

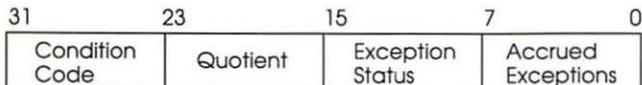


Figure 29-2
The MC68881's Floating-Point Status register (FPSR)

The MC68881 bytes that include the equivalent of the SANE packages' environmental information are the FPCR's Exception Enable and Mode Control bytes and the FPSR's Exception Status and Accrued Exception bytes. The next two sections describe those bytes in detail.

❖ *Note:* The environment bits in the MC68881's FPCR and FPSR are similar to the environment bits in the SANE software for the MC68000, but they aren't in the same locations in the bytes.

The Exception Enable and Mode Control bytes

The FPCR's Exception Enable and Mode Control bytes contain bits for enabling traps (halts) and for setting the rounding modes—that is, rounding precision and direction. Those bits have near counterparts in the MC68000 SANE Environment word.

Figure 29-3 and Table 29-1 identify the individual bits in the Exception Enable and Mode Control bytes.

❖ *Note:* The trap mechanism on the MC68881 is similar to the halt mechanism used by the software SANE packages. See Chapter 30, "The MC68881 Trap Mechanism."

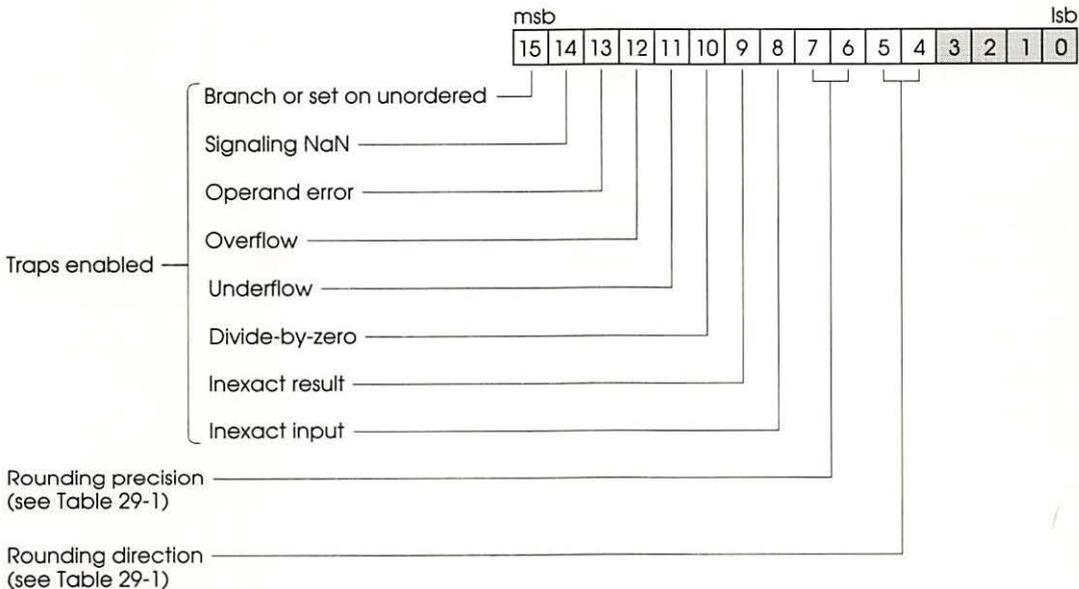


Figure 29-3
The MC68881's Exception Enable and Mode Control bytes

Table 29-1

Bits in the MC68881's Exception Enable and Mode Control bytes

Group name	Mask bits	Mask value	Description
Traps enabled (Bit group \$FF00)	15 14 13 12 11 10 9 8		
	1 0 0 0 0 0 0 0	\$8000	Branch or set on unordered
	0 1 0 0 0 0 0 0	\$4000	Signaling NaN
	0 0 1 0 0 0 0 0	\$2000	Operand error
	0 0 0 1 0 0 0 0	\$1000	Overflow
	0 0 0 0 1 0 0 0	\$0800	Underflow
	0 0 0 0 0 1 0 0	\$0400	Divide-by-zero
	0 0 0 0 0 0 1 0	\$0200	Inexact result
	0 0 0 0 0 0 0 1	\$0100	Inexact input
Rounding precision (Bit group \$00C0)	7 6		
	0 0	\$0000	Extended
	0 1	\$0040	Single
	1 0	\$0080	Double
	1 1	\$00C0	(Undefined)
Rounding direction (Bit group \$0030)	5 4		
	0 0	\$0000	To-nearest
	0 1	\$0010	Toward-zero
	1 0	\$0020	Downward
	1 1	\$0030	Upward

Note: Bits 0, 1, 2, and 3 are not used.

❖ *Note:* To trap on all operations that generate the inexact exception, you must enable traps on both inexact result and inexact input. Similarly, to trap on all operations that generate the invalid exception, enable traps on the following three exceptions:

- branch or set on unordered
- signaling NaN
- operand error

The Exception Status and Accrued Exception bytes

The MC68881's FPSR contains bits for two kinds of exception flags: exception status flags and accrued exception flags. The exception status flags reflect the status of the last instruction; they are cleared at the start of each instruction. The accrued exception flags reflect the history of all previous instructions since the last time they were cleared by the application. The accrued exception flags are equivalent to the exception flags in the MC68000 SANE software. Figure 29-4 and Table 29-2 identify the individual bits in the FPSR's Exception Status and Accrued Exception bytes.

At the end of each operation that can affect the Accrued Exception bits, each of the Current Exception bits that is set in turn causes the setting of the corresponding Accrued Exception bit. Either the inexact-result bit or the inexact-input bit can cause setting of the inexact Accrued Exception bit. Similarly, any of the following three exception bits causes the setting of the invalid Accrued Exception bit:

- branch or set on unordered
- signaling NaN
- operand error

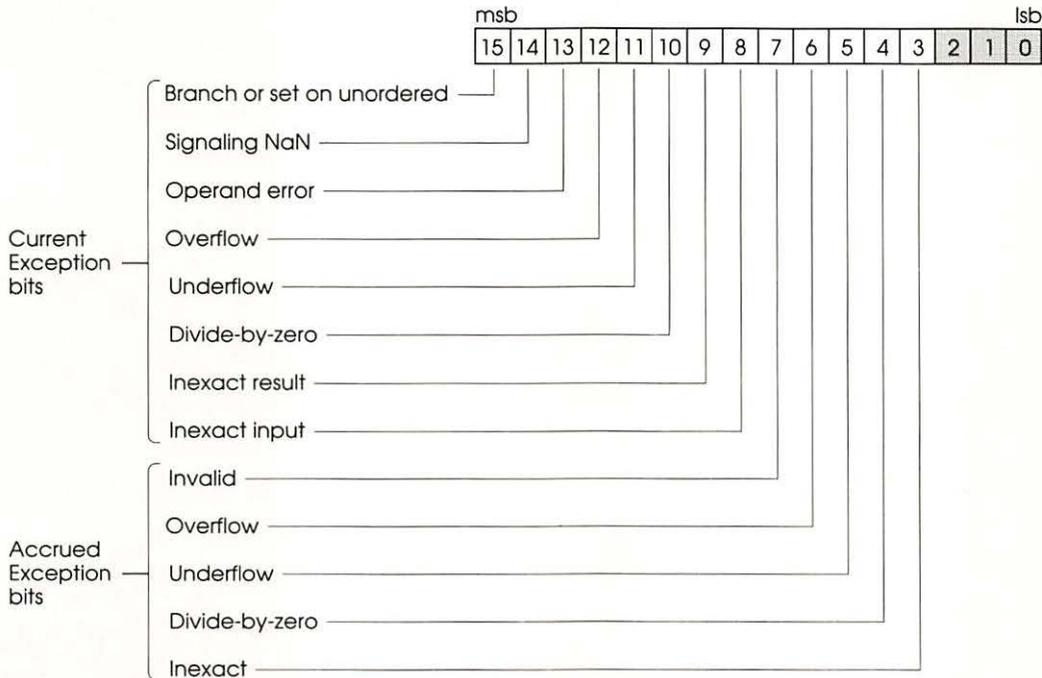


Figure 29-4
The MC68881's Exception Status and Accrued Exception bytes

Table 29-2

Bits in the MC68881's Exception Status and Accrued Exception bytes

Group name	Mask bits	Mask value	Description
Current Exception flags (Bit group \$FF00)	15 14 13 12 11 10 9 8		
	1 0 0 0 0 0 0 0	\$8000	Branch or set on unordered
	0 1 0 0 0 0 0 0	\$4000	Signaling NaN
	0 0 1 0 0 0 0 0	\$2000	Operand error
	0 0 0 1 0 0 0 0	\$1000	Overflow
	0 0 0 0 1 0 0 0	\$0800	Underflow
	0 0 0 0 0 1 0 0	\$0400	Divide-by-zero
	0 0 0 0 0 0 1 0	\$0200	Inexact result
0 0 0 0 0 0 0 1	\$0100	Inexact input	
Accrued Exception flags (Bit group \$00F8)	7 6 5 4 3		
	1 0 0 0 0	\$0080	Invalid operation
	0 1 0 0 0	\$0040	Overflow
	0 0 1 0 0	\$0020	Underflow
	0 0 0 1 0	\$0010	Divide-by-zero
0 0 0 0 1	\$0008	Inexact	

Note: Bits 0, 1, and 2 are not used.



Chapter 30



The MC68881 Trap Mechanism

Traps in the MC68881 and halts in the software SANE packages are two different mechanisms for handling exceptional events. The MC68881 implements the traps that are an optional part of the IEEE Standard. For more information about the way the MC68881 handles traps, please refer to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*. For information about halts and SANE software, please refer to Chapter 23, "Halts in MC68000 SANE."

Halts and traps

Halts and traps respond to the SANE exceptions in different ways. In the event of a halt, the SANE packages put information about the halt onto the stack and pass control through one halt vector to a single halt-handling routine. (For a complete description, please refer to Chapter 23.) The MC68881 deals with a trap by passing control through one of seven trap vectors to separate trap handlers, which find information about the trap in the MC68881's registers.

Calls to hybrid SANE packages, which use the MC68881 coprocessor, use the same one-vector halt mechanism as the software SANE packages, thus maintaining compatibility with programs written for other models of the Macintosh. On the other hand, programs that call the MC68881 directly must use the MC68881's trap mechanism.

MC68881 exception handling

Exception handlers for programs that make direct calls to the MC68881 must use the trap mechanism described in Motorola's *MC68881 Floating-Point Coprocessor User's Manual* and use the routines SetTrapVector and GetTrapVector in the SANE library. Several functions in the SANE library are implemented internally by calls to the package software because the MC68881 does not perform those functions. To avoid forcing your program to deal with two different mechanisms, the MPW libraries for the MC68881 protect the calling program against package-type halts by substituting an MC68881 trap whenever an exception occurs that would otherwise have caused a package halt.

❖ *Note:* Programmers using high-level languages needn't be concerned about mixing halts and traps. Programs compiled with the MC68881 options and linked to the SANE libraries for the MC68881 always use the MC68881's trap mechanism. Any exception handlers must be coded for the appropriate mechanism.

Assembly-language programs that use the MC68881 directly can make calls to the software packages by using macros for the MC68881, which automatically map the software halts into hardware traps. (See "SANE Macros for the MC68881" in Chapter 27.) Alternatively, programmers can use a similar approach in their own code.

The Pascal and C SANE libraries in MPW include routines for use with the MC68881. Those routines bracket SANE package calls with routines that map the software exceptions to the equivalent hardware flags. One of those routines saves MC68881 exception and trap flags, clears the package flags, then calls the SANE packages. Thus, the packages never see enabled halts. Before returning to the caller, the other routine restores the saved MC68881 exception and halt flags and signals any exceptions pending from the packages. In this way the package halt mechanism is disabled and exceptions are handled by the MC68881's trap mechanism.



Chapter 31



Examples: Using the MC68881 SANE Engine

The following examples illustrate direct calls to the MC68881 coprocessor. The examples are similar to those in Chapter 26, "Examples: Using the MC68000 SANE Engine."

In the comments, $\&x$ means the address of x and $STACK: x < y < z \dots$ means x is on the top-of-stack, y is next deeper in the stack, and z is next deeper after y .

Example: polynomial evaluation

This example illustrates the evaluation of a polynomial

$$c_0x^n + c_1x^{n-1} + \dots + c_n$$

using Horner's recurrence:

$$res \leftarrow c_0$$

$$res \leftarrow res \times x + c_j, \text{ for } j = 1 \text{ to } n$$

On entry, A1 points to the coefficient table of extended values and FP1 contains the extended function argument x . The example returns its result in FP0. The coefficient table consists of a leading word that is a positive integer equal to $n - 1$, where n is equal to the degree of the polynomial, followed by $n + 1$ extended coefficients, starting with c_0 . For example, for the polynomial $x^3 + 2x^2 - 5$, A1 points to the following table:

DC.W	2	; n-1 = 2 (one less than last subscript)
DC.X	"1"	; c0 = 1
DC.X	"2"	; c1 = 2
DC.X	"0"	; c2 = 0
DC.X	"-5"	; c3 = -5

Entry is by a JSR instruction to location PolyEval.

```

PolyEval Proc  Entry
    MOVE.W    (A1)+,D0      ; copy n into D0
                          ; and advance A1 to point to first coeff
    FMOVE.X  (A1)+,FP0     ; initialize the result space with c0
                          ; and advance A1 to point to c1

PolyLoop
    FMUL.X   FP1,FP0       ; multiply accumulated result by x
    FADD.X   (A1)+,FP0     ; add next coefficient, advance coeff pointer
    DBRA    D0,PolyLoop   ; decrement loop counter, continue if D0 ≥ 0

    RTS

```

Example: language interface

This example illustrates the kind of code required to implement a high-level language interface to the MC68881 coprocessor. This example implements the following Pascal function:

```
FUNCTION ScalbNew(n : integer; x : extended) : extended;
```

The calling routine performs the following steps:

- push a 4-byte pointer to a 12-byte space for the return value
- push a 2-byte value for n
- push the 4-byte address of the 12-byte value x
- execute a JSR instruction to SCALBNew

Thus, on entry, the stack contains $\&ret < \&x < n < \&res < \dots$. The called routine will clear the stack back to the result pointer.

```
SCALBNEW Proc      Entry
                MOVEM.L  (SP)+,A0/A1    ; return address -> A0,
                ; address of x -> A1,
                ; removing both from the stack
                FMOVE.X  (A1),FP0       ; copy x to working register
                FSCALE.W (SP)+,FP0      ; scale x by n and remove n from stack
                MOVE.L   (SP),A1        ; collect result address, leave on stack
                FMOVE.X  FP0,(A1)       ; return result
                JMP      (A0)           ; return to caller
```

Example: scanning and formatting

The following example shows the mixing of package calls with direct MC68881 calls. It must use 80-bit extended for calls to the software packages, and 96-bit extended for calls to the MC68881. The last two equates (`xOfff80` and `xOfff96`) are exploited to convert in place between the two extended formats; Figure 27-5 shows how this is done. Concurrency between the MC68020 and the MC68881 sometimes allows this format conversion to take zero time.

The example illustrates the use of the numeric scanner and formatter. It accepts as argument an ASCII string representing a number of degrees and returns the trigonometric sine of its argument as a numeric ASCII string. Both input and output are Pascal strings: the zeroth byte gives the length; the first byte contains the first character in the string. The caller of the procedure pushes the address of the input string and executes a JSR instruction to location `SINE`. The procedure overwrites the input string with the result, whose length may be as large as 80, and clears the stack. The procedure `Sine` could be declared in Pascal as follows:

```
PROCEDURE Sine(var s : DecStr);
```

Notice that the example includes two methods for computing the sine. It makes a call to the MC68881, but it also includes code, in the form of comments, for making the equivalent package call.

```

SINE  PROC  Entry

; Offsets from A6 for i/o pointer and temporaries -
sOff  EQU   8           ; s : i/o string
iOff  EQU  -2           ; i : 16-bit integer index
dOff  EQU  iOff-26      ; d : decimal record
                        ; note extra word-alignment byte
vOff  EQU  dOff-2      ; v : 16-bit boolean for valid prefix
                        ; note extra word-alignment byte
xOff80 EQU  vOff-10     ; x : when viewed as 80-bit extended
xOff96 EQU  xOff80-2   ; x : when viewed as 96-bit extended

LINK  A6,#xOff96      ; STACK: x < v < d < i < A6 < &ret < &s
MOVE.W #1,iOff(A6)    ; initialize i to 1
MOVE.L sOff(A6),-(A7) ; push s address
PEA   iOff(A6)        ; push i address
PEA   dOff(A6)        ; push d address
PEA   vOff(A6)        ; push v address
FPSTR2DEC              ; call Pack7 to convert string to decimal record

PEA   dOff(A6)        ; push d address
PEA   xOff80(A6)      ; push x address (pack call, so delivers 80-bits)
FDEC2X                 ; call Pack4 to convert decimal to extended

LEA   Pi180,A0        ; collect address of  $\pi/180$  (below)
FMOVE.X (A0),FP0      ; move  $\pi/180$  to a working register
MOVE.W xOff80(A6),xOff96(A6) ; convert x to 96-bit extended
FMUL.X xOff96(A6),FP0 ; x *  $\pi/180$  (881 call, so use 96-bits)

```

```

; deliver sin (x *  $\pi$ /180) to x as an 80-bit extended
FSIN.X FP0          ; sin (x *  $\pi$ /180)
FMOVE.X FP0,xOff96(A6) ; deliver binary result to x (881 call, 96-bits)
MOVE.W xOff96(A6),xOff80(A6) ; convert x to 80-bit extended

; or use this alternate code for more precise (but slower) sine calculation:
; FMOVE.X FP0,xOff96(A6) ; copy x *  $\pi$ /180 to x
; MOVE.W xOff96(A6),xOff80(A6) ; convert x to 80-bit extended
; PEA xOff80(A6) ; push x address (pack call, use 80-bit extended)
; FSINX ; call pack5 to effect sin (x) -> x

PEA DecForm ; push DecForm address
PEA xOff80(A6) ; push x address (pack call, 80-bits)
PEA dOff(A6) ; push d address
FX2DEC ; call Pack4 to convert extended to decimal

PEA DecForm ; push DecForm address
PEA dOff(A6) ; push d address
MOVE.L sOff(A6),-(A7) ; push s address
FDEC2STR ; call Pack7 to convert decimal to string

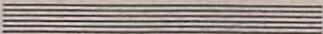
UNLK A6 ; restore stack to: &ret < &s
MOVE.L (SP)+,(SP) ; leaving only &ret on stack
RTS

Pi180 DC.X "$3FF93FF98EFA351294E9C8AE";  $\pi$ /180, 96-bit format
DecForm DC.W $0100 ; style = fixed
DC.W $000A ; digits = 10

```



Appendixes





Appendix A



SANE in High-Level Languages

Much of the Standard Apple Numerics Environment, as specified in this manual, can be provided to the high-level language programmer through a run-time library. However, to use extended-precision arithmetic efficiently and effectively, and to handle NaNs and Infinities, some extensions to standard languages are required. Apple has designed SANE extensions for languages, including Pascal and C, following the design goals of

- full and scrupulous conformance to the IEEE floating-point standard
- maximum exploitation of extended-precision arithmetic
- upward compatibility from current language standards
- minimum deviation from current language standards

This appendix describes the SANE extensions to high-level languages and includes SANE library interfaces. A programming language that makes the features of SANE available to its users by providing SANE extensions and a SANE library is said to *support* SANE. (A programming language may also *use* SANE without fully supporting SANE; that is, it may use the SANE engine to perform arithmetic without making all features of SANE available to its users.)

- ❖ *Note:* These extensions and interfaces are implemented in Apple's MPW and APW languages. Other language products that support SANE have similar extensions and interfaces.

Low-level SANE arithmetic (as well as the floating-point chips Intel 8087, 80287, and 80387, and Motorola 68881 and 68882) evaluates arithmetic operations to the range and precision of the 80-bit extended type. Thus, extended naturally becomes the basic arithmetic type for computing purposes. The types single, double, and comp serve as space-saving storage types. Adding floating-point types and installing extended in the role of the basic arithmetic type are the most salient changes to standard languages.

❖ *Note:* The extended type can be used as long double in ANSI C.

The SANE extensions expand the syntax for input/output to accommodate NaNs and Infinities, and include the treatment of NaNs in relationals as required by the IEEE Standard.

The SANE extensions are upward compatible from current language standards: most programs using only the traditional floating-point types and operations may be compiled and run without modification; the rest require only setting of rounding precision to single or double in a few places. (See “Example: Is x Negligible?” in Chapter 3.) SANE does not affect integer arithmetic.

SANE programming

Automatic use of the extended type by SANE language systems produces results that are generally better than those of language systems without extended-based IEEE floating point. Extended precision yields more accuracy, and extended range avoids unnecessary underflow and overflow of intermediate results. The programmer can further exploit the extended type by declaring all floating-point temporary variables, formal value parameters, and function results to be type extended. This is generally both time-efficient and space-efficient, because it reduces the number of automatic conversions between types. External data should be stored in one of the three smaller SANE types (single, double, comp), not just for economy but also because the extended format may vary between SANE implementations. As a general rule, use single, double, or comp data as program input; extended arithmetic for computations; and single, double, or comp data as program output.

In many instances, IEEE arithmetic allows simpler algorithms than were possible without IEEE arithmetic. The handling of Infinities enlarges the domain of some formulas. For example, $1 + 1/x^2$ computes correctly even if x^2 overflows. Running with halts disabled (the IEEE Standard default), a program will never crash because of a floating-point exception. Hence, by monitoring exception flags, a program can test for exceptional cases after the fact. The alternative, screening out bad input, is often infeasible, sometimes impossible.

Part I of this book gives several examples of the way IEEE arithmetic can make programs simpler.

SANE library

A SANE language system includes a SANE library. This library rounds out the IEEE Standard implementation and provides basic tools for developing a wide range of applications. A SANE library includes the following features:

- logarithmic, exponential, and trigonometric functions
- financial functions
- random number generation
- binary-decimal conversion
- numeric scanning and formatting
- environment control
- other functions required or recommended by the IEEE Standard

Pascal SANE extensions

With the extensions described below (and a SANE library), ANSI Pascal supports SANE.

Data types

The Pascal real type and three new Pascal types correspond to the four SANE numeric data types, as shown in Table A-1.

Table A-1
Pascal data types

Pascal name	SANE type
Real	IEEE single
Double	IEEE double
Comp	SANE comp
Extended	IEEE extended

Constants

Two types of numeric constants are of type extended: those that include floating-point syntax—a point (.) or an exponent field—and those that lie outside the range of `longint`.

Decimal-to-binary conversions for numeric constants are done at compile time in the IEEE default numeric environment—see “Numeric Environment,” later in this section. Exceptions that arise from those conversions are not signaled at run time.

In SANE Pascal, assignment of constants to variables is always done at run time.

You may need a constant in executable code rounded some way other than the default rounding set by the compiler. To cause conversions of decimal constants in executable code to occur at run time, use `Str2Num`. For example, replace the first of these statements by the second:

```
x := 1.234;  
x := Str2Num('1.234');
```

Expressions

The SANE types `real`, `double`, `comp`, and `extended` can be mixed in expressions with each other and with integer types. An expression consisting solely of a SANE-type variable, constant, or function is of type extended. An expression formed by subexpressions and an arithmetic operation is of type extended if either of its subexpressions is of that type. Extended expressions are evaluated by using extended-precision SANE arithmetic, with conversions to extended type generated automatically as needed. An extended expression can be assigned to any SANE-type variable, but cannot be assigned to an integer-type variable.

Expressions in constant definitions are evaluated at compile time; all other evaluation of extended expressions is done at run time.

Comparisons

In IEEE Standard arithmetic, the result of a comparison involving a NaN operand is unordered. Thus, the usual trichotomy of numbers is expanded to less, greater, equal, and unordered. The Pascal relational operator `<>` means *not equal*, rather than *less or greater*. The other Pascal relational operators (`<`, `<=`, `=`, `>`, `>=`) retain their traditional interpretations. Note, however, that the negation of *a less than b* is not *a greater than or equal to b* but *(a greater than or equal to b) OR (a and b unordered)*.

Functions and procedures

Parameter passing follows the same rules of types that govern assignments. (See the earlier section “Expressions.”)

Input/output

In addition to the usual syntax accepted for numeric input, the read routine recognizes `INF` as Infinity and `NAN` as a NaN. `NAN` may be followed by parentheses, which may contain an integer (a code indicating the NaN’s origin). `INF` and `NAN` are optionally preceded by a sign and are case insensitive.

The write routine formats values of extended expressions in the manner of standard Pascal number formatting, with the following exceptions:

- Infinities are written as `[-]INF` and NaNs as `[-]NAN(ddd)`, where `ddd` is the NaN code.
- Exponent-digits fields are four characters wide (numbers from the extended type may require this); the first exponent digit is 0 only if the exponent is zero, and the field is padded on the right with space characters.
- The ANSI restriction that the second colon parameter must be positive may be removed to facilitate formatting large integral values from SANE types.

Numeric environment

The numeric environment contains rounding direction, rounding precision, halt enables, and exception flags. IEEE Standard defaults—rounding to nearest, rounding to extended precision, and all halts disabled—are in effect for compile-time arithmetic (including decimal-to-binary conversion). A Pascal system may begin programs with these defaults and with all exception flags clear. Alternatively, to conform more nearly to ANSI specifications for floating-point errors, a Pascal system may begin programs with halts on invalid, overflow, and divide-by-zero enabled, and IEEE default settings otherwise. Regardless of the Pascal system defaults, programmers can easily modify the environment by using the functions for managing the environment included in the run-time SANE library. Optimizing compilers should not rearrange the order of floating-point evaluation in any way that might change either the computed value or the side effects (such as exception flag settings) from what may have been intended in the source code.

Pascal SANE library

Following is the interface for a Pascal library using the SANE packages. This interface is part of Apple's MPW Pascal; other Pascal language products that support SANE have similar interfaces.

In this interface, `INTEGER` refers to 16-bit integers and `LONGINT` refers to 32-bit integers.

- ❖ *Note:* This Pascal interface includes sections for machines with and without an MC68881 coprocessor. Conditional compilation selects the appropriate section based on the settings of compiler options `MC68881` and `Ellems881`.

```
{ SANE.p - Pascal interface for Standard Apple Numeric Environment

Copyright Apple Computer, Inc. 1983 - 1987
All rights reserved. }

UNIT SANE;

INTERFACE

{ Ellems881 mode set by -d Ellems881=true on Pascal command line }

{$IFC UNDEFINED Ellems881}
    {$SETC Ellems881 = FALSE}
{$ENDC}

{$IFC OPTION(MC68881)}

(*=====*)
* The interface specific to the MC68881 SANE library *
*=====*)
CONST
{-----}
* Exceptions.
{-----}

    Inexact = 8;
    DivByZero = 16;
    Underflow = 32;
    Overflow = 64;
    Invalid = 128;

    CurInex1 = 256;
    CurInex2 = 512;
    CurDivByZero = 1024;
    CurUnderflow = 2048;
    CurOverflow = 4096;
    CurOpError = 8192;
    CurSigNaN = 16384;
    CurBSONUnor = 32768;
```

```

{-----
* Environmental control.
-----}
TYPE
  TrapVector = RECORD
    Unordered: LONGINT;
    Inexact   : LONGINT;
    DivByZero: LONGINT;
    Underflow: LONGINT;
    OpError  : LONGINT;
    Overflow : LONGINT;
    SigNaN   : LONGINT;
  END;

  Exception = LONGINT;
  Environment = RECORD
    FPCR: LONGINT;
    FPSR: LONGINT;
  END;

FUNCTION IEEEDefaultEnv: Environment;
{ return IEEE default environment }

PROCEDURE GetTrapVector(VAR Traps: TrapVector);
{ Traps <-- FPCP trap vectors }

PROCEDURE SetTrapVector(Traps: TrapVector);
{ FPCP trap vectors <-- Traps }

{-----
* TYPES and FUNCTIONS for converting between SANE Extended formats
-----}
TYPE
  Extended80 = ARRAY [0..4] OF INTEGER;

FUNCTION X96toX80(x: EXTENDED): Extended80;
{ X96toX80 <-- 96 bit x in 80 bit extended format }

FUNCTION X80toX96(x: Extended80): EXTENDED;
{ X80toX96 <-- 80 bit x in 96 bit extended format }

```

```

{-----}
* Compatible Transcendental functions - bypasses direct MC68881 calls
{-----}
{$IFC Elems881 = false}

FUNCTION Sin(x: EXTENDED): EXTENDED;
{ sine }

FUNCTION Cos(x: EXTENDED): EXTENDED;
{ cosine }

FUNCTION ArcTan(x: EXTENDED): EXTENDED;
{ inverse tangent }

FUNCTION Exp(x: EXTENDED): EXTENDED;
{ base-e exponential }

FUNCTION Ln(x: EXTENDED): EXTENDED;
{ base-e log }

FUNCTION Log2(x: EXTENDED): EXTENDED;
{ base-2 log }

FUNCTION Ln1(x: EXTENDED): EXTENDED;
{ ln(1+x) }

FUNCTION Exp2(x: EXTENDED): EXTENDED;
{ base-2 exponential }

FUNCTION Exp1(x: EXTENDED): EXTENDED;
{ exp(x) - 1 }

FUNCTION Tan(x: EXTENDED): EXTENDED;
{ tangent }

{$ENDC}

{$ELSEC}

{ *=====*
* The interface specific to the software SANE library *
*=====*}

CONST
{-----}
* Exceptions.
{-----}

Invalid = 1;
Underflow = 2;
Overflow = 4;
DivByZero = 8;
Inexact = 16;

```

```

{-----
* IEEE default environment constant.
-----}
    IEEEDefaultEnv = 0;

{-----
* Environmental control.
-----}
TYPE
    Exception    = INTEGER;
    Environment  = INTEGER;

FUNCTION GetHaltVector: LONGINT;
{ return halt vector }

PROCEDURE SetHaltVector(v: LONGINT);
{ halt vector <-- v }

{-----
* TYPEs and FUNCTIONs for converting between SANE Extended formats
-----}
TYPE
    Extended96 = ARRAY [0..5] OF INTEGER;

FUNCTION X96toX80(x: Extended96): EXTENDED;
{ 96 bit x in 80 bit extended format }

FUNCTION X80toX96(x: EXTENDED): Extended96;
{ 80 bit x in 96 bit extended format }

{-----
* SANE library functions
-----}
FUNCTION Log2(x: EXTENDED): EXTENDED;
{ base-2 log }

FUNCTION Ln1(x: EXTENDED): EXTENDED;
{ ln(1+x) }

FUNCTION Exp2(x: EXTENDED): EXTENDED;
{ base-2 exponential }

FUNCTION Exp1(x: EXTENDED): EXTENDED;
{ exp(x) - 1 }

FUNCTION Tan(x: EXTENDED): EXTENDED;
{ tangent }

{$ENDC}

```

```

{=====*
* The common interface for the SANE library *
*=====*}

CONST
  DecStrLen = 255;
  SigDigLen = 20; { for 68K; use 28 in 6502 SANE }

TYPE
{-----}
* Types for handling decimal representations.
{-----}
  DecStr = STRING[DecStrLen];

  CStrPtr = ^CHAR;

  Decimal = RECORD
    sgn: 0..1;
    exp: INTEGER;
    sig: STRING[SigDigLen]
  END;

  DecForm = RECORD
    style: (FloatDecimal, FixedDecimal);
    digits: INTEGER;
  END;

{-----}
* Ordering relations.
{-----}
  RelOp = (GreaterThan, LessThan, EqualTo, Unordered);

{-----}
* Inquiry classes.
{-----}
  NumClass = (SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum);

  RoundDir = (ToNearest, Upward, Downward, TowardZero);

  RoundPre = (ExtPrecision, DblPrecision, RealPrecision);

```

```

{ *-----*
* The functions and procedures of the SANE library      *
*-----* }

{-----
* Conversions between numeric binary types.
-----}

FUNCTION Num2Integer(x: EXTENDED): INTEGER;

FUNCTION Num2Longint(x: EXTENDED): LONGINT;

FUNCTION Num2Real(x: EXTENDED): real;

FUNCTION Num2Double(x: EXTENDED): DOUBLE;

FUNCTION Num2Extended(x: EXTENDED): EXTENDED;

FUNCTION Num2Comp(x: EXTENDED): comp;

{-----
* Conversions between binary and decimal.
-----}

PROCEDURE Num2Dec(f: DecForm; x: EXTENDED; VAR d: Decimal);
{ d <-- x according to format f }

FUNCTION Dec2Num(d: Decimal): EXTENDED;
{ Dec2Num <-- d }

PROCEDURE Num2Str(f: DecForm; x: EXTENDED; VAR s: DecStr);
{ s <-- x according to format f }

FUNCTION Str2Num(s: DecStr): EXTENDED;
{ Str2Num <-- s }

```

```

{-----}
* Conversions between decimal formats.
{-----}

PROCEDURE Str2Dec(s: DecStr; VAR Index: INTEGER; VAR d: Decimal;
                VAR ValidPrefix: BOOLEAN);
{ On input Index is starting index into s, on output Index is
  one greater than index of last character of longest numeric
  substring;
  d <-- Decimal rep of longest numeric substring;
  ValidPrefix <-- "s, beginning at Index, contains valid numeric
  string or valid prefix of some numeric string" }

PROCEDURE CStr2Dec(s: CStrPtr; VAR Index: INTEGER; VAR d: Decimal;
                 VAR ValidPrefix: BOOLEAN);
{ Str2Dec for character buffers or C strings instead of Pascal
  strings: the first argument is the the address of a character
  buffer and ValidPrefix <-- "scanning ended with a null byte" }

PROCEDURE Dec2Str(f: DecForm; d: Decimal; VAR s: DecStr);
{ s <-- d according to format f }

{-----}
* Arithmetic, auxiliary, and elementary functions.
{-----}

FUNCTION Remainder(x, y: EXTENDED; VAR quo: INTEGER): EXTENDED;
{ Remainder <-- x rem y; quo <-- low-order seven bits of integer
  quotient x/y so that -127 < quo < 127 }

FUNCTION Rint(x: EXTENDED): EXTENDED;
{ round to integral value }

FUNCTION Scalb(n: INTEGER; x: EXTENDED): EXTENDED;
{ scale binary; Scalb <-- x * 2^n }

FUNCTION Logb(x: EXTENDED): EXTENDED;
{ Logb <-- unbiased exponent of x }

FUNCTION CopySign(x, y: EXTENDED): EXTENDED;
{ CopySign <-- y with sign of x }

FUNCTION NextReal(x, y: real): real;

FUNCTION NextDouble(x, y: DOUBLE): DOUBLE;

FUNCTION NextExtended(x, y: EXTENDED): EXTENDED;
{ return next representable value from x toward y }

```

```

FUNCTION XpwrI(x: EXTENDED; i: INTEGER): EXTENDED;
{ XpwrI <-- x^i }

FUNCTION XpwrY(x, y: EXTENDED): EXTENDED;
{ XpwrY <-- x^y }

FUNCTION Compound(r, n: EXTENDED): EXTENDED;
{ Compound <-- (1+r)^n }

FUNCTION Annuity(r, n: EXTENDED): EXTENDED;
{ Annuity <-- (1 - (1+r)^(-n)) / r }

FUNCTION RandomX(VAR x: EXTENDED): EXTENDED;
{ returns next random number and updates argument;
  x integral, 1 <= x <= (2^31)-2 }

{-----}
* Inquiry routines.
{-----}

FUNCTION ClassReal(x: real): NumClass;

FUNCTION ClassDouble(x: DOUBLE): NumClass;

FUNCTION ClassComp(x: comp): NumClass;

FUNCTION ClassExtended(x: EXTENDED): NumClass;
{ return class of x }

FUNCTION SignNum(x: EXTENDED): INTEGER;
{ 0 if sign bit clear, 1 if sign bit set }

{-----}
* NaN function.
{-----}

FUNCTION NAN(i: INTEGER): EXTENDED;
{ returns NaN with code i }

```

```

{-----
* Environment access routines.
-----}

PROCEDURE SetException(e: Exception; b: BOOLEAN);
{ set e flags if b is true, clear e flags otherwise; may cause halt }

FUNCTION TestException(e: Exception): BOOLEAN;
{ return true if any e flag is set, return false otherwise }

PROCEDURE SetHalt(e: Exception; b: BOOLEAN);
{ set e halt enables if b is true, clear e halt enables otherwise }

FUNCTION TestHalt(e: Exception): BOOLEAN;
{ return true if any e halt is enabled, return false otherwise }

PROCEDURE SetRound(r: RoundDir);
{ set rounding direction to r }

FUNCTION GetRound: RoundDir;
{ return rounding direction }

PROCEDURE SetPrecision(p: RoundPre);
{ set rounding precision to p }

FUNCTION GetPrecision: RoundPre;
{ return rounding precision }

PROCEDURE SetEnvironment(e: Environment);
{ set environment to e }

PROCEDURE GetEnvironment(VAR e: Environment);
{ e <-- environment }

PROCEDURE ProcEntry(VAR e: Environment);
{ e <-- environment; environment <-- IEEE default env }

PROCEDURE ProcExit(e: Environment);
{ temp <-- exceptions; environment <-- e;
  signal exceptions in temp }

{-----
* Comparison routine.
-----}

FUNCTION Relation(x, y: EXTENDED): RelOp;
{ return Relation such that "x Relation y" is true }

```

END.

C SANE extensions

With the extensions described below (and a SANE library), conventional Kernighan and Ritchie-style C supports SANE. Future versions of SANE C may have minor changes to conform to the ANSI C standard.

Data types

The usual C data types, float and double, and two new C types correspond to the four SANE numeric data types, as shown in Table A-2.

Table A-2
C data types

C name	SANE type
float	IEEE single
double	IEEE double
comp	SANE comp
long double	IEEE extended
extended	IEEE extended

Constants

Decimal-to-binary conversions for numeric constants are done at compile time in the IEEE default numeric environment—see “Numeric Environment,” later in this section. Exceptions that arise from those conversions are not signaled at run time.

Expressions

The SANE types float, double, comp, and extended can be mixed in expressions with each other and with integer types in the same manner that float and double can be mixed in conventional C. An expression consisting solely of a SANE-type (that is, float, double, comp, or extended) variable, constant, or function is of type extended. An expression formed by subexpressions and an arithmetic operation is of type extended if either of its subexpressions is of that type. Extended expressions are evaluated using extended-precision SANE arithmetic, with conversions to extended type generated automatically as needed. Parentheses in extended-type expressions are honored. Initialization of external and static variables, which may include expression evaluation, is done at compile time; all other evaluation of extended expressions is done at run time.

Comparisons

In IEEE Standard arithmetic, the result of a comparison involving a NaN operand is unordered. Thus, the usual trichotomy of numbers is expanded to less, greater, equal, and unordered. The C relational operators (<, <=, ==, >, >=, !=) retain their traditional interpretations. Note, however, that the negation of *a less than b* is not *a greater than or equal to b* but *(a greater than or equal to b) OR (a and b unordered)*.

Functions

A numeric actual parameter passed by value is an expression and hence is of extended or integer type. All extended-type arguments are passed as extended values. Similarly, all results of functions declared float, double, comp, or extended are returned in the extended format.

Input/output

In addition to the usual syntax accepted for numeric input, the standard C library function `scanf` recognizes `INF` as Infinity and `NAN` as a NaN. `NAN` may be followed by parentheses, which may contain an integer (a code indicating the NaN's origin). `INF` and `NAN` are optionally preceded by a sign and are case insensitive. `scanf` specifiers for SANE types extend conventional C as follows: conversion characters *f*, *e*, and *g* indicate type float; *lf*, *le*, and *lg* indicate type double; *mf*, *me*, and *mg* indicate type comp; and *ne*, *nf*, and *ng* indicate type extended.

❖ *ANSI C note:* The `scanf` specifiers for type extended may be different for SANE ANSI C.

`printf` writes Infinities as `[-]INF` and NaNs as `[-]NAN(ddd)`, where *ddd* is the NaN code.

Numeric environment

The numeric environment contains rounding direction, rounding precision, halt enables, and exception flags. IEEE Standard defaults—rounding to nearest, rounding to extended precision, and all halts disabled—are in effect for compile-time arithmetic (including decimal-to-binary conversion). Each program begins with these defaults and with all exception flags clear. Functions for managing the environment are included in the run-time SANE library. Optimizing compilers should not rearrange the order of floating-point evaluation in any way that might change either the computed value or the side effects (such as exception flag settings) from what may have been intended in the source code.

C SANE library

Following is the interface for a C library using the SANE packages. This interface is part of Apple's MPW C; other C language products that support SANE have similar interfaces.

In this interface, `short` refers to 16-bit signed integers and `long` refers to 32-bit signed integers.

❖ *Note:* This C interface includes sections for machines with and without an MC68881 coprocessor. Conditional compilation selects the appropriate section based on the settings of compiler option `-MC68881`.

```
/*
    SANE.h - Standard Apple Numeric Environment

    C Interface to the Macintosh Libraries
    Copyright Apple Computer, Inc.    1985-1988
    All rights reserved.
*/

#ifndef __SANE__
#define __SANE__

#ifdef mc68881
/* MC68881 Exceptions */

#define INEXACT            8
#define DIVBYZERO        16
#define UNDERFLOW        32
#define OVERFLOW         64
#define INVALID          128
#define CURINEX1         256
#define CURINEX2         512
#define CURDIVBYZERO     1024
#define CURUNDERFLOW     2048
#define CUROVERFLOW      4096
#define CUROPERRROR      8192
#define CURSIGNAN        16384
#define CURBSONUNOR      32768

typedef long exception;
typedef struct trapvector {
    void (*unordered)();
    void (*inexact)();
    void (*divbyzero)();
    void (*underflow)();
    void (*operror)();
    void (*overflow)();
    void (*signan)();

```

```

} trapvector;
typedef struct environment {
    long FPCR;
    long FPSR;
} environment;

environment IEEEDEFAULTENV = {0L,0L};

void gettrapvector(trapvector);
void settrapvector(trapvector);

#else

/* Software SANE Exceptions */

#define INVALID        1
#define UNDERFLOW     2
#define OVERFLOW       4
#define DIVBYZERO      8
#define INEXACT        16
#define IEEEDEFAULTENV 0

typedef short exception;
typedef short environment;
typedef void (*haltvector)();

haltvector gethaltvector();
void sethaltvector(haltvector);

#endif

/* Decimal Representation Constants */

#define SIGDIGLEN      20    /* significant decimal digits */
#define DECSTROUTLEN  80    /* max length for decimal string output */

/* Decimal Formatting Styles */

#define FLOATDECIMAL  0
#define FIXEDDECIMAL  1

/* Ordering Relations */

#define GREATERTHAN  0
#define LESSTHAN     1
#define EQUALTO      2
#define UNORDERED    3

/* Inquiry Classes */

#define SNAN          0
#define QNAN          1
#define INFINITE      2

```

```

#define ZERONUM      3
#define NORMALNUM   4
#define DENORMALNUM 5

/* Rounding Directions */

#define TONEAREST   0
#define UPWARD      1
#define DOWNWARD    2
#define TOWARDZERO  3

/* Rounding Precisions */

#define EXTPRECISION 0
#define DBLPRECISION 1
#define FLOATPRECISION 2

typedef short relop;      /* relational operator */
typedef short numclass;  /* inquiry class */
typedef short rounddir;  /* rounding direction */
typedef short roundpre;  /* rounding precision */
typedef struct decimal {
    char sgn, unused;     /* sign 0 for +, 1 for - */
    short exp;           /* decimal exponent */
    struct {unsigned char length, text[SIGDIGLEN], unused;} sig;
                        /* significant digits */
} decimal;

typedef struct decform {
    char style, unused;  /* FLOATDECIMAL or FIXEDDECIMAL */
    short digits;
} decform;

/* Conversions between Binary and Decimal Records */

void num2dec(decform * f, extended x, decimal * d);
                        /* d <-- x, according to format f */
extended dec2num(decimal * d);
                        /* returns d as extended */

/* Conversions between Decimal Records and ASCII Strings */

void dec2str(decform * f, decimal * d, char * s);
                        /* s <-- d, according to format f */
void str2dec(char * s, short * ix, decimal * d, short * vp);
                        /* on input ix is starting index into s, on */
                        /* output ix is one greater than index of last */
                        /* character of longest numeric substring; */
                        /* boolean vp = "s beginning at given ix is a */
                        /* valid numeric string or a valid prefix of */
                        /* some numeric string" */

```

```

/* Arithmetic, Auxiliary, and Elementary Functions */

extended fabs(extended x); /* absolute value */
extended remainder(extended x, extended y, short * quo);
/* IEEE remainder; quo <-- 7 low */
/* order bits of integer quotient x/y, */
/* -127 <= quo <= 127 */

extended sqrt(extended x); /* square root */
extended rint(extended x); /* round to integral value */
extended scalb(short n, extended x);
/* binary scale: x * 2^n; */
/* first coerces n to short */
extended logb(extended x); /* binary log: binary exponent of */
/* normalized x */

extended copysign(extended x, extended y);
/* returns y with sign of x */

extended nextfloat(extended x, extended y);
/* next float representation after */
/* (float) x in direction of (float) y */

extended nextdouble(extended x, extended y);
/* next double representation after */
/* (double) x in direction of (double) y */

extended nextextended(extended x, extended y);
/* next extended representation after */
/* x in direction of y */

extended log2(extended x); /* base-2 log */
extended log(extended x); /* base-e log */
extended log1(extended x); /* log(1 + x) */
extended exp2(extended x); /* base-2 exponential */
extended exp(extended x); /* base-e exponential */
extended expl(extended x); /* exp(x) - 1 */
extended power(extended x, extended y);
/* general exponential: x ^ y */

extended ipower(extended x, short i);
/* integer exponential: x ^ i */

extended compound(extended r, extended n);
/* compound: (1 + r) ^ n */

extended annuity(extended r, extended n);
/* annuity: (1 - (1 + r) ^ (-n)) / r */

extended tan(extended x); /* tangent */
extended sin(extended x); /* sine */
extended cos(extended x); /* cosine */
extended atan(extended x); /* arctangent */
extended randomx(extended * x);
/* returns next random number; updates x */
/* x must be integral, 1 <= x <= 2^31 - 2 */

```

```

/* Inquiry Routines */

numclass classfloat(extended x); /* class of (float) x */
numclass classdouble(extended x); /* class of (double) x */
numclass classcomp(extended x); /* class of (comp) x */
numclass classextended(extended x); /* class of x */

long signnum(extended x); /* returns 0 for +, 1 for - */

/* Environment Access Routines */
/* An exception variable encodes the exceptions whose sum is its value */

void setexception(exception e, long s);
/* clears e flags if s is 0, sets */
/* e flags otherwise; may cause halt */
long testexception(exception e); /* returns 1 if any e flag is set, */
/* returns 0 otherwise */
void sethalt(exception e, long s); /* disables e halts if s is 0, enables e */
/* halts otherwise */
long testhalt(exception e); /* returns 1 if any e halt is enabled, */
/* returns 0 otherwise */
void setround(rounddir r); /* sets rounding direction to r */
rounddir getround(); /* returns current rounding direction */
void setprecision(roundpre p); /* sets rounding precision to p */
roundpre getprecision(); /* returns current rounding precision */
void setenvironment(environment e); /* sets SANE environment to e */
void getenvironment(environment * e);
/* e <-- SANE environment */
void procentry(environment * e); /* e <-- SANE environment; */
/* SANE environment <-- IEEEdefaultenv */
void procexit(environment e); /* temp <-- current exceptions; */
/* SANE environment <-- e; */
/* signals exceptions in temp */
haltvector gethaltvector(); /* returns SANE halt vector */
void sethaltvector(haltvector v); /* SANE halt vector <-- v */

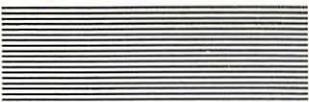
/* Comparison Routine */

relop relation(extended x, extended y);
/* returns relation such that */
/* "x relation y" is true */

/* NaNs and Special Constants */

extended nan(unsigned char c); /* returns NaN with code c */
extended inf(); /* infinity */
extended pi(); /* pi */

```



Appendix B



The SANE Engines: Availability

Different kinds of Apple computers require different implementations of SANE. Here is information about how to obtain the SANE engines for the different machines, based on the type of microprocessor they have.

SANE for the 6502

You can obtain the object code for the 6502 SANE software (FP6502, Elems6502, and DecStr6502) by writing to the following address and asking for 6502 assembly-language SANE:

Apple Software Licensing
20525 Mariani Avenue
Cupertino, CA 95014

Included with the object code are the macros mentioned in this manual and complete instructions for use with ProDOS[®] and DOS assemblers and with the Apple Pascal Assembler. The code can be used on any Apple II computer with at least 64K of memory, or on an Apple III.

SANE for the 65C816

Support for SANE is built-in on all Apple computers that use the 65C816: the SANE functions are part of the system tools.

Most development systems, such as the Apple IIGS Programmer's Workshop (APW), provide macros for convenient use of SANE.

SANE for the MC68000

Support for SANE is built-in on all Apple computers that use the MC68000: the SANE functions are part of the system tools. The Package Manager automatically makes available the object code for the 68000 SANE software (FP68K, Elems68K, and DecStr68K), as needed.

If you are using the Macintosh Programmer's Workshop Assembler, you should include the file SANEMacs, which contains the macros mentioned in this manual. Other assemblers provide similar macros; consult the accompanying manual for instructions on using them.

SANE for the MC68020 and MC68881

Support for SANE is built-in on all Apple computers that use the MC68020: the SANE functions are part of the system tools. The Package Manager makes available the MC68881 versions of the floating-point (SANE) packages.

The Macintosh Programmer's Workshop, version 2.0, supports the MC68881 in its assembler as well as its high-level languages. Other development systems for Apple computers also support SANE; consult the accompanying manuals for details.

❖ *Note:* There are non-Apple hardware products for adding the MC68020 and MC68881 processors to MC68000-based Apple computers. Those products also include support for SANE; consult the accompanying manuals.



Appendix C



Porting Programs to SANE

Porting programs to run in the Apple numerics environment is easier than porting to other computers. Expressions that produce good results on other computers usually give at least as good results with SANE.

This appendix contains information of interest to programmers who are porting programs from some other machine to run on an Apple computer with SANE. If you are such a programmer and you think you're getting problems because of differences in numerics, you should read this appendix.

Semantics of arithmetic evaluation

When you translate programs from one language to another, you must be aware of the hidden pitfalls in translation. For example, certain operations in different languages may have similar syntax without being similar semantically. Here's an example of similar functions with different syntaxes:

- in Fortran, `SIGN(A, B)` (two operands)
- in BASIC, `SIGN(A)` (one operand)

Different languages have different ways of dealing with mixed integers and reals. For example, Fortran truncates integer quotients to integers, so $3/7 = 0$ (you have to write $3.0/7.0$ for a fraction). The programmer doing the translating must be aware that such expressions are not independent of language.

In the case of conversion from real to integer, different languages have different semantics. For example, in Fortran, assigning a floating-point value to an integer rounds toward zero.

Here are the operations used to truncate a real to an integer in three different languages:

- in C: assignments and casts
- in Fortran: AINT, INT
- in Pascal: Trunc

Mixed formats

On some other computers, the formats for single and double are identical except for length. On those machines, for arguments passed by address, a calling routine can store data in one format and a called routine can read data in the other format without apparent error.

If you have a program that exploits this confusion, you'll have to revise it before you can run it on a machine that uses SANE. (Type checking is no help here; if the discrepancy was such that type checking could detect it, the original compiler would have caught it.)

Floating-point precision

You should be aware of differences between the floating-point precision on the original machine and on the target machine.

Wider precision

Some computers have floating-point formats that have more precision than the current SANE extended formats. These include the VAX H format, the IBM Q format, and the HP Spectrum quad format. Programs use these wide formats for computation involving input data from a narrower format to minimize the occurrence of overflow and underflow and to preserve accuracy. The IEEE extended data type was designed to be sufficiently large to provide these benefits in most cases. However, for a given algorithm, it is difficult to be sure that the SANE extended format is adequate, so the possibility of problems arising with programs that used formats wider than SANE extended should not be ignored.

CDC and Cray computers have a single format that is wider than SANE single and a double format that is wider than SANE extended. When porting code from those machines, you should consider changing type declarations from single to double and from double to extended.

Double rounding

SANE implementations evaluate expressions with real variables in extended format, then round the results. This sometimes leads to double rounding, but the double rounding doesn't increase errors in the results, because the difference in magnitudes is so large that the second rounding swamps errors in the first.

Computed error bounds

In applications that compute convergence criteria and error bounds by doing arithmetic, evaluation in extended can cause problems. For example, with x and y of type single, you might have your program evaluate

$$z := ((x + y) - x) - y$$

to find the error due to rounding $(x + y)$. This expression gives 0 on Apple machines, so it can't be used to find the rounding error.

Using such expressions is bad programming practice because it causes many other machines to malfunction. If you have this problem after porting your code to SANE, you should find out the evaluation rules for the original environment so that you can make appropriate changes to the code.

The rules of evaluation

There are many possible evaluation rules. Here are three reasonable ones:

- Rule 1: Round the result to the wider of the two operand formats.
- Rule 2: Round the result to the widest available format.
- Rule 3: Round the result to the widest format in the expression.

Rule 1 is instant rounding: It is the rule on computers with many registers the same width as memory. This rule has been used by IBM and CDC Fortran since 1963. It is not part of the Fortran standard, though often thought to be.

Rule 2 is what Apple does by evaluating in extended precision. Other machines using this approach include the PDP-11C (using double precision) and floating-point coprocessors such as the 8087 and the MC68881. This approach doesn't take best advantage of machines with separate processing units for each floating-point type.

Rule 3 is the way you do it when computing by hand. It was the rule in Fortran until 1963. By this rule, if you see an expression with mixed precision, you assume the user wants the widest visible precision.

With SANE, you can write code to simulate any of these rules. To simulate Rule 1, use separate assignments when computing subexpressions. To simulate Rule 3, you have to examine the expression, find the widest format, and set the rounding precision accordingly.

For transported code, either you have to understand the programmer's tricks or you have to mimic the way rounding works on the programmer's machine. With SANE, you can set the precision and rounding direction to mimic other machines.

The invalid-operation flag

For the many improvements SANE provides, there is a price to be paid: Some old things no longer work the way they used to.

Many computers used to stop on an invalid operation, such as $0/0$. Programmers have made the best of this and not bothered to test in advance for values that could cause an invalid operation. It's better to stop than to give a plausible but incorrect answer.

When a program written that way runs on SANE, it may produce a NaN where it formerly would have stopped. The NaN might cause the program to take an unplanned branch and thus produce an erroneous answer. Because the program doesn't test for invalid operations, the user won't know whether the answers the program finally delivers have been influenced by exceptional events that formerly would have stopped the computer.

Programs sometimes contain code that depends on an ill-documented effect or one that varies from machine to machine. If you have inherited such a program and you don't know what it does about exceptional conditions, here are some possible strategies:

- Insert tests on operands that could cause invalid operations.
- Change the program to make sure that NaNs propagate as NaNs, rather than as plausible answers.
- After evaluations, add code to test the invalid flag and deliver a meaningful result or message and then lower the flag.

If you have a program with code you can't change and you distrust the results it gives when invalid operations occur, you should set halts to stop on those invalid operations and set the environment to simulate the environment in which the program was designed to run.

Table D-1
Format diagram symbols

Symbol	Description
v	Value of number
s	Sign bit
e	Biased exponent
i	Explicit one's bit (extended type only)
f	Fraction
d	Nonsign bits (comp type only)

Single: 32 bits

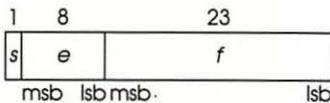


Figure D-2
Single format

Table D-2
Values of single-format numbers (32 bits)

Biased exponent e	Fraction f	Value v	Class of v
$0 < e < 255$	(any)	$v = (-1)^s \times 2^{(e-127)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-126)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 255$	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 255$	$f \neq 0$	v is a NaN	NaN

Double: 64 bits

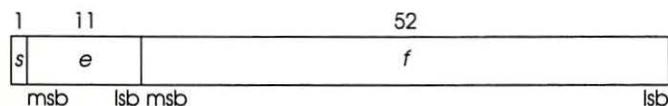


Figure D-3
Double format

Table D-3
Values of double-format numbers (64 bits)

Biased exponent e	Fraction f	Value v	Class of v
$0 < e < 2047$	(any)	$v = (-1)^s \times 2^{(e-1023)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-1022)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 2047$	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 2047$	$f \neq 0$	v is a NaN	NaN

Comp: 64 bits

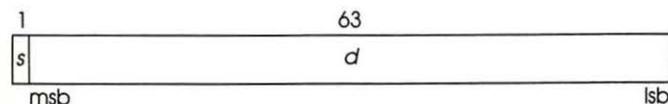


Figure D-4
Comp format

Table D-4
Values of comp-format numbers (64 bits)

Sign bit s	Nonsign bits d	Value v
$s = 1$	$d = 0$	v is the unique NaN in the comp type.
$s = 1$	$d \neq 0$	v is the two's-complement value of the 64-bit representation.
$s = 0$	(any)	v is the two's-complement value of the 64-bit representation.

Extended: 80 bits

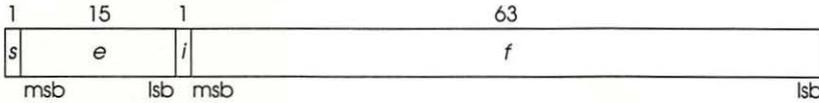


Figure D-5
Extended format

Table D-5
Values of extended-format numbers (80 bits)

Biased exponent e	Integer i	Fraction f	Value v	Class of v
$0 \leq e \leq 32766$	1	(any)	$v = (-1)^s \times 2^{(e-16383)} \times (1.f)$	Normalized
$0 \leq e \leq 32766$	0	$f \neq 0$	$v = (-1)^s \times 2^{(e-16383)} \times (0.f)$	Denormalized
$0 \leq e \leq 32766$	0	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 32767$	(any)	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 32767$	(any)	$f \neq 0$	v is a NaN	NaN

Operations

Tables D-6 through D-10 define the abbreviations and symbols used in the operation tables that follow. Tables D-11 through D-24 show the mnemonic name, opword, operand types, and exceptions for each operation.

In the opword, the first byte is the operand format code and the second is the operation code. For some operations, the first byte (xx) of the opword is ignored.

Abbreviations and symbols

The symbols and abbreviations in this section closely parallel those in the text, although some are shortened. In some cases, the same symbol has different meanings, depending on context; for example, in data types, x stands for *extended*; in exceptions, x stands for *inexact*.

Operands

Table D-6
Operands

Abbreviation	Description
DST	Destination operand (passed by address)
DST2	Second destination operand (passed by address)
SRC	Source operand (passed by address)
SRC	Source operand (passed by value)
SRC2	Second source operand (passed by address)

Note: Push operands in the order SRC2 (if any), SRC, DST2 (if any), DST.

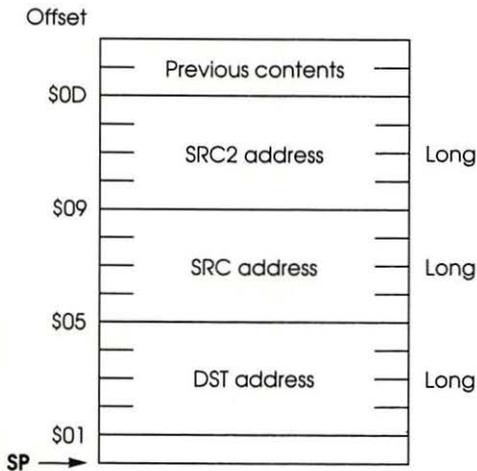


Figure D-6
SANE operands on the 65C816 stack

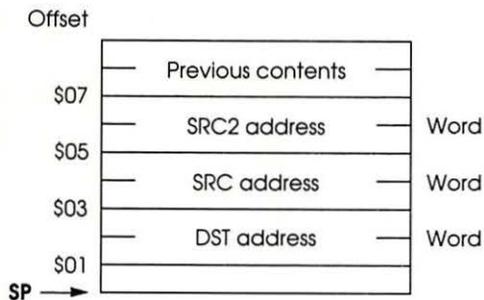


Figure D-7
SANE operands on the 6502 stack

Data types

Table D-7
Data types

Abbreviation	Description
X	Extended (80 bits, passed by address)
D	Double (64 bits, passed by address)
S	Single (32 bits, passed by address)
I	Integer (16 bits, passed by address)
I	Integer (16 bits, passed by value)
L	Long integer (32 bits, passed by address)
C	Comp (64 bits, passed by address)
Dec	Decimal record (33 bytes, passed by address)
DecForm	Decform record (4 bytes, passed by address)
DecStr	Pascal or C decimal string

65C816 and 6502 processor registers

Table D-8
65C816 and 6502 processor registers

Abbreviation	Description
Xreg	X register*
Yreg	Y register*
Preg	Processor Status register
Nbit	Negative bit of Processor Status register
Zbit	Zero bit of Processor Status register

- * 8-bit values. For the 65C816, these values occupy the low-order bits of 16-bit registers; high-order bits are set according to Figure D-8.

Exceptions

Table D-9
Exceptions

Abbreviation	Description
Xcps	Exceptions, collectively
I	Invalid operation
U	Underflow
O	Overflow
D	Divide-by-zero
X	Inexact

Environment and halts

Table D-10
Environment and halts

Abbreviation	Description
EnvWrđ	SANE Environment word (16-bit integer)
HltVctr	SANE halt vector (32-bit address*)

* The halt vector in 6502 SANE is a 16-bit address.

Arithmetic operations (entry points FP816, FP6502)

Table D-11

Arithmetic operations (entry points FP816, FP6502)

Name	Opword	Operation	SRC type	Exceptions
FADDX	\$0000	$DST \leftarrow DST + SRC$	X	I - O - X
FADDD	\$0100	$DST \leftarrow DST + SRC$	D	I - O - X
FADDS	\$0200	$DST \leftarrow DST + SRC$	S	I - O - X
FADDC	\$0500	$DST \leftarrow DST + SRC$	C	I - O - X
FADDI	\$0400	$DST \leftarrow DST + SRC$	I	I - O - X
FADDL	\$0300	$DST \leftarrow DST + SRC$	L	I - O - X
FSUBX	\$0002	$DST \leftarrow DST - SRC$	X	I - O - X
FSUBD	\$0102	$DST \leftarrow DST - SRC$	D	I - O - X
FSUBS	\$0202	$DST \leftarrow DST - SRC$	S	I - O - X
FSUBC	\$0502	$DST \leftarrow DST - SRC$	C	I - O - X
FSUBI	\$0402	$DST \leftarrow DST - SRC$	I	I - O - X
FSUBL	\$0302	$DST \leftarrow DST - SRC$	L	I - O - X
FMULX	\$0004	$DST \leftarrow DST * SRC$	X	I U O - X
FMULD	\$0104	$DST \leftarrow DST * SRC$	D	I U O - X
FMULS	\$0204	$DST \leftarrow DST * SRC$	S	I U O - X
FMULC	\$0504	$DST \leftarrow DST * SRC$	C	I - O - X
FMULI	\$0404	$DST \leftarrow DST * SRC$	I	I - O - X
FMULL	\$0304	$DST \leftarrow DST * SRC$	L	I - O - X
FDIVX	\$0006	$DST \leftarrow DST / SRC$	X	I U O D X
FDIVD	\$0106	$DST \leftarrow DST / SRC$	D	I U O D X
FDIVS	\$0206	$DST \leftarrow DST / SRC$	S	I U O D X
FDIVC	\$0506	$DST \leftarrow DST / SRC$	C	I U - D X
FDIVI	\$0406	$DST \leftarrow DST / SRC$	I	I U - D X
FDIVL	\$0306	$DST \leftarrow DST / SRC$	L	I U - D X
FSQRTX	\$0012	$DST \leftarrow \text{sqrt}(DST)$	-	I - - - X
FREMX [†]	\$000C	$DST \leftarrow DST \text{ rem } SRC$	X	I - - - -
FREMD [†]	\$010C	$DST \leftarrow DST \text{ rem } SRC$	D	I - - - -
FREMS [†]	\$020C	$DST \leftarrow DST \text{ rem } SRC$	S	I - - - -
FREMC [†]	\$050C	$DST \leftarrow DST \text{ rem } SRC$	C	I - - - -
FREMI [†]	\$040C	$DST \leftarrow DST \text{ rem } SRC$	I	I - - - -
FREML [†]	\$030C	$DST \leftarrow DST \text{ rem } SRC$	L	I - - - -
FRINTX	\$0014	$DST \leftarrow \text{rnd}(DST)$	-	I - - - X
FTINTX	\$0016	$DST \leftarrow \text{chop}(DST)$	-	I - - - X

Note: For all arithmetic operations, the destination is of type extended.

[†] Also, the 7 low-order bits of x_{reg} contain the corresponding bits of $|n|$,
 $Nbit \leftarrow \text{sign bit of } n$, and $\text{sign}(y_{reg}) \leftarrow \text{sign of } n$,
 where $n = \text{integer quotient } DST/SRC$.

Auxiliary routines (entry points FP816, FP6502)

Table D-12

Auxiliary routines (entry points FP816, FP6502)

Name	Opword	Operation	SRC type	Exceptions
FSCALBX	\$0018	$DST \leftarrow DST \times 2^{SRC}$	I	I U O - X
FLOGBX	\$001A	$DST \leftarrow \log_b(DST)$	-	I - - D -
FNEGX	\$000D	$DST \leftarrow -DST$	-	- - - - -
FABSX	\$000F	$DST \leftarrow DST $	-	- - - - -
FCPYSGNX	\$0011	$DST \leftarrow DST$ with sign of SRC	X	- - - - -
FCPYSGND	\$0111	$DST \leftarrow DST$ with sign of SRC	D	- - - - -
FCPYSGNS	\$0211	$DST \leftarrow DST$ with sign of SRC	S	- - - - -
FCPYSGNC	\$0511	$DST \leftarrow DST$ with sign of SRC	C	- - - - -
FCPYSGNI	\$0411	$DST \leftarrow DST$ with sign of SRC	I	- - - - -
FCPYSGNL	\$0311	$DST \leftarrow DST$ with sign of SRC	L	- - - - -
FNEXTX	\$001E	$DST^\dagger \leftarrow$ Nextafter DST toward SRC	X [†]	I U O - X
FNEXTD	\$011E	$DST^\dagger \leftarrow$ Nextafter DST toward SRC	D [†]	I U O - X
FNEXTS	\$021E	$DST^\dagger \leftarrow$ Nextafter DST toward SRC	S [†]	I U O - X

Note: For most auxiliary routines, the destination is of type extended.

[†] For Nextafter, SRC and DST are of the same type.

Conversions (entry points FP816, FP6502)

Table D-13

Binary-to-binary conversions (entry points FP816, FP6502)

Name	Opword	Operation	DST type	SRC type	Exceptions
FX2X	\$0010	DST ← SRC	X	X	I - - - -
FX2D	\$0110	DST ← SRC	D	X	I U O - X
FX2S	\$0210	DST ← SRC	S	X	I U O - X
FX2C	\$0510	DST ← SRC	C	X	I - - - X
FX2I	\$0410	DST ← SRC	I	X	I - - - X
FX2L	\$0310	DST ← SRC	L	X	I - - - X
FD2X	\$010E	DST ← SRC	X	D	I - - - -
FS2X	\$020E	DST ← SRC	X	S	I - - - -
FC2X	\$050E	DST ← SRC	X	C	- - - - -
FI2X	\$040E	DST ← SRC	X	I	- - - - -
FL2X	\$030E	DST ← SRC	X	L	- - - - -

Table D-14

Binary-to-decimal conversions (entry points FP816, FP6502)

Name	Opword	Operation	DST type	SRC type	SRC2 type	Exceptions
FX2DEC	\$000B	DST ← SRC using SRC2	Dec	X	SRC2	I - - -X
FD2DEC	\$010B	DST ← SRC using SRC2	Dec	D	SRC2	I - - -X
FS2DEC	\$020B	DST ← SRC using SRC2	Dec	S	SRC2	I - - -X
FC2DEC	\$050B	DST ← SRC using SRC2	Dec	C	SRC2	- - - -X
FI2DEC	\$040B	DST ← SRC using SRC2	Dec	I	SRC2	- - - -X
FL2DEC	\$030B	DST ← SRC using SRC2	Dec	L	SRC2	- - - -X

Note: First push SRC2, then SRC, then DST.

Table D-15

Decimal-to-binary conversions (entry points FP816, FP6502)

Name	Opword	Operation	DST type	SRC type	Exceptions
FDEC2X	\$0009	DST ← SRC	X	Dec	- U O - X
FDEC2D	\$0109	DST ← SRC	D	Dec	- U O - X
FDEC2S	\$0209	DST ← SRC	S	Dec	- U O - X
FDEC2C	\$0509	DST ← SRC	C	Dec	I - - - X
FDEC2I	\$0409	DST ← SRC	I	Dec	I - - - X
FDEC2L	\$0309	DST ← SRC	L	Dec	I - - - X

Comparisons (entry points FP816, FP6502)

The comparison operations in Table D-17 return relation information in the P register and in the low bytes of the X and Y registers, as shown in Table D-16.

Table D-16
Relation information

Relation	P-register bit			X register*	Y register*
	Z	N	V		
SRC > DST	0	0	1	\$40	\$40
SRC < DST	0	1	0	\$80	\$80
SRC = DST	1	0	0	\$02	\$00
SRC, DST unordered	0	0	0	\$01	\$01

* 1-byte value, in the low byte of register on the 65C816

Table D-17
Comparisons (entry points FP816, FP6502)

Name	Opword	Operation	DST type	SRC type	Exceptions
FCMPX*	\$0008	SRC <relation> DST	X	X	I - - - -
FCMPD*	\$0108	SRC <relation> DST	D	X	I - - - -
FCMPS*	\$0208	SRC <relation> DST	S	X	I - - - -
FCMPC*	\$0508	SRC <relation> DST	C	X	I - - - -
FCMPI*	\$0408	SRC <relation> DST	I	X	I - - - -
FCMPL*	\$0308	SRC <relation> DST	L	X	I - - - -
FCPXX†	\$000A	SRC <relation> DST	X	X	I - - - -
FCPXD†	\$010A	SRC <relation> DST	D	X	I - - - -
FCPXS†	\$020A	SRC <relation> DST	S	X	I - - - -
FCPXC†	\$050A	SRC <relation> DST	C	X	I - - - -
FCPXI†	\$040A	SRC <relation> DST	I	X	I - - - -
FCPXL†	\$030A	SRC <relation> DST	L	X	I - - - -

* These comparisons don't signal invalid for unordered but do signal invalid for signaling NaN inputs.

† These comparisons signal invalid for unordered.

Inquiries: class and sign (entry points FP816, FP6502)

The classify operations shown in Table D-20 return class information in the low byte of the X register and sign in the low byte of the Y register and in the N bit of the P register, as shown in Tables D-18 and D-19.

Table D-18
Class information

Class	X register*
Signaling NaN	\$FC
Quiet NaN	\$FD
Infinity	\$FE
Zero	\$FF
Normalized	\$00
Denormalized	\$01

* 1-byte value, in the low byte of register on the 65C816

Table D-19
Sign information

Sign	Y register*	N bit
Positive	\$00	0
Negative	\$80	1

* 1-byte value, in the low byte of register on the 65C816

Table D-20
Classify (entry points FP816, FP6502)

Name	Opword	Operation		SRC type	Exceptions
FCLASSX	\$001C	Xreg, Yreg, Nbit ← SRC	X	- - - - -	
FCLASSD	\$011C	Xreg, Yreg, Nbit ← SRC	D	- - - - -	
FCLASSS	\$021C	Xreg, Yreg, Nbit ← SRC	S	- - - - -	
FCLASSC	\$051C	Xreg, Yreg, Nbit ← SRC	C	- - - - -	
FCLASSI	\$041C	Xreg, Yreg, Nbit ← SRC	I	- - - - -	
FCLASSL	\$031C	Xreg, Yreg, Nbit ← SRC	L	- - - - -	

Environmental control (entry points FP816, FP6502)

Table D-21 shows the first byte of the opword as xx to indicate that the byte is ignored.

Table D-21
Environmental control (entry points FP816, FP6502)

Name	Opword	Operation	SRC type	Exceptions
FGETENV	\$xx03	Xreg ← low byte of EnvWrd* Yreg ← high byte of EnvWrd*		- - - - -
FSETENV†	\$xx01	EnvWrd ← SRC	I	- - - - -
FTESTXCP	\$xx1B	Zbit ← SRC Xcps clear	I	- - - - -
FSETXCP	\$xx15	EnvWrd ← EnvWrd OR SRC*256	I	I U O D X
FPROCENTRY	\$xx17	DST ← EnvWrd, EnvWrd ← 0	I	- - - - -
FPROCEXIT	\$xx19	EnvWrd ← SRC OR current Xcps	I	I U O D X

* Figure D-8 shows how these 8-bit values are returned in the 16-bit registers of the 65C816.

† Exceptions set by Set-Environment do not cause halts.

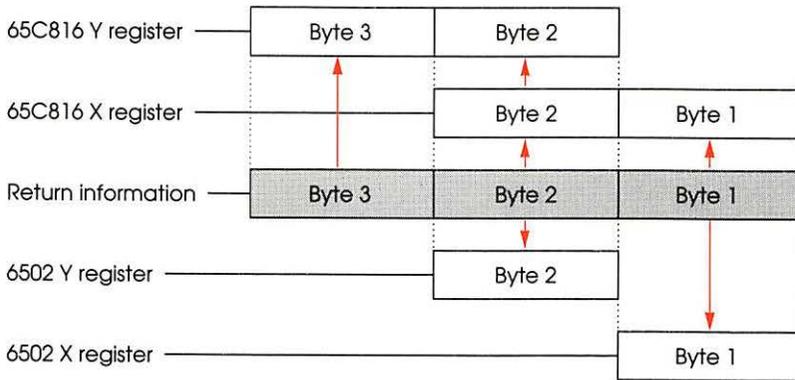


Figure D-8
Data returned in X and Y registers

Halt control (entry points FP816, FP6502)

Table D-22 shows the first byte of the opword as xx to indicate that the byte is ignored.

Table D-22

Halt control (entry points FP816, FP6502)

Name	Opword	Operation	SRC type	Exceptions
FSETHV	\$xx05	HltVctr ← SRC*	I	- - - - -
FGETHV	\$xx07	Xreg ← low bytes of HltVctr† Yreg ← high bytes of HltVctr†		- - - - -

* On the 65C816, SRC is a 32-bit value with a 24-bit address in its low-order bytes; on the 6502, SRC is a 16-bit value.

† Figure D-8 shows how the halt vector is returned. HltVctr is three bytes for the 65C816, two bytes for the 6502.

Elementary functions (entry points Elems816, Elems6502)

Table D-23 shows the first byte of the opword as xx to indicate that the byte is ignored.

Table D-23

Elementary functions (entry points Elems816, Elems6502)

Name	Opword	Operation	DST type	SRC type	SRC2 type	Exceptions
FLNX	\$xx00	DST ← ln(DST)	X	X	-	I - - D X
FLOG2X	\$xx02	DST ← log ₂ (DST)	X	X	-	I - - D X
FLN1X	\$xx04	DST ← ln(1 + DST)	X	X	-	I U - D X
FLOG21X	\$xx06	DST ← log ₂ (1 + DST)	X	X	-	I U - D X
FEXPX	\$xx08	DST ← e ^{DST}	X	X	-	I U O - X
FEXP2X	\$xx0A	DST ← 2 ^{DST}	X	X	-	I U O - X
FEXP1X	\$xx0C	DST ← e ^{DST-1}	X	X	-	I U O - X
FEXP21X	\$xx0E	DST ← 2 ^{DST-1}	X	X	-	I U O - X
FXPWRI	\$xx10	DST ← DST ^{SRC}	X	I	-	I U O D X
FXPWRY	\$xx12	DST ← DST ^{SRC}	X	X	-	I U O D X
FCOMPOUND	\$xx14	DST ← compound(SRC2, SRC)*	X	X	X	I U O D X
FANNUITY	\$xx16	DST ← annuity(SRC2, SRC)*	X	X	X	I U O D X
FATANX	\$xx18	DST ← atan(DST)	X	X	-	I U - - X
FSINX	\$xx1A	DST ← sin(DST)	X	X	-	I U - - X
FCOSX	\$xx1C	DST ← cos(DST)	X	X	-	I U - - X
FTANX	\$xx1E	DST ← tan(DST)	X	X	-	I U - D X
FRANDX†	\$xx20	DST ← randomx(DST)	X	X	-	I U O - X

* SRC2 is the rate; SRC is the number of periods.

† No exceptions for valid arguments.

Scanners and formatter (entry points DecStr816, DecStr6502)

Table D-24 shows the first byte of the opword as xx to indicate that the byte is ignored.

Table D-24

Scanners and formatter (entry points DecStr816, DecStr6502)

Name	Opword	Operation	DST type	DST2 type	SRC type	SRC2 type	Exceptions
FPSTR2DEC	\$xx02	DST2* ← SRC2 beginning at SRC SRC ← new index DST ← valid prefix flag	I	Dec	I	DecStr	- - - - -
FCSTR2DEC	\$xx04	DST2* ← SRC2 beginning at SRC SRC ← new index DST ← valid prefix flag	I	Dec	I	DecStr	- - - - -
FDEC2STR	\$xx03	DST ← SRC according to SRC2	DecStr	-	Dec	DecForm	- - - - -

* Push SRC2, then SRC, then DST2, then DST.

The Environment word

Rounding direction and precision are stored as 2-bit encoded values; exception and halt-enabled flags are set as individual bits. Note that the default environment is represented by the integer value zero.

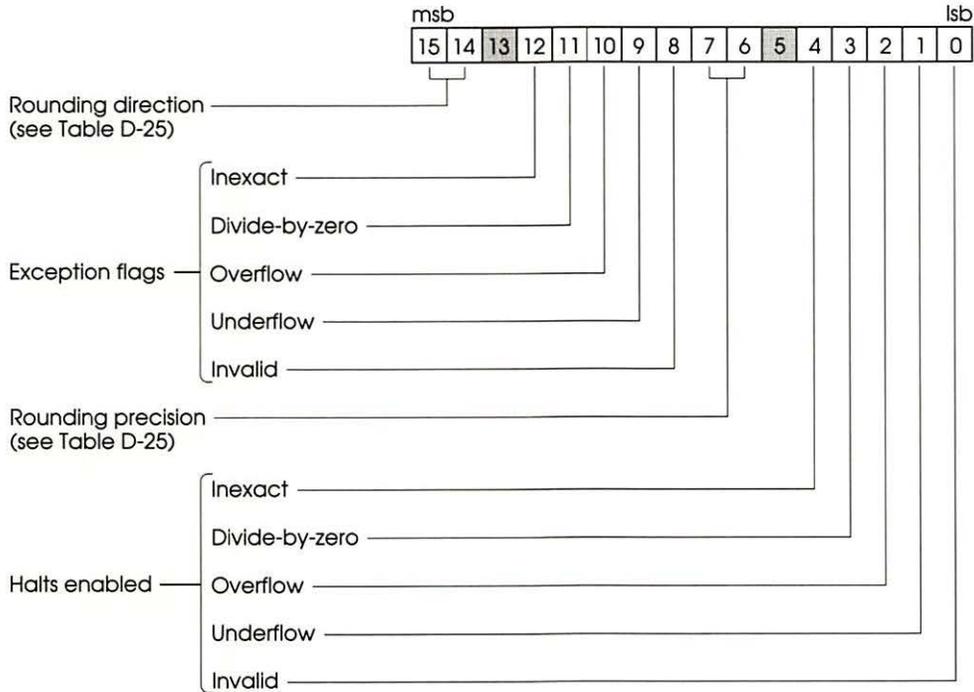


Figure D-9
The Environment word for the 65C816 and 6502

Table D-25
Bits in the Environment word for the 65C816 and 6502

Group name	Mask bits	Mask value	Description
Rounding direction (Bit group \$6000)	15 14 0 0 0 1 1 0 1 1	\$0000 \$4000 \$6000 \$8000	To-nearest Upward Downward Toward-zero
Exception flags (Bit group \$1F00)	12 11 10 9 8 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$1000 \$0800 \$0400 \$0200 \$0100	Inexact Divide-by-zero Overflow Underflow Invalid
Rounding precision (Bit group \$00C0)	7 6 0 0 0 1 1 0 1 1	\$0000 \$0040 \$0080 \$00C0	Extended Double Single (Undefined)
Halts enabled (Bit group \$001F)	4 3 2 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$0010 \$0008 \$0004 \$0002 \$0001	Inexact Divide-by-zero Overflow Underflow Invalid

Note: Bits 5 and 13 are not used.

Appendix E

MC68000 SANE Quick Reference Guide

This guide is your quick reference for the MC68000 SANE engine. It contains diagrams of the SANE formats and tables of the SANE operations and Environment word. The operation tables show the macro names and opwords for all operations.

Formats of SANE types

Each of the diagrams below is followed by a table that gives the rules for evaluating the number v .

In each field of each diagram, the leftmost bit is the msb and the rightmost is the lsb. The SANE engine for the MC68000 uses the convention that the most significant bytes are stored at the lowest addresses.

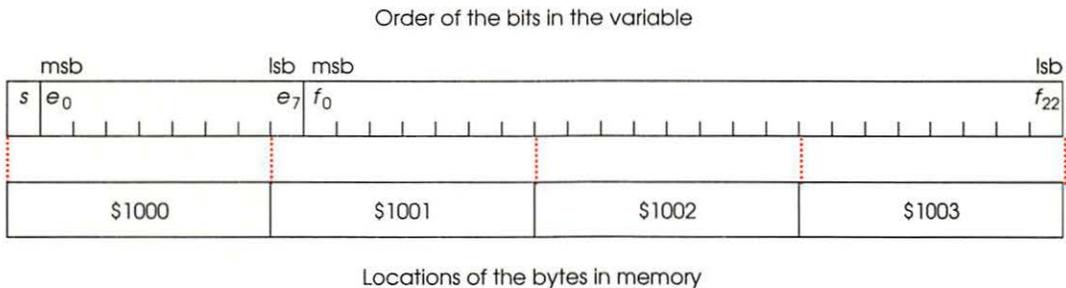


Figure E-1
Memory format of a variable of type single

Table E-1
Format diagram symbols

Symbol	Description
v	Value of number
s	Sign bit
e	Biased exponent
i	Explicit one's bit (extended type only)
f	Fraction
d	Nonsign bits (comp type only)

Single: 32 bits

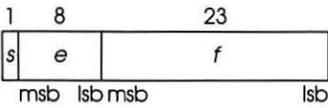


Figure E-2
Single format

Table E-2
Values of single-format numbers (32 bits)

Biased exponent e	Fraction f	Value v	Class of v
$0 < e < 255$	(any)	$v = (-1)^s \times 2^{(e-127)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-126)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 255$	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 255$	$f \neq 0$	v is a NaN	NaN

Double: 64 bits

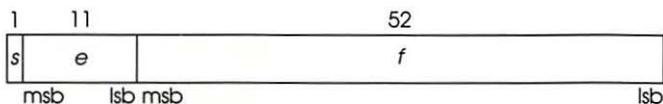


Figure E-3
Double format

Table E-3
Values of double-format numbers (64 bits)

Biased exponent e	Fraction f	Value v	Class of v
$0 < e < 2047$	(any)	$v = (-1)^s \times 2^{(e-1023)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-1022)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 2047$	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 2047$	$f \neq 0$	v is a NaN	NaN

Comp: 64 bits

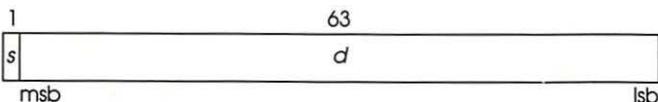


Figure E-4
Comp format

Table E-4
Values of comp-format numbers (64 bits)

Sign bit s	Nonsign bits d	Value v
$s = 1$	$d = 0$	v is the unique NaN in the comp type.
$s = 1$	$d \neq 0$	v is the two's-complement value of the 64-bit representation.
$s = 0$	(any)	v is the two's-complement value of the 64-bit representation.

Extended: 80 bits

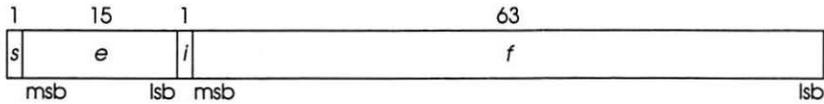


Figure E-5
Extended format

Table E-5
Values of extended-format numbers (80 bits)

Biased exponent e	Integer i	Fraction f	Value v	Class of v
$0 \leq e \leq 32766$	1	(any)	$v = (-1)^s \times 2^{(e-16383)} \times (1.f)$	Normalized
$0 \leq e \leq 32766$	0	$f \neq 0$	$v = (-1)^s \times 2^{(e-16383)} \times (0.f)$	Denormalized
$0 \leq e \leq 32766$	0	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 32767$	(any)	$f = 0$	$v = (-1)^s \times \text{Infinity}$	Infinity
$e = 32767$	(any)	$f \neq 0$	v is a NaN	NaN

Operations

Tables E-6 through E-10 define the abbreviations and symbols used in the operation tables that follow. Tables E-11 through E-24 show the mnemonic name, opword, operand types, and exceptions for each operation.

In the opword, the first byte is the operand format code and the second is the operation code.

Abbreviations and symbols

The symbols and abbreviations in this section closely parallel those in the text, although some are shortened. In some cases, the same symbol has different meanings, depending on context; for example, in data types, x stands for *extended*; in exceptions, x stands for *inexact*.

Operands

Table E-6
Operands

Abbreviation	Description
DST	Destination operand (passed by address)
DST2	Second destination operand (passed by address)
SRC	Source operand (passed by address)
SRC2	Second source operand (passed by address)

Note: Push operands in the order SRC2 (if any), SRC, DST2 (if any), DST.

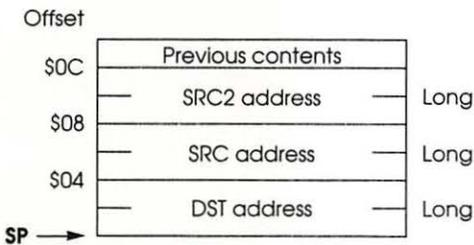


Figure E-6
SANE operands on the MC68000 stack

Data types

Table E-7
Data types

Abbreviation	Description
X	Extended (80 bits, passed by address)
D	Double (64 bits, passed by address)
S	Single (32 bits, passed by address)
B	Byte (8 bits, passed by address)
I	Integer (16 bits, passed by address)
L	Long integer (32 bits, passed by address)
C	Comp (64 bits, passed by address)
Dec	Decimal record (25 bytes, passed by address)
DecForm	Decform record (4 bytes, passed by address)
DecStr	Pascal or C decimal string

MC68000 processor registers

Table E-8
MC68000 processor registers

Abbreviation	Description
D0	Data register 0
X	Extend bit of Processor Status register
N	Negative bit of Processor Status register
Z	Zero bit of Processor Status register
V	Overflow bit of Processor Status register
C	Carry bit of Processor Status register

Exceptions

Table E-9
Exceptions

Abbreviation	Description
Xcp	Exception
I	Invalid operation
U	Underflow
O	Overflow
D	Divide-by-zero
X	Inexact

Environment and halts

Table E-10
Environment and halts

Abbreviation	Description
EnvWrD	SANE Environment word (16-bit integer)
HltVctr	SANE halt vector (32-bit long integer)

Arithmetic operations (entry point FP68K)

Table E-11
Arithmetic operations (entry point FP68K)

Name	Opword	Operation	SRC type	Exceptions
FADDX	\$0000	DST ← DST + SRC	X	I - O - X
FADDD	\$0800	DST ← DST + SRC	D	I - O - X
FADDS	\$1000	DST ← DST + SRC	S	I - O - X
FADDC	\$3000	DST ← DST + SRC	C	I - O - X
FADDI	\$2000	DST ← DST + SRC	I	I - O - X
FADDL	\$2800	DST ← DST + SRC	L	I - O - X
FSUBX	\$0002	DST ← DST - SRC	X	I - O - X
FSUBD	\$0802	DST ← DST - SRC	D	I - O - X
FSUBS	\$1002	DST ← DST - SRC	S	I - O - X
FSUBC	\$3002	DST ← DST - SRC	C	I - O - X
FSUBI	\$2002	DST ← DST - SRC	I	I - O - X
FSUBL	\$2802	DST ← DST - SRC	L	I - O - X
FMULX	\$0004	DST ← DST * SRC	X	I U O - X
FMULD	\$0804	DST ← DST * SRC	D	I U O - X
FMULS	\$1004	DST ← DST * SRC	S	I U O - X
FMULC	\$3004	DST ← DST * SRC	C	I - O - X
FMULI	\$2004	DST ← DST * SRC	I	I - O - X
FMULL	\$2804	DST ← DST * SRC	L	I - O - X
FDIVX	\$0006	DST ← DST / SRC	X	I U O D X
FDIVD	\$0806	DST ← DST / SRC	D	I U O D X
FDIVS	\$1006	DST ← DST / SRC	S	I U O D X
FDIVC	\$3006	DST ← DST / SRC	C	I U - D X
FDIVI	\$2006	DST ← DST / SRC	I	I U - D X
FDIVL	\$2806	DST ← DST / SRC	L	I U - D X
FSQRTX	\$0012	DST ← sqrt(DST)	-	I - - - X
FREMX [†]	\$000C	DST ← DST rem SRC	X	I - - - -
FREMD [†]	\$080C	DST ← DST rem SRC	D	I - - - -
FREMS [†]	\$100C	DST ← DST rem SRC	S	I - - - -
FREMC [†]	\$300C	DST ← DST rem SRC	C	I - - - -
FREMI [†]	\$200C	DST ← DST rem SRC	I	I - - - -
FREML [†]	\$280C	DST ← DST rem SRC	L	I - - - -
FRINTX	\$0014	DST ← rnd(DST)	-	I - - - X
FTINTX	\$0016	DST ← chop(DST)	-	I - - - X

Note: For all arithmetic operations, the destination is of type extended.

[†] Also, $D0 \leftarrow \text{low-order } 7 \text{ bits of } |n|$, negated if $n < 0$, where $n = \text{integer quotient } \text{DST}/\text{SRC}$.

Auxiliary routines (entry point FP68K)

Table E-12

Auxiliary routines (entry point FP68K)

Name	Opword	Operation	SRC type	Exceptions
FSCALBX	\$0018	$DST \leftarrow DST * 2^{SRC}$	I	I U O - X
FLOGBX	\$001A	$DST \leftarrow \log_b(DST)$	-	I - - D -
FNEGX	\$000D	$DST \leftarrow - DST$	-	- - - - -
FABSX	\$000F	$DST \leftarrow DST $	-	- - - - -
FCPYSGNX	\$0011	SRC \leftarrow SRC with sign of DST	*	- - - - -
FNEXTX	\$0013	SRC \leftarrow Nextafter SRC toward DST	X [†]	I U O - X
FNEXTD	\$0813	SRC \leftarrow Nextafter SRC toward DST	D [†]	I U O - X
FNEXTS	\$1013	SRC \leftarrow Nextafter SRC toward DST	S [†]	I U O - X

Note: For most auxiliary routines, the destination is of type extended.

* For CopySign, SRC and DST can be of type X, D, or S.

† For Nextafter, SRC and DST are of the same type.

Conversions (entry point FP68K)

Table E-13

Binary-to-binary conversions (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	Exceptions
FX2X	\$0010	DST ← SRC	X	X	I - - - -
FX2D	\$0810	DST ← SRC	D	X	I U O - X
FX2S	\$1010	DST ← SRC	S	X	I U O - X
FX2C	\$3010	DST ← SRC	C	X	I - - - X
FX2I	\$2010	DST ← SRC	I	X	I - - - X
FX2L	\$2810	DST ← SRC	L	X	I - - - X
FX2X*	\$000E	DST ← SRC	X	X	I - - - -
FD2X	\$080E	DST ← SRC	X	D	I - - - -
FS2X	\$100E	DST ← SRC	X	S	I - - - -
FC2X	\$300E	DST ← SRC	X	C	- - - - -
FI2X	\$200E	DST ← SRC	X	I	- - - - -
FL2X	\$280E	DST ← SRC	X	L	- - - - -

* Included for completeness; functionally identical to first FX2X in table.

Table E-14

Binary-to-decimal conversions (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	SRC2 type	Exceptions
FX2DEC	\$000B	DST ← SRC using SRC2	Dec	X	Decform	I - - - X
FD2DEC	\$080B	DST ← SRC using SRC2	Dec	D	Decform	I - - - X
FS2DEC	\$100B	DST ← SRC using SRC2	Dec	S	Decform	I - - - X
FC2DEC	\$300B	DST ← SRC using SRC2	Dec	C	Decform	- - - - X
FI2DEC	\$200B	DST ← SRC using SRC2	Dec	I	Decform	- - - - X
FL2DEC	\$280B	DST ← SRC using SRC2	Dec	L	Decform	- - - - X

Note: First push SRC2, then SRC, then DST.

Table E-15

Decimal-to-binary conversions (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	Exceptions
FDEC2X	\$0009	DST ← SRC	X	Dec	- U O - X
FDEC2D	\$0809	DST ← SRC	D	Dec	- U O - X
FDEC2S	\$1009	DST ← SRC	S	Dec	- U O - X
FDEC2C	\$3009	DST ← SRC	C	Dec	I - - - X
FDEC2I	\$2009	DST ← SRC	I	Dec	I - - - X
FDEC2L	\$2809	DST ← SRC	L	Dec	I - - - X

Comparisons (entry point FP68K)

The comparison operations return the relation information in the Processor Status register, as shown in Table E-16.

Table E-16
Relation information

Relation	Status bit				
	X	N	Z	V	C
DST > SRC	0	0	0	0	0
DST < SRC	1	1	0	0	1
DST = SRC	0	0	1	0	0
DST, SRC unordered	0	0	0	1	0

Table E-17
Comparisons (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	Exceptions
FCMPX*	\$0008	DST <relation> SRC	X	X	I - - - -
FCMPD*	\$0808	DST <relation> SRC	X	D	I - - - -
FCMPS*	\$1008	DST <relation> SRC	X	S	I - - - -
FCMPC*	\$3008	DST <relation> SRC	X	C	I - - - -
FCMPI*	\$2008	DST <relation> SRC	X	I	I - - - -
FCMPL*	\$2808	DST <relation> SRC	X	L	I - - - -
FCPXX†	\$000A	DST <relation> SRC	X	X	I - - - -
FCPXD†	\$080A	DST <relation> SRC	X	D	I - - - -
FCPXS†	\$100A	DST <relation> SRC	X	S	I - - - -
FCPXC†	\$300A	DST <relation> SRC	X	C	I - - - -
FCPXI†	\$200A	DST <relation> SRC	X	I	I - - - -
FCPXL†	\$280A	DST <relation> SRC	X	L	I - - - -

* These comparisons don't signal invalid for unordered but do signal invalid for signaling NaN inputs.

† These comparisons signal invalid for unordered.

Inquiries: class and sign (entry point FP68K)

The classify operations set the sign of the destination to the sign of the source and the value of the destination according to the class of the source, as shown in Table E-18. The destination is an integer variable.

Table E-18
Class information

Class of SRC	Value
Signaling NaN	1
Quiet NaN	2
Infinity	3
Zero	4
Normalized	5
Denormalized	6

Table E-19
Sign information

Sign of SRC	Sign of DST
Positive	Positive
Negative	Negative

Table E-20
Classify (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	Exceptions
FCLASSX	\$001C	DST ← sign and <class> of SRC	I	X	- - - - -
FCLASSD	\$081C	DST ← sign and <class> of SRC	I	D	- - - - -
FCLASSS	\$101C	DST ← sign and <class> of SRC	I	S	- - - - -
FCLASSC	\$301C	DST ← sign and <class> of SRC	I	C	- - - - -

Environmental control (entry point FP68K)

Table E-21

Environmental control (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	Exceptions
FGETENV	\$0003	DST ← EnvWrd	I	-	- - - - -
FSETENV*	\$0001	EnvWrd ← SRC	-	I	- - - - -
FTESTXCP	\$001B	DST high byte ← DST Xcp set	I	-	- - - - -
FSETXCP†	\$0015	EnvWrd ← EnvWrd OR SRC Xcp	-	I	I U O D X
FPROCENTRY	\$0017	DST ← EnvWrd, EnvWrd ← 0	I	-	- - - - -
FPROCEXIT	\$0019	EnvWrd ← SRC OR current Xcps	-	I	I U O D X

* Exceptions set by Set-Environment do not cause halts.

† 0 = invalid, 1 = underflow, 2 = overflow, 3 = divide-by-zero, 4 = inexact

Halt control (entry point FP68K)

Table E-22

Halt control (entry point FP68K)

Name	Opword	Operation	DST type	SRC type	Exceptions
FSETHV	\$0005	HltVctr ← SRC	-	L	- - - - -
FGETHV	\$0007	DST ← HltVctr	L	-	- - - - -

Elementary functions (entry point Elems68K)

Table E-23
Elementary functions (entry point Elems68K)

Name	Opword	Operation	DST type	SRC type	SRC2 type	Exceptions
FLNX	\$0000	DST ← ln(DST)	-	X	-	I - - D X
FLOG2X	\$0002	DST ← log ₂ (DST)	-	X	-	I - - D X
FLN1X	\$0004	DST ← ln(1 + DST)	-	X	-	I U - D X
FLOG21X	\$0006	DST ← log ₂ (1 + DST)	-	X	-	I U - D X
FEXPX	\$0008	DST ← e ^{DST}	-	X	-	I U O - X
FEXP2X	\$000A	DST ← 2 ^{DST}	-	X	-	I U O - X
FEXP1X	\$000C	DST ← e ^{DST-1}	-	X	-	I U O - X
FEXP21X	\$000E	DST ← 2 ^{DST-1}	-	X	-	I U O - X
FXPWRI	\$8010	DST ← DST ^{SRC}	X	I	-	I U O D X
FXPWRY	\$8012	DST ← DST ^{SRC}	X	X	-	I U O D X
FCOMPOUND	\$C014	DST ← compound(SRC2,SRC)*	X	X	X	I U O D X
FANNUITY	\$C016	DST ← annuity(SRC2,SRC)*	X	X	X	I U O D X
FSINX	\$0018	DST ← sin(DST)	X	X	-	I U - - X
FCOSX	\$001A	DST ← cos(DST)	X	X	-	I U - - X
FTANX	\$001C	DST ← tan(DST)	X	X	-	I U - D X
FATANX	\$001E	DST ← atan(DST)	X	X	-	I U - - X
FRANDX	\$0020	DST ← randomx(DST)	X	X	-	I U O - X

* SRC2 is the rate; SRC is the number of periods.

Scanners and formatter (entry point DecStr68K)

Table E-24
Scanners and formatter (entry point DecStr68K)

Name	Opword	Operation	DST type	DST2 type	SRC type	SRC2 type	Exceptions
FPSTR2DEC	\$0002	DST2 ← SRC2 beginning at SRC SRC ← new index DST ← valid prefix flag	B	Dec	I	Decstr	- - - - -
FCSTR2DEC	\$0004	DST2 ← SRC2 beginning at SRC SRC ← new index DST ← valid prefix flag	B	Dec	I	Decstr	- - - - -
FDEC2STR	\$0003	DST ← SRC according to SRC2	DecStr	-	Dec	DecForm	- - - - -

Note: Push SRC2, SRC, DST2, then DST.

The Environment word

Rounding direction and precision are stored as 2-bit encoded values; exception and halt-enabled flags are set as individual bits. Note that the default environment is represented by the integer value zero.

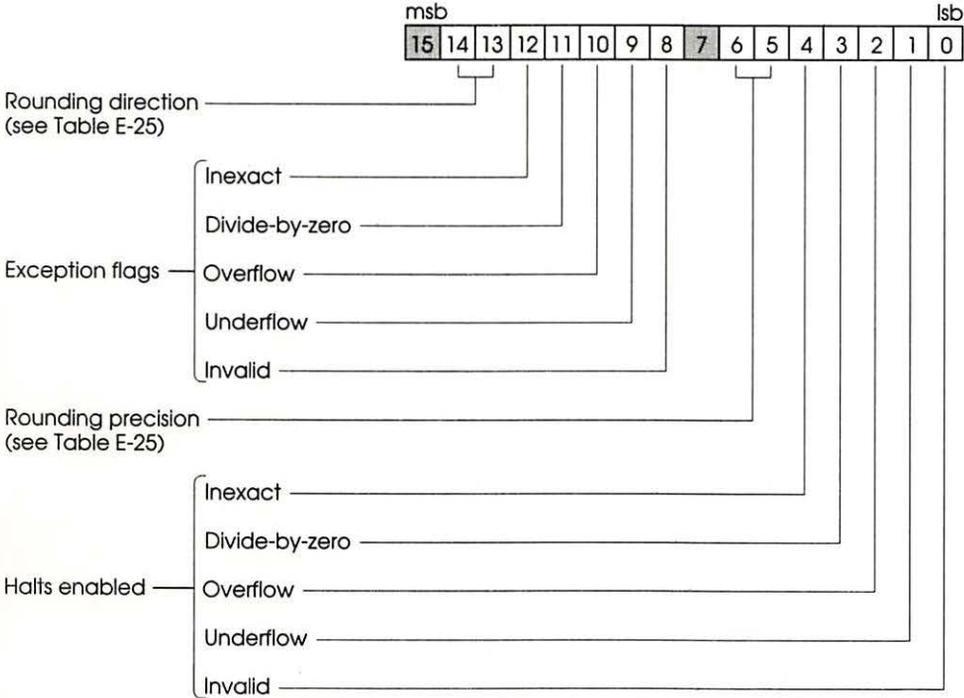


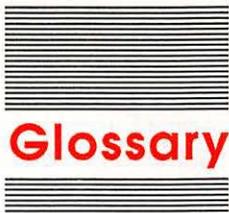
Figure E-7
The Environment word for the MC68000

Table E-25

Bits in the Environment word for the MC68000

Group name	Mask bits	Mask value	Description
Rounding direction (Bit group \$6000)	14 13 0 0 0 1 1 0 1 1	\$0000 \$2000 \$4000 \$6000	To-nearest Upward Downward Toward-zero
Exception flags (Bit group \$1F00)	12 11 10 9 8 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$1000 \$0800 \$0400 \$0200 \$0100	Inexact Divide-by-zero Overflow Underflow Invalid
Rounding precision (Bit group \$0060)	6 5 0 0 0 1 1 0 1 1	\$0000 \$0020 \$0040 \$0060	Extended Double Single (Undefined)
Halts enabled (Bit group \$001F)	4 3 2 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1	\$0010 \$0008 \$0004 \$0002 \$0001	Inexact Divide-by-zero Overflow Underflow Invalid

Note: Bits 7 and 15 are not used.



Glossary

application type: Any data type, other than extended, used to store data for applications. SANE application types are single, double, and comp.

arithmetic type: Any data type used to hold results of calculations inside the computer. The SANE arithmetic type, extended, has greater range and precision than the application types.

binade: The collection of numbers that lie between two successive powers of 2.

binary floating-point number: A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and 2 raised to the power of the exponent.

comp type: A 64-bit application data type for storing integral values of up to 18- or 19-decimal-digit precision. It is used by accounting applications, among others.

decform record: A data type for specifying the formatting for decimal results (of conversions). It specifies fixed- or floating-point form and the number of digits.

decimal record: A Pascal record type for storing decimal data. It consists of three fields: sign (16 bits), exponent (16 bits), and significand (a Pascal string).

decimal string: A decimal number represented as a string of ASCII characters with a length byte, like a Pascal string.

default environment: The environment settings when a SANE implementation starts up: rounding is to-nearest, rounding precision is extended, and all exception flags and halts are off.

delta guide: A description of something new in terms of its differences from something the reader already knows about, so-called from mathematicians' use of the Greek letter delta (Δ) to represent a difference.

denormalized number: A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type. Also called *denorm*.

double type: A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used by statistical and financial applications, among others.

environmental settings: The rounding direction, rounding precision, and the exception flags and their respective halt-enables.

exception flag: Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

exceptions: Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

exponent: The part of a binary floating-point number that indicates the power to which 2 is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range the type will handle.

extended type: An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

flush-to-zero: A system that excludes denormalized numbers. Results smaller than the smallest normalized number are rounded to zero.

gradual underflow: A system that includes denormalized numbers.

halt: Each exception has a halt-enable that can be set, cleared, or tested. When an exception is signaled and the corresponding halt-enable flag is set, the SANE engine will transfer control to the address in a halt vector. A high-level language need not pass on to its user the facility to assign the halt vector, but may halt the user's program. Halt-enable flags remain set until explicitly cleared.

infinity: A special value produced when a floating-point operation should produce a mathematical infinity or when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

integer types: System types for integral values. Integer types typically use 16- or 32-bit two's-complement integers. Integer types are not SANE types but are available to SANE users.

integral value: A value, perhaps in a SANE type, that is exactly equal to a mathematical integer: ..., -2, -1, 0, 1, 2, ...

NaN (Not-a-Number): A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs propagate through arithmetic operations.

normalized number: A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

quiet NaN: A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

rounding direction: When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: to-nearest (the default), upward, downward, and toward-zero.

SANE engine: Software or hardware that implements some or all of the SANE functionality.

signaling NaN: A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

sign bit: The bit of a single, double, comp, or extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

significand: The part of a binary floating-point number that indicates where the number falls between two successive powers of 2. The wider the significand field in a numeric type, the more resolution the type has.

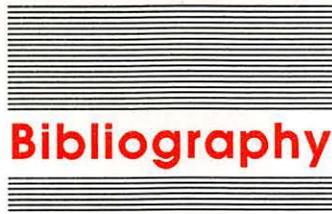
single type: A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used by engineering applications, among others.

supports SANE: (of an application or language) Uses the SANE engine for all calculations except operations on machine types, and gives the user access to all data types, operations, and environmental controls. Compare **uses SANE**.

tiny: Characteristic of a number whose magnitude is smaller than the smallest positive normalized number in the format of the number.

unit in the last place (ULP): Magnitude of the smallest possible change in a binary value.

uses SANE: (of an application or language) Uses the SANE engine for all calculations except operations on machine types, without necessarily giving the user access to all data types, operations, and environmental controls. Compare **supports SANE**.



Bibliography

- [1] Alefeld, G., and J. Hertzberger. *Introduction to Interval Computations*. New York: Academic Press, 1983.

This book presents a mathematically rigorous description of interval arithmetic.

- [2] “Appendix D: The Standard Apple Numeric Environment and the SANE Library,” *Macintosh Pascal Technical Appendix*. Lexington, MA: THINK Technologies, Inc., and Cupertino, CA: Apple Computer, Inc., 1984.

Included are an early version of Part I of the *Apple Numerics Manual* and an interface to a Pascal SANE library.

- [3] *Apple Pascal Numerics Manual: A Guide to Using the Apple Pascal SANE and Elems Units*. Cupertino, CA: Apple Computer, Inc., 1983.

This manual describes the Apple II and Apple III Pascal implementation of the Standard Apple Numerics Environment (SANE) through procedure calls to the SANE and Elems units.

- [4] *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units*. Cupertino, CA: Apple Computer, Inc., 1983.

This manual describes the Apple III Pascal implementation of the Standard Apple Numerics Environment (SANE) through procedure calls to the SANE and Elems units. This was Apple’s first full implementation of IEEE arithmetic.

- [5] *Apple III Pascal Programmer’s Manual*, Volume 2. “Appendix A: The TRANSCEND and REALMODES Units” and “Appendix E: Floating-Point Arithmetic.” Cupertino, CA: Apple Computer, Inc., 1981.

These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

- [6] *Apple IIGS Programmer’s Workshop C Reference*, version 1.0. Renton, WA: Apple Programmer’s and Developer’s Association, 1987.

This is the reference manual for the C language in the Apple IIGS Programmer’s Workshop.

- [7] *Apple IIGS Toolbox Reference*, Volumes 1 and 2. Reading, MA: Addison-Wesley, Inc., and Cupertino, CA: Apple Computer, Inc., 1987.
- Volume 1 includes information about starting and using the tool sets. Volume 2 tells about specific tool sets, including the SANE tool set.
- [8] *Apple II Instant Pascal Language Reference Manual*. Reading, MA: Addison-Wesley, Inc., and Cupertino, CA: Apple Computer, Inc., 1985.
- This manual documents SANE extensions to Pascal. Appendix E introduces SANE and the SANE Pascal library functions.
- [9] Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." *IEEE Computer* Vol. 14, No. 3 (March 1981).
- This paper compares the several contending proposals presented to the Working Group.
- [10] Cody, W. J., et al. "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic." *IEEE Micro* Vol. 4, No. 4 (August 1984).
- This article makes the proposed IEEE 854 Standard available for public comment and discusses implementation problems. SANE implementations conform to the more general proposed Standard 854, as well as to Standard 754.
- [11] Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 13, No. 1 (January 1980).
- This paper is a forerunner to the work on the draft Standard.
- [12] Coonen, Jerome T. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic." Ph.D. Thesis, University of California at Berkeley, 1984. (Available from University Microfilm, Ann Arbor, MI.)
- The thesis, developed alongside the standard itself, is a set of clarifications and elaborations of the terse 754 document [17]; it is an aid to implementors and a demonstration that the implementation is feasible.
- [13] Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer* Vol. 14, No. 3 (March 1981).
- [14] Demmel, James. "The Effects of Underflow on Numerical Computation." *SIAM Journal on Scientific and Statistical Computing*, Vol. 5, No. 4 (December 1984), pp. 887-919.
- These two papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.
- [15] Farnum, Charles. "Compiler Support for Floating-Point Computation." Submitted to *Software Practices and Experience*, 1988.
- This paper describes many of the things a compiler writer should know about floating-point arithmetic.

- [16] Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." *ACM Transactions on Programming Languages and Systems* Vol. 4, No. 2 (April 1982).

This paper describes the significance to high-level languages, especially Fortran, of various features of the IEEE proposed Standard.

- [17] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This is Draft 8.0 of the proposed Standard, which was offered for public comment. The final Standard [21] is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

- [18] Harbison, S. T., and G. L. Steele, Jr. *C Reference Manual*, Second Edition. Englewood Cliffs, NJ: Prentice-Hall, 1987.

- [19] Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 14, No. 3 (March 1981).

This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

- [20] *HP-15C Advanced Functions Handbook*. Corvallis, OR: Hewlett-Packard Company, 1982.

An appendix, "Accuracy of Numerical Calculations," gives a good analysis of rounding, albeit in decimal.

- [21] *IEEE Std 754-1985. IEEE Standard for Binary Floating-Point Arithmetic*. New York: IEEE, Inc., 1985.

SANE is based on this standard.

- [22] Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard." In *Interval Mathematics* 1980, edited by K. E. L. Nickel. New York: Academic Press, 1980.

This paper attempts to allay certain widespread misconceptions about computer arithmetic.

- [23] Kahan, W. "To Solve a Real Cubic Equation." Berkeley, CA: Report No. PAM-352, Center for Pure and Applied Mathematics, University of California, 1986.

This paper compares the behavior of different numerical methods for extracting cube roots of complex numbers.

- [24] Kahan, W. "Rational Arithmetic in Floating-Point." Berkeley, CA: Report No. PAM-343, Center for Pure and Applied Mathematics, University of California, 1986.

This paper describes preconditioning and the use of the inexact flag.

- [25] Kahan, W. "Branch Cuts for Complex Elementary Functions." In *The State of the Art of Numerical Analysis*, edited by A. Iserles and M. J. D. Powell. Oxford University Press, 1987.
- [26] Kahan, W., and Jerome T. Coonen. "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments." In *The Relationship between Numerical Computation and Programming Languages*, edited by J. K. Reid. New York: North Holland, 1982.

This paper describes high-level language issues relating to the proposed IEEE Standard, including expression evaluation and environment handling.

- [27] *LightspeedC User's Guide and Reference Manual*. Lexington, MA: THINK Technologies, Inc., 1986.
- [28] *Lightspeed Pascal User's Guide and Reference Manual*. Lexington, MA: THINK Technologies, Inc., 1986.
- [29] *Mac C and Mac C Toolkit: A Programmer's Guide*. Version 2.0. Portola Valley, CA: Consulair Corp., 1985.

LightspeedC and Lightspeed Pascal are development systems that support SANE.

- This manual documents SANE extensions to C and a SANE C library.
- [30] *Macintosh Pascal Reference Manual*. Lexington, MA: THINK Technologies, Inc., and Cupertino, CA: Apple Computer, Inc., 1984.
- [31] *MC68881 Floating-Point Coprocessor User's Manual*. Phoenix, AZ: Motorola, Inc., 1985.

This is the reference manual for the MC68881, a hardware implementation of the IEEE Standard for Floating-Point Arithmetic, P754, for the MC68000 family of microprocessors.

- [32] Moore, R. E. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, 1979.
- [33] *MPW C Reference*, version 2.0. Renton, WA: Apple Programmer's and Developer's Association, 1987.

This is the reference manual for the C language in Apple's Macintosh Programmer's Workshop.

- [34] *MPW Pascal Reference*, version 2.0. Renton, WA: Apple Programmer's and Developer's Association, 1987.

This is the reference manual for the Pascal language in Apple's Macintosh Programmer's Workshop.

- [35] Rice, John R. *Numerical Methods, Software, and Analysis*. New York: McGraw-Hill, 1983.

This book is a compendium on currently-available numerical software libraries, mostly in Fortran. It discusses numerical methods and gives advice—some of it good. It also includes a large bibliography.

- [36] Sterbenz, Pat H. *Floating-Point Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

This book is obsolescent. It describes some things you have to know and others you used to have to know; it gives a good idea of why floating-point computation is simpler nowadays.

- [37] *Turbo Pascal for the Mac User's Guide and Reference Manual*. Scotts Valley, CA: Borland International, Inc., 1986.



Index

- A**
- Absolute Value function 49, 55, 95, 150, 197
 - Accrued Exception byte 202, 203, 205–206
 - Accrued Exception flag 205, 206
 - accuracy
 - conversions 23
 - maximizing for 65C816 103
 - maximizing for 68000 159
 - MC68881 elementary functions 199
 - addition 55, 93, 196
 - AIN_T 243
 - Annuity function 64–65, 123, 174, 198
 - special cases 65
 - Annuity(*r*, *n*) 123
 - A1 register 142, 180, 212
 - Apple Pascal Assembler 240
 - Apple II 240
 - Apple IIGs 85
 - Apple IIGs Programmer's Workshop (APW) 82–88, 98, 219, 240
 - Apple IIGs Toolbox 85
 - Apple III 240
 - application types 12
 - APW. *See* Apple IIGs Programmer's Workshop
 - arccosine 71, 198
 - arccosine, hyperbolic 74
 - arcsine 71, 198
 - arcsine, hyperbolic 74
 - arctangent 66, 67, 120, 172, 197
 - arctangent, hyperbolic 74
 - area, triangle 77–78
 - A register 86, 116
 - argument reduction 47, 66
 - arithmetic, IEEE Standard 4–10, 220
 - arithmetic operations 36, 46–47, 196, 253, 269
 - arithmetic type 12. *See also*
 - extended type
 - assembly-language calls 82
 - assembly-language macros 88–89, 92, 144
 - Atan(*x*) 120, 172
 - atomic operations 58, 110
 - ATOMICPROC 110, 165
 - A2 register 180
 - auxiliary routines
 - 65C816 94, 95, 254
 - 68000 150–151, 270
 - 68881 197, 198
 - A0 register 142, 180
- B**
- base-*e* exponential 63, 120, 172, 197
 - base-*e* exponential minus 1 63, 120, 172, 197
 - base-*e* logarithm 62, 120, 172, 197
 - base-10 exponential 198
 - base-10 logarithm 198
 - base-2 exponential 63, 120, 172, 197, 198
 - base-2 logarithm 62, 120, 172, 197, 198
 - BASIC 242
 - biased exponent 16
 - binade 42
 - binary exponent 50, 94, 150
 - binary logarithm 94, 150, 197
 - binary mantissa 198
 - binary operations 83
 - binary scale 94, 150, 181, 197
 - bytes 246
 - bytes in memory, order 90, 102, 145, 246
- C**
- calling conventions 82, 140
 - calling sequence
 - for 65C816 86–89, 92, 100–102, 121–123, 127
 - for 68000 143, 148, 156–158, 172–174, 176
 - CCR 168
 - CDC computers 243
 - C language
 - data types 233
 - interface 235
 - SANE extensions 233–239
 - SANE library 235–239
 - scaling functions 50
 - classify operation 97, 197, 257, 273
 - class inquiry 97
 - comparison
 - in C extension 234
 - F_{CPX}, F_{CMF} operations 96–97, 151–152
 - and invalid operations 55
 - involving NaNs and Infinities 48
 - in Pascal extension 222
 - in 65C816 SANE 84, 96–97, 256
 - in 68000 SANE 142, 151–152, 277
 - using 68881 196

- compatibility 187, 202
 - comp format 13, 17, 193, 221, 233
 - compilers 55
 - options 224, 235
 - compound conditional statements 7–8
 - compounded value Q 63
 - Compound function 64, 174, 198
 - special cases 65
 - Compound (r, n) 123
 - computer approximation of real numbers 23
 - computer voice xxvi
 - Condition Code byte 202
 - constant C 70
 - constant definitions, evaluating 222
 - constants
 - in C extensions 231
 - for exceptions 54
 - extended 222
 - in Pascal extensions 220
 - for Sinh and Cosh 72
 - CONST DivByZero 54
 - CONST Inexact 54
 - CONST Invalid 54
 - CONST Overflow 54
 - CONST Underflow 54
 - continued fractions 76–77
 - controlling the environment 52–59
 - with 65C816 SANE 106–110, 258
 - with 68000 SANE 162–165, 274
 - with 68881 202–206
 - convergence criteria 244
 - conversions 20–34
 - accuracy of decimal to binary 23
 - between binary formats 100–101, 156–157
 - binary to decimal 22–29, 102–103, 157–159
 - between decimal formats 29–34
 - between decimal strings and records 176
 - between extended formats 193
 - to comp and other integral formats 22
 - cycles (binary-decimal) 24
 - decimal to binary 22–29, 102–103, 157–159
 - to extended 100, 156
 - from extended 21, 101, 157
 - invalid conditions 55
 - numeric constants 222, 233
 - with 65C816 84, 100–103, 255
 - with 68000 141, 156–159, 271
 - with 68881 196–198
 - CopySign 55, 95, 150, 198
 - cosecant 70
 - cosine 66, 120, 172, 197
 - cotangent 71
 - Cray computers 243
 - C SANE extensions 233–239
 - C SANE library 235–239
 - C status bit 151
 - Cstr2dec 126, 176
 - current exception flags 206
 - current rounding direction 47, 52
- D**
- data types 12–18
 - with 65C816 70
 - with 68000 145
 - with 68881 190
 - DecForm.digits 26–27
 - decform records
 - in conversions 31–32, 84, 101–102, 157, 158
 - definition 26–27
 - digits field 26–27
 - as numeric formatter input 127, 177
 - style field 26–27, 102, 157
 - DecForm.style 26–27, 102, 157
 - decimal data 24–25
 - Decimal.exp 27, 102
 - decimal fractions 23
 - decimal records 27–33
 - in conversions 27–33, 101–102, 157–159
 - definition 27
 - exp field 27
 - as numeric formatter input 127, 177
 - as numeric scanner output 126, 176
 - sgn field 27
 - sig field 27
 - Decimal.sgn 27, 102, 157
 - Decimal.sgn 27, 102, 157
 - Decimal.sig 27, 101, 102, 157, 158, 159
 - decimal strings 25
 - DecStr 25
 - DecStr816 82, 85, 86, 112, 127
 - DecStr68K 140, 176, 241
 - DecStr6502 82, 85, 86, 87, 127, 135, 240
 - Dec2Str 127, 177
 - default environment 52, 53, 106, 162, 223
 - default rounding 223
 - direction 52
 - precision 53
 - delta guide 183
 - denormalized numbers 13, 15, 40, 42–43, 50
 - destination operand (*DST*)
 - definition 83
 - in 65C816 operation forms 83–84, 86–87, 92
 - in 68000 operation forms 141–143, 148, 151
 - development systems 88
 - direct page 85, 113, 114
 - divide and round 199
 - divide-by-zero exception 56
 - Divide function 93
 - division 55, 93, 196
 - by zero 9, 77
 - D1 register 142
 - DOS assemblers 240
 - double format 13, 17, 221, 233, 248, 265
 - double precision 244
 - double rounding 244
 - downward rounding 52
 - DST*. See destination operand
 - D0 register 142, 148, 149, 168
- E**
- 80-bit extended format 18, 191–194, 213, 220
 - 8087 coprocessor 244
 - elementary functions 10, 62, 120, 172, 259, 275
 - accuracy using 68881 199
 - Elms881 190, 224
 - Elms816 82, 85, 86, 112, 120

Elms68K 140, 172, 190, 241
Elms6502 82, 85–87, 135, 240
environment 52–59, 223, 234, 245
environmental control 52–59
 with 65C816 SANE 106–110, 250
 with 68000 SANE 162–165, 274
 with 68881 202–206
environment flags 106, 162
environment registers (68881) 202–206
Environment word 85
 for 65C816 106–110
 for 68000 162–163
 for 68881 202–203
EnvWrD 252, 267
equal (comparison) 47, 49, 96
error bounds 244
evaluation rules 244
exceptional events 7, 77
Exception Enable byte 202, 203–204
exception handling 10, 70, 208–209
exceptions 10, 49, 85, 116, 150, 168, 209, 233
 with C extensions 233, 234
 in Environment word 106–107, 162–163
 in Exception word 109, 164–165
 and halt mechanism 54, 116, 168
 with Pascal extensions 222, 223
 setting 58
 during sign manipulation 49
 spurious 59, 70
 stimulating 109, 110, 165
 types of 55–56
Exception Status byte 202, 203, 205–206
Exception Status flags 205
Exception word 109, 164
Exp field 63
explicit one's bit 16
Exp1 63
exponent 13
 binary 50
exponential functions 63–64
Exp1(x) 120, 172
expressions 222, 233

Exp21(x) 120, 172
Exp2 63
Exp2(x) 120, 172
Exp(x) 120, 172
extended type
 advantages 4, 12–13, 77
 in arithmetic operations 46
 in C extension 233
 constants in 222
 conversions between 80- and 96-bit 193
 80-bit format 18
 format 18
 96-bit format 18, 192
 in Pascal extension 221
 when porting programs 243
 precision 12–13
 range 12–13
 using temporaries 37

F

FADD 190
FADDD 190
FADDS 144
FALSE 48
FBIN 98
FBLE 97, 98
FBLES 152
FBNE 97, 98
FBNES 152
FCMP 96, 151, 152
FCMPD 152
FCOMPOUND 123, 174
FCPX 96, 151, 152
FCPXS 152
FCPYSGNC 95
FCPYSGNX 150
FCSTR2DEC 176
FC2DEC 158
FC2X 100, 156
FDEC2D 102, 158
FDEC2STR 177
FDIVD 93, 148
FD2X 100, 144, 156
FGETENV 108, 164
FI2X 100, 156
fixed-format overflow 102
Floating-Point Control register 202
 Exception Enable/Mode Control bytes 202, 203–204

floating-point coprocessors 244.
 See also MC68881
floating-point registers (68881) 191
Floating-Point Status register 202
 Exception Status/Accrued Exception bytes 203, 205–206
floating-point storage formats 12–13
float type (C) 233
FL2X 100, 156
flush-to-zero systems 43
FNEXTD 95, 151
formatters, numeric 29, 32, 260, 275
formatting 198
 MC68881 213–214
 MC68000 181–182
 65C816 132–133
 6502 135–136
Fortran 242, 243, 244
FPCR (Floating-Point Control register) 202
FP881 190
FP816 82, 85, 86, 112, 127, 131
FP1 register 212
FPROCENTRY 110, 165
FPROCEXIT 110, 165
FP7 register 190, 191
FP68K 140, 141, 142, 150, 151, 156, 158, 159, 168, 170, 176, 190, 241
FP6502 82, 85, 86, 87, 127, 134, 135, 240
FPSR (Floating-Point Status register) 202
FP0 register 190, 191, 212
fractions 16
FREMS 94, 149
frexp 50
FRINTX 93
FSCALEBX 94, 150
FSETENV 108, 164
FSETHV 116, 117, 168
FSINX 121, 173
FSQRTX 93, 144, 149
FS2X 100, 156
FSUBS macro 82
FTESTXCP 109, 165
FTINTX 93

FUNCTION Annuity 65
 FUNCTION ArcCos 71
 FUNCTION ArcCosh 74
 FUNCTION ArcSin 71
 FUNCTION ArcSinh 74
 FUNCTION ArcTan 66
 FUNCTION ArcTanh 74
 FUNCTION ClassComp 44
 FUNCTION ClassDouble 44
 FUNCTION ClassExtended 44
 FUNCTION ClassReal 44
 FUNCTION ClassSignNum 44
 FUNCTION Compound 64
 FUNCTION CopySign 49
 FUNCTION Cos 66
 FUNCTION CoSecant 70
 FUNCTION Cosh 73
 FUNCTION CoTangent 71
 FUNCTION Dec2Num 28
 FUNCTION Exp 63
 FUNCTION Exp1 63
 FUNCTION Exp2 63
 FUNCTION FPFunc 193
 FUNCTION GetHaltVector 54
 FUNCTION Ln 62
 FUNCTION Ln1 62
 FUNCTION Logb 50
 FUNCTION Log2 62
 FUNCTION NAN 41
 FUNCTION NextDouble 50
 FUNCTION NextExtended 50
 FUNCTION NextReal 50
 FUNCTION NumFcn 59
 FUNCTION Num2Comp 22
 FUNCTION Num2Double 21
 FUNCTION Num2Integer 22
 FUNCTION Num2LongInt 22
 FUNCTION Num2Real 21
 FUNCTION RandomX 67
 FUNCTION Relation 49
 FUNCTION Remainder 46
 FUNCTION Rint 47
 FUNCTION Scalb 50, 181
 FUNCTION ScalbNew 213
 FUNCTION Secant 70
 FUNCTION SetHaltVector 54
 FUNCTION Sin 66
 FUNCTION Sinh 72
 FUNCTION Str2Num 25
 FUNCTION Tan 66

FUNCTION Tanh 73
 FUNCTION XpwrI 63
 FUNCTION XpwrY 63
 fundamental operations 5,
 187–189, 196
 FXPWRI 122, 173
 FX2C 101, 157
 FX2D 101, 144, 157
 FX2I 101, 157
 FX2L 101, 157
 FX2S 101, 157
 FX2X 100, 101, 156, 157

G

gamma 72
 general exponentiation function
 122, 173, 198
 Get-Environment 107, 108, 163,
 164
 GetHaltVector 115, 169
 GetTrapVector 208
 gradual underflow 5, 42–43
 greater (comparison) 47, 49, 96
 greater-or-equal (comparison) 47

H

halt 109, 112, 113, 114, 168
 halt address 84
 halt bit 109
 halt control 259, 274
 halt-enable bit 112, 168
 halt-enable flags 106, 162, 261,
 276
 halt flags 209
 halt handlers 114, 116, 168, 170,
 208
 halt mechanism 112, 116–117,
 168–170
 halts enabled 107, 163, 223, 234,
 262, 277
 halt settings 57
 halt status information 113
 halt (trap) vector 85, 112, 114,
 116, 169, 170, 208
 operations 115
 hardware exception trapping 54
 Heron's formula 77–78
 high-level languages 219–239
 comparisons 48
 halts and traps 208

HltVctr 252, 267
 Horner's recurrence 130, 133,
 180, 212
 HP Spectrum quad format 243
 HROUTINE 170
 hyperbolic arccosine 74
 hyperbolic arcsign 74
 hyperbolic arctangent 74, 198
 hyperbolic cosine 73, 198
 hyperbolic sine 72, 198
 hyperbolic tangent 198

I, J

IBM Q format 243
 IEEE double type 13, 221, 233
 IEEE extended type 13, 221,
 233, 243
 IEEE rounding 22
 IEEE single type 13, 221, 233
 IEEE Standard 12, 23, 38, 96, 152
 IEEE Standard arithmetic 4–10,
 220
 IEEE Standard defaults 223, 234
 IEEE Standard 854 50
 IEEE Standard numerics 186
 IEEE Standard 754 50, 186, 187,
 196
 binary floating-point arithmetic
 xxv, 22, 137
 index 126, 176
 Index 30
 inexact exception 56
 inexact flag 36
 INF 223, 234
 Infinity 7–10, 22, 26, 28, 40, 44,
 77, 220, 223, 234
 comparisons 48
 negative 40
 positive 40
 inquiries 84, 97–98, 153, 257, 273
 instant rounding 244
 INT 243
 integer exponentiation function
 122, 173, 198
 integral values 22, 47, 52
 internal rate of return 63
 interval arithmetic 9
 invalid-operation exception 41, 55
 invalid-operation flag 48, 245
 inverse operations 6, 199

K

Kahan, William 70

L

lambda 72

language interface, MC68881 213

ldexp 50

less (comparison) 47, 49, 96

less-or-equal (comparison) 47

limits, theory 40

Ln(1 + x) 62

Ln1(x) 62, 120, 172

Ln(x) 62, 120, 172

logarithmic functions 62

Logb 94, 150

Log21(x) 120, 172

Log2(x) 62, 120, 172

M

Macintosh 186

Macintosh Programmer's Workshop

(MPW) 140, 148, 152, 172,

194, 208, 241

C 235

C compiler 189

Pascal 8, 170, 224

Pascal and C SANE libraries 209

Pascal compiler 189

revision 2.0 241

macro calls 122

macro names 140

macros 148, 152, 172, 194, 240

mantissa 50

MC68881 compiler option 224,
235

MC68881 coprocessor 18, 54, 183,
196, 197, 244

accuracy 199

calls 190, 213

comp format 193

data types 190–193

environment registers 202–203

exception and halt flags 209

exception handling 208–209

floating-point registers 191

functions 196–199

functions not performed 189

language interface 213

96-bit extended format 191–193

numerics packages 190

polynomial evaluation 212

SANE libraries 208

SANE macros 194

SANE software 186–189, 241

scanning and formatting

213–214

status and control registers 202

traps 208

MC68020 microprocessor 187

MC68000 microprocessor 18, 137,

183, 191, 196

assembly-language 141, 142,

172

CCR flags 142

floating-point packages 186

language interface 181

polynomial evaluation 180

registers 142, 267

stack 267

MC68000 SANE engine, 137, 140,

145, 156, 157, 180, 181, 263

data types 145

Environment word 203

software 202, 241

scanning and formatting

181–182

memory, order of bytes 90, 102,

145, 246

Memory Manager 85, 130

Miscellaneous Tool Set 85

MISC record 168

mixed formats 243

Mode Control byte 202, 203–204

modulo function 46–47

modulo remainder 198

monotonic packages 199

mu 72

multiplication 55, 93, 196

multiply and round 199

Multiply function 93

NAN 26, 223, 234

NaN codes 25, 28, 41, 223

SANE 196

NaNs 7, 8, 10, 22, 28, 32, 40, 41,

77, 196, 222, 223, 234, 245

comparisons 48

quiet 41, 55

signaling 41, 48, 49, 55, 150

needle-shaped triangles 78

Negate function 49, 55, 95, 150

negation 197

negative Infinity 40

Nextafter function 50, 151, 198

96-bit extended format 18,

191–194, 213

96-bit interface 193

nonsign bits 16

normalized numbers 13–15, 40, 42

Not-a-Number. *See* NaNs

not-equal (comparison) 47, 48

N status bit 96, 97, 151

numeric ASCII strings 30

numeric constants, conversions

222, 233

numeric formatter 127, 132, 135,

177, 181

numeric scanner 126–127, 132,

135, 176, 181

O

operands 82

classes 153

destination 83, 86, 92, 96, 141,
148, 151

format code 88

order 49

passed by address 83, 101, 141,
148

passed by value 83

source 86, 88, 96, 141, 148,
151, 153

operation forms 83–84, 140–142

opword 88, 143

order of operands 49

output

fixed-style 32

floating-style 31

overflow exception 56

P

package calls 194, 209, 213

package halt mechanism 209

Package Manager 241

parser 103

Pascal 20, 21, 31, 243

Pascal data types 221

Pascal SANE extensions 221–232

Pascal SANE library 224–232
 Pascal strings 176, 181
 Pascal-type rounding 22
 PDP-11C, double-precision 244
 periodic functions 66
 pi 66
 PolyEval 212
 POLYEVAL 131, 134, 180
 polynomial evaluation
 MC68000 180
 MC68881 212
 65C816 130–131
 6502 133–134
 porting programs 242
 positive Infinity 40
 P register 257
 PROCEDURE Dec2Str 31
 Procedure-Entry 57, 58, 70, 107,
 110, 163, 165
 Procedure-Exit 57, 58, 70, 107,
 110, 163, 165
 PROCEDURE GetEnvironment 57
 PROCEDURE GetPrecision 53
 PROCEDURE GetRound 52
 PROCEDURE Num2Dec 28
 PROCEDURE Num2Str 26
 PROCEDURE ProcEntry 57
 PROCEDURE ProcExit 57
 PROCEDURE SetEnvironment 57
 PROCEDURE SetException 54
 PROCEDURE SetHalt 54
 PROCEDURE SetPrecision 53
 PROCEDURE SetRound 52
 PROCEDURE Sine 135, 181, 213
 PROCEDURE Str2Dec 30
 PROCEDURE TestException 54
 PROCEDURE TestHalt 54
 processor status bits 92, 96
 CCR 148
 Processor Status register 109, 272
 ProDOS assemblers 240
 PROGRAM invop; 6
 Pstr2dec 126, 176
 PUSH 82
 PUSHLONG macro 82, 86
 PUSHWORD macro 86

Q

quiet NaN 41, 55
 Quotient byte 202

R

radians 66
 random number generator 67, 198
 RandomX(x) 120, 172
 real numbers 48
 computer approximation 23
 rectangular distribution 67
 registers 114
 relational operators 48, 96
 relations 96
 Remainder function 46–47, 55, 94,
 149, 196
 remainder magnitude 47
 result, tiny 56
 rounding 4, 5, 9, 245
 default 6, 222
 double 244
 IEEE 22
 instant 244
 Pascal-type 22
 rounding direction 9, 52–53, 103,
 106, 107, 149, 162, 163, 203,
 204, 223, 234, 245, 261, 262,
 276, 277
 current 47
 default 52
 setting 57
 rounding error 244
 rounding modes 203
 rounding precision 53, 57, 107,
 162, 163, 203, 204, 223, 234,
 261, 262, 276, 277
 rounding upward 53
 Round-to-Integer 93, 149, 197
 round-to-integral value 196
 run-time library 219

S

SANE xxv, 4, 6
 comp type 13, 221, 233
 engine calls 140
 Environment word 85
 function number 85
 hardware 186
 hybrid packages 186, 187, 199,
 202, 208
 implementations for different
 microprocessors 22
 language products supporting
 219

language systems 220, 221
 MC68881 macros 194
 NaN codes 196
 operations 82
 opword 113
 Pascal interface 224
 types 14–15, 20, 220
 SANE library 219, 221
 C 209, 235–239
 MC68881 208
 Pascal 209, 224–232
 SANEShutdown 130
 SANE software 193, 198, 202
 functions 196–198
 halts 208
 MC68000 241
 MC68020 241
 MC68881 186–189, 241
 65C816 240
 6502 240
 SANEStartUp 85, 130
 SANE Tool Set 85
 Scalb 94, 150, 181
 scanf 234
 specifiers 234
 scanners 29, 31, 114, 126–127,
 260, 275
 scanning 198
 MC68000 181–182
 MC68881 213–214
 routines 126–127, 176
 65C816 132–133
 6502 135–136
 Secant 70
 seed 67
 set condition 199
 Set-Environment 107, 108, 163,
 164
 Set-Exception 107, 109, 163,
 164–165
 SetHaltVector 112, 115, 169
 SetTrapVector 208
 sgn field 27
 short-circuit option (\$\$C+) 8
 sig field 27
 SIGN(A) 242
 SIGN(A,B) 242
 signaling NaN 41, 48, 49, 55, 150
 sign bit 16
 signed-integer format 199
 significand 13

- significant digits, number of 158, 159
 - sign inquiry 97
 - sign manipulations 49, 55
 - sign of zero 43–44
 - sine 66, 120, 172
 - in examples 132, 135, 244
 - sine, hyperbolic 72
 - single format 13, 15, 16, 221, 233
 - single-precision arithmetic 37
 - 65C816 microprocessor 79
 - addresses 115
 - direct page 113
 - halt example 116
 - polynomial evaluation 130–131
 - processor registers 251
 - stack 250
 - 65C816 SANE engine 79, 82, 85, 89, 90, 92, 103, 112, 121
 - data types 90
 - halt mechanism 112
 - software 240
 - scanning and formatting 132–133
 - 6502 microprocessor 79
 - halt example 117
 - polynomial evaluation 133–134
 - processor registers 251
 - stack 250
 - status bit 114
 - 6502 SANE engine 79, 82, 85–90, 92, 94, 100, 103, 108, 112, 121
 - halt status record 114
 - macros 88
 - software 240
 - scanning and formatting 135–136
 - 68000. *See* MC68000
 - 68881. *See* MC68881
 - software packages 197, 208
 - calls 187
 - source operand 86, 88, 96, 141, 148, 151, 153
 - passed by address 92
 - passed by value 92, 94
 - speed improvement with MC68881 188, 197
 - spurious exceptions 59, 70
 - square root 55, 93, 149, 196
 - SRC*. *See* source operand
 - stack frame 168, 169
 - Standard Apple Numerics Environment. *See* SANE
 - status flags 86
 - stopping 8
 - stopping computation 77
 - stopping program 245
 - string 126, 127, 176, 177
 - conversions 26
 - Str2Num 222
 - style field (of decform record) 26–27, 102, 157
 - Subtract function 93
 - subtraction 55, 93, 196
- T**
- tangent 66, 120, 172, 197
 - tangent, hyperbolic 73
 - temporary variables 37
 - Test-Exception 107, 109, 163, 164–165
 - theory of limits 40
 - time value of money 64
 - to-nearest rounding 52
 - Tool Dispatcher 85–88
 - Tool Locator 85
 - tool set number 85
 - toward-zero rounding 52
 - transcendental functions 70, 188
 - transcendental operations 189, 197, 199
 - transported code 245
 - trap handlers 208
 - trap mechanism 208, 209
 - traps 203, 204
 - traps enabled 204
 - trap vectors 208
 - triangle
 - area 77–78
 - needle-shaped 78
 - trigonometric functions 66–67
 - TRUE 48
 - Trunc 243
 - Truncate-to-Integer 93, 149, 197
 - type comp 12, 13, 20, 36, 90, 145, 220, 221, 233
 - type double 12, 13, 20, 90, 145, 220, 221, 233
 - TYPE Environment 57
 - TYPE Exception 54
- U**
- unary operations 83
 - underflow 4
 - gradual 5, 42–43
 - underflow exception 56
 - underflow halt 56
 - unit in last place 52
 - UNIT SANE 224
 - University of California (Berkeley) 70
 - unordered (comparison) 48, 49, 96
 - upward rounding 52
- V, W**
- ValidPrefix 30
 - valid prefix 126, 176
 - values
 - integral 47, 52
 - number 16
 - variables, temporary 37
 - VAX H format 243
 - V status bit 96, 151
- X**
- xOff80 213
 - xOff96 213
 - XpwrI 63
 - XPwrI(x, i) 122, 173
 - XpwrY 63
 - XPwrY(x, y) 122, 173
 - X register 86, 92, 96, 97, 108, 112, 114, 115, 257
 - X status bit 151

Y

Y register 86, 92, 96, 97, 112,
114, 115, 257

Z

zero

division by 9, 77

sign of 43–44

zero bank 130

zero page 85

Z flag 109

Z status bit 96, 151

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word. Proof pages were created on the Apple LaserWriter® Plus. Final pages were created on the Varityper® VT600™. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated. Some of the illustrations were created using Adobe Illustrator™.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.