

CoSort v9.5.3

User Manual & Programmer's Guide

© INNOVATIVE ROUTINES INTERNATIONAL (IRI), INC.
1978 - 2016. All rights reserved. The material
herein is confidential, proprietary property,
protected by, and enforced under, U.S.
and international copyright,
trademark and trade secret laws.
No part of this guide may
be disclosed, disseminated,
or reproduced without
the express written
consent of IRI.



September 2013

CONFIDENTIAL: Use of this document is restricted by the terms of a non-disclosure and/or license agreement binding you and your company. For your own protection, **DO NOT** transfer or disclose any part of this manual, **CoSort** software or trial information without the prior written consent of Innovative Routines International (“IRI”), Inc.

IRI has made every effort to ensure that this document is correct and accurate, but reserves the right change it without notice.

CoSort software licenses are serialized and usage is registered. Anyone wishing to expand the use, or integrate and/or distribute all or any part of the software, must first execute an appropriate license agreement with IRI.

Restricted rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

No warranty, expressed or implied, is made by IRI as to the accuracy of the material and functioning of the software. Any warranty of fitness for any particular purpose is expressly excluded and in no event will IRI be liable for any direct or consequential damages.

Trademarks: **CoSort** of IRI; all other brand or product names are trademarks or registered trademarks of their respective holders/companies in the United States and other jurisdictions.

© 1978-2016 IRI. All rights reserved. No part of this document or the **CoSort** programs may be used or copied without the express written permission of IRI. Please contact:

INNOVATIVE ROUTINES INTERNATIONAL (IRI), INC.

2194 Highway A1A

Suite 303, Atlantis Center

Melbourne, Florida 32937 USA

Tel (321) 777-8889, Fax (321) 777-8886

Email support@cosort.com

URL www.iri.com

TABLE OF CONTENTS

INTRODUCTION	21
1 PURPOSE.....	21
2 COMPATIBLE APPLICATIONS	22
3 NOTATION	24
3.1 Fonts	24
3.2 Symbols	25
3.3 CoSort Entities	25
3.4 Hyperlinks.....	26
4 PRODUCT OVERVIEW	27
4.1 Sort/Merge Module.....	29
4.2 User Interfaces	30
4.3 User Programs (API)	32
4.4 CoSort Technical Specifications.....	33
4.5 Compatible Platforms	36
 sortcl PROGRAM	 37
1 PURPOSE.....	37
2 EXECUTION.....	40
3 CONVENTIONS	41
3.1 Naming Conventions	41
3.2 Abbreviations	42
3.3 Optional Statement Parameters.....	42
3.4 Environment Variables	42
3.5 Line Continuation	42
3.6 Comments.....	43
3.7 Italics.....	43
4 FILES	44
4.1 Resource Control File	44
4.2 Specification Files.....	45
4.3 Data Definition Files.....	46
4.4 Data Files	48

4.4.1	Input Files	48
4.4.2	Output Files	51
5	DATA SOURCE AND TARGET FORMATS (/PROCESS)	53
5.1	RECORD or RECORD_SEQUENTIAL	54
5.2	MFVL_SMALL	56
5.3	MFVL_LARGE	56
5.4	VARIABLE_SEQUENTIAL (or VS)	57
5.5	LINE_SEQUENTIAL (or LS)	57
5.6	V (variable).....	58
5.7	VISION.....	58
5.8	VSAM.....	58
5.9	MFISAM.....	59
5.10	UNIVBF	60
5.11	CSV.....	60
5.11.1	Special Considerations.....	61
5.12	LDIF	61
5.12.1	Special Considerations.....	62
5.13	ODBC	62
5.13.1	Syntax: Using /PROCESS=ODBC	63
5.13.2	EXT_FIELD	64
5.13.3	Extraction.....	65
5.13.4	Loading	66
5.13.5	Special Considerations.....	67
5.13.6	EXT_FIELD	68
5.14	BLOCKED	68
5.15	XML	68
5.15.1	Special Considerations.....	70
	Example 1: Using /PROCESS=XML on Input	70
	Example 2: Using /PROCESS=XML on Output	72
5.16	RANDOM (Generating Test Data).....	73
	Example 3: Random Data Generation	74
	Example 4: Random Data Selection	75
5.17	ELF (W3C Extended Log Format).....	77
6	/CHARSET.....	78
7	/ENDIAN	79
8	STATISTICS	81
9	AUDITING.....	82
	Example 5: Creating an /AUDIT file	84
	Example 6: Querying the Audit File	87

10	/ALIAS	89
	Example 7: Using /ALIAS	89
11	FIELDS	92
11.1	Syntax	92
11.2	Field Name	93
11.3	POSITION	93
11.4	SEPARATOR	95
	Example 8: Using a Multi-Character Separator	96
	Example 9: Using Different Separators	98
11.5	SIZE	99
11.6	CHARS	100
11.7	Precision	100
11.8	FRAME	101
	Example 10: Using FRAME	102
11.9	Alignment	103
	Example 11: Using RIGHT_ALIGN on Output	103
	Example 12: Using LEFT_ALIGN and RIGHT_ALIGN	105
11.10	MILL	106
11.11	FILL	106
11.12	Null Assignment	107
	Example 13: Using Null Assignments	107
11.13	Numeric Attributes	109
	Example 14: Using the EXPONENT Attribute	110
	Example 15: Using the IMPLIED_DECIMAL Attribute	112
11.14	Composite Fields and Templates	113
	11.14.1 Syntax	113
	11.14.2 Symbols	113
	11.14.3 Mapping Composites	115
	Example 16: Subfield to a subfield of the same name using different TEMPLATE	115
	Example 17: Subfield to a regular field	116
	Example 18: Regular field to a subfield	116
	Example 19: Mapping regular fields to a composite field	117
11.15	RANDOM	118
	Example 20: Random Field Attribute	118
11.16	FIELD ENDIANNES	120
11.17	Data Types (Single-Byte)	121
	Example 21: Converting Data Types	123
	Example 22: Sorting by IP Address	125
	Example 23: Using the BIT Data Type	125
11.18	Multi-Byte Character Types	127
	Example 24: Native Form 6 data type - Chinese GBK Simplified Sort ...	129
	Example 25: Native Form 6 data type - Chinese Big5 Sort	130

Example 26: Native Form 6 data type - Korean Hangul sort	131
Example 27: Unicode Form 6 data type - Japanese SJIS sort	132
Example 28: Unicode Form 6 data type - UTF16 Unicode sort	133
Example 29: Form 0 ASCII to Form 6 UTF16_UNICODE conversion ...	134
Example 30: Form 6 - Joining files with Chinese GBK Data	135
Example 31: Form 6 - Joining Files with Chinese Big5 Data	137
Example 32: Form 6 - Joining Files with Korean Hangul Data	139
Example 33: Form 6 - Joining Files with SJIS Japanese Data	141
12 FIELD_PREDICATE	143
12.1 Usage	143
Example 34: Using FIELD_PREDICATE	144
13 FIELD EXPRESSIONS (CROSS-CALCULATION)	146
Example 35: Mathematical Expressions	147
14 /INREC	150
Example 36: Using /INREC	150
15 /DATA	152
Example 37: Using /DATA	153
16 INTERNAL VARIABLES	154
17 CONTROL (ESCAPE) CHARACTERS	155
18 NON-ASCII FIELD EVALUATION AND CONVERSION SPECIFIERS	156
18.1 Using a Conversion Specifier within a Condition	157
18.2 Using a Conversion Specifier within a /DATA statement	158
19 ACTION	159
20 KEYS	160
20.1 Syntax	161
20.2 Field Name Reference	161
20.3 Unnamed Reference	162
20.4 Alternate Data Type	162
20.5 Alternate Collating Sequence: /ALTSEQ	163
Example 38: Using ALTSEQ	164
20.6 Direction	165
20.7 ASCII Options	166
20.8 Stability	167
Example 39: Using /STABLE	167
20.9 No Duplicates, Duplicates Only	168

21	CONDITIONS	169
21.1	Syntax	169
21.2	Unary Logical Expressions (Change Test)	170
21.3	Binary Logical Expressions	171
21.4	Compound Logical Expressions	172
21.5	Evaluation Order	172
21.6	Compound Conditions	173
22	INCLUDE-OMIT (RECORD SELECTION)	175
22.1	Syntax	175
22.2	Include-Omit Evaluation	175
22.3	Include-Omit Examples	176
	Example 40: Using /INCLUDE with WHERE	176
	Example 41: Using /INCLUDE with Named Conditions	177
	Example 42: Using Two /OMIT Statements	177
	Example 43: Condition Ordering: /INCLUDE first	178
	Example 44: Condition Ordering: /OMIT first	179
23	CONDITIONAL FIELD AND DATA STATEMENTS	180
23.1	Multiple IF THEN clauses	181
23.2	Function Compares in Conditions (iscompares)	182
	Example 45: Using iscompares	184
23.3	Conditional Field and Data Statement Examples	185
	Example 46: Simple /DATA Condition	185
	Example 47: Multi-Level Condition Based on One Field	186
	Example 48: Multi-Level Condition Based on Two Fields	187
	Example 49: Multi-Level Condition with Empty Output	188
	Example 50: Multi-Level Condition with Delimited Fields	189
24	FIELD FUNCTIONS	191
24.1	Field Length Function	192
	Example 51: Using the Field Length Function	192
24.2	ASCII Substrings	193
	Example 52: Using ASCII Substrings	194
24.2.1	INSTR	195
	Example 53: Using INSTR	195
24.3	Replace characters	196
24.4	Format Strings	196
24.5	Universally Unique Identifier	197
24.6	URL Encoding	197
24.7	Set files	198
24.7.1	Using Multi-Column Set Files	202
	Example 54: Substituting Sensitive Data Values	202
	Example 55: Pseudonymization	204

Example 56: Multiple Column Set File Example	206
24.7.2 Slowly Changing Dimensions Reporting	208
Example 57: Two-column SCD reporting	209
Example 58: Four-column SCD reporting	210
24.8 Find and Replace	212
Example 59: Using All Case Transfer Functions	216
24.9 Date Intervals	216
Example 60: Calculating Date Intervals	217
24.10 Named Fields With a Literal (fixed) Value	218
Example 61: Using a Literal Value	218
24.11 De-identification and Re-identification	220
Example 62: De-identifying Data	221
Example 63: Re-identifying Data	222
24.12 Custom Field-Level Functions (External Transformations).....	223
25 /JOIN.....	224
25.1 Syntax	225
25.2 /JOIN Examples	228
Example 64: INNER Join	229
Example 65: LEFT_OUTER and RIGHT_OUTER Join	229
Example 66: FULL OUTER Join	231
25.3 JOIN ONLY	232
25.4 Unordered (Unconditional) Joins	233
26 SUMMARY FUNCTIONS (AGGREGATION)	234
26.1 Summary, Average, and Standard Deviation	235
Example 67: Using the running lag count for windowing	237
26.2 Maximum and Minimum	239
Example 68: Calculating Maximum and Minimum Values	240
26.3 Counting	241
Example 69: Using /COUNT	242
26.4 Ranking.....	242
Example 70: Ranking Values	243
26.5 Creating New Files Based on BREAK (/NEWFILE).....	245
Example 71: Using /NEWFILE for Multiple Output File Creation	246
Example 72: Using /NEWFILE with Aggregation	247
26.6 Reports with Summaries.....	249
26.7 Running (Accumulating) Aggregates.....	251
Example 73: Running Summary	251
Example 74: Running Summary with Subtotals	252
Example 75: Running Using a Lag Count	253
27 SEQUENCER.....	255
Example 76: Using Sequencer	256

28	RECORD FILTERS	259
28.1	Input Options	260
	Example 77: Using /HEADREAD and /HEADWRITE	262
	Example 78: Using /TAILREAD and /TAILWRITE	263
	Example 79: Using /INSKIP	264
	Example 80: Using /INSKIP and /INCOLLECT	265
28.2	Output Options	268
	Example 81: Using /RECSPPERPAGE	273
	Example 82: Using /HEADREC, /OUTSKIP, and /OUTCOLLECT	275
29	MISCELLANEOUS OPTIONS	276
29.1	/RC	276
29.2	/EXECUTE	277
29.3	/MONITOR	277
29.4	Runtime Warnings (/WARNINGSON and /WARNINGSOFF)	280
29.5	/ERRORCOUNT	280
29.6	/ROUNDING	281
29.7	/LOCALE	281
	Example 83: Using /LOCALE with Currency	282
	Example 84: Using /LOCALE with Dates	284
30	SORTCL JOB SCRIPTING EXAMPLES	285
	Example 85: Simple Script Form	285
	Example 86: Two-Key Sort	286
	Example 87: Less Verbose	286
	Example 88: Two-Key Sort with Delimited Fields	287
	Example 89: Delimited-to-Fixed Format Change	288
	Example 90: Using /CHECK	288
	Example 91: Using /MERGE	289
	Example 92: Using a Data Definition File	290
	Example 93: Multiple Output Files and Formats	291
	Example 94: Conditional Output Selection	292
	Example 95: Multiple Output Files with Conditional Selection	293
	Example 96: Carrying a Condition Forward	294
	Example 97: Report with Complex Selection	295
	Example 98: Alternate Report with Complex Selection	296
	Example 99: Multiple Input Files with Same Formats	296
	Example 100: Multiple Input Files with Different Formats (/INREC)	297
	Example 101: Removing Duplicate Records	298
	Example 102: Summary Fields with Breaks	299
	Example 103: Summary Functions with Detail, Break, and Total Records	300
	Example 104: Using WHERE with Sums, Cross-Calcs on Sums	301
	Example 105: Multiple Output Files with Summaries	303
	Example 106: /OUTSKIP and /OUTCOLLECT - Mixed Record Types	304
	Example 107: Piping from a Sort to an Inner Join; Sort with Cross-Calcs	306
	Example 108: Multi-Table Join	309

Example 109: Joining a Fourth Table	311
Example 110: Using ALTSEQ with Includes	312
Example 111: Creating an HTML-formatted output file	314
Example 112: Calling sortcl from a Batch Script, Using /EXECUTE	316
Example 113: Pivoting Columns to Rows, with Summaries	322
Example 114: Updating Data, separate current and history records	324
Example 115: Updating Data, current and historical values in a single record .	326
 31 SORTCL STATEMENTS	 328
 CUSTOM TRANSFORMS	 345
1 WRITING LIBRARIES FOR CUSTOM DATA PROCESSING	346
1.1 Syntax	346
1.2 Custom Procedure Declaration	347
1.3 Custom Procedure Arguments	348
1.3.1 Source Arguments	348
1.3.2 Result Argument	350
1.3.3 Return Value	351
1.4 Sequence of Custom Procedure Calls	352
1.4.1 Initialization Call	352
1.4.2 Processing field level transformations for each record	352
1.4.3 Termination	353
1.4.4 Example	354
2 ASCII/ALPHANUM ENCRYPTION AND DECRYPTION	358
2.1 Usage: ASCII Encryption and Decryption	359
2.2 Syntax: ASCII/ALPHANUM Encryption	359
2.3 Syntax: ASCII Decryption	361
2.4 Advanced Encryption Standard (AES)	362
Example 116: Standard Encryption	362
Example 117: Standard Decryption	365
2.5 GPG Encryption and Decryption	366
Example 118: GPG Encryption	367
Example 119: GPG Decryption	369
2.6 Format-Preserving Encryption and Decryption	369
Example 120: Width-Preserving Encryption	370
Example 121: Width-Preserving Decryption	371
2.7 Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256)	372
Example 122: Alpha-Numeric Format-Preserving Encryption	372
Example 123: Alpha-Numeric Format-Preserving Decryption	374
Example 124: Using ASCII Encryption on Database Columns	375

2.8	Triple Data Encryption Standard (3DES).....	378
	Example 125: Triple DES encryption	378
	Example 126: Triple DES decryption	380
2.9	Encoding and Decoding	380
2.10	SHA-2 Hashing.....	381
	Example 127: Using SHA-2	381
3	MELISSA DATA CLEANSING ROUTINE.....	383
3.1	Input Arguments	383
3.2	Syntax	384
	Example 128: Address Standardization Using Melissa Data	385
4	TRILLIUM CLEANSING ROUTINE.....	387
4.1	Input Arguments	387
4.2	Syntax	388
4.3	Table of Errors	390
	Example 129: Address Standardization Using Trillium	391
sortel TOOLS		393
cob2ddf		395
1	PURPOSE.....	395
2	USAGE.....	396
3	EXAMPLE	397
ctl2ddf		399
1	PURPOSE.....	399
2	USAGE.....	399
3	EXAMPLE	400
csv2ddf		401
1	PURPOSE.....	401
2	USAGE.....	402

3	EXAMPLE	403
	elf2ddf	405
1	PURPOSE.....	405
2	USAGE.....	406
3	EXAMPLE	407
	ldif2ddf	409
1	PURPOSE.....	409
2	USAGE.....	409
3	EXAMPLE	410
	odbc2ddf	411
1	PURPOSE.....	411
2	USAGE.....	412
3	EXAMPLE	413
	xml2ddf	417
1	PURPOSE.....	417
2	USAGE.....	418
3	EXAMPLE	419
	CLF TEMPLATES	421
1	PURPOSE.....	421
2	USAGE.....	422
3	EXAMPLES	423
3.1	Common (Access) Log.....	423

3.2	Referral Log	424
3.3	Agent Log	425
sorti2scl		427
1	PURPOSE	427
2	USAGE	428
2.1	Execution	428
3	EXAMPLES	429
3.1	Example #1	429
3.2	Example #2	430
4	ADDITIONAL FUNCTIONS OF SORTCL	432
mvs2scl		433
1	PURPOSE	433
2	USAGE	434
2.1	Execution	434
3	CONVERSION RULES	436
4	EXAMPLES	438
4.1	Example #1—OMIT Statement	438
4.2	Example #2—Summary Output	441
vse2scl		445
1	PURPOSE	445
2	USAGE	446
2.1	Execution	446
3	CONVERSION RULES	448
4	EXAMPLES	450
4.1	Example #1 - Two-Key Sort	450
4.2	Example #2 - Omit Condition	451

COBOL TOOLS	453
1 OVERVIEW	453
2 WORKBENCH 4.1 FOR UNIX	455
3 WORKBENCH 4.0 AND NET EXPRESS 2.X FOR WINDOWS	457
4 MF COBOL SERVER EXPRESS SORT REPLACEMENT FOR UNIX	459
4.1 JCL Sort Replacement	460
5 MF COBOL NET EXPRESS SORT REPLACEMENT FOR WINDOWS	461
6 ACUCOBOL SORT REPLACEMENT FOR UNIX	462
7 BATCH-MODE SORTS	463
7.1 Using sortcl from the Prompt	464
7.2 Calling sortcl from within a COBOL Program	464
7.2.1 Literal-String Passing	464
7.2.2 Concurrent Processing Extensions in Unix	465
7.2.3 Supported COBOL Data Types	465
7.2.4 Micro Focus Line-Sequential Records	467
7.2.5 Micro Focus Variable-Length Records	467
7.2.6 Using Copy-In Source Files	468
8 SORTCL INCLUDE FILES	471
9 ACUCOBOL-GT VISION RECORDS	473
9.1 Enabling CoSort Support for Vision Files	473
9.2 Usage	475
10 MICRO FOCUS COBOL ISAM RECORDS	478
10.1 Enabling CoSort Support for MF ISAM Records	478
10.2 Usage	478
sort PROGRAM	481
1 PURPOSE	481
2 PERFORMANCE CONSIDERATIONS	482
3 EXECUTION	484
3.1 Input Sources	484
3.1.1 Single Input File	485
3.1.2 Multiple Input Files	485

3.1.3	Standard Input	486
4	FEATURES	487
4.1	Ordering Options	488
4.1.1	Dictionary Order	488
4.1.2	Fold Lowercase into Uppercase	488
4.1.3	Ignore	488
4.1.4	Numeric Sort	488
4.1.5	Reverse Order	489
4.2	Field Separator Options	489
4.2.1	Blanks	489
4.2.2	Field Separator Character	490
4.3	Defining Key Fields	490
4.3.1	Field Specifiers	490
4.3.2	Alternate Field Specifiers	493
4.4	Output Flags	495
4.4.1	Output File	495
4.4.2	Standard Output	495
4.5	Performance Flags	496
4.5.1	Temporary Directory Assignment	496
4.5.2	Memory Allocation Technique	497
4.5.3	I/O Buffer Setting	497
4.5.4	Version Display	497
	USING COSORT WITH THE SAS SYSTEM	499
1	PURPOSE	499
2	EXECUTION	499
3	MAKING COSORT AVAILABLE	499
3.1	For AIX	499
3.2	For Compaq Tru64 Unix, Intel ABI, IRIX, and Solaris	500
3.3	For HP-UX	500
4	OPTION STATEMENTS	501
	USING COSORT WITH SOFTWARE AG NATURAL	503
1	PURPOSE	503
2	DIRECT CALLS TO SORTCL	503
3	USING THE COSORT SORT REPLACEMENT	504

COSORT LOAD ACCELERATOR FOR DB2 (CLA4DB2)	505
1 PURPOSE	505
2 ENABLING CLA4DB2 ON YOUR SYSTEM	505
sorti PROGRAM	507
1 PURPOSE	507
2 AN INTERACTIVE SESSION	508
2.1 Start-up	508
2.2 A Sample Sort	511
3 SORTING	516
3.1 Record Length: 0 (variable (default)) / fixed:	517
3.2 Number of files: 0 (proc) / +- numb (1 default):	518
3.3 Input file #x:	518
3.4 Keys [number] [stable] [unique / dups only] (default) 1:	519
3.5 Key x Direction: ASCENDING (default) or DESCENDING:	519
3.6 Location: FIXED (default) / <separator char>:	520
3.7 Starting byte position: 1 (default):	521
3.8 Field Size: 65535 (maximum) 50 (default):	522
3.9 Form: ALPHA (def), NUM, DATE, TIMESTAMP:	522
3.10 Type ASCII (default) or LIST to show options:	523
3.11 Alignment: NONE (default), LEFT, or RIGHT:	523
3.12 Case fold letters: NO (default) or YES:	523
3.13 OUTPUT: TERMINAL (default), FILE, BOTH or PROGRAM:	524
3.14 Output file name:	525
3.15 EXECUTE with these specifications? YES (default) or NO:	526
3.16 Execution Time	526
3.17 Save specifications for reruns? YES or NO (default):	526
3.18 Specifications file name:	526
4 MERGING	527
4.1 A Sample Merge	528
5 DISPLAY	531
6 SPECIFICATION ENTRY GUIDE	532
7 ENDING INTERACTIVE	533

8	BATCH OPERATIONS	534
9	BATCH EXECUTION	535
10	SPECIFICATION SOURCES	536
11	SPECIFICATIONS FORMAT	537
12	SPECIFICATION SUMMARY	538
13	RUN-TIME ERROR CONDITIONS	540
API (Application Program Interface)		541
1	OVERVIEW	541
1.1	Calling the CoSort APIs	542
2	sortcl_routine()	543
2.1	Usage	543
2.1.1	sortcl_alloc()	543
2.1.2	sortcl_routine()	544
2.1.3	sortcl_free()	545
2.2	Linking with sortcl_routine()	545
2.2.1	Linking on Unix/Linux	545
2.2.2	Linking on Windows	546
2.3	sortcl_routine() Examples	546
2.3.1	Basic sortcl_routine() Example	547
2.3.2	Using sortcl_routine() with a Summary Function	548
2.3.3	Calling sortcl_routine() Using Java	551
2.3.4	Calling sortcl from within a Java Program	554
3	CUSTOM INPUT, OUTPUT AND COMPARE PROCEDURES	556
3.1	Custom Input	556
3.1.1	Syntax: /INPROCEDURE	556
3.1.2	Custom Input Procedure Declaration	557
3.1.3	Registering the Custom Input Procedure	557
3.1.4	Building the Custom Input Procedure for Runtime Linking	558
3.2	Custom Output	559
3.2.1	Syntax: /OUTPROCEDURE	559
3.2.2	Custom Output Procedure Declaration	560
3.2.3	Registering the Custom Output Procedure	560
3.2.4	Building the Custom Output Procedure for Runtime Linking	561
3.2.5	Examples	561
3.3	Custom Compares	562
3.4	Custom Record Compare Using /KEYPROCEDURE	562

3.5	Custom Key Compares Using /KEY	562
3.6	Implementation of Custom Compares	563
3.6.1	Custom Compare Procedure Declaration	563
3.6.2	Building the Compare Procedure for Runtime Linking.	564
3.7	Examples using Custom Input, Output, and Compare Procedures.	565
3.7.1	Custom Input and Output using sortcl_routine	565
3.7.2	Custom Compare Using sortcl_routine	567
4	cosort_r()	570
4.1	Usage	570
4.1.1	cosort_alloc()	571
4.1.2	cosort_r()	571
4.1.3	cosort_free()	572
4.2	cosort_r() Initialization Stage	572
4.3	cosort_r() Input Definition Stage	573
4.4	cosort_r() Compare Keys Setup Stage	574
4.4.1	CoSort Key Structure Format	575
4.4.2	Non-C Specification	577
4.4.3	cosort_r() Custom Compares	578
4.4.4	Custom Compare Procedure Declaration	579
4.4.5	Building the Compare Procedure for Runtime Linking.	580
4.4.6	Example	580
4.5	cosort_r() Output Definition Stage	580
4.6	cosort_r() Execution Stage	582
4.6.1	Simple Execution	582
4.6.2	Passing Records to cosort_r() via a Buffer	582
4.6.3	Receiving Records from cosort_r() via a Buffer	583
4.6.4	End of Job	583
4.7	cosort_r() Error Handling	584
4.8	Final Record Count	584
4.9	Linking with cosort_r() on Unix or Linux	584
4.9.1	Linking with libcosort.a	584
4.10	Linking with cosort_r() On Windows	585
4.10.1	Linking with the CoSort Library	585
4.11	cosort_r() API Examples	586
4.11.1	Simple Sort	587
4.11.2	Passing Records to cosort_r() Using a Buffer	590
4.11.3	Multi-File, Multi-Key Sort	594
4.11.4	Custom Input Selection	597
4.11.5	Custom Record Compare using cosort_r()	601
4.11.6	Custom Key Compare using cosort_r()	605
4.11.7	COBOL Program with a C Routine that Calls cosort_r()	608
	APPENDIX	611

A	DATA TYPES	611
A.1	Form 0 -- Alphabetic Data Types	612
A.2	Form 1 -- The Numeric Types	620
A.2.1	C Types	621
A.2.2	RM COBOL Data Types	626
A.2.3	MF COBOL Data Types	627
A.2.4	EBCDIC Native RM COBOL Data Types	630
A.2.5	EBCDIC Native MF COBOL Data Types	631
A.3	Forms 2-4 -- Date/Time/Timestamp Data Types	631
B	RECORD TYPES	633
B.1	Variable-length Records	633
B.1.1	Single Records	633
B.1.2	Multiple Logical Records (File Names)	633
B.2	Fixed-length Records	636
C	SYSTEM ROUTINES	637
C.1	Library Routines	637
C.2	Signal Processing	638
D	PERFORMANCE TUNING	639
D.1	Tuning CoSort for Windows	641
D.2	Tuning CoSort for Unix	642
D.3	Using Customized Resource Control Files	643
D.4	Search Order for Resource Controls	645
D.5	Resource Control Settings	647
D.5.1	Optional Settings	654
D.6	Benchmarking	659
E	ERROR and RUNTIME MESSAGES	660
E.1	Table of Error Values	661
E.2	Detailed Error and Runtime Messages	669
E.3	Unknown Exception Error -- Logging	680
F	ASCII COLLATING SEQUENCE	680
G	EBCDIC PRINTING CHARACTERS	681
	GLOSSARY	683

INTRODUCTION

1 PURPOSE

Selecting, sorting, joining, aggregating, type-converting, and reformatting are among the basic elements of transforming data into information. The fundamental action of sorting consumes processing time, disk space, memory and other computer resources exponentially. So when data sizes double, hardware needs can triple.

These considerations have led Innovative Routines International (IRI) to design and introduce the world's first commercial sort packages for open systems: **CoSort** for CP/M (1978), MS-DOS (1980), Unix (1986), Windows (1996), IBM eServer iSeries - AS/400 (2001), and IBM's eServer zSeries mainframe (Linux, 2003).

CoSort meets the needs of database and data warehouse installations with large data volumes, many data types, complex queries, and multiple CPU cores. **CoSort** also delivers:

- tools for converting from legacy applications and data types to open systems
- runtime libraries for embedded sorting and application acceleration
- a formatting language for complex transformation and reporting

As a result of its history of user adoption and integration, **CoSort** has become the world's most widely licensed sort package off the mainframe. **CoSort** users include data warehouse (ETL) architects, data center managers, DBAs, MIS report writers, systems analysts, application developers, and consultants.

The **CoSort** package contains the *coroutine sort* engine and a large collection of user interfaces and third-party sort replacements that use that routine. Most powerful among these is an award-winning¹ sort control language (**sortcl**) program. Programmers can also write their own calls to **CoSort**'s utilities and/or libraries directly.

The material in this manual describes:

- **CoSort**'s standalone utility programs (command-line interfaces)
- calls, conversion programs, and user exits for **sortcl** specifications
- third-party sort replacements
- **CoSort**'s **API** (Application Program Interface) calls

Other than the Introduction, Appendix, and Glossary, each chapter describes a specific **CoSort** interface or script/metadata conversion utility. A cursory reading of each chapter will allow you to find the most applicable tools for your requirements.

1. DM Review Readership Award, 2000.

2 COMPATIBLE APPLICATIONS

Additional IRI products are available to run in conjunction with **CoSort** and make use of its **sortcl** program language metadata in particular:

IRI Workbench



Workbench is a Graphical User Interface (GUI) and Integrated Development Environment (IDE) for **CoSort** built on Eclipse™.

FACT is high-performance unload utility for Oracle, DB2, Sybase, SQL Server, MySQL, Altibase, and Tibero. **FACT** is typically used in data warehouse extract-transform-load (ETL) and ELT operations, database archiving and migration scenarios, and to improve the performance of offline reorg operations.

- **FACT** extracts huge tables to flat files in parallel, using simple config files with SQL SELECT queries.
- **FACT** features built-in data conversion and reformatting options to change data types and layouts.
- **FACT** writes the extract metadata for CoSort staging and reporting scripts, and for bulk, pre-CoSorted, loads, making **FACT** an essential acquisition component in big data integration and staging.
- **FACT** runs with CoSort and DB loaders in the IRI Workbench (Eclipse GUI) for offline reorg and ETL. ♦

FieldShield

FieldShield is a targeted data protection solution for data at rest. **FieldShield** is designed to assist in data privacy law compliance, data loss prevention, and breach risk mitigation.

- **FieldShield** secures data at risk in databases and flat files by applying protections to each field according to the business rules.
- **FieldShield** can encrypt, hash, mask, randomize, redact, and otherwise de-identify data without cutting off access to data sources or changing their formats.
- **FieldShield** protection rules secure pattern-matched column names across tables in the same way, saving specification time and preserving referential integrity.
- **FieldShield** creates an XML audit log of each operation to prove that you protected the data. ♦

NextForm

NextForm converts, replicates, and federates data in databases and files. **NextForm** is typically used to migration data from mainframes and legacy databases for use in new applications and platforms.

- **NextForm** moves and maps data between DB2, MySQL, Oracle, SQL Server and Sybase, and facilitates new database creation.
- **NextForm** changes files into different formats, including: LDIF and CSV, MFVL and XML, MF-ISAM and Vision, structured and unstructured text.
- **NextForm** translates data types, including: EBCDIC to ASCII, packed decimal to numeric, ISO to European timestamp, Big5 to Unicode, etc.
- **NextForm** can also create multiple targets for replication or federation, and reformat and validate field and record layouts.

RowGen

RowGen builds structurally and referentially correct test data in the same form and format of realtables, files, and reports – without real data. **RowGen** is used to build and test applications, populate data warehouses, simulate and outsource file and report formats, and build benchmarking data sets when production data is classified or unavailable.

- **RowGen** simulates production tables, files, and reports with realistic and referentially correct test data faster than any other method.
- **RowGen** automatically parses data models, generates the test data, and populates its targets all at once.
- **RowGen** combines random data generation, lookup table selection, and data manipulation to enhance test data realism and protect privacy.
- **RowGen** uses your data models, the metadata in CoSort (and all IRI tools), or Meta Integration Model Bridge (MIMB) repositories, to match your production formats.

These tools are available as separate applications. For more information, and to sign up for a free trial, visit www.iri.com/products.

3 NOTATION

To distinguish files, procedures, user entries, and so on, certain typeface conventions are applied throughout the documentation:

- Fonts
- Symbols *on page 25*
- CoSort Entities *on page 25*
- Hyperlinks *on page 26*

3.1 Fonts

The following fonts are used throughout the documentation:

<i>italics</i>	are used for replaceable entries, cross-references, or for adding emphasis to a special word or term, e.g.: <ul style="list-style-type: none">• type <i>filename</i>• see Syntax <i>on page 92</i>• as described on the Unix <i>man</i> page
literal	is used for entries that you make exactly as shown, variable names, function names, or any output text as it appears literally on screen, e.g.: <ul style="list-style-type: none">• to initiate a merge, type <code>merge</code>• the <code>COSORT_TUNER</code> variable• the <code>memcmp()</code> function• a <code>Permission denied</code> message is displayed
bold	indicates a directory name or file name, e.g.: <ul style="list-style-type: none">• /export/home/cosort directory• chiefs data file• sortcl program file
[<i>option...</i>]	when placed in brackets, indicates an option. The ellipses show possible repetition, e.g.: <pre>sortcl /spec=<i>filename</i> [/stat / ...]</pre>
<u>underlined blue</u>	is used for external links.

3.2 Symbols

The following symbols are used throughout the **CoSort** documentation:



denotes the beginning of text that applies only to Unix systems.



denotes the beginning of text that applies only to Windows systems (that is, Windows NT, XP, 2000, 2003, 2008. and 7).



denotes text that applies only to multi-processor jobs



denotes the end of text that applies to the above symbols

3.3 CoSort Entities

Several entities whose name is, in part, **CoSort** are used throughout the documentation, as follows:

CoSort is a general reference to the full package and its capabilities, e.g.:

CoSort can sort binary records.

\$COSORT_HOME refers to **CoSort**'s upper-level directory on Unix. For example, if **CoSort** were installed in **/export/home/cosort95**, then **\$COSORT_HOME** would reference that directory.

On Windows, *install_dir* is used to refer to the location where the product is installed (by default, this is `\Program Files\IRI\CoSort95`).

cosort_r() is the thread-safe engine for the sort/merge coroutine used by **CoSort**'s utility programs, and is referenced in application software. An example of a

C or Java program call would be:

```
cosort_r(cs_cosort_t* cosort)
```

sortcl.1 is the man page for the **sortcl** program executable.



libcosort.a

is an archive file, found in **\$COSORT_HOME/lib** on Unix, which is often placed in the system library which contains the **cosort_r()** procedure. The **sort**, **sorti**, and **sortcl** programs, and your application programs link to **libcosort.a**, or the shared library (dynamic) version, **libcosort**. ◆


W**libcosort_static.lib**

is an archive file, found in the *install_dir*\lib subdirectory on Windows, which contains the `cosort_r()` procedure. The **sorti** program, as well as **unixsort**, **sortcl**, and your application program links to **libcosort_static.lib**, or the shared (dynamic) library version, **libcosort.dll**. ♦

cosort.h is the header file which contains defines, messages, and declarations that are used when application programmers call the `cosort_r()` procedure in C with:

```
#include "cosort.h"
```

3.4 Hyperlinks

When viewing the **CoSort** User Manual with Adobe Acrobat Reader, you can click an italicized cross-reference (beginning with see...) to link to its destination. A pointing finger symbol  appears as you mouse over each hyperlink.

4 **PRODUCT OVERVIEW**

CoSort, for **Coroutine Sort**, is a general-purpose data processing solution set for:

- data warehouse Extraction, Transformation, and Loading (ETL)
- Very Large Database (VLDB) reorganizations and load operations
- detail, summary and delta (change data capture) reporting
- mainframe sort migration and third-party sort upgrades
- new applications requiring sort functionality
- heterogeneous data integration (files and tables)

Most **CoSort** users are concerned with high-volume sort performance. **CoSort** produces uniformly excellent results on randomly distributed key values for a variety of sequential file and table formats. In large sorting operations on a single CPU, **CoSort** runs significantly faster than system sort commands and can use (but does not require) more of the memory and disk storage resources of the system. **CoSort** can be used when input data exceed memory, and can take direct advantage of multiple CPU cores.

In traditional operations, particularly those converted from mainframe environments, the task is to sort or merge records from one or more input files to one or more output files using one or more keys. All three native **CoSort** utilities do this: **sort** (**unixsort** on Windows), **sorti**, and **sortcl**¹. They allow you to quickly specify, test, and execute a sort, merge, join, or report, and to save the specifications for batch jobs. In addition, **CoSort** drop-in sort replacements are available for many third-party sort applications. **CoSort** also includes conversion tools to translate z/OS, VSE JCL, and Windows sort parms, as well as several third-party metadata formats, into **sortcl** data definitions. The **CoSort** sort control language (**sortcl**) program is also familiar to VAX VMS sort users, and is easily called from ETI*Extract and other applications that support the integration of string parameters and system calls.

Although these utilities have different interfaces with increasing levels of capabilities, they all call the same `cosort()` coroutine sort engine. **CoSort**'s **APIs** allow software developers to call this coroutine in their own language for unlimited job definition flexibility.

The engine of **CoSort** is a minimal-time, multi-threaded sort/merge algorithm in a coroutine architecture. **CoSort**'s unique coroutine allows a continuous exchange of control where records are moved in memory, either one-by-one or in blocks. The process can be visualized as a stream of records moving through memory from the caller to the sort, and as ordered records returning from the sorter back to the caller for output.

There are many benefits to **CoSort**'s coroutine approach, including:

-
1. **sortcl** is the most powerful **CoSort** data manipulation utility. **sortcl** also features a Java GUI and callable API. Several metadata and script conversion tools facilitate migrations to, and the use of, **sortcl**.

Flexibility

- unlimited selection criteria
- hundreds of data types
- any number of keys
- unlimited complexity of keys

Efficiency

- single pass through the database (flat files)
- virtual record processing
- no intermediate I/O (for transfer files)
- immediate access to returning records
- no duplicate data files
- smaller sort program
- smaller application program

Ease of Use

- supports file logic, record logic, or both
- no intermediate file programming
- solution language is the user's

Control

- to service real-time events
- access to global program data

Report Generation

- The immediate send/receive logic and flexibility of conditions make **CoSort** desirable for interactive SQL and third-party report generators.
- At request time, the reporting program can control the input record, the compare keys, and the output presentation

Reduced Development Time

- By eliminating the intermediate files that hold both sort input and sort output, the program is simplified and development time is reduced.
- Where selection criteria, special compares, etc. are required, the programmer works in his own language.

Real-Time Operations

- When control is returned to the caller to satisfy a sort need, the caller can service real-time events before returning to the sort.

Encrypted, Mixed Forms, and User-Defined Data Types

- Because **CoSort** can process binary data and the user can control input, compare, and output functions, the contents of records need not be known to **CoSort**.
- Encrypted data can be sorted.
- Keys with mixed data types and different measures can be compared to each other, and collating sequences can be defined at run-time.

4.1 Sort/Merge Module

The central `cosort()` routine¹ performs all sorting and merging functions for all interfaces in the **CoSort** package. End users do not communicate directly with the `cosort()` routine. Once specifications are received from a calling program, `cosort()`:

- 1) receives input records
- 2) sorts them according to specified keys
- 3) generates output records

The unique coroutine capability allows the input, compare, and output functions to be performed either by the `cosort()` procedure or by the caller.

CoSort has facilities for an infinite sort and will accept input records as long as there is sufficient overflow space. In large sort/merge operations when the incoming data overflows the prescribed memory, the data that has been read thus far is sorted and written to temporary files for later merging. Unless **CoSort**'s on-screen monitor is on, this overlay processing is transparent to the user. If there is no more input, the sorted and/or merged data is written to the specified output files or returned to the user. At the conclusion of the sort or merge, all work files are purged automatically.

1. **CoSort**'s `cosort()` library is central to the package's standalone utility programs and other API calls (see *User Programs (API)* on page 32).

4.2 User Interfaces

The **CoSort** package consists of callable sort engines, several standalone utilities, and third-party sort replacements.

The front-end interfaces to the **CoSort** sort engine are:

sortcl An SQL-oriented, mainframe-familiar *sort control language* to define simultaneous, multi-file, multi-format sort, join, aggregate, and reformat jobs for database loading, integration, data warehouse staging, and reporting (see the *sortcl PROGRAM chapter on page 37*).



For standalone use, it is strongly recommended that you use the **sortcl** interface because it supports the full range of data manipulation features offered by **CoSort**. Re-usable data definitions (metadata) can be stored in centralized repositories. The **sort** and **sorti** programs (described below) continue to be supported for legacy users of **CoSort**.

sort **unixsort.exe** on Windows. Drop-in replacement for the *sort* supplied with Unix operating systems, but is several times faster and works in large volume (see the *sort PROGRAM chapter on page 481*).

sorti A query-response *sort interactive* session suitable for non-programmers with in-context help to define parameters for immediate and batch execution (see the *sorti PROGRAM chapter on page 507*).

Several **sortcl** tools leverage and upgrade third-party metadata sources and sort specifications:

cob2ddf Converts COBOL copybook layouts into a **sortcl** data definition file (see the *cob2ddf chapter on page 395*).

csv2ddf Examines headers of a Microsoft CSV (Comma-Separated Values) file and generates a **sortcl** data definition file (see the *csv2ddf chapter on page 401*).

ctl2ddf Converts Oracle SQL*Loader control file metadata into a **sortcl** data definition file (see the *ctl2ddf chapter on page 399*).

elf2ddf Examines headers of a web transaction file in *Extended Log Format* and generates a **sortcl** data definition file (see the *elf2ddf chapter on page 405*).

- ldif2ddf** Converts the column layouts specified in an LDIF (Lightweight Directory Interchange) format into a **sortcl** data definition file (see the *ldif2ddf chapter on page 409*). LDIF is the format of data exported from an LDAP database.
- odbc2ddf** Converts database table layouts into a **sortcl** data definition file, provided the database is compatible with ODBC (Open Database Connectivity). See the *odbc2ddf chapter on page 411*.
- xml2ddf** Scans **XML** (eXtensible Markup Language) files to produce descriptive file name and field-layout text that can be referenced by, or pasted directly into, a **sortcl** job specification file.
- mvs2scl** Converts mainframe z/OS JCL sort parms to **sortcl** job specifications (see the *mvs2scl chapter on page 433*).
- vse2scl** Converts mainframe VSE JCL sort parms to **sortcl** job specifications (see the *vse2scl chapter on page 445*).

Each of these interfaces can be executed from the command line, a shell script, or batch file. In addition, **sorti** and **sortcl** recognize environment variables and references to *stdin* and *stdout* to allow a continuous flow between processes.

sorti and **sortcl** can also be customized with user exits -- procedures you write for special input, output, data conversion, and/or key compare criteria.

There is also a graphical user interface (GUI) built on Eclipse called **IRI Workbench**. The **IRI Workbench** facilitates the specification, execution, tuning and maintenance of **sortcl** job scripts through job creation and metadata definition wizards, a dynamic job outline, and a fully syntax-aware editor for manual **sortcl** specification. The **IRI Workbench** also facilitates database data access, viewing, and integration into **sortcl** jobs, and includes extensions for team contributions, job version control, and remote system data sourcing.

Following are additional sort tool replacements within all **CoSort** packages:

- acucosort** Drop-in replacement for ACUCOBOL-GT's sort for Unix (see the *COBOL TOOLS chapter on page 453*).
- cla4db2** Drop-in replacement for the sort engine in IBM's UDB load utility for DB2 v6-9.5 on Unix and Windows. By invoking **CoSort**, bulk load speeds can double.
- mfcosort** Drop-in replacement for the sort verbs supplied with Micro Focus COBOL compilers on Unix and Windows. COBOL programmers can link statically or dynamically to **mfcosort** and the **CoSort** engine to accelerate sort speed

and reduce temporary sort space in new executables or a full RTS (see the COBOL TOOLS *chapter on page 453*).

PROCsort SAS System 7 and 8 users can link dynamically to shared `cosort()` libraries on Unix systems to replace the sort function in SAS (see the USING COSORT WITH THE SAS SYSTEM *chapter on page 499*).

SORT Software AG Natural users can also use the **CoSort** library on Unix systems to sort faster by linking to a modified makefile (see the USING COSORT WITH SOFTWARE AG NATURAL *chapter on page 503*).

Similar **CoSort** facilities are also available for:

- ACUCOBOL-GT
- Amdocs Ensemble
- IBM WebSphere (Ascential) DataStage
- Cincom Supra
- Informatica PowerMart/Center
- Micro Focus COBOL
- SAS
- Software AG Natural
- Clarity Unikix MBM and MTP
- Pervasive's DataJunction / DataRush (as a utility call)

and other third-party products calling an external sort (see PRODUCT OVERVIEW *on page 27*). ETI*EXTRACT template functions can also be written to build and run **sortcl** scripts.

Note that other, additional CoSort drop-in sort replacement facilities are available for IBM WebSphere DataStage Server Edition and Informatica PowerCenter.

4.3 User Programs (API)

The ultimate software component is the user's own program. Your programs offer the best opportunities for integrating sorts and/or merges directly into application streams via **CoSort API** calls. Calls to **CoSort** libraries can be written in C, C++, COBOL, Fortran, Java, Visual Basic, and so on.

CoSort's unique coroutine architecture allows conventional sorting as well as record-level operations for ad hoc report generation and other unique situations. The caller can direct the thread-safe `cosort_r()` routine to read from files, take input from the caller, or both. See the API (Application Program Interface) *chapter on page 541* for complete details.



CoSort continues to support the use of the `mcs()` routine (Multi-CoSort API), which was a precursor to the current `cosort_r()` routine.

The `sortcl` API is also available to **CoSort** developers. The `sortcl_routine()` call provides a direct thread-safe interface between your application and **sortcl**. This gives you inline and online access to all the **sortcl** functionality that is available in the standalone program (see `sortcl_routine()` on page 543).

4.4 CoSort Technical Specifications

FUNCTIONS

- | | | | | |
|-----------|---------------|-----------|----------------|-------------|
| • Audit | • De-dupe | • Lookup | • Pseudonymize | • Select |
| • Average | • De-identify | • Mask | • Rank | • Sequence |
| • Check | • Derive | • Match | • Reformat | • Sort |
| • Convert | • Encrypt | • Math | • Re-identify | • Stability |
| • Copy | • Find | • Maximum | • Replace | • Substring |
| • Count | • Generate | • Merge | • Report | • Sum |
| • Decrypt | • Join | • Minimum | • Round | • Trig |

INPUTS

Sources	stdin and/or any number of files, ODBC-connected relational database tables, records from an input procedure, named and unnamed pipes.
Size	No limit.
Length	$1 \leq \text{fixed-length} \leq 65,535$ bytes per record. $0 \leq \text{variable-length} \leq 65,535$ bytes per record.
Program	Optional, user-written selection processing and/or special sources.
Bulk RDBMS	Unload tables via IRI's FAsT extraCT (FACT) product.

KEYS

Number	Unlimited.
Location	Fixed-position and variable (floating field).
Direction	Ascending/descending.
Types:	
Character strings	ASCII, EBCDIC, ASCII in EBCDIC order, days of the week, months of the year, natural, and 8-

	bit clean. Also, certain double-byte types such as Unicode, Big5, Shift JIS, and EHANGUL.
External numerics	Signed integers, reals and floating point values intermixed, IP_ADDRESS, whole numbers.
Internal numerics	C, COBOL, FORTRAN, signed/unsigned, characters, shorts, floats, doubles, COBOL packed, computationals, packed and zoned decimals, etc.
Timestamps	American, European, ISO, Japanese, and current time, date, and timestamp.
Alignment	Left/none/right adjustment option.
Case	Uppercase-lowercase equality option.
Duplicates	No duplicates option on keys.
Stability	First-In-First-Out (FIFO).
Program	Optional user-written key comparisons.

OUTPUTS

Terminal	Screen and/or stdout .
File	Multiple user-selected names or devices, including input file overwrite, and create or append logic, plus named and unnamed pipes.
Both	Standard output and file to observe and abort unintended executions.
Program	Optional user-written record processing
Table	ODBC-connected relational database targets.

JOB SPECIFICATION SETTINGS

Parameter Sources:

- Save option in **sorti** interactive mode.
- Text processor, converter, legacy sort parms, "**IRI Workbench**" GUI, **stdin**, or program.
- Procedure calls from calling programs.

Runtime Mode:

- Standalone interactive CLI or GUI execution.
- Standalone or embedded batch execution.
- Silent or verbose runtime progress monitoring.
- User program integration.

PROGRAMMING

`cosort_r()` and `sortcl_routine()`:

Calls	Subroutines for file operations; coroutines for user-written file and record-level procedures.
Languages	Library linkage to C, C++, COBOL, FORTRAN, Java, VB, et al.

SYSTEM CONTROLS

Tuning	Threads, RAM, blocksize, and disk usage.
Overflow	Directed to multiple file systems on different drives.
Checking	Error and warning messages with execution toggles.
Monitor	User-specified levels of on-screen event detail.
Logging	Runtime performance records directed to a file.

W The file **CoSort9.hlp** is provided in the directory `\install_dir\Docs`. This contains information on each main topic in **CoSort**. Topics are linked to from the main Welcome page. ♦

4.5 Compatible Platforms

Platforms compatible with **CoSort** include:

- AIX 5.1 and above on IBM Power architecture
- HP-UX 11.0 and above on the Intel Itanium architecture
- Linux kernel 2.6 and above on Intel x86 & x64 architecture
- Linux kernel 3.6 and above on ARMv6 and above architecture
- Solaris 5.8 and above on Sun Sparc architecture
- Solaris 10 on Intel x64 architecture
- Windows XP and above on Intel x86 & x64 architecture
- Windows Server 2003 and above on Intel x86 & x64 architecture

CoSort is 88open, MIPS ABI, Unix International, SPARCware certified, and ia64-compliant. **CoSort** has been affiliated with, and approved by, *all* commercial Unix OEM ISV programs since 1987. CoSort and IRI's metadata-compatible tools also run successfully in any virtual machine environment (e.g. Citrix VMWare) provided that the OS is otherwise supported (e.g. Linux or Windows).

sortcl PROGRAM

1 PURPOSE

This chapter describes the CoSort Sort Control Language (SORTCL) and the **sortcl** program. The SORTCL syntax uses familiar key words and logical expressions to define, transform, and map large volumes of disparate data. The **sortcl** program exploits **CoSort**'s coroutine sort engine and multiple cores to optimize throughput.

In addition to the **sortcl** program, `sortcl_routine()` is available as an API call for software developers interested in using **sortcl** functions in their applications (see `sortcl_routine()` on page 515).

A single pass through the **sortcl** program can perform and combine major functions of an Operational Data Store (ODS) and data warehouse Extraction, Transformation, and Loading (ETL) system, including:

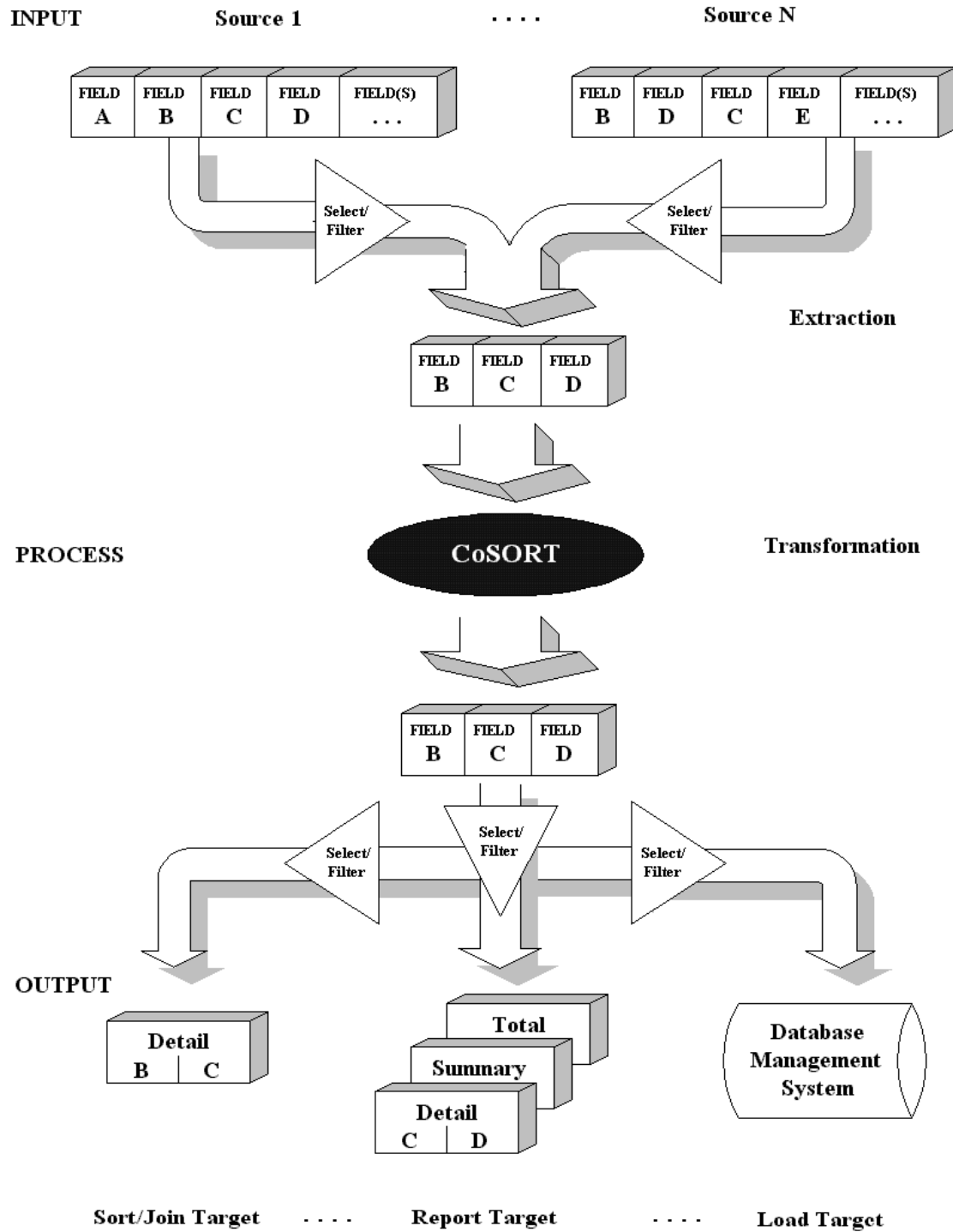
- select
- join
- sort
- merge
- aggregate
- cross-calculate
- type-convert
- report

One source for SORTCL is the VMS Sort Utility language. DEC suggested that IRI leverage this syntax, which IRI found to be familiar to mainframe sort users, more explicit than other legacy sort product syntax, and functionally extensible. For use in open systems, it was necessary to add structure and symbolic references (field names) to the record definitions. This, and various SQL constructs, allow users to reference and map data by field. Source data descriptions can be centrally maintained in data definition files, providing the basis for a shared data environment, and the creation of views and metadata repositories.

More specifically, **sortcl** provides the following functionality, in the same I/O pass:

- multiple input files, tables, and procedures defining data in different formats can be filtered, mapped, sorted, joined, and segmented into multiple targets and formats
- records and fields can be conditionally read and written
- summary results can be generated at multiple levels at the same time, and can include running or break-time maximum, minimums, totals, counts, and average values.

The **sortcl** data flow diagram on *page 39* illustrates the functionality of **sortcl** by following the flow of data from input to output. Inputs can be mapped, sorted, and remapped into multiple output targets. The ellipses (. . .) signify that any number of input and output layouts can be described and/or referenced.



2 **EXECUTION**

sortcl (**sortcl.exe** on Windows) is a standalone program that is executed from the command line, a batch script, the **IRI Workbench**, as a system call from a program (see COBOL TOOLS chapter *on page 453*, or as an API library call (see **sortcl_routine** *on page 541*).

To begin execution from the command line, enter the program name **sortcl** followed by either actual specifications, a job specification file, or a combination of both.

To display the version and other information, enter:

```
sortcl /version
```

This will display something like:

```
CoSort Sort Control Language (SortCL) Program Version 9.5.1  
R91090204-1205 32B Copyright 1978-2011 IRI, Inc. www.cosort.com  
Eastern Standard Time 09:51:43 Fri Feb 26 2011 #99999.9999 2 CPU
```

W The syntax required for Windows users referencing the path to a *script_file* depends on whether the drive letter is used. The following is an example of the syntax required in each case:

```
sortcl /SPECIFICATION=C:\home\test\job10.scl  
sortcl /SPECIFICATION=/home/test/job10.scl
```

Note that in earlier versions of **sortcl**, double backslashes were required when using a drive letter with the path in Windows. The double backslashes will still work, but are not required. If the drive letter is not used, use either a single forward slash (for consistency with Unix) or a single backslash (the standard Windows convention). See Spaces within File Names/Paths -- Windows Users Only *on page 50* if the name of the job script, or its path, contain a space. ◆

During **sortcl** execution, any syntax errors are reported by the line number in your script. The script also provides a way to re-execute, share, and modify your data definitions and/or job specifications.



Adjust the **MONITOR_LEVEL** resource control setting that determines the degree of verbosity for reporting the progress of a job (see Resource Control Settings *on page 647*). For details on all on-screen **CoSort** messages, see **ERROR** and **RUNTIME MESSAGES** *on page 660*.

3 CONVENTIONS

This section describes notation that is used in documenting the **sortcl** program. For documentation conventions that apply to the entire **CoSort** manual, and not just to **sortcl** specifically, see NOTATION *on page 24*.

3.1 Naming Conventions

The following rules apply both to names and statements recognized by **sortcl**, and to file names and field names that you define:

- must start with an alphabetic character
- can be any length
- can contain any combination of the following characters:
 - alphabetic
 - numeric
 - embedded underscore (`_`)

Statements and field names are not case-sensitive, that is, upper- and lower-case letters are interchangeable, for example:

- `POSITION` is the same as `position`
- `/FIELD=(PARTY)` is the same as `/field=(Party)`

U Unix paths and file names, however, are case-sensitive, so the file **chicago** is *not* the same as the file **CHICAGO**, **Chicago**, etc. ◆

W For Windows users with Fast Access Table (FAT) and NT File Systems (NTFS), file names are not case-sensitive, so the file **chicago** *is* the same as the file **CHICAGO**, **Chicago**, etc.

Refer to your operating system manual for the acceptable maximum length and naming format of a file name. ◆

The file names **stdin** and **stdout** are used for *standard input* and *standard output* (pipes), respectively. When input and output files are not specified, the defaults are **stdin** and **stdout**.

3.2 Abbreviations

Generally, **sortcl** words in statements can be truncated by deleting trailing letters. The control word must only be long enough to distinguish it from any other control word, for example:

```
/FIELD= (Goods , POSITION=10)  is the same as  
/FIELD= (Goods , POS=10)
```

and:

```
/SPECIFICATIONS  is the same as  
/SPEC
```

3.3 Optional Statement Parameters

Square brackets [] are used to describe optional parameters or values.

3.4 Environment Variables

The character \$ preceding a variable name directs **sortcl** to replace the environment variable with its current value. This syntax is used for both Windows and Unix. Use any of the following conventions:

- `$variable`
- `${variable}`
- `[$variable]`

U Unix users defining environment variables in a Bourne or Korn shell must export the variables for **sortcl** to recognize them. See Example: *on page 316* for an example. ◆

3.5 Line Continuation

The backslash character \ is the continuation character for **sortcl** specification statements longer than one line. In a **sortcl** script, include blanks, comments (starting with #), or tabs on the same line following the \ character. However, the first character on the next line must begin with the continuation of the syntax that was interrupted by the \. On the command line, Unix users can use a backslash as a line continuation character, but spaces, tabs and comments are not supported beyond the \.

W Windows users cannot use \ on the command line. Long lines should wrap around to the next line. ◆



You cannot use a \ line continuation character to separate a *[path] filename* reference, even if it contains spaces (see Spaces within File Names/Paths -- Windows Users Only *on page 50*) Also, you cannot use the line continuation character to break up any word. You must place the \ before or after the complete word, and complete the statement on the next line.

3.6 Comments

The character # marks the beginning of comments on a line in a **sortcl** script. Comments can begin after a statement on the same line, or can be on a line by themselves. The comment continues until the end of the line, and is ignored during processing.

3.7 Italics

A word that is italicized represents a variable that is to be replaced by a literal string, for example:

```
/INFILE=filename
```

indicates that the user might have, for example:

```
/INFILE=Inventory.dat
```

4 FILES

This section describes the types of files that can be referenced by **sortcl**:

- *Resource Control File* below
- *Specification Files on page 45*
- *Data Definition Files on page 46*
- *Data Files on page 48*

4.1 Resource Control File

This file can be used to set local or global system resources that **CoSort** will use at runtime. Adjust and control several settings to improve efficiency, such as the:

- number of threads
- amount of memory
- overflow areas
- verbosity of the monitor

U **CoSort** searches for a **.cosortrc** or **cosortrc** file within specific directories. Normally, a default **cosortrc** is built in the **\$COSORT_HOME/etc** directory by your system administrator at installation time. ♦

W On Windows systems, the resource control file is called **cosort.rc**, and its values take precedence over default registry settings only when it is in the same directory as the executable, such as **sortcl** or **sorti**. ♦

Set the environment variable **COSORT_TUNER** to the path (required) and name for a *resource control* file. And, create multiple *resource control* files for specific users and/or jobs. To ensure that a particular file is used with a specific job, use the following statement within a job script:

```
/MEMORY-WORK=" [path] filename "
```

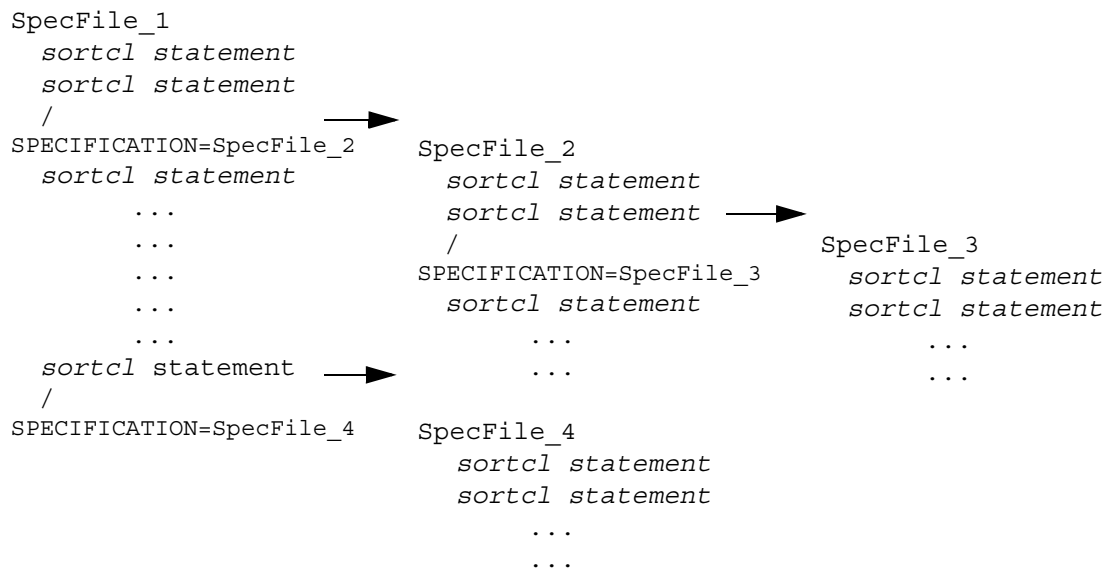
The values in the tuning file named in the **/MEMORY-WORK** statement have a higher priority than those in **COSORT_TUNER**. After values in **COSORT_TUNER** are checked, any values in **cosortrc** or **.cosortrc** in the **CoSort** search path will be read. If there are still values that have not been set, *factory default* values will be used (see *Search Order for Resource Controls on page 645*).

4.2 Specification Files

These files are used to organize **sortcl** statements. These named files are referenced by the `/SPECIFICATION` statement as shown below. One or more specification files can be used in the execution of the **sortcl** program, for example:

```
sortcl /SPECIFICATION=[path]file_1 /SPECIFICATION=[path]file_2 ...
```

A specification file, or *script*, can have other specification files referenced within it, and these can be nested as deeply as necessary. The following diagram demonstrates three levels of nesting.



For discussion purposes, we will refer to the *job specification file* as the principal group (or first level) of statements.

When a `/SPECIFICATION=filename` occurs within a specification file, the contents of the referenced file are read into the job at that point. For example, a job specification file might contain:

```

/INFILE=chiefs
/SPECIFICATION=key1.spec
/OUTFILE=parties

```

The file **key1.spec** could be defined as follows:

```

/KEY= (POSITION=40,SIZE=3)
/KEY= (POSITION=1,SIZE=22)

```

This is the equivalent of writing the following job specification:

```
/INFILE=chiefs
  /SORT
    /KEY= (POSITION=40,SIZE=3)
    /KEY= (POSITION=1,SIZE=22)
  /OUTFILE=parties
```

Note that instead of using the name of a specification file, you can use an environment variable that references the file. In that case, the job specification file might be as follows:

```
/INFILE=chiefs
/SPECIFICATION=$KEY1
/OUTFILE=parties
```



Windows users should ensure that a carriage return/linefeed (CRLF) exists at the end of the last entry in a **sortcl** specification file.

4.3 Data Definition Files

For application independence, it is recommended that you develop and use a specification file that contains data definitions only - *Data Definition File* (DDF). They are an ideal way to create user views and share common data.



Several **sortcl** tools convert third-party metadata into **sortcl** data definition files. See **sortcl** TOOLS chapter *on page 37* for conversion from Micro Focus COBOL data dictionaries, Microsoft Comma-Separated Values (CSV) header records, Oracle SQL*Loader control files (CTL), W3C Extended Log (web) Format (ELF), LDAP Interchange Format (LDIF), and flat eXtensible Markup Language (XML) file layouts.

In addition, **CoSort**'s FAst extraCT (**FACT**) unload tool for Oracle creates **sortcl** data definitions from table extracts for integrating **sortcl** into database reorg and data warehouse Extract-Transform-Load (ETL) operations. For more details on **FACT**, see <http://www.iri.com/fact>.

IRI's **FieldShield**, **NextForm**, and **RowGen** products also use the same data layouts (and job specifications) to define the layout of synthetic files. For more details, see http://www.iri.com/products/CoSort/SortCL_Metadata.

Meta Integration Technology, Inc., (MITI) has developed and released a Meta Integration Model Bridge (MIMB) for **sortcl** data definition files (DDF). The MIMB bridge to **.ddf** allows users of many third-party applications, where flat file definitions are stored in proprietary formats (such as DataStage .dsx), to automatically convert their metadata for use in **sortcl** and its spinoffs.

For more details on this layout migration option, see:

<http://www.metaintegration.net/Partners/IRI.html>

and <http://www.metaintegration.net/Products/MIMB>.

Place the DDF under the /INFILE or /OUTFILE statement for which the definitions apply.

For example, the data definition files **chiefs.ddf** and **parties.ddf** could contain:

```
# chiefs.ddf
  /FIELD= (president, POSITION=1, SIZE=22)
  /FIELD= (votes, POSITION=24, SIZE=3)
  /FIELD= (service, POSITION=28, SIZE=9)
  /FIELD= (party, POSITION=40, SIZE=3)
  /FIELD= (state, POSITION=45, SIZE=2)
```

```
# parties.ddf
  /FIELD= (party, POSITION=5, SIZE=3)
  /FIELD= (president, POSITION=10, SIZE=22)
```

and the job specification file could be:

```
/INFILE=chiefs
  /SPECIFICATION=chiefs.ddf
/SORT
  /KEY=party
  /KEY=president
/OUTFILE=parties
  /SPECIFICATION=parties.ddf
```

4.4 Data Files

This section describes flat file **sortcl** data sources and targets:

- Input Files *on page 48*
- Output Files *on page 51*

An input or output file description consists of a file name and set of attributes such as process type, fields, and filters. Relational database management systems (RDBMS) table sources and targets are declared as input and output files in **sortcl** through the `/PROCESS=ODBC` specification (see *ODBC on page 62*).

4.4.1 Input Files

Input files are named with one or more `/INFILE` statements:

```
/INFILE=[path] filename
```

Statements that describe the file are then placed under the `/INFILE` statement. A valid input file name is **stdin** (standard input) when records will be piped into **sortcl**.

If there is more than one input file and the attributes under each file name are the same, use the following syntax to define the files:

```
/INFILES=( [path] filename1, [path] filename2, ... )
```



You cannot insert a space directly before a comma in `/INFILES` declarations. See *Spaces within File Names/Paths -- Windows Users Only on page 50* for details on when spaces are accepted within file names and paths to file names.

Parentheses are not required around filenames in an `/INFILES` statement, but their usage is recommended to better differentiate from an `/INFILE` statement.

For an example of the `/INFILE` statement, consider records from the input file **miami**:

Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95

which can be referenced within your script as follows:

```
/INFILE=miami                # names the input file
  /FIELD=(title,POSITION=1,SIZE=15)    # book title
  /FIELD=(publisher,POSITION=16,SIZE=13)# book publisher
  /FIELD=(quantity,POSITION=30,SIZE=3)  # number of books sold
  /FIELD=(price,POSITION=38,SIZE=5,NUMERIC) # book price
```



If your input file is empty, use a resource control setting to determine the disposition of the output file (see `ON_EMPTY_INPUT` option *on page 626*). By default, if an input file is empty, an output file is created assuming zero-value input data. The output file can contain zero values for summary columns and other requested formatting data.

Using gzip Format as an INFILE source

sortcl supports input files in compressed gzip format. Deflated files are automatically recognized by the unique gzip header information at the beginning of the input file.

To process an input file in gzip format, name the gzip (deflated) file in the `/INFILE` line. You can mix regular and deflated files in the same `/INFILES` section because each file is scanned separately (see Input Files *on page 48* for details on `/INFILES`).



`HEADSKIP` and `/TAILSKIP` are not supported when using gzip input sources. Results will be unpredictable with either of these statements.

Multiple Formats

If you are sorting, merging, checking, or copying (/REPORT), and there are multiple input files where the declared attributes are not identical across all files, you must have an /INFILE declaration (with the specific attributes) for each file, for example.

```
/INFILE= [path] filename_1
          attributes_1
/INFILE= [path] filename_2
          attributes_2
/INFILE= [PATH] filename_3
          attributes_3
```



To provide a uniform format for processing disparate input record formats, as shown above, you must provide an /INREC section (see /INREC on page 150).

Spaces within File Names/Paths -- Windows Users Only

W In some cases, you might want **sortcl** to reference a path name or file name that contains a space. **sortcl** supports the use of a space (but not two or more in a row) within a path component and/or file name reference. This feature applies to the statements /INFILE (see Input Files on page 48), /OUTFILE (see Output Files on page 51), /SPEC (see Specification Files on page 45), and /FILE (see Data Definition Files on page 46).

If the file name or path containing a space is to be referenced again in your script (particularly in a join script), it is recommended that you define an alias for it beneath its first usage and use the alias when referenced later in the script (see /ALIAS on page 89). For join scripts that have path/file names with a space, it is required that you use ALIAS.

The use of a space is also supported within **sortcl** command-line references, for example:

```
sortcl /infile=C:\iri\cosort95\test file1
```



sortcl continues to support the pre-version 9.5 convention of using a tilde (~) followed by a 1 to substitute for the space. For example, within a **sortcl** script, use:

```
/INFILE=C:\iri\cosort95\chiefs ◆
```

Wildcards used with /INFILES

Specify wildcard characters such as `*` and `?` for your `/INFILES` sources. In this case, **sortcl** expands the file name and uses the expanded name during transformation. When defining `/INFILES` attributes, you must ensure that they pertain to all files referenced by the wildcard expansion (see *Input Files on page 48* for `/INFILES` syntax). If there is no matching file name available, **sortcl** terminates with the error `No such file`. For example, if you have a directory containing the files **chiefs**, **thiefs**, **chiefs_sep**, and you want all of these to be referenced by your `/INFILES` statement, use:

```
/INFILES=?hiefs*
```

U Unix users can specify a pattern bracket expression as part of `/INFILES`. For example, use `/INFILES=chiefs.[ab]` to reference **chiefs.a** and **chiefs.b**. ♦

/INPROCEDURE

Invoke your own custom input procedures and link them to **sortcl** to accommodate special job criteria (see *Custom Input on page 528*). All custom input routines are invoked using `/INPROCEDURE`, and attributes are provided directly beneath this statement.

4.4.2 Output Files

As input records are processed through **sortcl**, they are mapped to one or more output files. A possible output file is **stdout** (standard out), which, by default, sends results to the console.

Output files are declared with one or more `/OUTFILE` statements:

```
/OUTFILE=[path] filename
```

Further statements for describing the file are placed beneath this statement. Define multiple output files with a new `/OUTFILE` statement for each, for example:

```
/OUTFILE=apples
  attributes
/OUTFILE=peaches
  attributes
/OUTFILE=pears
  attributes
```

If a single output file such as **pears** requires multiple record formats, use an `/OUTFILE` statement to describe the attributes of each format, using the same file name:

```
/OUTFILE=pears
attributes_a
/OUTFILE=pears
attributes_b
...
```

This is typical of multi-level output reports containing both summary and detail records (see Example: *on page 300*).



If the input file is empty, use a resource control setting to determine the disposition of the output file (see `ON_EMPTY_INPUT` option *on page 626*).

By default, when an input file is empty, an output file is created assuming zero-value input data. The output file can contain zero values for summary columns and other requested formatting data.

Using gzip Format as an OUTFILE target

sortcl supports the writing of output files in compressed gzip format. To produce an output file in gzip format, give the filename a **.gz** or **.gzip** extension in the `/OUTFILE` entry, and the output file will be deflated and stored in gzip format.

/OUTPROCEDURE

Invoke your custom output procedures and link them to **sortcl** to accommodate special job criteria (see Custom Output *on page 531*). All custom output routines are invoked using `/OUTPROCEDURE`, and attributes are provided directly beneath this statement.

5 DATA SOURCE AND TARGET FORMATS (/PROCESS)

The structure of an input or output file is described using the /PROCESS statement. This statement is used to identify those special file structures which **sortcl** can *process* natively. You can also convert from one file structure to another; specify one process type on input, and a different process type on output.

If no process is specified for either the input or output, the default file structure is RECORD.

The syntax for a /PROCESS declaration is:

```
/PROCESS=process_type
```

sortcl recognizes the following process types:

- RECORD or RECORD_SEQUENTIAL *on page 54*
- MFVL_SMALL *on page 56*
- MFVL_LARGE *on page 56*
- VARIABLE_SEQUENTIAL (or VS) *on page 57*
- LINE_SEQUENTIAL (or LS) *on page 57*
- V (variable) *on page 58*
- VISION *on page 58*
- VSAM *on page 58*
- MFISAM *on page 59*
- UNIVBF *on page 60*
- CSV *on page 60*
- LDIF *on page 61*
- ODBC *on page 62*
- BLOCKED *on page 68*
- XML *on page 68*
- RANDOM (Generating Test Data) *on page 73*
- ELF (W3C Extended Log Format) *on page 77*

5.1 RECORD or RECORD_SEQUENTIAL

If no process is given, or the process type is explicitly RECORD, every record is either of fixed or variable length:

fixed A length statement can be given when all records are of the same length. This is given in the following form:

/LENGTH=*n*

where every record is length *n*, which can be a maximum of 65,535 bytes. In COBOL, this is equivalent to RECORD_SEQUENTIAL.

If your input is linefeed-terminated, and all input records are of equal length, then in some cases you can improve job efficiency by specifying a fixed length for the input file (or as part of /INREC). Be sure to add one or two bytes to the record length to accommodate each record's LF or CRLF, respectively. If the entire record is to be output, then specify the same length on output as on input to avoid having double linefeeds after each output record.

If you declare a fixed length as an attribute for an input file, and you do not specify a field layout for its respective output file, you must include the same LENGTH=*n* statement from the input file as an attribute for the output file.



If you have upgraded to **CoSort** version 9.5 from an older version, see *Optional Settings on page 628*.

FIXED, DELIMITED, and TEXT are aliases of the process type RECORD.

The process type RECORD_SEQUENTIAL or RS is an alias of the process type RECORD.



If one or more records in the input data is not the record length you have declared, your output results might be offset.

If the last record in the input data is shorter than the record length you have declared, this last record is discarded. This can happen if your last record is either a tail record (see Input Options *on page 260*), or contains only the excess characters of a previous record (in the event of offset data).

It is therefore suggested that you declare records as fixed-length only when you are certain that every record in your input data is of equal length. To determine this, divide the file size by the record length. If the result is not a whole number, then either the record length declaration is incorrect, or there is a wrong-sized record within your input data.

variable If no `/LENGTH` is specified, or you specify `/LENGTH=0`, then records can vary from 0 to 65,535 bytes in length and terminate with a line-feed character. This is equivalent to line sequential where there are no data byte values between `x00` and `x1F`.

Special Considerations

When binary numeric data type columns, such as `INTEGER (INT)` or `SHORT`, are used from a database in delimited fields, those columns must be converted to the `NUMERIC` data type. Many of the binary data types, such as the `SHORT` and `INT`, do not carry a decimal. Therefore, you must set `PRECISION=0` when you specify `NUMERIC` in `sortcl` (See `PRECISION ON PAGE 101`). Binary floating point types, such as `DOUBLE`, should also be converted to `NUMERIC`. If you do not set the `PRECISION` then a `PRECISION` of 2 is used. If you specify any of these fields as `ASCII`, `sortcl` cannot perform mathematical operations on the fields; however, you can extract and load such data in the database with the data type set to `ASCII`.



Depending on the operating system, the length of a variable-length record is the position number of the last character of the last field on the line, plus one for the line-feed character, or plus two bytes for the carriage-return and the line-feed.

When processing variable-length records with binary values, data must not have a binary 10 (hex 0a), the line-feed character, before the end of the record. The value would be treated as the record terminator.

5.2 MFVL_SMALL

This describes short Micro Focus variable-length records. The file has a 128-byte header record, and each record is prepended with a binary short integer of two bytes which holds the size of the record. The maximum record length is 4,096 bytes. Each record begins at an offset address which is evenly divisible by four.

When using this process type on output, you can set your own minimum and maximum record lengths, the values of which will be written to the header record to support inter-process conversions.

The syntax for using this process on output therefore supports additional options:

```
/PROCESS=MFVL_SMALL[,min_length][,max_length]
```

where *min_length* and *max_length* are the minimum and maximum record lengths contained in the output file. If /PROCESS=MFVL_SMALL on input, this information is taken from the input file if you do not specify these attributes.



For backward-compatibility with **CoSort** versions earlier than 8.2.2, /PROCESS=MFVL is accepted and is equivalent to /PROCESS=MFVL_SMALL.

5.3 MFVL_LARGE

This describes long Micro Focus variable-length records. In most cases, the file has a 128-byte header record, and each record is prepended with a binary int of four bytes which holds the size of the record. The maximum record length is 268,435,455 bytes. Each record begins at an offset address which is evenly divisible by four.

When using this process type on output, you can set your own minimum and maximum record lengths, the values of which will be written to the header record to support inter-process conversions.

The syntax for using this process on output therefore supports additional options:

```
/PROCESS=MFVL_LARGE[,min_length][,max_length]
```

where *min_length* and *max_length* are the minimum and maximum record lengths contained in the output file. If /PROCESS=MFVL_LARGE on input, this information is taken from the input file if you do not specify these attributes.

5.4 VARIABLE_SEQUENTIAL (or VS)

This describes variable-length records that are a string of characters prepended by a short integer. The value of the short is the length of the record, e.g., ABC is (x03 x00 x41 x42 x43) on a computer that is little-endian.

The format of the short integer is machine-dependent. Therefore, VS data between dissimilar computers (for example, between RISC and CISC) might be incompatible due to the difference in endianness. Therefore, when using this process type, you can indicate the endianness of the source and target file. If you prefer to set these options on a global level, and not for each job, see the ENDIANNESS option *on page 631*.

The syntax is:

```
/PROCESS=VS [,ENDIAN=option]
```

where option can be LITTLE or BIG. For example, the following script converts the contents of a VS file from little-endian to big-endian:

```
/INFILE=data.in
/PROCESS=VS,ENDIAN=LITTLE
/FIELD=(all,POSITION=1,SIZE=14)
/REPORT
/OUTFILE=data.out
/PROCESS=VS,ENDIAN=BIG
/FIELD=(all,POSITION=1,SIZE=14)
```



To perform an endian-specific transformation such as a sort:

- 1) Specify the actual source attributes in the input section.
- 2) Convert to the machine's default endianness in /INREC (see /INREC *on page 150*).
- 3) Specify the desired processing in the action phase.
- 4) Declare your output fields in the required endianness for display purposes.

5.5 LINE_SEQUENTIAL (or LS)

This describes variable-length records that are strings of characters where each low value data byte (x00 through x1f) is *protected* (or prepended) by a null byte (x00). The terminating linefeed character is not protected.

5.6 V (variable)

This is used for the IBM *unblocked* variable record format containing a 4-byte Record Descriptor Word (RDW) preceding each record. Blocked (or VB) data of this format are typically unblocked as a result of extraction from the mainframe.

For example, the following script converts an IBM variable record format file (the entire record, as specified with POSITION=1 and no SIZE attribute) to record sequential, and from the EBCDIC data type to ASCII.

```
/INFILE=v.dat
/PROCESS=V
/FIELD=(a, POSITION=1, EBCDIC)
/REPORT
/OUTFILE=v.out
/PROCESS=RECORD
/FIELD=(a, POSITION=1, ASCII)
```

This format also handles spanned records. Spanned records have a bit set in the RDW indicating that the next record is a continuation of the current record.

5.7 VISION

This describes ACUCOBOL-GT Vision files, which is a proprietary index file format originally from Acucorp, Inc. For complete details on using /PROCESS=VISION (including examples), see ACUCOBOL-GT VISION RECORDS *on page 446*.

When using /PROCESS=VISION on output, you must specify the index /KEY in the **sortcl** job script in order to create an indexed file.



To obtain the requisite Vision-enabled **CoSort** libraries for linking to Vision libraries, contact IRI or your IRI agent.

5.8 VSAM

This describes records in Clerity's proprietary VSAM format. When available with Clerity's Mainframe Batch Manager (MBM) and Mainframe Transaction Processing (MTP) software, **sortcl** is linked to Clerity's C-ISAM library. VSAM records can be moved from, and to, **sortcl** for processing.



To obtain the requisite VSAM-enabled **CoSort** library for linking to your VSAM libraries, contact IRI or your IRI agent.

When you obtain the library file **libsortcl_vsam.a**, save it to **\$COSORT_HOME/lib**, and execute the appropriate command to link all the libraries and create an updated **sortcl** executable. The compiler flags vary according to the operating system. The following are valid commands for some systems:

Solaris 10 `cc -xtarget=ultra -xarch=v9 -xcode=pic13 -Bdynamic
-Kpic -Xt -o sortclvs ./libsortcl_vsam.a
./libcosort.a $UNIKIX/lib/libbcisam.a -L$UNIKIX/
lib -lsocket -lnsl -lpthread -lrt -lm -ldl -llang_c`

HP-UX11 ia64 `cc +DD64 -Ae +z -o sortclvs ./libsortcl_vsam.a
./libcosort.a $UNIKIX/lib/libbcisam.a -L$UNIKIX/
lib -lkxtables -lpthread -lrt -llang_c -lm -ldl`

AIX 5.3 `cc -q64 -o sortclvs ./libsortcl_vsam.a
./libcosort.a $UNIKIX/lib/libbcisam.a $UNIKIX/lib/
libkxtables.so $UNIKIX/lib/liblang_c.so -l curses
-lnsl -lpthread -lrt -lm -ldl`

Linux x86 `cc -m64 -fPIC -o sortclvs ./libsortcl_vsam.a
./libcosort.a $UNIKIX/lib/libbcisam.a -L$UNIKIX/
lib -lnsl -lpthread -lrt -lm -ldl -llang_c`

This will create a new executable, **sortclvs**, which allows you to use **/PROCESS=VSAM** for both **/INFILE** and **/OUTFILE** purposes in a Clarity MBM application environment.

5.9 MFISAM

This describes records in Micro Focus indexed file format (MF ISAM). For details, see **MICRO FOCUS COBOL ISAM RECORDS** on page 478 .

When using **/PROCESS=MFISAM** on output, specify the index **/KEY** in the **sortcl** job script in order to create an indexed file.



Process MFISAM uses third party libraries requiring licensing that is provided by IRI when the COBOL MFISAM and VISION file support option is purchased. The Micro Focus License Manager is installed with NextForm and must be running to validate the use of the third party libraries.

5.10 UNIVBF

This describes variable-length records that are prepended by a 4-byte ASCII field giving the length of the record, including the four bytes. This format can be generated by Unisys mainframes writing to tapes.

5.11 CSV

Specifying `/PROCESS=CSV` instructs **sortcl** to recognize the data file as a Microsoft **.csv** (comma-separated values) file. CSV files always contain ASCII fields separated by commas, and a header that names the file's fields in order, from left to right. CSV data can contain commas within individual fields if such fields are enclosed in double quotes. **sortcl** will skip a file's header when `/PROCESS=CSV` is specified. To force **sortcl** to generate a header based on the file's field names and positions, specify `/PROCESS=CSV` in the output file. See *EXAMPLE on page 377* for details on using `/PROCESS=CSV`.



The maximum header length supported by the use of `PROCESS=CSV` is 4,096. If the header length exceeds 4,096 bytes, use the `/HEADSKIP` command (see */HEADSKIP on page 260*).

For the purposes of **sortcl**, it is expected that the first record of a Microsoft **.csv** file data is a header record. Therefore, if the data file does not have a header, you cannot use the `/PROCESS=CSV` statement. See *FRAME on page 101* for a useful way to handle CSV fields enclosed in quotes.

To write specifications for sorting CSV data files, use the utility **csv2ddf** that is provided with the **CoSort** package in the `$COSORT_HOME/bin` directory. In Windows, **csv2ddf** is located in `\install_dir\bin` (see the **csv2ddf** sub-chapter *on page 375*). The utility examines file headers and generates a **sortcl** data definition file based on the input field names found in the file header.

Its syntax is:

```
csv2ddf datafile [data definition filename]
```

5.11.1 Special Considerations

When binary numeric data type columns, such as INTEGER (INT) or SHORT, are used from a database in delimited fields, those columns must be converted to the NUMERIC data type. Many of the binary data types, such as the SHORT and INT, do not carry a decimal. Therefore, you must set PRECISION=0 when you specify NUMERIC in *sortcl* (See PRECISION ON PAGE 101). Binary floating point types, such as DOUBLE, should also be converted to NUMERIC. If you do not set the PRECISION then a PRECISION of 2 is used. If you specify any of these fields as ASCII, *sortcl* cannot perform mathematical operations on the fields; however, you can extract and load such data in the database with the data type set to ASCII.

5.12 LDIF

Use /PROCESS=LDIF for data in LDIF (the format of data exported from an LDAP database). In the input section of the *sortcl* script, you must define each desired LDIF attribute as a /FIELD statement, and if a value does not exist for a specific attribute, the value will be null on output. When using /PROCESS=LDIF, data columns must be defined as delimited fields, for example:

```
/INFILE=test1.ldif
/PROCESS=LDIF
  /FIELD=(cn, POSITION=1, SEPARATOR='|')
  /FIELD=(address1, POSITION=2, SEPARATOR='|')
  /FIELD=(address2, POSITION=3, SEPARATOR='|')
  /FIELD=(zipcode, POSITION=4, SEPARATOR='|')
  /FIELD=(phonenumber, POSITION=5, SEPARATOR='|')
  ...
```

On output, when using /PROCESS=LDIF, the attribute name will be the same as the field name. If the field value is null then the attribute will not appear in the output.

To write specifications for sorting LDIF data files, use the utility **ldif2ddf** that is provided with the **CoSort** package in the **\$COSORT_HOME/bin** directory. In Windows, **ldif2ddf** is located in **\install_dir\bin** (see the *ldif2ddf* sub-chapter on page 383). The utility examines LDIF files and generates a *sortcl* data definition file based on the detected layout.

Its syntax is:

```
ldif2ddf [options] source_filename [target_filename]
```

5.12.1 Special Considerations

When binary numeric data type columns, such as INTEGER (INT) or SHORT, are used from a database in delimited fields, those columns must be converted to the NUMERIC data type. Many of the binary data types, such as the SHORT and INT, do not carry a decimal. Therefore, you must set PRECISION=0 when you specify NUMERIC in **sortcl** (See PRECISION ON PAGE 101). Binary floating point types, such as DOUBLE, should also be converted to NUMERIC. If you do not set the PRECISION then a PRECISION of 2 is used. If you specify any of these fields as ASCII, **sortcl** cannot perform mathematical operations on the fields; however, you can extract and load such data in the database with the data type set to ASCII.

5.13 ODBC

Use /PROCESS=ODBC in a **sortcl** job script to process (on input) and populate (on output) table columns in databases supported by ODBC (Open Database Connectivity). For **CoSort** version 9.5, the following database types have been tested for compliance with /PROCESS=ODBC:

- Oracle
- SQL Server
- DB2
- MySQL



On some systems, the ODBC specification has been updated such that a certain value (SQLLEN) has changed from a 32-bit integer to a 64-bit integer when the software is compiled in 64-bit mode. Some older drivers will expect this value to be 32-bit, and newer drivers will expect it to be 64-bit. To address this, a separate file format must be used, /PROCESS=ODBC_LEGACY, which supports the older standard. Current PostgreSQL and MySQL ODBC drivers use the 64-bit value (/PROCESS=ODBC). Oracle 11i ODBC drivers use the 32-bit value (/PROCESS=ODBC_LEGACY). This is not an issue on Windows or MacOSX (iODBC always defines it as 64-bit).

W Requirements for Windows Users Using /PROCESS=ODBC

In order to use /PROCESS=ODBC or **odbc2ddf** (see the **odbc2ddf** sub-chapter on page 385), you must first create a Database Source Name (DSN) for each source or target you intend to connect to. Connection requires the presence of an ODBC driver. The ODBC driver must be installed on the specific machine or server on which **CoSort** resides. Windows machines typically contain an ODBC manager called the ODBC DATA Source Administrator, which manages database drivers and data sources, and maintains a list of DSNs. On 64-bit Windows systems, there is a native 64-bit ODBC manager which also offers a 32-bit ODBC manager. If you have a 32-bit **CoSort**, you must create DSNs using a 32-bit ODBC manager. If you have a 64-bit operating system while running a 32-bit **CoSort**, you must create DSNs using a 32-bit ODBC Data Source Administrator. ♦

U Requirements for UNIX/Linux Users Using /PROCESS=ODBC

In order to use /PROCESS=ODBC or **odbc2ddf** (see the **odbc2ddf** sub-chapter on page 385), you must manually set up the Unix ODBC driver manager on your system. The version of **odbcunix** must be 2.2.14 or higher, and it must match the bit architecture of your **CoSort** version (32- or 64-bit). You must also have an ODBC data source driver (for each database you are connecting with) that matches the bit architecture of **CoSort**. Using the ODBC driver manager, you must create a DSN for each source or target you intend to connect to. ♦

5.13.1 Syntax: Using /PROCESS=ODBC

For all **sortel** job scripts that contain /PROCESS=ODBC, you must move or copy the library file **libcsodbc.so** (UNIX/Linux) or **libcsodbc.dll** (Windows) from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

When using /PROCESS=ODBC to extract and process a database table, or to specify a target table to be populated, you must indicate the table name and the DSN in the /INFILE or /OUTFILE statement, respectively, and enclose these details in quotes for example:

```
/INFILE="tablea;DSN=oracledb[;UID=username;PWD=password;]"
    /PROCESS=ODBC
```

or

```
/OUTFILE="tableb;DSN=oracledb[;UID=username;PWD=password;]"
    /PROCESS=ODBC
```

where `tablea` is the source target to be extracted, and `tableb` is the target table to be populated. `oracledb` is the DSN. The `UID` and `PWD` are dependent on the system `odbc` configuration. If they are specified in the system ODBC configuration, they are not required in the `/INFILE` or `/OUTFILE` statement.

All data columns must be defined as delimited fields, as in the following example where `/PROCESS=ODBC` is used to identify both a database source and a database target:

```
/INFILE="tablea;DSN=oracle_tester;UID=scott;PWD=tiger;"
/PROCESS=ODBC
/FIELD=(field1,POSITION=1,SEPARATOR=',')
/FIELD=(field2,POSITION=2,SEPARATOR=',')
/FIELD=(field3,POSITION=3,SEPARATOR=',')
/OUTFILE="tableb;DSN=oracle_tester;UID=scott;PWD=tiger;"
/PROCESS=ODBC
/FIELD=(field1,POSITION=1,SEPARATOR=',')
/FIELD=(field2,POSITION=2,SEPARATOR=',')
/FIELD=(field3,POSITION=3,SEPARATOR=',')
```

Note that you should choose a separator character that will not appear in any of the data values. Common choices are comma (,), tab (t) , and vertical pipe(|).

5.13.2 EXT_FIELD

In some cases, you might want to specify a **sortcl** `/FIELD` name that is specifically meaningful to **sortcl** in the context of that job script. The desired **sortcl** field name will differ from its corresponding column name in the database. In this case, use the field attribute `EXT_FIELD` to assign the database column name to the field (to ensure that the field data will be extracted and loaded properly when using `/PROCESS=ODBC` on input and output, respectively), while using the desired **sortcl** field name that you choose.

For example, if you are extracting data from a table with the column name `SS_NO`, and prefer to refer to this column as `ssn` in **sortcl**, use the following for example:

```
/FIELD=(ssn,EXT_FIELD="SS_NO",POSITION=1,SEPARATOR=',')
```



If the **sortcl** `/FIELD` name differs from its corresponding name in the database, the field data will not be extracted or loaded and an error will occur. You must use `EXT_FIELD` to prevent this.

To write specifications for sorting ODBC data files, use the utility **odbc2ddf** that is provided with the **CoSort** package in the `$COSORT_HOME/bin` directory. In Windows, **odbc2ddf** is located in `\install_dir\bin` (see the `odbc2ddf` sub-chapter

on page 385). The utility examines ODBC-supported database tables and generates a **sortcl** data definition file based on the detected layout.

5.13.3 Extraction

In the input section of a **sortcl** script, you must define each table column you want to extract using corresponding `/FIELD` statements for those columns. See the `odbc2ddf` sub-chapter on page 385 for details on using **odbc2ddf**, which will automatically provide a complete list of `/FIELD` statements based on the column descriptions from the database. Several ODBC data types have **sortcl** data type equivalents, as shown in Table 10 on page 389, so you must specify `/FIELD` statements for only those database columns whose data types have **sortcl** equivalents.

QUERY

The `/QUERY=` statement is a string that contains a valid SQL SELECT statement. All of the columns that are returned from the query are mapped to the declared fields in the section by position, not by field name. The field or column names are ignored.

When using `PROCESS=ODBC`, fieldshield creates a query statement to process information in the script. You can create a customized query statement that overrides this internal query statement. The following is an example of a valid query:

```
/INFILE="SCOTT.CHIEFS;DSN=oratester"
/PROCESS=ODBC
/ALIAS=SCOTT_CHIEFS
/QUERY="SELECT LNAME,FNAME,TERM,PARTY FROM SCOTT.CHIEFS WHERE PARTY='DEM'"
/FIELD=(LNAME, TYPE=ASCII, POSITION=1, EXT_FIELD="LNAME", SEPARATOR="|")
/FIELD=(FNAME, TYPE=ASCII, POSITION=2, EXT_FIELD="FNAME", SEPARATOR="|")
/FIELD=(TERM, TYPE=ASCII, POSITION=3, EXT_FIELD="TERM", SEPARATOR="|")
/FIELD=(PARTY, TYPE=ASCII, POSITION=4, EXT_FIELD="TERM", SEPARATOR="|")

/REPORT

/OUTFILE="SCOTT.CHIEFS;DSN=oratester"
/PROCESS=ODBC
/ALIAS=SCOTT_CHIEFS
/FIELD=(LNAME, TYPE=ASCII, POSITION=1, EXT_FIELD="LNAME", SEPARATOR="|")
/FIELD=(FNAME, TYPE=ASCII, POSITION=2, EXT_FIELD="FNAME", SEPARATOR="|")
/FIELD=(TERM, TYPE=ASCII, POSITION=3, EXT_FIELD="TERM", SEPARATOR="|")
/FIELD=(PARTY, TYPE=ASCII, POSITION=4, EXT_FIELD="TERM", SEPARATOR="|")
```



sortcl is designed to process structured, flattened data. Errors will occur if carriage returns, linefeeds, frame characters, etc. are detected within data values when processing database tables.

5.13.4 Loading

When using `PROCESS=ODBC` on output, the output `/FIELD` statements must contain data types that have ODBC equivalents, as shown in Table 10 *on page 389*. Populate only those tables that have already been created in the database; `/PROCESS=ODBC` does not generate a *create table* statement.

When database tables are populated using `/PROCESS=ODBC` on output, tables will be populated using an *append* (`/APPEND`) option by default, where existing rows of table data will remain intact, and new table rows will be appended. You can change this default behavior by using either of the following statements: `/CREATE` or `/UPDATE`.

`/APPEND`

Associate an `/APPEND` with any target to cause output data to be placed after the existing data in a file or table. If the file does not exist or is empty, `/CREATE` will be invoked.

The syntax is:

```
/OUTFILE=output_filename  
/APPEND
```

`/CREATE`

This is the default specification for an ODBC output target. It indicates that a new output file will be created or an existing table will be truncated. If the file name already exists, all previous data in the file will be lost, even if nothing is written by this job.

The syntax is:

```
/OUTFILE=output filename  
/CREATE
```

`/UPDATE`

The `/UPDATE` statement introduces a third option to the choice of `/APPEND` and `/CREATE`. The `/UPDATE` command must include one or more key fields to use in the `WHERE` clause of the resulting `UPDATE` command in SQL. Note that the key fields must exist somewhere in the `INPUT`.

The syntax is:

```
/OUTFILE=output_filename
/PROCESS=ODBC
/UPDATE=(field)
```



If you have large data, and your load speeds are not satisfactory using /PROCESS=ODBC on output, it is recommended that you instead use the bulk load facility supported by your target database and use **sortcl** to pre-sort the data on the longest index or primary key.

For this purpose, tab-delimited /FIELDS are generally supported (SEPARATOR=' \t '), with /PROCESS=RECORD in the output section, rather than /PROCESS=ODBC. You can then load the **sortcl** output into your database table with the bulk load facility.

5.13.5 Special Considerations

Binary numeric data type columns from the database, such integer and double, must also be specified as delimited fields in **sortcl**, and given a data type of NUMERIC. In the case of ODBC data types INTEGER and SHORT with no decimal places, you must specify NUMERIC with PRECISION=0 in **sortcl** (see *Precision on page 100*). Binary floating point types such as DOUBLE should be specified as NUMERIC (though precision can be set optionally). If you specify any of these field types as ASCII, **sortcl** can not perform any mathematical operations on the field, however you can extract and load such data in the database when defined as ASCII.

Regarding database date and timestamp columns, depending on how you specify the field in **sortcl** on the input side, /PROCESS=ODBC will ensure that the column data is converted into that specific format, for example ISO_DATE will convert the database column to a format like 2006-09-19 (see Table 40 *on page 603* for the format of date and timestamp data types). These specifications are made in field statements on the input side because using /PROCESS=ODBC retrieves date and timestamp values in a binary format, and converts them to whichever **sortcl** data type or string format is required. You can also declare such fields as ASCII on input, however any sorting over a date or timestamp column declared as ASCII will collate in ASCII order. If using /PROCESS=ODBC on output, do not perform any data type conversion on these date and timestamp fields. Such columns will load into the target correctly, regardless of the declaration used on the input side.

All other database data types should be specified as ASCII. Table 10 *on page 389* lists the ODBC data types that have **sortcl** data type equivalents.

5.13.6 EXT_FIELD

In some cases, you might want to specify a **sortcl** /FIELD name that is specifically meaningful to **sortcl** in the context of that job script. The desired **sortcl** field name will differ from its corresponding column name in the database. In this case, use the field attribute EXT_FIELD to assign the database column name to the field (to ensure that the field data will be extracted and loaded properly when using /PROCESS=ODBC on input and output, respectively), while using the desired **sortcl** field name that you choose.

For example, if you are extracting data from a table with the column name SS_NO, and prefer to refer to this column as ssn in **sortcl**, use the following for example:

```
/FIELD=(ssn,EXT_FIELD="SS_NO",POSITION=1,SEPARATOR=',')
```



If the **sortcl** /FIELD name differs from its corresponding name in the database, the field data will not be extracted or loaded and an error will occur. You must use /EXT_FIELD to prevent this.

To write specifications for sorting ODBC data files, use the utility **odbc2ddf** that is provided with the **CoSort** package in the \$COSORT_HOME/bin directory. In Windows, **odbc2ddf** is located in *\install_dir\bin* (see the odbc2ddf sub-chapter on page 385). The utility examines ODBC-supported database tables and generates a **sortcl** data definition file based on the detected layout.

5.14 BLOCKED

When /PROCESS=BLOCKED, variable-length records are enclosed in variable-length blocks of records. This format is employed by various tape drives.

The 4-byte block-size indicator is encountered first. It is then followed by a stream of 4-byte record-size indicators followed by record data. When the accumulated records reach the current block size, a new block is introduced with its size indicator, and a new record stream follows.

/PROCESS=BLOCKED is not supported on output.

5.15 XML

When /PROCESS=XML on input, flat records with non-repeating detail elements formatted with XML are extracted and processed by **sortcl**. When /PROCESS=XML on output, data records are generated within XML tags. The XML attribute and tag names, for both **sortcl** input and output purposes, are specified in an XDEF attribute for each /FIELD statement to define the XML data elements.

When defining the XDEF attribute, the syntax for `/FIELD` statements is:

```
/FIELD=(fieldname, POSITION=n, SEPARATOR='x', XDEF="/node1/node2/[ /node3] /tag_name")
```

Optionally, use the `@` sign to specify that data is an attribute of the XML tag preceding it, for example:

```
/FIELD=(fieldname, POSITION=n, SEPARATOR='x', XDEF="/node1/node2/@attribute_name")
```

Note that any level of *nodes* can be specified, depending on how nested the source XML tag is (for input purposes), or how nested you want the XML target tag to be (for output purposes).



Generally, the first two nodes listed in an XDEF attribute will be the same for all fields you are defining because the actual data elements do not typically reside within the two uppermost node levels in the XML hierarchical structure. The examples below demonstrates this.

Alternatively, for either input or output purposes, you can specify a single XDEF for one field, and the same specified nodes will be applied to all remaining fields, where the tag names will assume the `/FIELD` names by default (either the input or output `/FIELD` names for the input or output XML process, respectively).



If you do not define any XDEF attributes when `/PROCESS=XML` on output, the XML tag definitions will default to a generic `/File/Record` naming convention in the target XML file. However, if the input section contains its own XDEF attributes that define an XML source, these `FIELD` statements and XDEFs will map automatically to output, and be applied to the target, if you do not explicitly specify output fields.

The `SIZE` and `POSITION` (or the `SEPARATOR` and `POSITION`) attributes are required for XML input fields defined in **sortcl**, but they have only internal significance. When using `/PROCESS=XML` on input, it is therefore recommended that you specify delimited input fields in order to accommodate field values of any size within the source (see `POSITION` on page 93 and `SEPARATOR` on page 95).

The `/FIELD` statements that are defined on output will determine the size and position of the field elements within the records that are output by **sortcl**. However, if no output fields are explicitly specified, the size and position attributes from the input section will be applied automatically to the output target.

5.15.1 Special Considerations

When binary numeric data type columns, such as INTEGER (INT) or SHORT, are used from a database in delimited fields, those columns must be converted to the NUMERIC data type. Many of the binary data types, such as the SHORT and INT, do not carry a decimal. Therefore, you must set PRECISION=0 when you specify NUMERIC in sortcl (See *Precision on page 100*). Binary floating point types, such as DOUBLE, should also be converted to NUMERIC. If you do not set the PRECISION then a PRECISION of 2 is used. If you specify any of these fields as ASCII, sortcl cannot perform mathematical operations on the fields; however, you can extract and load such data in the database with the data type set to ASCII.

Example: Using /PROCESS=XML on Input

Consider this excerpt of input data in XML format:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<chiefs>
  <chief>
    <president>Washington, George</president>
    <term>1789-1797</term>
    <party>FED</party>
    <state>VA</state>
  </chief>
  <chief>
    <president>Adams, John</president>
    <term>1797-1801</term>
    <party>FED</party>
    <state>MA</state>
  </chief>
  ....
</chiefs>
```

The following script, **xml_in.scl**, extracts this XML data, and sorts it over the president field:

```
/INFILE=chiefs10a_huge.xml
/PROCESS=XML
/FIELD=(president, POSITION=1, SEPARATOR='|', XDEF="/chiefs/chief/president")
/FIELD=(term, POSITION=2, SEPARATOR='|', XDEF="/chiefs/chief/term")
/FIELD=(party, POSITION=3, SEPARATOR='|', XDEF="/chiefs/chief/party")
/FIELD=(state, POSITION=4, SEPARATOR='|', XDEF="/chiefs/chief/state")
/SORT
/KEY=president
/OUTFILE=chiefs10a.out
```

Alternatively, the following convention can be applied to produce identical results:

```
/INFILE=chiefs10a_huge.xml
/PROCESS=XML
/FIELD=(president,POSITION=1,SEPARATOR='|')
/FIELD=(term,POSITION=2,SEPARATOR='|')
/FIELD=(party,POSITION=3,SEPARATOR='|')
/FIELD=(state,POSITION=4,SEPARATOR='|',XDEF="/chiefs/chief/state")
/SORT
/KEY=president
/OUTFILE=chiefs10a.out
```

This produces the following, for example:

```
Adams, John|1797-1801|FED|MA
Adams, John Quincy|1825-1829|D-R|MA
Harrison, William Henry|1841-1841|WHG|VA
Jackson, Andrew|1829-1837|DEM|SC
Jefferson, Thomas|1801-1809|D-R|VA
Madison, James|1809-1817|D-R|VA
Monroe, James|1817-1825|D-R|VA
Tyler, John|1841-1845|WHG|VA
Van Buren, Martin|1837-1841|DEM|NY
Washington, George|1789-1797|FED|VA
```

As shown above, the field values were extracted from the XML source based on the XDEF location references in the field statements, and the records were sorted over the president field. Because no output fields were defined, the pipe-delimited record layout defined on input was used for the output layout.



In the above example, a comma (,) separator can be used instead of a pipe (|), but a `FRAME=' '` attribute would be required in the president `/FIELD` definition to ensure that the commas found within presidents' names would not be considered as field separator characters (see *FRAME on page 101*).

Ensure that your XML source file is well formed if you are using `/PROCESS=XML` on input.

Example: Using /PROCESS=XML on Output

Consider the following input data:

```
Adams, John|1797-1801|FED|MA
Adams, John Quincy|1825-1829|D-R|MA
Van Buren, Martin|1837-1841|DEM|NY
Jackson, Andrew|1829-1837|DEM|SC
Tyler, John|1841-1845|WHG|VA
Jefferson, Thomas|1801-1809|D-R|VA
Monroe, James|1817-1825|D-R|VA
Washington, George|1789-1797|FED|VA
Madison, James|1809-1817|D-R|VA
Harrison, William Henry|1841-1841|WHG|VA
```

The following script, **xml_out.scl**, sorts this data by president, and generates an XML file with the above data elements inserted within tags specified by the XDEF entries in the output /FIELD statements:

```
/INFILE=data.in
/PROCESS=record
/FIELD=(president, POSITION=1, SEPARATOR='|')
/FIELD=(term, POSITION=2, SEPARATOR='|')
/FIELD=(party, POSITION=3, SEPARATOR='|')
/FIELD=(state, POSITION=4, SEPARATOR='|')
/SORT
/KEY=president
/OUTFILE=data.xml
/PROCESS=xml
/FIELD=(president, POSITION=1, SEPARATOR="^", XDEF="/chiefs/chief@president")
/FIELD=(term, POSITION=2, SEPARATOR="^", XDEF="/chiefs/chief/term")
/FIELD=(state, POSITION=3, SEPARATOR="^", XDEF="/chiefs/chief/state")
/FIELD=(party, POSITION=4, SEPARATOR="^", XDEF="/chiefs/chief/party")
```

This produces the following XML document, **data.xml** (middle section omitted for readability):


```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
  <chiefs>
    <chief president="Adams, John">
      <term>1797-1801</term>
      <state>MA</state>
      <party>FED</party>
    </chief>
    <chief president="Adams, John Quincy">
      <term>1825-1829</term>
      <state>MA</state>
      <party>D-R</party>
    </chief>
    ...
    <chief president="Van Buren, Martin">
      <term>1837-1841</term>
      <state>NY</state>
      <party>DEM</party>
    </chief>
    <chief president="Washington, George">
      <term>1789-1797</term>
      <state>VA</state>
      <party>FED</party>
    </chief>
  </chiefs>

```

5.16 RANDOM (Generating Test Data)

When /PROCESS=RANDOM on input, field contents are randomly generated based on input /FIELD specifications that you specify (with respect to the data type and other field attributes such as size, position, and separator). **sortcl** can also randomly select field contents from a text-based set file that you provide, in order to produce realistic looking data items. /PROCESS=RANDOM cannot be used on output.

An /INFILE statement is required, but the file you name is not recognized as an input source because the records are being *created* by **sortcl**.

The /INCOLLECT statement determines the number of records to be generated, for example set /INCOLLECT=50 when using /PROCESS=RANDOM to generate 50 records (the default is 100 records).

sortcl allows you to transform the randomly generated, or selected, data in the same way you would transform records from an actual input source, as described throughout this chapter. This allows you to integrate data and application prototyping into your existing data processing (transformation), presentation (reporting), and protection (encryption, de-identification, etc.) operations.



The /PROCESS=RANDOM feature allows you to make use of features found in the standalone **RowGen** product for test data generation, application prototyping, safe file outsourcing, and benchmarking. The job scripting syntax of **RowGen** is the same as **sortcl** to allow for interoperability and interchangeability between the products. The same job script can be applied to either **RowGen** (when the input file is not present, but test data must be created that mimics actual input) or **CoSort**'s **sortcl** (when the infile exists and is able to be processed).

RowGen offers a full range of test data generation and ranging functions, and a GUI to specify both individual test structures and multiple, DDL-compliant tables for safe database and enterprise data warehouse population. Many more set files and dedicated documentation is included for users requiring high volumes of database test data with referential integrity. Contact your IRI agent for details about the **RowGen** product.

Example: Random Data Generation

To create randomly generated field contents, consider the following script, **random.scl**:

```
/INFILE=simple.in           # placeholder only; no real input is recognized
/INCOLLECT=5
/PROCESS=RANDOM             # create random data based on specs below
  /FIELD=(code,POSITION=1,SIZE=5,ALPHA_DIGIT) # alpha_digit field; size 5
  /FIELD=(value,POSITION=10,SIZE=7,PRECISION=2,NUMERIC) # numeric field
  /INCLUDE WHERE value > 10 # filter values
/REPORT
/OUTFILE=sample_data
  /FIELD=(code,POSITION=1,SIZE=5,ASCII) # alpha_digit field; size 5
  /FIELD=(value,POSITION=10,SIZE=12,PRECISION=2,NUMERIC,FILL='0')
                                # re-format results
```

This **sortcl** script produces the following:

```
3AE2S      000001702.07
5Jik4      000002935.14
864qt      000006721.16
p01d3      000003634.73
t1a58      000001706.85
```

As shown above:

- Five records were generated (/INCOLLECT=5).
- The code field contains only alphabetic and numeric characters, as determined by the ALPHA_DIGIT data type. Note that this data type is specific to users of /PROCESS=RANDOM.
- The /INCLUDE condition ensured that only values greater than 10 were generated.
- The output layout extended the length of the value field and padded the results with leading zeroes (see FILL on page 106).

Example: Random Data Selection

To randomly select from a user-provided set file, consider the following text file, **first_names.set**:

```
Bill
Carole
Jane
Jill
John
Peter
Rachel
Roger
```

The following script, **rand_set.scl**, will select values, at random, from **first_names.set** above, and produce a report with realistic record contents:

```
/INFILE=names.in          # placeholder only; no real input is recognized
/PROCESS=RANDOM           # create random data based on specs below
  /FIELD=(name,SET=first_names.set,POSITION=1,SIZE=9)  # select from
SET
  /FIELD=(salary,POSITION=10,SIZE=5,digit) # digit field
  /INCLUDE WHERE salary > 50000 AND salary < 99999 # filter values
/SORT
  /KEY=name
  /NODUPPLICATES          # Ensures that each name is selected once
/OUTFILE=sample_data_names
  /OUTCOLLECT=8           # Produces one record for each name
  /FIELD=(name,POSITION=1)
  /FIELD=(salary,POSITION=10,CURRENCY) # convert to CURRENCY
```

This produces:

Bill	\$72,171.00
Carole	\$95,248.00
Jane	\$67,679.00
Jill	\$52,672.00
John	\$59,568.00
Peter	\$55,355.00
Rachel	\$67,348.00
Roger	\$74,299.00

Note that:

- The name field contains randomly selected items from **first_names.set**.
- The /NODUPPLICATES entry, together with /OUTCOLLECT=8, ensured that each of the eight unique names from the set file was selected and returned (see No Duplicates, Duplicates Only *on page 168* and /OUTCOLLECT *on page 275*).
- The salary field is comprised of random digits from 0 through 9, as determined by the DIGIT data type. Note that this data type is specific to users of /PROCESS=RANDOM.
- The salary field was converted to CURRENCY on output (see CURRENCY *on page 124*).



The SET= attribute within a /FIELD statement when using /PROCESS=RANDOM (as illustrated in this example) is applied differently by **sortcl** than the way the SET= attribute is applied for all other /PROCESS types. In all other cases, the SET= attribute is used for table look up purposes (see Set files *on page 198*).

5.17 ELF (W3C Extended Log Format)

sortcl supports internet web transaction files in W3C Extended Log Format (ELF). These files have a header containing lines of comments, followed by a line naming the data fields. To instruct **sortcl** to skip the header before sorting, specify `/PROCESS=ELF` in the input file. To force **sortcl** to generate a header based on the file's field names and positions, specify `/PROCESS=ELF` in the output file.

To write specifications for sorting ELF data files, use the utility **elf2ddf** that is provided with the **CoSort** package in the `$COSORT_HOME/bin` directory. In Windows, **elf2ddf** is located in `\install_dir\bin` (see the **elf2ddf** sub-chapter *on page 379*). This utility reads ELF file headers and generates **sortcl** data definition files accordingly, giving users a base for writing specifications. Its syntax is:

```
elf2ddf datafile [data-definition-filename]
```

CLF (NCSA Common Log Format)

Web logs in CLF format can also be processed within **sortcl**. Example data definition files **CLF_Referrer.ddf**, **CLF_Agent.ddf**, and **CLF_Access.ddf** are provided in the **examples/SORTCL** directory (on Unix) and `\install_dir\examples\sortcl` (on Windows).

6 /CHARSET

Currently, only UTF16 record support is implemented for /CHARSET. Single-byte characters are used when the statement is not present. This statement is needed after each Unicode file is named.

The statement: /CHARSET=UTF16 must be placed after each /INFILE or /OUTFILE statement. For variable length records, the file uses Unicode record termination characters of 0x000D for CR (if present) and 0x000A for LF.

7 /ENDIAN

The /ENDIAN statement in a file description defines the byte order of the numerical record prefix.

The /ENDIAN allows **sortcl** to correctly read the multi-byte value where it was generated in a machine of a different endianness. Individual field endianness can be specified to override the file endianness. See FIELD ENDIANNES on page 120.

For CHARSET UTF16, which is used to specify Unicode record termination, the byte order mark (BOM) can also be specified by the /ENDIAN statement.

The statement:

```
/ENDIAN= [BOM] [, BIG/LITTLE/MACHINE/INPUT]
```

is placed after the /INFILE or /OUTFILE statement to specify the byte order of the file.

The following are valid input files options:

/ENDIAN=BOM	File is read in the endian order as defined by the Byte Order Mark in the file.
/ENDIAN=BIG	File is read in the Big-Endian byte order.
/ENDIAN=LITTLE	File is read in the Little-Endian byte order.
/ENDIAN=MACHINE	Use machine endianness. This is the default if the /ENDIAN statement is not used.

The following are valid for specific output files:

/ENDIAN=BOM [,INPUT]	A Byte Order Mark for the INFILE data endianness is added to the output file. The output is written in the endianness of the input data.
/ENDIAN=BOM, BIG	File is written in Big-Endian byte order and Byte Order Mark is added. The output data endianness is converted if not already in that state.
/ENDIAN=BOM, LITTLE	File is written in Little-Endian byte order and Byte Order Mark is added. The output data endianness is converted if not already in that state.
/ENDIAN=BOM, MACHINE	A Byte Order Mark for the MACHINE endianness is added to the output file. The output data endianness is converted if not already in that state.

/ENDIAN=BIG	No BOM is written. File is written in Big-Endian byte order. The output data endianness is converted if not already in that state.
/ENDIAN=LITTLE	No BOM is written. File is written in Little-Endian byte order. The output data endianness is converted if not already in that state.
/ENDIAN=MACHINE	No BOM is written. The output data endianness is converted if not already in that state.
/ENDIAN=INPUT	This is the default if the /ENDIAN statement is not used. The output is written in the endianness of the input data.

Data types effected by endianness

Unicode data types and all numeric forms other than types CHAR, SCHAR, UCHAR, ASCNUM, and NUMERIC, as well as COBOL types MF_CMP5 and UMF_CMP5, are effected by endianness.

8 STATISTICS

To produce runtime statistics, use the following statement:

```
/STATISTICS [= [path] filename]
```

sortcl will output to a specified *filename*, or, by default, to standard out. All runtime statistics comprise the following, but additional information is returned depending on the action performed (such as a `/JOIN`):

- Input File Names, and for each input file:
 - `/HEADREAD` or `/HEADSKIP` byte length
 - `/TAILREAD` or `/TAILSKIP` byte length
 - record length
 - number of records rejected
 - number of records accepted
- Process Record (`/INREC`) information, including:
 - record length
 - key number, direction, position, size, and format
 - `/NODUPLICATES`
 - `/STABLE`
- Output File Names, and for each output file:
 - `/HEADWRITE` and `/HEADREC` byte lengths
 - `/TAILWRITE` and `/FOOTREC` byte lengths
 - record length
 - number of records rejected
 - number of records accepted
- Job Efficiency (Tuner) information, including:
 - total memory used
 - internal buffer (I/O block) size
 - number of records in memory
 - work area directories
- Job Total information, which can include:
 - number of records read
 - number of records sorted
 - number of records output
 - time job finished and began
 - real, user and system times

Statistics are also produced for jobs that terminate with an error, in which case the error message is also included with the above information.

9 **AUDITING**

sortcl can produce a self-appending log file, in XML format, that contains comprehensive job information for the purposes of auditing. Auditing is enabled when the **AUDIT** entry is present in the **cosortrc** file on Unix or the Windows Registry (see **AUDIT [path]filename** *on page 627*). An audit record is produced for every **sortcl** execution, and these records append to the XML *[path]filename* specified by the **AUDIT** entry.

Example: *on page 84* demonstrates how a **sortcl** audit record is generated based on the job script contents and user environment at the time of execution. It also shows what the audit file looks like when opened in both a text editor and an XML browser utility.

You can also use **sortcl** to process, and obtain specific information from, an audit log (see *Querying the Audit File on page 87*).



AUDIT is not compatible with scripts that require the recursive use of **sortcl**, including scripts that JOIN using the **NOT_SORTED** keyword, or that JOIN more than two files.

Supplied DDF File

To facilitate the use of **sortcl** for generating reports based on audit information, the file **audit_log.ddf** is provided in **\$COSORT_HOME/etc** (Unix) or **install_dir/etc** (Windows). It includes **/FIELD** references to all the elements contained in the **sortcl**-generated audit log. It is provided to keep the metadata separate from the application (see *Querying the Audit File on page 87* for an example).

The data elements that comprise the audit log are as follows:

Product	Software product used for job execution, for example CoSort.
Version	Product version number, for example 9.5.1.
VersionTag	Product version tag, for example S110425-1508.
Serial	Product serial number, for example 11027.9518.
OperatingSystem	For example, windows XP.
User	The end user running the job, for example John_Thomas.
ProcessId	The operating system process ID, for example 3992.

Terminal	The ID of the CoSort user's connection to the host.
Program	Path name and file name of the executable used to perform the job, such as <code>c:\Work_area\sortcl</code> .
Command	References the command line entry that launched the job, which specifies the job script name, for example <code>/ specification=csaudit.scl</code> .
StartTime	Start time and date of sortcl job execution, in ISO_TIMESTAMP format (see Table 40 <i>on page 603</i>), for example <code>2011-04-19 21:47:15</code> .
EndTime	End time and date of sortcl job execution in ISO_TIMESTAMP format, for example, <code>2011-04-19 21:49:06</code> .
RunTime	The runtime duration of the sortcl job (the difference between StartTime and EndTime) in HH:MM:SS format, for example <code>00:01:51</code> .
ReturnCode	For example, 0, if the job completed without error.
ErrorMessage	The error message associated with job execution. <code>normal</code> return indicates no error.
RecordsProcessed	The number of input records processed (after any pre-processing record filter logic was applied).
Script	The complete text of the job script, where each line of entry is separated by a space.
Environment	All environment variables at the time of execution, which will show the literal equivalents of any environment variables referenced in the job script.

Example: Creating an /AUDIT file

Consider that the following entry is set in the Windows Registry (or set in the **cosortrc** file on Unix):

```
AUDIT=C:\CoSort\Tests\mytests\audit\csaudit.xml
```

Consider that the following **sortcl** job script, **audit.scl**, is executed:

```
/INFILE=$CSINPUT  
/SPECIFICATION=fields.ddf  
/REPORT  
/OUTFILE=$CSOUTPUT  
/SPECIFICATION=fields.ddf
```

where **fields.ddf** contains the first two **/FIELD** statements and a nested **/SPEC** entry that refers to the remaining two fields of the input record:

```
/FIELD= (name, POSITION=1, SIZE=27, ASCII)  
/FIELD= (year, POSITION=28, SIZE=12, ASCII)  
/SPECIFICATION=fields2.ddf
```

and where **fields2.ddf** contains:

```
/FIELD= (party, POSITION=40, SIZE=5, ASCII)  
/FIELD= (state, POSITION=45, SIZE=2, ASCII)
```

When the above job script is executed, an entry in the audit file **csaudit.xml**, is created/added. When **csaudit.xml** is opened in a text editor, the **.ddf** file names and their components are tabbed within the **<Script>** element to allow for easier readability of nested script specifications, as follows:

```

...
<Script>
/INFILE=C:\CoSort\Tests\mytests\audit\chiefs
/SPECIFICATION=fields.ddf
    /FIELD= (name, POSITION=1, SIZE=27, ASCII)
    /FIELD= (year, POSITION=28, SIZE=12, ASCII)
    /SPECIFICATION=fields2.ddf
        /FIELD= (party, POSITION=40, SIZE=5, ASCII)
        /FIELD= (state, POSITION=45, SIZE=2, ASCII)
/REPORT
/OUTFILE=C:\CoSort\Tests\mytests\audit\chiefs.out
/SPECIFICATION=fields.ddf
    /FIELD= (name, POSITION=1, SIZE=27, ASCII)
    /FIELD= (year, POSITION=28, SIZE=12, ASCII)
    /SPECIFICATION=fields2.ddf
        /FIELD= (party, POSITION=40, SIZE=5, ASCII)
        /FIELD= (state, POSITION=45, SIZE=2, ASCII)
</Script>
...

```

However, when opened in an XML-supported browser, the `<script>` entry displays as a series of space-delimited entries and there are no tabs. An XML browser will therefore display **csaudit.xml** as follows (note that the `<Environment>` entry near the bottom has been truncated for readability purposes):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <AuditTrail>
- <AuditRecord>
  <Product>CoSort</Product>
  <Version>9.5.1</Version>
  <VersionTag>D90110623-0000</VersionTag>
  <Serial>12345.6789</Serial>
  <OperatingSystem>Windows XP</OperatingSystem>
  <User>cosort_user</User>
  <ProcessId>4784</ProcessId>
  <Program>C:\CoSort\Tests\mytests\audit\sortcl</Program>
  <Command>/sp=audit.scl</Command>
  <StartTime>2008-01-19 21.47.15</StartTime>
  <EndTime>2008-01-19 21.48.06</EndTime>
  <RunTime>00:01:51</RunTime>
  <ReturnCode>0</ReturnCode>
  <ErrorMessage>normal return</ErrorMessage>
  <RecordsProcessed>42</RecordsProcessed>
  <Script>/STATISTICS=C:\CoSort\Tests\mytests\audit\csstat.log /
INFILE=C:\CoSort\Tests\mytests\audit\chiefs /SPEC=fields.ddf /
FIELD=(name,POS=1,SIZE=27,ASCII) /FIELD=(year,POS=28,SIZE=12,ASCII)
/SPEC=fields2.ddf /FIELD=(party,POS=40,SIZE=5,ASCII) /
FIELD=(state,POS=45,SIZE=2,ASCII) /SORT /OUT-
FILE=C:\CoSort\Tests\mytests\audit\chiefs.out /SPEC=fields.ddf /
FIELD=(name,POS=1,SIZE=27,ASCII) /FIELD=(year,POS=28,SIZE=12,ASCII)
/SPEC=fields2.ddf /FIELD=(party,POS=40,SIZE=5,ASCII) /
FIELD=(state,POS=45,SIZE=2,ASCII)</Script>
  <Environment>ALLUSERSPROFILE=C:\Documents and Settings\All Users APP-
DATA=C:\Documents and Settings\cosort_user\Application Data CLASS-
PATH=.;C:\Program Files\Java\j2re1.4.2_03\lib\ext\QTJava.zip
CLIENTNAME=Console CommonProgramFiles=C:\Program Files\Common Files
...
</Environment>
</AuditRecord>
</AuditTrail>
```



This audit file contains only one record. Subsequent jobs that are written to **csaudit.xml** are appended to the bottom, and always begin with a new **<AuditRecord>** tag.

Example: Querying the Audit File

Use `/PROCESS=XML` in the input section of a **sortcl** job script to process the records contained in a **sortcl** audit log (see *XML on page 68*).

Consider the following XML audit log, **catalog.xml**, which was generated by **sortcl** (note that the `<Environment>` entry has been truncated for readability purposes):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <AuditTrail>
- <AuditRecord>
  <Product>CoSort</Product>
  <Version>9.5.1</Version>
  <VersionTag>D95110623-0000</VersionTag>
  <Serial>12345.6789</Serial>
  <OperatingSystem>Windows 2000</OperatingSystem>
  <User>Bill_Evans</User>
  <ProcessId>1792</ProcessId>
  <Terminal>console</Terminal>
  <Program>C:\IRI\CoSort95\bin\sortcl</Program>
  <Command>/spec=catalog_plant.scl</Command>
  <StartTime>2011-01-12 09.59.27</StartTime>
  <EndTime>2011-01-12 10.01.17</EndTime>
  <RunTime>00:01:50</RunTime>
  <ReturnCode>0</ReturnCode>
  <ErrorMessage></ErrorMessage>
  <RecordsProcessed>0</RecordsProcessed>
  <Script>/INFILE=plant_catalog.txt /PROCESS=record /FIELD=(common, POS=1,
SEP='^') /FIELD=(botanical, POS=2, SEP='^') /FIELD=(zone, POS=3, SEP='^')
/FIELD=(light, POS=4, SEP='^') /FIELD=(price, POS=5, SEP='^') /FIELD=(availabil-
ity, POS=6, SEP='^') /KEY=botanical /OUTFILE=catalog_plant.xml</Script>
  <Environment>ALLUSERSPROFILE=C:\Documents and Settings\All Users\WINNT APP-
DATA=C:\Documents and Settings\Bill_Evans\Application Data CommonProgram-
Files=C:\Program Files\Common Files COMPUTERTNAME=ATHENA
  ...
</Environment>
</AuditRecord>
</AuditTrail>
```



For the purposes of this example, the complete **catalog.xml** is comprised of several **sortcl** jobs (audit records), the first of which is shown above.

The following script, **catalog.scl**, reads the data in the XML audit file, **catalog.xml**, sorts the file by user, and outputs records in CSV format that can be read using a spreadsheet utility such as Microsoft Excel (see *CSV on page 60*):

```
/INFILE=catalog.xml
  /PROCESS=XML
  /SPECIFICATION=audit_log.ddf
      # Copied into current directory from install_dir\etc
/SORT
  /KEY=User
/OUTFILE=csaudit.csv
  /PROCESS=CSV
  /INCLUDE WHERE Command CT "catalog"    # returns records where the sortcl
      # job script contains this name
  /FIELD=(User,POSITION=1,SEPARATOR=',',FRAME='')
  /FIELD=(Command,POSITION=2,SEPARATOR=',',FRAME='')
  /FIELD=(StartTime,POSITION=3,SEPARATOR=',',FRAME='')
```

This produces **csaudit.csv**, which might look like this for example (assuming the input source, **catalog.xml**, consists of several records):

```
User,Command,StartTime,StartDate
"Bill_Evans","/spec=catalog_plant.scl","2008-01-12 09.59.27"
"Dave_Johnston","/spec=catalog_survey.scl","2008-01-10 08.16.13"
"John_Thompson","/spec=catalog_plant.scl","2008-01-14 10.35.23"
"Marcy_Wallace","/spec=catalog_survey.scl","2008-01-08 07.23.54"
"Roger_Smith",",","/spec=catalog_report.scl","2008-01-10 22.59.22"
```

As shown above, the audit file was processed, sorted by user, and only the desired output records (those with **catalog** in the job script name) were produced in CSV format.



Using **/PROCESS=XML** on input, and referring to the input **/FIELD** specifications provided in **audit_log.ddf**, query and analyze the audit log file using any of the **sortcl** options described throughout this chapter such as record filter logic, field-level IF THEN ELSE logic, aggregation, and reformatting.

10 /ALIAS

Use the /ALIAS statement to create a logical name for a physical file that can be used wherever that file is referenced in the script. By using a short mnemonic name, the resultant script is easier to write and understand. Additionally, the logical script that you create is easy to maintain if your physical file changes because you need only change the name of the physical file in your specifications once (or make no changes if your physical file is referenced by an environment variable).

The syntax for declaring this logical name is:

```
/ALIAS [=] name
```

where /ALIAS appears after a /FILE or /INFILE statement for the file it is referencing, and before any /FIELD statements.

In addition to ascribing a more logical name to a physical file, use the declared alias as a field value when defining an /INREC or an output file (see /INREC *on page 150* and Output Files *on page 51*). When multiple input files are used, create a field, `alias`, that indicates the input-file source of the current record.

Example: Using /ALIAS

In this example, `alias` is used to give logical names to three input file sources, and to create derived fields based on the input-file origin of the records. Three input files are used.

dept10:	dept21:	dept50:
10251 203.54 2003-03-01	91520 95.25 2003-03-01	21101 143.53 2003-03-02
23201 1043.23 2003-03-01	23201 543.23 2003-03-02	13251 21.00 2003-03-04
03261 521.50 2003-03-03	32546 24.00 2003-03-04	89342 19.25 2003-03-05
91520 689.35 2003-03-05	13251 521.50 2003-03-04	
32546 714.00 2003-03-07	91520 189.05 2003-03-05	
	13251 915.50 2003-03-06	

Using the job script **alias.scl**:

```
/INFILE=dept10
  /ALIAS=D10
  /FIELD=(acct, POSITION=1, SIZE=5)
  /FIELD=(amount, POSITION=7, SIZE=8)
  /FIELD=(date, POSITION=16, SIZE=10)
/INFILE=dept21
  /ALIAS=D21
  /FIELD=(acct, POSITION=1, SIZE=5)
  /FIELD=(amount, POSITION=7, SIZE=8)
  /FIELD=(date, POSITION=16, SIZE=10)
/INFILE=dept50
  /ALIAS=D50
  /FIELD=(acct, POSITION=1, SIZE=5)
  /FIELD=(amount, POSITION=7, SIZE=8)
  /FIELD=(date, POSITION=16, SIZE=10)
/INREC
  /FIELD=(acct, POSITION=1, SIZE=5)
  /FIELD=(alias, POSITION=7, SIZE=3)
  /FIELD=(amount, POSITION=10, SIZE=8)
  /FIELD=(date, POSITION=20, SIZE=10)
/SORT
  /KEY=acct
  /KEY=date
  /KEY=amount
/OUTFILE=alias.out
  /HEADREC="Acct      Dept      Amount    Date\n\n"
  /FIELD=(acct, POSITION=1, SIZE=5)
  /FIELD=(dept, POSITION=9, IF alias EQ "D50" THEN "Promo" ELSE alias)
  /FIELD=(amount, POSITION=16, SIZE=8)
  /FIELD=(date, POSITION=27, SIZE=10)
```

alias.out is as follows:

Acct	Dept	Amount	Date
03261	D10	521.50	2003-03-03
10251	D10	203.54	2003-03-01
13251	Promo	21.00	2003-03-04
13251	D21	521.50	2003-03-04
13251	D21	915.50	2003-03-06
21101	Promo	143.53	2003-03-02
23201	D10	1043.23	2003-03-01
23201	D21	543.23	2003-03-02
32546	D21	24.00	2003-03-04
32546	D10	714.00	2003-03-07
89342	Promo	19.25	2003-03-05
91520	D21	95.25	2003-03-01
91520	D21	189.05	2003-03-05
91520	D10	689.35	2003-03-05

Use of alias in a join script allows logical references to the input files, making the script easier to read. Below is an example:

```
/INFILE=chiefs10h.sorted
/ALIAS=First
/FIELD=(pres, POSITION=1, SIZE=27)
/FIELD=(party, POSITION=40, SIZE=3)
/FIELD=(state, POSITION=45, SIZE=2)
/INFILE=chiefs10h_rev.sorted
/alias=Second
/FIELD=(pres, POSITION=1, SIZE=27)
/FIELD=(party, POSITION=40, SIZE=3)
/FIELD=(state, POSITION=45, SIZE=2)
/JOIN First Second WHERE First.pres == Second.pres
/OUTFILE=out
/FIELD=(First.pres, POSITION=1, SIZE=23)
/FIELD=(First.party, POSITION=25, SIZE=3)
/FIELD=(Second.state, POSITION=30, SIZE=2)
/FIELD=(Second.party, POSITION=34, SIZE=3)
```

In this case, the pres and party fields from the **chiefs10h.sorted** file (with the alias First), along with the state and party fields of the **chiefs10h_rev.sorted** file (with the alias Second), will appear in the output at the stated positions.

11 FIELDS

Data files consist of records; records consist of fields. The location of a field is either:

- fixed position, where the starting byte for a field is always in the same column
- floating or delimited, where the starting position for any field after the first field is to the right of a separator character

In both cases, numbering begins with 1. **sortcl** allows both fixed and floating fields in the same record. Defined fields can also overlap each other in an input record.

11.1 Syntax

The following is the syntax for defining a fixed-position field. The **POSITION** and **SIZE** are required, in addition to field name, when defining input attributes:

```
/FIELD=(fieldname or field_function,POSITION=n,  
[SIZE=n.[n]] [,PRECISION=n] [,SET=set_filename[options]  
[,FRAME='char'] [,alignment] [,MILL]  
[,FILL='char' or FILL=n] [,null assignment] [,XDEF=tag]  
[,numeric_attribute(s)] [,ENDIAN=option] [,data type])
```

The following is the syntax for defining delimited fields. A **POSITION** and **SEPARATOR** are required, in addition to the field name, when defining input attributes:

```
/FIELD=(fieldname or field_function,POSITION=n,  
SEPARATOR='Char(s)' [,SIZE=n.[n]] [,PRECISION=n]  
[,SET=set_filename[options] [,FRAME='char']  
[,alignment] [,MILL] [,FILL='char' or FILL=n] [,null assignment]  
[,XDEF=tag] [,numeric_attribute(s)] [,ENDIAN=option] [,data  
type])
```



The SET attribute is used with output /FIELD statements only, and is described in Set files *on page 198*.

XDEF pertains only to /PROCESS=XML (see XML *on page 68*).

Use the /FIELD_PREDICATE option to set default attributes and values for the /FIELD statements that follow it. This makes it easier to write and read large scripts that have repeating field descriptions (see FIELD_PREDICATE *on page 143*).

When defining a field, field attributes are separated by commas. Field name appears first, but other attributes can appear in any order. The *n* shown for some of the parameter values represents a whole number. **sortcl** also supports several field-level functions, as described in FIELD FUNCTIONS *on page 191*.

11.2 Field Name

The field name identifies the field and can be referenced in other **sortcl** statements.

In the output file only, it is possible to have a field definition that is only an input field name, that is, without additional attributes. Each defined field will be mapped sequentially (one after the other). For example, if you had defined the fields `lastname` and `firstname` in the input file, you could have the following field definitions in an output file:

```
/FIELD=(lastname)
/FIELD=(firstname)
```

Filler

When there are areas in a record that you do not intend to reference in the output, use the input field name `filler`, one or more times, in order to account for all the record contents. The syntax is as follows:

```
/FIELD=(filler, POSITION=n, SIZE=n)
or
/FIELD=(filler, POSITION=n, SEPARATOR=' char' )
```

11.3 POSITION

This statement describes the starting location of each field in the record. The syntax is:

```
POSITION=n
```

In a fixed position field, *n* is the starting column for the first byte of the field. For example:

```
/FIELD=(lastname, POSITION=17, SIZE=10)
```

defines the `lastname` field beginning at byte position 17 and being 10 bytes wide.

In a variable-length (delimited) field, *n* is the field number when counting from left to right, with respect to the separator. For example:

```
/FIELD=(lastname, POSITION=2, SEPARATOR=', ')
```

defines the lastname field as being the second field from the left within the record when the fields are separated by a comma (see *SEPARATOR on page 95*).

Position Over-Defines: Defining Multiple Fixed-Length Fields

Over-define multiple fixed-position fields as a single `/FIELD` using the appropriate `POSITION` and `SIZE` attributes (see *SIZE on page 99*).

For example, considering the following input data:

George	Wallace	50000
George	Smith	51000
Henry	Jones	23000

Combine multiple fields into one `/FIELD` statement, for the purpose of processing them together as a single sort key, for example:

```
/FIELD=(firstname_lastname, POSITION=1, SIZE=16)
```

Position Over-Defines: Defining Contiguous Variable-Length Fields

Over-define contiguous delimited fields as a single `/FIELD` using a variant of the `POSITION` attribute. Consider the following input data:

Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Adams, John Quincy	1825-1829	D-R	MA

The options for defining multiple variable-length fields are:

POSITION=:*n* From the beginning of the record to field *n*. For example:

```
/FIELD=(name_term_party, POSITION=:3, SEPARATOR='| ')
```

defines, as one field, the first three fields within the record.

POSITION=*n*: From field *n* to the end of the record. For example:

```
/FIELD=(term_to_end, POSITION=2:, SEPARATOR='|')
```

defines, as one field, the second field to the end of the record.

This is particularly useful when you have a large number of fields in a record, and need to map them to the output record, but do not intend to process any individual fields (for example, as a key) within the multiple remaining fields of the record. In this way, over-define a single field to the end of a record rather than use multiple `/FIELD` statements to define every field that comprises the remainder of the record.

POSITION=*n1:n2* From field *n1* up to and including field *n2*. For example:

```
/FIELD=(term_party, POSITION=2:3, SEPARATOR='|')
```

defines, as one field, the second and third fields within the record.

11.4 SEPARATOR

The separator character is used as the delimiting character that separates floating fields. The syntax is:

```
SEPARATOR='option'
```

where `SEPARATOR` can be abbreviated to `SEP`, and where *option* can consist of any of the following:

- a single ASCII character such as a comma ,
- a multi-character ASCII string such as , * |
(see Multi-Character Separators *on page 96*)
- a control character such as a tab (\t)
- a multi-byte character (see Multi-Byte Character Separators *on page 97*)
- hexadecimal character such as a # sign: (\x23) (see ASCII COLLATING SEQUENCE *on page 654*)



If your separator character is a single quote ('), you must escape it with a backslash character as follows:

```
SEPARATOR=' \ ' '
```

See *Different Separators on page 98* to define fields with respect to different separator characters within the same record.

If you have records that use the comma (,) as the separator character, you might have records formatted as follows where you have two distinct fields:

```
Washington, George
Adams, John
Jefferson, Thomas
```

In this example, *field 1* could be called `lastname`, and *field 2* could be called `firstname`. In this case, your field definitions would be as follows:

```
/FIELD= (Lastname, POSITION=1, SEPARATOR=',')
/FIELD= (Firstname, POSITION=2, SEPARATOR=',')
```

Multi-Character Separators

Specify a multiple-character ASCII separator.

Example: Using a Multi-Character Separator

Consider the following data:

```
Apples,*|Chicago,*|Red
Pears,*|Philadelphia,*|Yellow
Peaches,*|Macon,*|Peach
```

Use the following script, **multi_sep.scl**, to define this input data and reformat it:

```
/INFILE=Produce
/FIELD= (Fruit, POSITION=1, SEPARATOR=',*|')
/FIELD= (City, POSITION=2, SEPARATOR=',*|')
/FIELD= (Color, POSITION=3, SEPARATOR=',*|')
/REPORT
/OUTFILE=stdout
/FIELD= (Fruit, POSITION=1, SEPARATOR=',')
/FIELD= (City, POSITION=2, SEPARATOR=',')
/FIELD= (Color, POSITION=3, SEPARATOR=',')
```

On execution, the screen displays:

```
Apples,Chicago,Red
Pears,Philadelphia,Yellow
Peaches,Macon,Peach
```




Incorporate control characters as part of a multi-character separator.
For example, specify

```
SEPARATOR=' , *\t '
```

to describe a separator that consists of a comma, an asterisk, and a tab.

Multi-Byte Character Separators

In cases where you are processing multi-byte characters (see Multi-Byte Character Types *on page 127*), specify a multi-byte separator character so that **sortcl** can search for the separator as a whole character, rather than byte-by-byte.



When using multi-byte data types, ensure that the last field in each record is terminated with the specified field separator. This is required by **sortcl** because the record terminator might not be in the same encoding as the field data.

When the separator character is ASCII-compatible, such as UTF8 or SJIS, insert the character directly in the SEP attribute using your text editor, for example:

```
/FIELD=(last_name, POSITION=2, SEPARATOR=' s' , utf16)
```

where *s* is the ASCII-compatible separator character typed using the text editor.

For the Unicode data types UTF16 or UTF32, you must include the % character (see Table 7 *on page 157* for conversion specifiers), the encoding type, and the UTF8 equivalent of the character within single quotes, for example:

```
/FIELD=(lname, POSITION=2, SEPARATOR=%utf16" , " , utf16)
```

or

```
/FIELD=(lname, POSITION=2, SEPARATOR=%utf32" , " , utf32)
```



A multi-byte separator character is always required for records containing fields of data type UTF16 or UTF32.

See Table 28 *on page 585* for a list of **sortcl**-supported multi-byte character data types.

Different Separators

sortcl allows the use of multiple separators within the same record definition. The fields delimited by one separator are independent of the fields delimited by another.

Example: Using Different Separators

The **Produce** file, used below, has two different separator characters:

- two fields delimited by a comma (,)
- three fields delimited by a pipe (|)

as shown here:

```
Apples|Chicago, IL|Red
Pears|Philadelphia, PA|Yellow
Peaches|Macon, GA|Peach
```

The following script file maps two fields to the output when there are two different separator characters defined in the input:

```
/INFILE=Produce
/FIELD=(Fruit, POSITION=1, SEPARATOR='|')
/FIELD=(Address, POSITION=2, SEPARATOR='|')
/FIELD=(State, POSITION=2, SEPARATOR=',', SIZE=2)
/FIELD=(Color, POSITION=3, SEPARATOR='|')
/SORT
/KEY=State
/OUTFILE=stdout
/FIELD=(State, POSITION=1)
/FIELD=(Fruit, POSITION=5)
```

The Address field is given as `POSITION=2` because it is the second field in the record with respect to the pipe (|) separator. The State field is also specified at `POSITION=2` because it is the second field in the record with respect to the comma (,) separator. (See *Multi-Byte Character Types on page 127* for the exception to this convention.). The `SIZE` attribute is necessary here to ensure that only the first two characters are considered as the State.

On execution, the screen displays:

```
GA  Peaches
IL  Apples
PA  Pears
```

11.5 SIZE

This statement sets the width, in bytes, of a given field. The syntax is:

```
SIZE=width, PRECISION=precision
```

For example, to specify the SKU field as 13 bytes long, the statement could be:

```
/FIELD= (SKU, POSITION=16, SIZE=13)
```

Numeric Precision

The `SIZE=width.precision` option allows additional control of output data, for example, `SIZE=6.3`. When used with numeric data, `.precision` is used to indicate the number of digits required to the right of the decimal point, and `width` is the total length of the field, including the decimal point and precision decimal places.



When defining a size attribute you must ensure that the width is large enough to accommodate any sign (+ or -) and/or decimal point in addition to the values on both sides of the decimal.

You must use `SIZE=width, PRECISION=precision` in IRI Workbench. The `SIZE=width.precision` option continues to work in the command line tool.

For additional numeric formatting options, see Numeric Attributes *on page 109*.

To demonstrate the use of numeric precision given a six-character input field, *Table 1* shows output based on size and type.

Table 1: Output Based on Size/Type

Input	Size	Data Type=NUMERIC
123.45	8.2	_ _ 1 2 3 . 4 5
123.45	7.3	1 2 3 . 4 5 0
123.45	6.4	* * * * *
abcdef	6.4	0 . 0 0 0 0



precision can not exceed 18 bytes.

The * characters in the third row of *Table 1* signify that an overflow error has occurred in a numeric field. That is, the size specified is not sufficient to display the entire value of the field.

When working with delimited fields, it is recommended that you use the PRECISION option separately (see *Precision on page 100*) without a *SIZE* attribute, so that numeric field values of varying lengths need not conform to a fixed byte width.

Precision Limits of Numeric Fields

On platforms that support 80-bit extended precision floating point, the precision limit of numeric fields is 18 digits. On platforms that support 128-bit extended precision floating point, such as Solaris, the precision limit is 33 digits. For all other platforms, including Windows, the limit is 15 digits.

11.6 CHARS

This statement is used for fields in fixed format (non delimited) records containing data types with variable length character encoding, currently UTF8 only.

CHARS is used instead of SIZE when you need to specify the field width as a number of characters rather than the number of bytes. When CHARS is used, the POSITION value of all fields becomes a character count offset. The syntax is:

`CHARS=width`

For example, to specify the SKU field as 5 characters long, the statement could be:

`/FIELD= (SKU, POSITION=16, CHARS=5, TYPE=UTF8)`

In one record SKU might contain “AEIOU” consuming 5 bytes, while in another record SKU might contain “ÃÊÏÓ” consuming 10 bytes.

Output fields may be remapped, but field positions in the output section of a script must be in ascending order.

11.7 Precision

To assign a uniform precision to numeric output values, without fixing the field size, use the PRECISION option. Using PRECISION instead of the `SIZE=width [.precision]` convention (see *SIZE on page 99*) ensures that each

numeric value is displayed with the precision required. There will be neither wasted white space for smaller values, nor overflow for larger values. The syntax is:

```
PRECISION=n
```

where *n* is the numeric precision applied to a field.

Although supported for both fixed-length and variable-length (delimited) fields, use of PRECISION is recommended for variable-length output fields. When using the *width.precision* convention (see *SIZE on page 99*), all field widths will conform to the specified size, regardless of whether some values are shorter or longer than that width.

For example, given these input records:

```
Harris,135,abc
Jones,5323.65,abc
Walters,923.0,abc
```

Define the numeric field on output with only a PRECISION attribute:

```
/FIELD=(value, POSITION=2, PRECISION=2, SEPARATOR=' ', ' , NUMERIC)
```

and the following is returned:

```
Harris,135.00,abc
Jones,5323.65,abc
Walters,923.00,abc
```

Note that if you use the SIZE attribute instead, for example

/FIELD=(value, POSITION=2, SIZE=7.2, SEPARATOR=' ', ' , NUMERIC), the output is:

```
Harris, 135.00,abc
Jones,5323.65,abc
Walters, 923.00,abc
```

The numeric field has a width of 7 in each case, regardless of each input value's width (displaying unnecessary white space).

11.8 FRAME

Use **sortcl**'s FRAME option to identify one or more fields. Typically, framed fields are found in RDBMS table or Microsoft CSV data. Framing is useful in the following cases:

- when field contents contain a character that is also used as a delimiter, and you want to prevent **sortcl** from treating such characters as field separators
- when you want to evaluate only the contents of the framed field, and ignore the frame character that encloses it
- when you want to add, or change, a frame character.

The **sortcl** frame field attribute syntax is:

```
FRAME=' char'
```

where *char* is the character that is used to frame the field.

Example: Using FRAME

Consider the following data:

```
Grapes, "Washington D.C.", Black  
Apples, "Chicago, IL", Red  
Pears, "Philadelphia, PA", Yellow
```

This script, **frame.scl**, can be used to sort these records by the framed field (Address):

```
/INFILE=Produce  
/FIELD=(Fruit, POSITION=1, SEPARATOR=', ')  
/FIELD=(Address, POSITION=2, SEPARATOR=', ', FRAME='"')  
/FIELD=(Color, POSITION=3, SEPARATOR=', ')  
/SORT  
/KEY=Address  
/OUTFILE=stdout  
/FIELD=(Address) # frame character removed
```

Note how the Color field is located at position 3 with respect to the separator. In the records where the address contains a comma (,) the field layout positions would be incorrect without using the **FRAME** attribute to define the Address field. On execution, the screen displays:

```
Chicago, IL  
Philadelphia, PA  
Washington D.C.
```

Note how the frame character (") was ignored for the purposes of sorting, and the frame character was removed on output because it was not specified in the output section of the script. The " or a different frame character could have been specified for the output.



It is recommended that you declare all framed fields. Otherwise, fields that can be referenced in your job that occur after undeclared framed fields might not be in their intended positions because any delimiter characters existing in the undeclared frame fields will be incorrectly treated as field separators.

11.9 Alignment

This field attribute is used for ASCII fields only. When used in the output or /INREC section of a script (see /INREC on page 150), it aligns a desired field string (not its leading or trailing fill characters) to either the left or right of the field. Leading or trailing spaces are moved to the opposite side of the string. The following alignment options are accepted:

- NONE_ALIGN** No change (the default, not required).
- LEFT_ALIGN** The string beginning with the first desired (non-space) character is aligned to the left of the target field. The remaining length to the right of the target field is populated with spaces.
- RIGHT_ALIGN** Spaces to the right of the source string are removed. The remaining source string is moved to the right side of the target field. The remaining length to the left of the target field is populated with spaces.



Use **LEFT_ALIGN**, **NONE_ALIGN**, and **RIGHT_ALIGN** in the /INFILE section of a script to define source ASCII fields. **LEFT_ALIGN** is the default.

Example: Using **RIGHT_ALIGN** on Output

To right-align the president's name field from the following input file:

```
McKinley William    1897-1901
Roosevelt Theodore 1901-1909
Taft William H.    1909-1913
Wilson Woodrow     1913-1921
Harding Warren G.   1921-1923
Coolidge Calvin     1923-1929
```

use a script like **right_align.scl**:

```
/INFILE=chiefs_some
  /FIELD= (pres, POSITION=1, SIZE=19)
  /FIELD= (term, POSITION=20, SIZE=9)
/REPORT
/OUTFILE=chiefs_some_right
  /FIELD= (pres, POSITION=1, SIZE=19, RIGHT_ALIGN)
  /FIELD= (term, POSITION=21, SIZE=9)
```

The output is as follows:

```
McKinley William 1897-1901
Roosevelt Theodore 1901-1909
Taft William H. 1909-1913
Wilson Woodrow 1913-1921
Harding Warren G. 1921-1923
Coolidge Calvin 1923-1929
```

As shown above, the name string is right-aligned. The spaces existing within the strings have not been affected by the alignment.

Padding and Reducing

If you apply a left or right align attribute to the target field (output or /INREC), and assign it a length greater than on input, extra fill characters are inserted on the opposite side as required (see *FILL on page 106*). Conversely, fill characters are removed from the opposite side when a lesser length is given on output. If no length is specified for the target field, and output fields are delimited, then any fill characters are removed (not added to the opposite side). See *Trimming* below.



When using an alignment option within a /KEY statement, the field is considered aligned only for the purpose of comparison, and no reformatting is performed (see *ASCII Options on page 166*).

Trimming

If your output fields are delimited, trim all leading or trailing fill characters from a fixed- or variable-length input field using `LEFT_ALIGN` or `RIGHT_ALIGN`, and not specifying a length for the target field. In this case, any fill characters are trimmed and the non-fill characters are preserved.

Additional trim functionality is automatically included via the **libcsutil** library in the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). The syntax is:

`trim(field)` – Removes the leading and trailing white space.
`trim_left(field)` – Removes the leading white space.
`trim_right(field)` – Removes the trailing white space.

Trim from both the left and right side of a field using `/INREC` and no-length, delimited output fields, as shown in the following example.

Example: Using LEFT_ALIGN and RIGHT_ALIGN

Consider the following fixed-length input data:

McKinley	1897-1901
Roosevelt	1901-1909
Taft	1909-1913
Wilson	1913-1921
Harding	1921-1923
Coolidge	1923-1929

Use the following script, **trim.scl**, to trim spaces from both the left and right side of the president's name field:

```
/INFILE=early_20th_chiefs
/FIELD=(pres, POSITION=1, SIZE=17)
/FIELD=(term, POSITION=18, SIZE=9)
/INREC
/FIELD=(pres, POSITION=1, SEPARATOR=',', LEFT_ALIGN) # trim leading
/FIELD=(term, POSITION=2, SEPARATOR=',')
/SORT
/KEY=pres
/OUTFILE=trimmed_early_20th_chiefs
/FIELD=(pres, POSITION=1, SEPARATOR=',', RIGHT_ALIGN) # trim trailing
/FIELD=(term, POSITION=2, SEPARATOR=',')
```

The leading spaces are trimmed as part of `/INREC`, and the trailing spaces are removed on output. The output is as follows:

Coolidge,1923-1929
Harding,1921-1923
McKinley,1897-1901
Roosevelt,1901-1909
Taft,1909-1913
Wilson,1913-1921

11.10 MILL

This statement in a numeric output field causes commas to be inserted at the appropriate places in a string of decimal digits. `MILL` is implied when `CURRENCY` or `MONEY` is used as a data type. Depending on the locale setting, the character inserted might not be a comma (see */LOCALE on page 281*)

Notice the effect of the `MILL` option in these examples:

```
/FIELD= (Value1, POSITION=01, SIZE=07, NUMERIC)
/FIELD= (Value1, POSITION=10, SIZE=07, MILL, NUMERIC)
/FIELD= (Value1, POSITION=19, SIZE=10, CURRENCY)
```

If the input field `Value1` is 12345, the display would be:

1	2	3	4	5						1	2	,	3	4	5						\$	1	2	,	3	4	5	.	0	0
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8			
0										1										2										



Be sure to increase the size of the field to accommodate the insertion of commas caused by the `MILL` statement; otherwise, an overflow can occur (see *SIZE on page 99*).

11.11 FILL

This statement is used to pad ASCII and (non-binary) numeric field statements. On input, it defines a pad character that must appear to the left of the field value. On output or `/INREC` (see */INREC on page 150*), it pads the left of the field with the character that you specify. If no `FILL` statement is used, the default fill character is a space.

Note that the default fill character for a non-binary field is a space; for a binary field, it is a binary `NULL`.

There are two forms of the `FILL` statement:

- `FILL= 'char'`
- `FILL= n`

where *char* is the fill character, and where *n* is the decimal weight of a character (see *ASCII COLLATING SEQUENCE on page 654* for equivalents).

Notice the effect of the FILL statement, when used on output, in these examples:

```
/FIELD=(Value, POSITION=01, SIZE=07, NUMERIC)
/FIELD=(Value, POSITION=10, SIZE=07, FILL='0', NUMERIC)
/FIELD=(Value, POSITION=18, SIZE=11, FILL='*', CURRENCY)
```

If the input field Value is 12345, the display would be:

```
      1 2 3 4 5      0 0 1 2 3 4 5  $ * 1 2 , 3 4 5 . 0 0
| | | | | | | | | | | | | | | | | | | | | | | | | |
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
0                      1                      2
```

11.12 Null Assignment

Most database management systems allow a special *null* character to describe a field value that has not yet been entered. In **sortcl**, that character can be assigned as an input field attribute in the following manner:

```
NULL_LOW='character' or NULL_HIGH='character'
```

where any field with the NULL_LOW character will have the lowest value in the collating sequence and any field with the NULL_HIGH character will have the highest value in the collating sequence. You can also use NULL to mean NULL_LOW.

NULL_LOW and NULL_HIGH can also be referenced in IF-THEN-ELSE logic (see CONDITIONAL FIELD AND DATA STATEMENTS *on page 180*).

Example: Using Null Assignments

This example uses NULL_LOW and NULL_HIGH in fields that are used as keys, and uses these fields for comparison in IF-THEN-ELSE logic:

-ashington, George	1789-1797	FED	VA
Adams, John	1797-1801	+ED	MA
Jefferson, Thomas	1801-1809	D-R	VA
-adison, James	1809-1817	+ -R	VA
Monroe, James	1817-1825	D-R	VA
-dams, John Quincy	1825-1829	D-R	MA
-ackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Harrison, William Henry	1841-1841	+HG	VA
Tyler, John	1841-1845	WHG	VA
-olk, James K.	1845-1849	DEM	NC
Taylor, Zachary	1849-1850	WHG	VA

Fillmore, Millard	1850-1853	WHG	NY
-ierce, Franklin	1853-1857	+EM	NH
Buchanan, James	1857-1861	DEM	PA
Lincoln, Abraham	1861-1865	REP	KY
Johnson, Andrew	1865-1869	REP	NC

The job script, **null.scl**, is as follows:

```
/INFILE=chiefs_null
  /FIELD=(pres,POSITION=1,SIZE=27,NULL_LOW='-')
  /FIELD=(party,POSITION=40,SIZE=3,NULL_HIGH='+')
/SORT
  /KEY=pres          # inherits NULL high/low
  /KEY=party         # status from declaration
/OUTFILE=out
  /FIELD=(pres,POSITION=1,SIZE=27)
  /DATA=(IF pres EQ NULL_LOW THEN " LOW")
  /FIELD=(party,POSITION=40,SIZE=3)
  /DATA=(IF party EQ NULL_HIGH THEN "  HIGH")
```

The resulting output file is:

-dams, John Quincy	LOW	D-R
-olk, James K.	LOW	DEM
-ackson, Andrew	LOW	DEM
-ashington, George	LOW	FED
-adison, James	LOW	+ -R HIGH
-ierce, Franklin	LOW	+EM HIGH
Adams, John		+ED HIGH
Buchanan, James		DEM
Fillmore, Millard		WHG
Harrison, William Henry		+HG HIGH
Jefferson, Thomas		D-R
Johnson, Andrew		REP
Lincoln, Abraham		REP
Monroe, James		D-R
Taylor, Zachary		WHG
Tyler, John		WHG
Van Buren, Martin		DEM

11.13 Numeric Attributes

When a field data type is numeric, specify the following additional formatting attributes:

SIGN_CONTROL=*value*

Applies to the data type **NUMERIC** in **/INREC** and **/OUTFILE** sections. Determines the display format of positive and negative values. The supported values are:

STANDARD	Default. A leading negative sign is displayed when applicable.
LEADING	A leading positive or negative sign is displayed, i.e., directly before the value.
TRAILING	A trailing positive or negative sign is displayed, i.e., directly after the value.
SURROUND	Encloses any negative values in parentheses.

MIN_DIGITS=*n* Applies to **NUMERIC** data in **/INREC** and **/OUTFILE** sections. Determines the number of digits to be displayed before the decimal point. *n* is the number of digits preceding the decimal point. The default for *n* is 0. For example, if you have an input value of 3.1, and you specify **MIN_DIGITS**=3, the converted value will be displayed as 003.1.

For example, consider an input field value of 3. With the following **/FIELD** statement on output:

```
/FIELD= (num1, POSITION=1, SIZE=7, PRECISION=1, SIGN_CONTROL=LEADING, MIN_DIGITS=3, NUMERIC)
```

the output value will display as +003.1. That is, the **SIGN_CONTROL=LEADING** attribute places the + sign in front of the value, and the **MIN_DIGITS=3** attribute ensures the display of three digits before the decimal point.

PLUS_CHAR=*char* Determines the character used to indicate positive values in **/INREC** and **/OUTFILE** sections. *char* is any single character. + is the default.

MINUS_CHAR=*char* Determines the character used to indicate negative values in **/INREC** and **/OUTFILE** sections. *char* is any single character. - is the default.

EXPONENT=*n* Multiplies the field value by 10^n . The syntax is **EXPONENT**=+*n* or -*n*, where *n* is the exponent value. + is the default. The exponent is applied to the numeric value of the data in the field. Can be used in

/INFILE, /INREC, and /OUTFILE sections. See Example: *on page 110*.

IMPLIED_DECIMAL=*n*

Intended for use with numeric values to define a hidden decimal point. The numeric value of the input data is restored by multiplying the input value by 10^{-n} . The decimal point is removed on output by multiplying the value by 10^n . This is similar to the exponent except for the exponent sign reversal. Can be used in /INFILE, /INREC, and /OUTFILE sections. See Example: *on page 112*.

Example: Using the EXPONENT Attribute

Given an input file **sampleNumeric.dat** that contains the value 123456789.12345, the following script, **exponent.scl**, demonstrates various applications of the EXPONENT attribute:

```
/INFILE=sampleNumeric.dat
  /FIELD=(f1,POSITION=1,Size=15,NUMERIC) # unmodified input
  /FIELD=(f2,POSITION=1,Size=15,NUMERIC,EXPONENT=1)
  /FIELD=(f3,POSITION=1,Size=15,NUMERIC,EXPONENT=-1)
/REPORT
/OUTFILE=out1 # exponent (+-3) applied to output with default precision (2)
  /FIELD=(f1,POSITION=1,SIZE=15,NUMERIC) # unmodified
  /FIELD=(f1,POSITION=18,SIZE=15,EXPONENT=3,NUMERIC)
  /FIELD=(f1,POSITION=35,SIZE=15,EXPONENT=-3,NUMERIC)
/OUTFILE=out2 # exponent (+-3) applied to output with precision (4)
  /FIELD=(f1,POSITION=1,SIZE=15,PRECISION=4,NUMERIC) # unmodified
  /FIELD=(f1,POSITION=18,SIZE=17,PRECISION=4,EXPONENT=3,NUMERIC)
  /FIELD=(f1,POSITION=37,SIZE=15,PRECISION=4,EXPONENT=-3,NUMERIC)
/OUTFILE=out3 # exponent (+-1) applied to input with default precision (2)
  /FIELD=(f1,POSITION=1,SIZE=15,NUMERIC) # unmodified
  /FIELD=(f2,POSITION=18,SIZE=15,NUMERIC)
  /FIELD=(f3,POSITION=35,SIZE=15,NUMERIC)
/OUTFILE=out4 # exponent (-1) applied to both input and output with
               # default precision (2)
  /FIELD=(f1,POSITION=1,SIZE=15,NUMERIC) # unmodified
  /FIELD=(f2,POSITION=18,SIZE=15,EXPONENT=-1,NUMERIC) # modified on input only
  /FIELD=(f3,POSITION=35,SIZE=16,EXPONENT=-1,NUMERIC) # modified in input and out
```

With X representing every 18th byte, **out1** values are displayed as:

```

      unmodified      exponent=3      exponent=-3
123456789.12345  123456789123.45    123456.79
-----X-----X-----X

```

out2:

```

      unmodified      exponent=3      exponent=-3
123456789.1235  123456789123.4500  123456.7891
-----X-----X-----X

```

out3:

```

      unmodified      exponent=1      exponent=-1
123456789.12345  1234567891.23    12345678.91
-----X-----X-----X

```

out4:

```

      unmodified      exponent=1      exponent=-1
123456789.12345  123456789.12    1234567.89
-----X-----X-----X

```

Example: Using the IMPLIED_DECIMAL Attribute

Given an input file **sampleImplied.dat** that contains the value 123456789, the following script, **implied.scl**, demonstrates various applications of the IMPLIED_DECIMAL attribute:

```
/INFILE=sampleImplied.dat
  /FIELD= (f1, POSITION=1, Size=15, NUMERIC) # unmodified input
  /FIELD= (f2, POSITION=1, Size=15, NUMERIC, IMPLIED_DECIMAL=2)
  /FIELD= (f3, POSITION=1, Size=15, NUMERIC, IMPLIED_DECIMAL=-2)
  /FIELD= (f4, POSITION=1, Size=15, NUMERIC, IMPLIED_DECIMAL=0)
/REPORT
/OUTFILE=out1      # implied decimal only on input
  /FIELD= (f1, POSITION=1, SIZE=18, NUMERIC) # unmodified
  /FIELD= (f2, POSITION=19, SIZE=18, NUMERIC)
  /FIELD= (f3, POSITION=38, SIZE=18, NUMERIC)
  /FIELD= (f4, POSITION=57, SIZE=18, NUMERIC)
/OUTFILE=out2      # implied decimal only on output
  /FIELD= (f1, POSITION=1, SIZE=18, NUMERIC) # unmodified
  /FIELD= (f1, POSITION=19, SIZE=18, IMPLIED_DECIMAL=2, NUMERIC)
  /FIELD= (f1, POSITION=38, SIZE=18, IMPLIED_DECIMAL=-2, NUMERIC)
  /FIELD= (f1, POSITION=57, SIZE=18, IMPLIED_DECIMAL=0, NUMERIC)
/OUTFILE=out3      # implied decimal on both input and output
  /FIELD= (f1, POSITION=1, SIZE=18, NUMERIC) # unmodified
  /FIELD= (f2, POSITION=19, SIZE=18, IMPLIED_DECIMAL=2, NUMERIC)
  /FIELD= (f3, POSITION=38, SIZE=18, IMPLIED_DECIMAL=-2, NUMERIC)
  /FIELD= (f4, POSITION=57, SIZE=18, IMPLIED_DECIMAL=0, NUMERIC)
```

With X representing every 18th byte, **out1** values are displayed as:

unmodified	(in)implied=2	(in)implied=-2	(in)implied=0
123456789.00	1234567.89	12345678900.00	123456789.00
-----X-----	-----X-----	-----X-----	-----X-----

out2:

unmodified	(out)implied=2	(out)implied=-2	(out)implied=0
123456789.00	12345678900	1234568	123456789
-----X-----	-----X-----	-----X-----	-----X-----

out3:

unmodified	(io)implied=2	(io)implied=-2	(io)implied=0
123456789.00	123456789	123456789	123456789
-----X-----	-----X-----	-----X-----	-----X-----

11.14 Composite Fields and Templates

A TEMPLATE is used to define the structure of a composite field and is composed of field symbols and literals. The field symbols contain format symbols and characters that further define the format symbols. These format symbols can be used to convert from one compatible symbol to another, such as from a 4-digit year to a 2-digit year. They can also be used for mapping. The literals show other characters that appear in the field. Each TEMPLATE is named and must be defined before being used in the field statement of a composite field.

11.14.1 Syntax

The syntax for TEMPLATE is:

```
/TEMPLATE=(<template_name>="<field symbols and literals>")
```

where field symbols are in the form % [x.y] S

- % indicates the start of a field symbol that contains a format symbol
- x controls the width of the value defined by the format symbol
- y controls the precision of the value defined by the format symbol
- S one of the format symbols

A composite field has the same syntax as a regular field but with an additional parameter called STRUCTURE. The STRUCTURE invokes a template name and gives sub-field the names to the parts of the TEMPLATE that have format symbols.

The syntax for a composite field is:

```
/FIELD=(<field_name>, POSITION=<n>, ... , STRUCTURE=<template_name>(sub_field1, subfield2, ...))
```

11.14.2 Symbols

The characters in a template are either literals or field symbols. Literals are copied directly into the output. Field symbols, that begin with '%', are paired with values in the value list. For additional information, search for `strftime` on the Internet.

The symbols below follow the % in the template to indicate that a field will be formatted.

A. Date/time fields

	Sample (US) or Range	Meaning
%a	Mon	locale's weekday name abbreviated
%A	Monday	locale's full weekday name
%u	[1,7]	day of week (1: Monday)
%b	Nov	locale's month name abbreviated
%B	November	locale's full month name
%m	[01,12]	month as a decimal (01:January)
%d	[01,31]	day of the month as a decimal
%e	[1,31]	day of the month as a decimal
%H	[00,23]	hour (24-hour clock)
%I	[01,12]	hour (12-hour clock)
%k	[0,23]	hour (24-hour) - no 0 fill (check with SS)
%l	[1,12]	hour (12-hour) - no 0 fill (check with SS)
%j	[001,366]	day of the year
%M	[00,59]	minute
%p	[AM,PM]	AM or PM (upper case)
%P	[am,pm]	am or pm (lower case)
%S	[00,60]	second
%W	[0,53]	week of year
%y	[0,99]	last two digits of the year
%Y	[0000,9999]	year as a decimal number
%Z	UTC	Time zone

B. Non-time date-related symbols

<code>%[0][x]i</code>	An integer field of length x. If no x, the field will be long enough to accommodate the value. If a 0 is given, the field will be as long as an x and be 0 filled.
<code>%[0][x.y]f</code>	A floating field of length x and decimal portion y. Same 0 rules.
<code>%[x.y]s</code>	A string field of length x shifted by y. If [x.y] not given, the field is as long as the internal length.

C. Exceptions

The exceptions to the above allow the insertion of a control character to the output:

Entry	causes the insertion of the
<code>%%</code>	% character
<code>%t</code>	tab character
<code>%n</code>	line feed character

11.14.3 Mapping Composites

Mapping can occur from a:

- subfield of a composite field to a subfield of the same name using a different TEMPLATE for the composite field
- subfield of a composite field to a regular field
- regular field to a subfield of a composite field

Example: Subfield to a subfield of the same name using different TEMPLATE

This example maps a subfield of a composite field to a subfield of the same name using a different TEMPLATE for the field.

The input file is **in1.dat**, and the script is **composite_map1.scl**.

```
01/29/2013 |aaaa
12/02/2001 |bb
```

```
/TEMPLATE=(m_first="%m/%d/%Y")
/TEMPLATE=(y_first="%Y-%m-%d")
/INFILE=in1.dat
  /FIELD=(comp_date1,POSITION=1,SEPARATOR='|',STRUCTURE=m_first(month,day,year))
  /FIELD=(f2,POSITION=2,SEPARATOR='|')
/REPORT
/OUTFILE=composite_map1.out
  /FIELD=(f2,POSITION=1,SEPARATOR='|')
  /FIELD=(comp_date1,POSITION=2,SEPARATOR='|',STRUCTURE=y_first(year,month,day))
```

The resulting output is:

```
aaaa|2013-01-29
bb|2001-12-02
```

Example: Subfield to a regular field

This example maps a subfield of a composite field to a regular field.

The input file is **in1.dat**, as in the previous example, and the script is **composite_map2.scl**.

```
/TEMPLATE=(m_first="%m/%d/%Y")
/INFILE=in.dat
  /FIELD=(comp_date1,POSITION=1,SEPARATOR='|',STRUCTURE=m_first(month,day,year))
  /FIELD=(f2,POSITION=2,SEPARATOR='|')
/REPORT
/OUTFILE=composite_map2.out
  /FIELD=(f2,POSITION=1,SEPARATOR='|')
  /FIELD=(year,POSITION=2,SEPARATOR='|')
```

The resulting output is:

```
aaaa|2013
bb|2001
```

Example: Regular field to a subfield

This example maps a regular field to a subfield of a composite field.

The input file is **in2.dat**, and the script is **composite_map3.scl**.

```
2013|aaaa|29|aa|01
2001|bb|02|bbb|12
```

```

/TEMPLATE=(m_first="%m/%d/%Y")
/INFILE=in2.dat
  /FIELD=(year, POSITION=1, SEPARATOR='|')
  /FIELD=(f1, POSITION=2, SEPARATOR='|')
  /FIELD=(day, POSITION=3, SEPARATOR='|')
  /FIELD=(f2, POSITION=4, SEPARATOR='|')
  /FIELD=(month, POSITION=5, SEPARATOR='|')
  /FIELD=(f3, POSITION=6, SEPARATOR='|')
/REPORT
/OUTFILE=composite_map3.out
  /FIELD=(cdate, POSITION=1, SEPARATOR='|', STRUCTURE=m_first(month, day, year))
  /FIELD=(f1, POSITION=2, SEPARATOR='|')

```

The resulting output is:

```

01/29/2013|aaaa
12/02/2001|bb

```

Example: Mapping regular fields to a composite field

This example maps regular fields to a composite field using different non-time date-related symbols and utilizes size, precision, and zero fill for numeric examples.

The input file is **inventory.dat**, and the script is **composite_map4.scl**.

```

, , 23456, Hat, Red, 14.95, ,
, , 1345, Gloves, Gray, 12.02, ,
, , 734755, Scarf, White, 125.45 , ,

```

```

/TEMPLATE=(Inventory="%5s # %6i %4s \ $%06.2f")
/INFILE=inventory.dat
  /FIELD=(SKU, POSITION=3, SEPARATOR=',')
  /FIELD=(Item, POSITION=4, SEPARATOR=',')
  /FIELD=(Color, POSITION=5, SEPARATOR=',')
  /FIELD=(Price, POSITION=6, SEPARATOR=',')
/REPORT
/OUTFILE=inventory.out
  /FIELD=(Sale, POSITION=1, SEPARATOR=',', STRUCTURE=Inventory(Item, SKU, Color, Price))

```

The resulting output is:

```

Hat # 23456 Red $014.95
Glove # 1345 Gray $012.02
Scarf #734755 Whit $125.45

```

To sort a composite field based on certain subfields from the TEMPLATE, create an INREC where the subfields are mapped to fields, and then those fields are used for the keys.

11.15 RANDOM

If you need a field with random data within a regular record, use the RANDOM attribute on the field definition so that you can generate random values for that field.

This attribute must be applied to fields that have been defined as part of an input file. The actual data from the input is ignored when the RANDOM attribute is used. A new field name cannot be defined, the RANDOM attribute must be applied to an existing field from the input records.

The syntax for the RANDOM field attribute is:

```
/FIELD=(input_field_name, POSITION=1, SIZE=4, RANDOM,  
MINSIZE=1, MAXSIZE=4, TYPE=ASCII)
```

Example: Random Field Attribute

Consider the following input file, **chiefs_1900**:

Roosevelt, Theodore	1901-1909	REP	NY
Taft, William H.	1909-1913	REP	OH
Harding, Warren G.	1921-1923	REP	OH
Coolidge, Calvin	1923-1929	REP	VT
Hoover, Herbert C.	1929-1933	REP	IA
Roosevelt, Franklin D.	1933-1945	DEM	NY
Eisenhower, Dwight D.	1953-1961	REP	TX
Kennedy, John F.	1961-1963	DEM	MA
Johnson, Lyndon B.	1963-1969	DEM	TX
Nixon, Richard M.	1969-1973	REP	CA
Ford, Gerald R.	1973-1977	REP	NE
Carter, James E.	1977-1981	DEM	GA
Reagan, Ronald W.	1981-1989	REP	IL
Bush, George H.W.	1989-1993	REP	TX
Clinton, William J.	1993-2001	DEM	AR

Use the following script, **random.scl**, to convert the term field to random 1- to 3-digit numbers, the party field to random 3-character uppercase values, and the state field to random 2-character lowercase values.

```
/INFILE=chiefs_1900
/FIELD= (name, POSITION=1, SIZE=27)
/FIELD= (term, POSITION=28, SIZE=4)
/FIELD= (party, POSITION=40, SIZE=3)
/FIELD= (state, POSITION=45, SIZE=2)
/REPORT
/OUTFILE=randomout
/FIELD= (name, POSITION=1, SIZE=27)
/FIELD= (term, POSITION=28, SIZE=4, RANDOM, MIN_SIZE=1, MAX_SIZE=3, DIGIT)
/FIELD= (party, POSITION=40, SIZE=3, RANDOM, uppercase)
/FIELD= (state, POSITION=45, SIZE=2, RANDOM, lowercase)
```

The resulting outfile is:

Roosevelt, Theodore	3	HJW	se
Taft, William H.	301	RYC	lv
Harding, Warren G.	836	SYF	rk
Coolidge, Calvin	495	KAX	ke
Hoover, Herbert C.	15	QNJ	kq
Roosevelt, Franklin D.	1	MZX	rx
Eisenhower, Dwight D.	7	OSS	bl
Kennedy, John F.	1	IHO	vb
Johnson, Lyndon B.	2	MLH	cw
Nixon, Richard M.	31	JWZ	pg
Ford, Gerald R.	70	KES	ej
Carter, James E.	613	ATO	vw
Reagan, Ronald W.	4	NSK	qa
Bush, George H.W.	106	PEW	vh
Clinton, William J.	538	EOH	tq

11.16 FIELD ENDIANNESS

In any section of a **sortcl** job script, add an attribute to any field to specify its endianness, either to describe the source data values (at the /INFILE or /INREC level) or to define the endianness to be used for the target (when used in an /OUTFILE section). This option applies only to fields of a data type where endianness varies on a machine-specific basis.

The syntax is:

ENDIAN=*option*

where *option* can be LITTLE or BIG.



When using /PROCESS=VS, apply a specific endianness to all applicable fields in the file (see VARIABLE_SEQUENTIAL (or VS) *on page 57*). The field-level ENDIAN=*option* attribute will override any file-level settings. The default for both file- and field-level endianness is the endianness used on the machine on which **CoSort** is running.

The script, **int_2_bin.scl**, converts from an ascii-numeric source field to a binary integer target field that is little-endian:

```
/INFILE=vals
  /FIELD= (VALUE, POSITION=1, SIZE=8, PRECISION=0, NUMERIC)
/REPORT
/OUTFILE=bin_out
  /LENGTH=4
  /FIELD= (VALUE, POSITION=1, SIZE=4, TYPE=INTEGER, ENDIAN=LITTLE)
```

The next script, **bin_2_bin.scl**, uses the output from the previous example (**bin_out**), and converts the little-endian integer field to big-endian.

```
/INFILE=bin_out
  /LENGTH=4
  /FIELD= (VALUE_LITTLE, POSITION=1, SIZE=4, INTEGER, ENDIAN=LITTLE)
/REPORT
/OUTFILE=BIN_BIN_OUT
  /LENGTH=4
  /FIELD= (VALUE_LITTLE, POSITION=1, SIZE=4, INTEGER, ENDIAN=BIG)
```


Sorting Based on Endianness

Arithmetic, multi-byte characters comparisons, sorting, etc. can be performed on data with a different endianness. The following example demonstrates the sorting of big-endian data on a little-endian machine:

```
/INFILE=LE_integers
/LENGTH=16
/FIELD=(record_nb, POSITION=1, SIZE=4)
/FIELD=(value_asc, POSITION=5, SIZE=6)
/FIELD=(value_ile, POSITION=13, SIZE=4, INTEGER, ENDIAN=LITTLE)
/SORT
/KEY = value_ile
/OUTFILE=BE_integers_sorted
/LENGTH=16
/FIELD=(record_nb, POSITION=1, SIZE=4)
/FIELD=(value_asc, POSITION=5, SIZE=6)
/FIELD=(value_ile, POSITION=13, SIZE=4, INTEGER, ENDIAN=BIG)
```

In the above example, data from the field `value_ile`, the source data for which is big-endian, is sorted in little-endian order, as per the machine requirements. This was achieved by declaring it `LITTLE`-endian in the input section. The field data is then converted back to its original form, `BIG`-endian, in the output section.

11.17 Data Types (Single-Byte)

It is important to declare the data type of a field so that **sortcl** can interpret the data correctly for comparisons, conversions, and/or display purposes. For a description of all data types recognized in the **CoSort** suite, see *DATA TYPES on page 583*. If the data type is not given for an input or output field, the field is assumed to be ASCII. A key field is assumed to have the same data type that was declared in the input.



See Multi-Byte Character Types *on page 127* for details on using **sortcl**-supported multi-byte data types.

Data-Type Conversion

When mapping from input to output, **sortcl** can convert between data types, for example:

- numeric data to currency
- EBCDIC character to ASCII
- packed decimal values to binary integer.

sortcl also has the ability to convert multi-byte data types (see Native Form 6 data type - Chinese GBK Simplified Sort *on page 129*).



If you are converting ASCII-Numeric data to a packed numeric format such as MF_CMP3, and your data has decimal values that you need to preserve, use the following scripting convention, for example, in the INREC and output sections:

```
/INFILE=numbers.in
  /FIELD=(f1, POSITION=1, SEPARATOR='|', PRECISION=2, NUMERIC)
/INREC
  /FIELD=(newf1=f1 * 100, POSITION=1, SIZE=12, PRECISION=0, NUMERIC)
/REPORT
/OUTFILE=numbers_packed.out
  /FIELD=(newf1, POSITION=1, SIZE=6, MF_CMP3)
```

In this example, a new field was defined in INREC that multiplied the original numeric field value by 100, and uses a precision of 0.

In the output section, the decimal place of the original value is now preserved when converting the field to MF_CMP3.

If converting from packed to ASCII-Numeric, you must convert to ASCII-numeric in the INREC section, and then divide that value by 100 on output, for example:

```
/INFILE=packed.in
  /FIELD=(f1, POSITION=1, SIZE=6, MF_CMP3)
/INREC
  /FIELD=(f1, POSITION=1, SIZE=12, PRECISION=0, NUMERIC)
/REPORT
/OUTFILE=numbers.out
  /FIELD=(f1 / 100, POSITION=1, SEPARATOR='|', PRECISION=2, NUMERIC)
```

Note that the PRECISION attribute on output can be customized according to your requirements.

Example: Converting Data Types

Consider the following input file, **chicago**, which contains numeric data:

5180	On Top	15.95	Harper-Row
3391	Married Young	24.95	Prentice-Hall
8835	Beginnings	8.50	Prentice-Hall
2272	Still There	13.05	Dell
1139	Greater Than	34.75	Valley Kill
3928	Not On Call	9.99	Harper-Row
4877	Going Nowhere	17.95	Valley Kill

The following script, **currency.scl**, converts the Price field from NUMERIC to CURRENCY:

```
/INFILE=chicago
/FIELD= (StockNum, POSITION=1, SIZE=4)
/FIELD= (Title, POSITION=6, SIZE=15)
/FIELD= (Price, POSITION=21, SIZE=8, NUMERIC)
/FIELD= (Publisher, POSITION=30, SIZE=15)
/OUTFILE=currency.out
/FIELD= (StockNum, POSITION=1, SIZE=4)
/FIELD= (Title, POSITION=6, SIZE=15)
/FIELD= (Price, POSITION=21, SIZE=8, CURRENCY)
/FIELD= (Publisher, POSITION=30, SIZE=15)
```

currency.out shows the result of sorting and converting **chicago**:

1139	Greater Than	\$34.75	Valley Kill
2272	Still There	\$13.05	Dell
3391	Married Young	\$24.95	Prentice-Hall
3928	Not On Call	\$9.99	Harper-Row
4877	Going Nowhere	\$17.95	Valley Kill
5180	On Top	\$15.95	Harper-Row
8835	Beginnings	\$8.50	Prentice-Hall

sortcl-Specific Data Types

This section describes the data types that are unique to **sortcl**.

sortcl has the following ASCII-numeric data types:

- CURRENCY *on page 124*
- WHOLE_NUMBER *on page 124*
- IP_ADDRESS *on page 124*
- BIT *on page 125*.

CURRENCY

CURRENCY, by default, displays right-justified with a precision of two. MONEY and CURRENCY can be used interchangeably. They have MILL set to *on* and display the monetary symbol for the active locale at the beginning of the field (see MILL *on page 106* and /LOCALE *on page 281*).

The following are examples of how the ASCII-numeric data types display in the USA:

ASCII	Numeric	Currency	Currency w/Fill
3.2	3.20	\$3.20	\$*****3.20
150	150.00	\$150.00	\$*****150.00
3.25	3.25	\$3.25	\$*****3.25
9.4562	9.46	\$9.46	\$*****9.46
1023.45	1023.56	\$1,023.45	\$***1,023.45
29384.56	29384.56	\$29,384.56	\$**29,384.56

WHOLE_NUMBER

WHOLE_NUMBER is an ASCII-numeric data type that displays right-justified and only contains whole numbers.

IP_ADDRESS

IP addresses are represented in dotted-decimal IPIV notation. They contain four numbers, each ranging from 0 to 255, separated by dots, for example 111.123.124.215. When sorting, each subfield is compared numerically starting with the left-most subfield. Subsequent sub-fields are compared only if all previous sub-fields have been determined to be equal.

Example: Sorting by IP Address

The following input file contains company names and an IP Address associated with each.

Cratehouse	201.21.255.255
Wheels4U	10.1.2.34
Lens City	201.23.255.255
Utensil Co	201.23.45.255
Sentinel	192.213.5.245
Import Rite	201.23.55.255
Novels2Go	111.123.124.245

To sort by IP Address, use the following job script, **ip.scl**:

```
/INFILE=ip.dat
  /FIELD=(company, POSITION=1, SIZE=13)
  /FIELD=(ip, POSITION=14, SIZE=15, IP_ADDRESS)
/SORT
  /KEY=ip
/OUTFILE=ip.out
```

Output will be as follows:

Wheels4U	10.1.2.34
Novels2Go	111.123.124.245
Sentinel	192.213.5.245
Cratehouse	201.21.255.255
Utensil Co	201.23.45.255
Import Rite	201.23.55.255
Lens City	201.23.255.255

BIT

Use the **BIT** data type to return the bit representation in ASCII or EBCDIC of one or more input characters.

Example: Using the BIT Data Type

Consider the following input data, **bits.in**:

```
B
C
A
```

The **sortcl** script, **bits.scl**, can be as follows:

```
/INFILE=bits.in
  /FIELD=(letter, POSITION=1, SIZE=1, BIT)
/REPORT
/OUTFILE=stdout
  /FIELD=(letter, POSITION=1, SIZE=8, ASCII)
```

This produces the following results:

```
01000010
01000011
01000001
```

which are the ASCII bit representations of the ASCII characters B, C, and A, respectively. On output, you must be sure to resize the output field to eight times its input field size to display the complete pattern.



If the corresponding output field is given a size that is smaller than eight times the input field, the results are truncated. If the output field size is given a size that is greater than eight times the input field, the results are padded to the right with extra spaces.

Note that any field you declare as **BIT** cannot be used as a **/KEY**. However, to order on those field values, redefine another ASCII field with the same size and position, and use that as the **/KEY** field, for example:

```
/INFILE=test
  /FIELD=(letter, POSITION=1, SIZE=1, BIT)
  /FIELD=(ascii_letter, POSITION=1, SIZE=1, ASCII)
/SORT
  /KEY=ascii_letter
/OUTFILE=stdout
  /FIELD=(letter, POSITION=1, SIZE=8, ASCII)
```

11.18 Multi-Byte Character Types

sortcl 9.5.1 introduces multi-byte characters in Form 6. While Form 0 multi-byte characters are sorted by the encoding order, Form 6 sorts in traditional order for the language type.

sortcl allows you to process, collate, and convert between several multi-byte data types, including multiple Chinese, Korean, and Japanese collation standards. See Table 29 on page 587 and onward for a complete list of supported multi-byte data types.

sortcl can convert from any Form 6 digit data type to another Form 6 digit data type as shown in the table below. The single-byte ASCII numeric data types and UTF16 digits have numeric properties while the other data types are digit characters only.

Table 2: Conversion between native multi-byte digit types

Data Type	ASCII_DIGITS	UTF16_DIGITS	CHINESE_BIG5_DIGITS	CHINESE_GBK_DIGITS	JAPANESE_DIGITS	KOREAN_DIGITS
ASCII_DIGITS	N/A	Yes	Yes	Yes	Yes	Yes
ALPHA_DIGITS	N/A	Yes	Yes	Yes	Yes	Yes
ASCII_ALPHA_DIGIT	N/A	Yes	Yes	Yes	Yes	Yes
DIGITS	N/A	Yes	Yes	Yes	Yes	Yes
UTF16_DIGITS	Yes	N/A	Yes	Yes	Yes	Yes
CHINESE_BIG5_DIGITS	Yes	Yes	N/A	Yes	Yes	Yes
HK_DIGITS	Yes	Yes	N/A	Yes	Yes	Yes
MO_DIGITS	Yes	Yes	N/A	Yes	Yes	Yes
ROC_DIGITS	Yes	Yes	N/A	Yes	Yes	Yes
TW_DIGITS	Yes	Yes	N/A	Yes	Yes	Yes
CHINESE_GBK_DIGITS	Yes	Yes	Yes	N/A	Yes	Yes
PRC_DIGITS	Yes	Yes	Yes	N/A	Yes	Yes
SG_DIGITS	Yes	Yes	Yes	N/A	N/A	Yes
JAPANESE_DIGITS	Yes	Yes	Yes	Yes	N/A	Yes
JP_DIGITS	Yes	Yes	Yes	Yes	N/A	Yes
KOREAN_DIGITS	Yes	Yes	Yes	Yes	Yes	N/A
KR_DIGITS	Yes	Yes	Yes	Yes	Yes	N/A

A field containing non-Unicode multi-byte characters can also include single-byte characters.

When using delimited records, non-Unicode data types can use a single- or multi-byte separator, while Unicode requires a multi-byte separator. Refer to Multi-Byte Character Separators *on page 97*.

To specify conditions involving multi-byte data types, you must use the %HEX conversion specifier. See NON-ASCII FIELD EVALUATION AND CONVERSION SPECIFIERS *on page 156*.

For non-delimited fields, the position for multi-byte characters is increased by the number of bytes, so a single 2-byte character is counted as two positions.

When using Form 0 multi-byte character separators, a trailing separator is required on the final field.

When specifying more than one separator string for multi-byte data types within each record, you cannot use the same POSITION value for multiple fields as with single-byte data types (as shown in Different Separators *on page 98*). Instead, you must use unique POSITIONS for each subsequent field, whether the separator is the same or not.

For example, using the script from Different Separators *on page 98*, note how the POS values on input are unique when using multi-byte data types:

```
/INFILE=Produce
  /FIELD= (Fruit, POSITION=1, SEPARATOR= ' | ' , GBK)
  /FIELD= (Address, POSITION=2, SEPARATOR= ' | ' , GBK)
  /FIELD= (State, POSITION=3, SEPARATOR= ' , ' , GBK)
  /FIELD= (Color, POSITION=4, SEPARATOR= ' | ' , GBK)
/SORT
  /KEY=State
/OUTFILE=stdout
  /FIELD= (State, POSITION=1, SEPARATOR= ' | ' , GBK)
  /FIELD= (Fruit, POSITION=2, SEPARATOR= ' | ' , GBK)
```

In the script above, note that each POS value in the input section is not necessarily the field's position in the record with respect to the specified separator, but rather the value of POS is advanced with each newly defined field, regardless of whether the separator character changes (from | to, in this case).

Example: Native Form 6 data type - Chinese GBK Simplified Sort

Consider the following input data, **GBK_data1.txt**, which contains unsorted records with name, age, and employer fields:

```
刘伟, 67, 中国船舶重工集团公司第七〇五研究所
曹天才, 75, 华中科技大学
许军, 48, 福州大学
李雅帝, 66, 中国航天科工集团第三研究院
秦林, 41, 北京中星微电子有限公司
孙琳, 44, 华中科技大学
沈建英, 44, 总参谋部第六十一研究所
王鸿娜, 52, 福州大学
蔡莹, 46, 中国矿业大学
夏学林, 69, 中国金属学会
```

The following script, **gbk_simplified_sort.scl** sorts the records over the employer field:

```
/INFILE=GBK_data1.txt
/FIELD=(name, POSITION=1, SEPARATOR=' ', CHINESE_GBK_SIMPLIFIED)
/FIELD=(age, POSITION=2, SEPARATOR=' ', ASCII)
/FIELD=(employer, POSITION=3, SEPARATOR=' ', CHINESE_GBK_SIMPLIFIED)
/SORT
/KEY=employer
/OUTFILE=GBK_data1.sorted
```

This produces **GBK_data1.sorted**:

```
秦林, 41, 北京中星微电子有限公司
许军, 48, 福州大学
王鸿娜, 52, 福州大学
曹天才, 75, 华中科技大学
孙琳, 44, 华中科技大学
刘伟, 67, 中国船舶重工集团公司第七〇五研究所
李雅帝, 66, 中国航天科工集团第三研究院
夏学林, 69, 中国金属学会
蔡莹, 46, 中国矿业大学
沈建英, 44, 总参谋部第六十一研究所
```

Example: Native Form 6 data type - Chinese Big5 Sort

Consider the following input data, **BIG5_data1.txt**, which contains unsorted records with name, age, and employer fields:

林琪蓁, 45, 行政院衛生署新竹醫院
翁廷璋, 64, 國軍新竹地區醫院附設民眾診療服務處
胡家發, 46, 財團法人長庚紀念醫院林口分院
黃奕翊, 36, 大園敏盛醫院
胡耿嘉, 41, 桃新醫院
蔡宏偉, 55, 財團法人長庚紀念醫院林口分院
李玠霖, 57, 慈祐醫院
陳元暘, 49, 國軍新竹地區醫院附設民眾診療服務處
李昀晏, 41, 南門綜合醫院
林哲佑, 54, 天成醫院

The following script, **big5_sort.scl** sorts the records over the employer field:

```
/INFILE=BIG5_data1.txt
  /FIELD= (name, POSITION=1, SEPARATOR=', ', CHINESE_BIG5)
  /FIELD= (age, POSITION=2, SEPARATOR=', ', ASCII)
  /FIELD= (employer, POSITION=3, SEPARATOR=', ', CHINESE_BIG5)
/SORT
  /KEY=employer
/OUTFILE=BIG5_data1.sorted
```

This produces **BIG5_data1.sorted**:

黃奕翊, 36, 大園敏盛醫院
林哲佑, 54, 天成醫院
林琪蓁, 45, 行政院衛生署新竹醫院
李昀晏, 41, 南門綜合醫院
胡耿嘉, 41, 桃新醫院
蔡宏偉, 55, 財團法人長庚紀念醫院林口分院
胡家發, 46, 財團法人長庚紀念醫院林口分院
翁廷璋, 64, 國軍新竹地區醫院附設民眾診療服務處
陳元暘, 49, 國軍新竹地區醫院附設民眾診療服務處
李玠霖, 57, 慈祐醫院

Example: Native Form 6 data type - Korean Hangul sort

Consider the following input data, **KSC_data1.txt**, which contains unsorted records with name, age, and employer fields:

```
김대중,54,농협
이순신,47,삼성
이명박,45,현대
최순호,32,대우
이승엽,28,중소기업은행
장나라,33,삼성
김동건,38,농심
심형래,31,스리동
이기영,29,현대
박주영,30,빙그레
```

The following script, **hangul_sort.scl** sorts the records over the employer field:

```
/INFILE=KSC_data1.txt
/FIELD=(name, POSITION=1, SEPARATOR=' ', KOREAN_HANGUL)
/FIELD=(age, POSITION=2, SEPARATOR=' ', ASCII)
/FIELD=(employer, POSITION=3, SEPARATOR=' ', KOREAN_HANGUL)
/SORT
/KEY=employer
/OUTFILE=KSC_data1.sorted
```

This produces **KSC_data1.sorted**:

```
김동건,38,농심
김대중,54,농협
최순호,32,대우
박주영,30,빙그레
이순신,47,삼성
장나라,33,삼성
심형래,31,스리동
이승엽,28,중소기업은행
이명박,45,현대
이기영,29,현대
```

Example: Unicode Form 6 data type - Japanese SJIS sort

Consider the following input data, **SJIS_data1.txt**, which contains unsorted records with name, age, and employer fields:

```
藤原 釜足,35,株式会社みちのく銀行
日野 陽仁,65,三井物産株式会社
岩井 俊二,40,日本コンピュータシステム株式会社
鈴木 一郎,58,西武鉄道株式会社
坂本 龍一,64,オムロン株式会社
小野 洋子,37,パイオニア株式会社
村上 春樹,27,ロート製薬株式会社
岡野 穂高,43,西濃運輸株式会社
土田 順一,54,三井物産株式会社
藤村 健一,66,オムロン株式会社
```

The following script, **sjis_sort.scl** sorts the records over the employer field:

```
/INFILE=SJIS_data1.txt
  /FIELD=(name, POSITION=1, SEPARATOR=',', JAPANESE_HIRAGANA)
  /FIELD=(age, POSITION=2, SEPARATOR=',')
  /FIELD=(employer, POSITION=3, SEPARATOR=',', JAPANESE_HIRAGANA)
/SORT
  /KEY=(employer, SEPARATOR=',', JAPANESE_HIRAGANA)
/OUTFILE=SJIS_data1.sorted
```

This produces **SJIS_data1.sorted**:

```
坂本 龍一,64,オムロン株式会社
藤村 健一,66,オムロン株式会社
小野 洋子,37,パイオニア株式会社
村上 春樹,27,ロート製薬株式会社
藤原 釜足,35,株式会社みちのく銀行
土田 順一,54,三井物産株式会社
日野 陽仁,65,三井物産株式会社
岡野 穂高,43,西濃運輸株式会社
鈴木 一郎,58,西武鉄道株式会社
岩井 俊二,40,日本コンピュータシステム株式会社
```

Example: Unicode Form 6 data type - UTF16 Unicode sort

Consider the following input data, **Unicode_sample.input**:

```
Japanese, 日本
Chinese Simplified, 简体中文
Chinese Traditional, 中國傳統
Korean, 한국어
Vietnamese, Việt
Thai, ไทย
French, Française
```

The following script, **unicode_sort.scl** sorts the records by English:

```
/INFILE=Unicode_sample.input
/CHARSET=UTF16
/ENDIAN=BOM # Endian determined by file Byte Order Mark
/FIELD_PREDICATE=(POSITION=1,SEPARATOR=%utf16",",TYPE=UTF16_UNICODE)
/FIELD=(English_Uni)
/FIELD=(TX_Uni)
/SORT
/KEY=English_Uni
/OUTFILE=Unicode_sorted.out
/CHARSET=UTF16
/ENDIAN=BOM # Output with Infile Endian format with Byte Order Mark
/FIELD_PREDICATE=(TYPE=UTF16_UNICODE)
/FIELD=(English_Uni,POSITION=1,SIZE=40)
/FIELD=(TX_Uni,POSITION=45,size=18)
```

This produces **Unicode_sorted.out**:

Chinese Simplified	简体中文
Chinese Traditional	中國傳統
French	Française
Japanese	日本
Korean	한국어
Thai	ไทย
Vietnamese	Việt

Example: Form 0 ASCII to Form 6 UTF16_UNICODE conversion

Consider the following input data, **chiefs10.ascii**:

Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Adams, John Quincy	1825-1829	D-R	MA
Jackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA

The following script, **ascii_to_unicode.scl** sorts ASCII input and converts it to Unicode:

```
/INFILE=chiefs10.ascii
/FIELD=(president,POSITION=1,SIZE=27)
/FIELD=(term,POSITION=28,SIZE=9)
/FIELD=(party,POSITION=40,SIZE=3)
/FIELD=(state,POSITION=45,SIZE=2)
/SORT
/KEY=president
/OUTFILE=Ascii_to_Unicode_BOM.out
/CHARSET=UTF16
/ENDIAN=BOM
/FIELD=(president,POSITION=1,SIZE=54,UTF16_UNICODE)
/FIELD=(term,POSITION=57,SIZE=18,UTF16_UNICODE)
/FIELD=(party,POSITION=79,SIZE=6,UTF16_UNICODE)
/FIELD=(state,POSITION=89,SIZE=4,UTF16_UNICODE)
```

The output file, **Ascii_to_Unicode_BOM.out**, contains a byte order mark preceding Unicode characters in the endian order of the system of which the script is run. It also has a null character following each readable character. The file below represents the data as it appears represented in ascii.

Adams, John	1797-1801	FED	MA
Adams, John Quincy	1825-1829	D-R	MA
Harrison, William Henry	1841-1841	WHG	VA
Jackson, Andrew	1829-1837	DEM	SC
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Tyler, John	1841-1845	WHG	VA
Van Buren, Martin	1837-1841	DEM	NY
Washington, George	1789-1797	FED	VA

Example: Form 6 - Joining files with Chinese GBK Data

Consider the following input data, **GBK_data1.sorted** (resulting from Example: *on page 129*), which contains two columns with Chinese GBK characters, the Name (first) and Employer (third) fields. It will be used as the left-side file for the join:

秦林, 41, 北京中星微电子有限公司
 许军, 48, 福州大学
 王鸿娜, 52, 福州大学
 曹天才, 75, 华中科技大学
 孙琳, 44, 华中科技大学
 刘伟, 67, 中国船舶重工集团公司第七〇五研究所
 李雅帝, 66, 中国航天科工集团第三研究院
 夏学林, 69, 中国金属学会
 蔡莹, 46, 中国矿业大学
 沈建英, 44, 总参谋部第六十一研究所

The following file, **GBK_data2.sorted**, contains two columns with Chinese GBK characters, the Business (first) and Address (second) fields. It will be used as the right-side file for the join:

北京中星微电子有限公司, 北京市海淀区学院路35号
 福州大学, 福州市工业路523号
 华中科技大学, 湖北省武汉市洪山区珞喻路1037号
 中国船舶重工集团公司第七〇五研究所, 陕西省西安市高新技术产业开发区高新一路18号
 中国船舶重工集团公司第七一一研究所, 上海市华宁路 3111 号
 中国电子科技集团公司第三十八研究所, 合肥市高新技术产业开发区香樟大道 199 号
 中国航天科工集团第三研究院, 天津市南开区航天道
 中国金属学会, 北京市东四西大街46号
 中国矿业大学, 江苏省徐州市三环南路
 总参谋部第六十一研究所, 海淀区田村山南路17号

The following join script, **gbk_join.scl**, joins the left- and right-side files together where the Employer field from **GBK_data1.sorted** matches the Business field from **GBK_data2.sorted**:

```
/INFILE=GBK_data1.sorted
/ALIAS=File1
  /FIELD= (Name, POSITION=1, SEPARATOR=' ', CHINESE_GBK_SIMPLIFIED)
  /FIELD= (Age, POSITION=2, SEPARATOR=' ', NUMERIC)
  /FIELD= (Employer, POSITION=3, SEPARATOR=' ', CHINESE_GBK_SIMPLIFIED)
/INFILE=GBK_data2.sorted
/ALIAS=File2
  /FIELD= (Business, POSITION=1, SEPARATOR=' ', CHINESE_GBK_SIMPLIFIED)
  /FIELD= (Address, POSITION=2, SEPARATOR=' ', CHINESE_GBK_SIMPLIFIED)
/JOIN INNER File1 File2 WHERE File1.Employer == File2.Business
/OUTFILE=GBK_data1_GBK_data2.joined
  /FIELD= (Name, POSITION=1, CHINESE_GBK_SIMPLIFIED)
  /FIELD= (Age, POSITION=8, NUMERIC)
  /FIELD= (Employer, POSITION=11, CHINESE_GBK_SIMPLIFIED)
  /FIELD= (Address, POSITION=48, CHINESE_GBK_SIMPLIFIED)
```

This produces **BIG5_data1_BIG5_data2.joined**:

秦林	41	北京中星微电子有限公司	北京市海淀区学院路35号
许军	48	福州大学	福州市工业路523号
王鸿娜	52	福州大学	福州市工业路523号
曹天才	75	华中科技大学	湖北省武汉市洪山区珞喻路1037号
孙琳	44	华中科技大学	湖北省武汉市洪山区珞喻路1037号
刘伟	67	中国船舶重工集团公司第七〇五研究所	陕西省西安市高新产业开发区高新一路18号
李雅帝	66	中国航天科工集团第三研究院	天津市南开区航天道
夏学林	69	中国金属学会	北京市东四西大街46号
蔡莹	46	中国矿业大学	江苏省徐州市三环南路
沈建英	44	总参谋部第六十一研究所	海淀区田村山南路17号

As shown above, the two files were joined successfully over the common key fields (see */JOIN on page 224*).

Example: Form 6 - Joining Files with Chinese Big5 Data

Consider the following input data, **BIG5_data1.sorted** (resulting from Example: *on page 130*), which contains two columns with Chinese Big5 characters, the Name (first) and Employer (third) fields. It will be used as the left-side file for the join:

黃奕翊, 36, 大園敏盛醫院
 林哲佑, 54, 天成醫院
 林琪蓁, 45, 行政院衛生署新竹醫院
 李昀晏, 41, 南門綜合醫院
 胡耿嘉, 41, 桃新醫院
 蔡宏偉, 55, 財團法人長庚紀念醫院林口分院
 胡家發, 46, 財團法人長庚紀念醫院林口分院
 翁廷璋, 64, 國軍新竹地區醫院附設民眾診療服務處
 陳元暘, 49, 國軍新竹地區醫院附設民眾診療服務處
 李玠霖, 57, 慈祐醫院

The following file, **BIG5_data2.sorted**, contains two columns with Chinese Big5 characters, the Business (first) and Address (second) fields. It will be used as the right-side file for the join:

大園敏盛醫院, 桃園縣大園鄉華中街二號
 天成醫院, 桃園縣楊梅鎮新成路 1 9 6 號中山北路一段 3 5 6 號
 行政院衛生署新竹醫院, 新竹市經國路一段 4 4 2 巷 2 5 號
 李綜合醫療社團法人苑裡李綜合醫院, 苗栗縣苑裡鎮和平路 1 6 8 號
 南門綜合醫院, 新竹市東區林森路 2 0 號
 桃新醫院, 桃園縣桃園市復興路九十八號
 財團法人長庚紀念醫院林口分院, 桃園縣龜山鄉公西村復興街 5 號 5 之 7 號
 國軍新竹地區醫院附設民眾診療服務處, 新竹市北區武陵路 3 號
 崇仁醫院, 苗栗縣頭份鎮東興路 1 1 0 號
 慈祐醫院, 苗栗縣竹南鎮民治街 1 7 號

The following join script, **BIG5_join.scl**, joins the left- and right-side files together where the Employer field from **BIG5_data1.sorted** matches the Business field from **BIG5_data2.sorted**:

```
/INFILE=BIG5_data1.sorted
/ALIAS=File1
  /FIELD= (Name, POSITION=1, SEPARATOR=' ', CHINESE_BIG5)
  /FIELD= (Age, POSITION=2, SEPARATOR=' ', NUMERIC)
  /FIELD= (Employer, POSITION=3, SEPARATOR=' ', CHINESE_BIG5)
/INFILE=BIG5_data2.sorted
/ALIAS=File2
  /FIELD= (Business, POSITION=1, SEPARATOR=' ', CHINESE_BIG5)
  /FIELD= (Address, POSITION=2, SEPARATOR=' ', CHINESE_BIG5)
/JOIN INNER File1 File2 WHERE File1.Employer == File2.Business
/OUTFILE=BIG5_data1_BIG5_data2.joined
  /FIELD= (Name, POSITION=1, SEPARATOR=' ', CHINESE_BIG5)
  /FIELD= (Age, POSITION=2, SEPARATOR=' ', NUMERIC)
  /FIELD= (Employer, POSITION=3, SEPARATOR=' ', CHINESE_BIG5)
  /FIELD= (Address, POSITION=4, SEPARATOR=' ', CHINESE_BIG5)
```

This produces **BIG5_data1_BIG5_data2.joined**:

黃奕翊 36	大園敏盛醫院	桃園縣大園鄉華中街二號
林哲佑 54	天成醫院	桃園縣楊梅鎮新成路196號中山北路一段356號
林琪縈 45	行政院衛生署新竹醫院	新竹市經國路一段442巷25號
李昀晏 41	南門綜合醫院	新竹市東區林森路20號
胡耿嘉 41	桃新醫院	桃園縣桃園市復興路九十八號
蔡宏偉 55	財團法人長庚紀念醫院林口分院	桃園縣龜山鄉公西村復興街5號5之7號
胡家發 46	財團法人長庚紀念醫院林口分院	桃園縣龜山鄉公西村復興街5號5之7號
翁廷璋 64	國軍新竹地區醫院附設民眾診療服務處	新竹市北區武陵路3號
陳元暘 49	國軍新竹地區醫院附設民眾診療服務處	新竹市北區武陵路3號
李玠霖 57	慈祐醫院	苗栗縣竹南鎮民治街17號

As shown above, the two files were joined successfully over the common key fields (see /JOIN on page 224).

Example: Form 6 - Joining Files with Korean Hangul Data

Consider the following input data, **KSC_data1.sorted** (resulting from Example: *on page 131*), which contains two columns with Korean Hangul characters, the Name (first) and Employer (third) fields. It will be used as the left-side file for the join:

```

김동건, 38, 농심
김대중, 54, 농협
최순호, 32, 대우
박주영, 30, 빙그레
이순신, 47, 삼성
장나라, 33, 삼성
심형래, 31, 스리동
이승엽, 28, 중소기업은행
이명박, 45, 현대
이기영, 29, 현대

```

The following file, **KSC_data2.sorted**, contains two columns with Korean Hangul characters, the Business (first) and Address (second) fields. It will be used as the right-side file for the join:

```

나운기업, 경남 창원시 조동동 987
농심, 경기도 안양시 동안구 776-1
농협, 서울시 동대문구 면목동 3665
대우, 수원시 정자동 파장동 234
빙그레, 경기도 구리시 3665
삼성, 서울시 중구 249-4
스리동, 부산시 남촌동 366
아리스, 수원시 구기동 동안리 652-9
중소기업은행, 부산시 안창동 3550
현대, 서울시 남대문구 종운동 22-3

```

The following join script, **hangul_join.scl**, joins the left- and right-side files together where the Employer field from **KSC_data1.sorted** matches the Business field from **KSC_data2.sorted**:

```
/INFILE=KSC_data1.sorted
/ALIAS=File1
  /FIELD= (Name, POSITION=1, SEPARATOR= ' ', KOREAN_HANGUL)
  /FIELD= (Age, POSITION=2, SEPARATOR= ' ', NUMERIC)
  /FIELD= (Employer, POSITION=3, SEPARATOR= ' ', KOREAN_HANGUL)
/INFILE=KSC_data2.sorted
/ALIAS=File2
  /FIELD= (Business, POSITION=1, SEPARATOR= ' ', KOREAN_HANGUL)
  /FIELD= (Address, POSITION=2, SEPARATOR= ' ', KOREAN_HANGUL)
/JOIN INNER File1 File2 WHERE File1.Employer == File2.Business
/OUTFILE=KSC_data1_KSC_data2.joined
  /FIELD= (Name, POSITION=1, KOREAN_HANGUL)
  /FIELD= (Age, POSITION=8, NUMERIC)
  /FIELD= (Employer, POSITION=11, KOREAN_HANGUL)
  /FIELD= (Address, POSITION=25, KOREAN_HANGUL)
```

This produces **KSC_data1_KSC_data2.joined**:

김동건 38 농심	경기도 안양시 동안구 776-1
김대중 54 농협	서울시 동대문구 면목동 3665
최순호 32 대우	수원시 정자동 파장동 234
박주영 30 빙그레	경기도 구리시 3665
이순신 47 삼성	서울시 중구 249-4
장나라 33 삼성	서울시 중구 249-4
심형래 31 스리동	부산시 남촌동 366
이승엽 28 중소기업은행	부산시 안창동 3550
이명박 45 현대	서울시 남대문구 종운동 22-3
이기영 29 현대	서울시 남대문구 종운동 22-3

As shown above, the two files were joined successfully over the common key fields (see /JOIN on page 224).

Example: Form 6 - Joining Files with SJIS Japanese Data

Consider the following input data, **SJIS_data1.sorted** (resulting from Example: *on page 132*), which contains two columns with Japanese SJIS characters, the Name (first) and Employer (third) fields. It will be used as the left-side file for the join:

```
坂本 龍一,64,オムロン株式会社
藤村 健一,66,オムロン株式会社
小野 洋子,37,パイオニア株式会社
村上 春樹,27,ロート製薬株式会社
藤原 釜足,35,株式会社みちのく銀行
土田 順一,54,三井物産株式会社
日野 陽仁,65,三井物産株式会社
岡野 穂高,43,西濃運輸株式会社
鈴木 一朗,58,西武鉄道株式会社
岩井 俊二,40,日本コンピュータシステム株式会社
```

The following file, **SJIS_data2.sorted**, contains two columns with Japanese SJIS characters, the Business (first) and Address (second) fields. It will be used as the right-side file for the join:

```
オムロン株式会社,京都府京都市下京区塩小路通堀川東入南不動堂町 801
パイオニア株式会社,神奈川県川崎市幸区新小倉 1 番 1 号
ロート製薬株式会社,大阪府大阪市生野区箕西一丁目 8 番 1 号
株式会社みちのく銀行,青森市勝田一丁目 3 番 1 号
株式会社東京マルイ,東京都足立区綾瀬 4 丁目 16 番地 16 号
三菱石油株式会社,東京都品川区東大井五丁目 22 番 5 号
三井物産株式会社,東京都千代田区大手町 1 丁目 2 番 1 号
西濃運輸株式会社,岐阜県大垣市田口町 1 番地
西武鉄道株式会社,埼玉県所沢市くすのぎ台一丁目 1 番地 1
日本コンピュータシステム株式会社,東京都新宿区西新宿二丁目 1 番 1 号
```

The following join script, **sjis_join.scl**, joins the left- and right-side files together where the Employer field from **SJIS_data1.sorted** matches the Business field from **SJIS_data2.sorted**:

```
/INFILE=SJIS_data1.sorted
/ALIAS=File1
/FIELD=(Name, POSITION=1, SEPARATOR=' ', JAPANESE_HIRAGANA)
/FIELD=(Age, POSITION=2, SEPARATOR=' ', ASCII )
/FIELD=(Employer, POSITION=3, SEPARATOR=' ', JAPANESE_HIRAGANA)
/INFILE=SJIS_data2.sorted
/ALIAS=File2
/FIELD=(Business, POSITION=1, SEPARATOR=' ', JAPANESE_HIRAGANA)
/FIELD=(Address, POSITION=2, SEPARATOR=' ', JAPANESE_HIRAGANA)
/JOIN INNER File1 File2 WHERE File1.Employer == File2.Business
/OUTFILE=SJIS_data1_SJIS_data2.joined
/FIELD=(Name, POSITION=1, JAPANESE_HIRAGANA)
/FIELD=(Age, POSITION=8, ASCII)
/FIELD=(Employer, POSITION=11, JAPANESE_HIRAGANA)
/FIELD=(Address, POSITION=35, JAPANESE_HIRAGANA)
```

This produces **SJIS_data1_SJIS_data2.joined**:

```
藤村 健一 66 オムロン株式会社 京都府京都市下京区塩小路通堀川東入南不動堂町 801
坂本 龍一 64 オムロン株式会社 京都府京都市下京区塩小路通堀川東入南不動堂町 801
小野 洋子 37 パイオニア株式会社 神奈川県川崎市幸区新小倉 1 番 1 号
村上 春樹 27 ロート製薬株式会社 大阪府大阪市生野区箕西一丁目 8 番 1 号
藤原 釜足 35 株式会社みちのく銀行 青森市勝田一丁目 3 番 1 号
土田 順一 54 三井物産株式会社 東京都千代田区大手町 1 丁目 2 番 1 号
日野 陽仁 65 三井物産株式会社 東京都千代田区大手町 1 丁目 2 番 1 号
岡野 穂高 43 西濃運輸株式会社 岐阜県大垣市田口町 1 番地
鈴木 一朗 58 西武鉄道株式会社 埼玉県所沢市くすのき台一丁目 1 番地 1
岩井 俊二 40 日本コンピュータシステム株式会社 東京都新宿区西新宿二丁目 1 番 1 号
```

As shown above, the two files were joined successfully over the common key fields (see */JOIN on page 224*).

12 FIELD_PREDICATE

In a given /INFILE or /OUTFILE section, the /FIELD_PREDICATE statement sets default attributes and their values for the /FIELD statements that follow it. This makes it easier to write and read large scripts that have repeating field descriptions.

The syntax is:

```
/FIELD_PREDICATE=(attribute[,attribute]...)
```

where `attribute` can be any supported /FIELD attribute (and its value, if applicable).

For example, the statement:

```
/FIELD_PREDICATE=(SEPARATOR='^')
```

indicates that all fields beneath this statement in a given /INFILE or /OUTFILE section will be separated by the carat (^) character unless overridden at the individual field statement, or until a new /FIELD_PREDICATE statement is specified (see *Usage on page 143*). Note that the field position will automatically start at 1 when no position is specified.

12.1 Usage

/FIELD_PREDICATE statements can be inserted in any /INFILE or /OUTFILE section of a **sortcl** job script.

Attributes and their values specified in a /FIELD_PREDICATE statement apply to all /FIELDS that appear beneath the statement. In the case of a POSITION attribute when used with a SEPARATOR (for delimited fields), the fields beneath the /FIELD_PREDICATE statement will begin at that position, with respect to the separator character, and increment by 1 with each subsequent /FIELD (see **out1** in *Using FIELD_PREDICATE on page 144*).

You can overwrite a given attribute value on a per-field basis inside a /FIELD statement, but the /FIELD_PREDICATE attribute and value will resume with subsequent fields, as shown in the second output file (**out2**) of *Using FIELD_PREDICATE on page 144*.

Change the default attributes and values for all subsequent fields at any time by inserting a new, additional /FIELD_PREDICATE statement. This will overwrite all previous default attributes and values, and create a new set of default attributes and values for all /FIELDS that appear beneath the new statement, as shown in **out3** of *Using FIELD_PREDICATE on page 144*.

Example: Using FIELD_PREDICATE

Given the input file addresses:

```
Dick Jones,1234 Maple St.,Philadelphia,Pennsylvania
Sam Henderson,1400 Highway A1A,Satellite Beach,Florida
Harry James,50 Elm Ave.,Boston,Massachusetts
Sarah Smith,300 Thornton Rd.,Frankfurt,Kentucky
```

The following job script, **predicates.scl**, shows various ways in which /FIELD_PREDICATE can be used.

```
/INFILE=addresses
# FIELD_PREDICATE for delimited fields
  /FIELD_PREDICATE=(SEPARATOR=',')
    /FIELD=(Name)
    /FIELD=(Address)
    /FIELD=(City)
    /FIELD=(State)
/REPORT
/OUTFILE=out1 # delimited fields
  /FIELD_PREDICATE=(POSITION=1,SEPARATOR='^',FILL='+',SIZE=15)
    /FIELD=(Name)
    /FIELD=(Address)
    /FIELD=(City)
    /FIELD=(State)
/OUTFILE=out2 # fixed-length fields
  /FIELD_PREDICATE=(size=15,fill='x')
    /FIELD=(Name,POSITION=1)
    /FIELD=(Address,POSITION=16,FILL='_') # override FILL
    /FIELD=(City,POSITION=31)
    /FIELD=(State,POSITION=46)
/OUTFILE=out3 # multiple FIELD_PREDICATES
  /FIELD_PREDICATE=(SIZE=15,FILL='^')
    /FIELD=(Name,POSITION=1)
    /FIELD=(Address,POSITION=17)
  /FIELD_PREDICATE=(SIZE=17,FILL='x') # new SIZE, new FILL
    /FIELD=(City,POSITION=33)
    /FIELD=(State,POSITION=51)
```

out1 contains

```
Dick Jones+++++^1234 Maple St.+^Philadelphia+++^Pennsylvania+++
Sam Henderson++^1400 Highway A1^Satellite Beach^Florida+++++++
Harry James++++^50 Elm Ave.++++^Boston+++++++^Massachusetts++
Sarah Smith++++^300 Thornton Rd^Frankfurt+++++^Kentucky++++++
```


Note that:

- the starting position is 1, and increments by 1 for each subsequent field
- a caret separator character (^) is used for all fields
- each field is a fixed size of 15
- a FILL character of + is used to pad the fields to their full widths of 15.

out2 contains:

```
Dick Jonesxxxxx 1234 Maple St._ Philadelphiaxxx Pennsylvaniaxxx
Sam Hendersonxx 1400 Highway A1 Satellite Beach Floridaxxxxxxxxx
Harry Jamesxxxx 50 Elm Ave._____ Bostonxxxxxxxxxx Massachusettsxx
Sarah Smithxxxx 300 Thornton Rd Frankfurtxxxxxx Kentuckyxxxxxxxx
```

As shown above:

- each field is a fixed size of 15, with a FILL character of x used to pad the fields to their full widths
- a FILL character of _ is used for the address field only, overriding the FILL character x for this field only
- fields begin at fixed byte positions, as indicated by the POS attributes given for each field, that is, the /FIELD_PREDICATE entry did not introduce a separator character (delimited fields) with a starting position, as for **out1**

out3 contains:

```
Dick Jones^^^^^ 1234 Maple St.^ Philadelphiaxxxxx Pennsylvaniaxxxxx
Sam Henderson^^ 1400 Highway A1 Satellite Beachxx Floridaxxxxxxxxxxx
Harry James^^^^ 50 Elm Ave.^^^^ Bostonxxxxxxxxxx Massachusettsxxxx
Sarah Smith^^^^ 300 Thornton Rd Frankfurtxxxxxx Kentuckyxxxxxxxx
```

As shown above:

- the first two fields, Name and Address, have a fixed size of 15, with a ^ fill character, as specified by the /FIELD_PREDICATE statement that precedes them.
- the next two fields, City and State, have a fixed size of 17, with a + fill character, as specified by the new /FIELD_PREDICATE statement that precedes them.

13 FIELD EXPRESSIONS (CROSS-CALCULATION)

In both /INREC and in the output files, it is possible to specify a mathematical expression in place of the field name. These expressions can reference multiple fields and use arithmetic operators and mathematical functions. Parentheses can be used to control operator precedence, and temporary fields can be created to hold intermediate values. These features are particularly useful for ad hoc financial calculations, spreadsheet-style presentations, and business intelligence requirements.



sortcl requires that all arithmetic symbols be preceded and followed by a space in /FIELD and /DATA expressions, as shown in the example below.

It is possible to use the /ROUNDING statement to change the way in which numeric values with several decimal places are rounded after an arithmetic **sortcl** operation (see /ROUNDING on page 281).

Table 3 shows the arithmetic operators in high-to-low-precedence order:

Table 3: Arithmetic Symbols and their Precedence

Operator	Meaning
()	parentheses
* / %	multiply, divide, whole number remainder of x/y
+ -	add, subtract, and unary operators (such as t=-5)

Example: Mathematical Expressions

The following script, **expr.scl**, demonstrates expression writing and its use of precedence:

```
/INFILE=X
  /FIELD=(a, POSITION=01, SIZE=3)
  /FIELD=(b, POSITION=04, SIZE=3)
  /FIELD=(c, POSITION=07, SIZE=3)
  /FIELD=(d, POSITION=10, SIZE=3)
/OUTFILE=stdout
  /HEADREC="a b c d t=a+b*(c+d) (t-1)/4\n\n"
  /DATA=a
  /DATA=b
  /DATA=c
  /DATA=d
  /FIELD=(t=a + b * (c + d), POSITION=13, SIZE=9, NUMERIC) # calculate t
  /FIELD=((t - 1) / 4, SIZE=12, NUMERIC) # calculate and display
```

Internal arithmetic is performed in floating-point precision. This example calculates a value for t and displays it. The arithmetic that produces t is performed in the following order:

- 1) Add c to d .
- 2) Multiply by b .
- 3) Add a .
- 4) Store t .

The following is the input file:

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

The output file is produced by the command:

```
sortcl /specification=expr.scl
```

It produces:

a	b	c	d	t=a+b*(c+d)	(t-1)/4
1	2	3	4	15	3.50
2	3	4	5	29	7.00
3	4	5	6	47	11.50
4	5	6	7	69	17.00

Cross-Calculation on Summary Fields

You can also include cross-calculation expressions with summary fields, which enables you to perform arithmetic across values that are themselves the result of aggregation. See Using WHERE with Sums, Cross-Cals on Sums *on page 301* for an example.

The functions supported by **sortcl** are shown in *Table 4*.

Table 4: Mathematical Functions

Function	Description
<code>abs (x)</code>	absolute value of x , $ x $. x can be a whole number or a floating point value
<code>acos (x)</code>	arc cosine of x ; range: $[0, \pi]$ radians; $-1 \leq x \leq 1$
<code>asin (x)</code>	arc sine of x ; range: $[-\pi/2, \pi/2]$ radians; $-1 \leq x \leq 1$
<code>atan (x)</code>	arc tangent of x ; range: $[-\pi/2, \pi/2]$ radians
<code>atan2 (x, y)</code>	arc tangent of y/x ; range: $[-\pi, \pi]$
<code>ceil (x)</code>	smallest whole number not less than x
<code>cos (x)</code>	cosine of x in radians
<code>cosh (x)</code>	hyperbolic cosine of x
<code>exp (x)</code>	e to the power of x
<code>floor (x)</code>	largest whole number (as a double-precision number) not greater than x
<code>gamma (x)</code>	$\ln(\text{GAMMA}(x))$
<code>hypot (x, y)</code>	$\text{sqrt}(x^2 + y^2)$
<code>j0 (x)</code>	Bessel function of x , first kind, order 0
<code>j1 (x)</code>	Bessel function of x , first kind, order 1
<code>jn (n, x)</code>	Bessel function of x , first kind, order n
<code>length (n)</code>	Return the field (n) length. See Field Length Function <i>on page 192</i> .
<code>log (x)</code>	natural logarithm of x ; $x > 0$
<code>log10 (x)</code>	logarithm base ten of x ; $x > 0$
<code>mod (x, y)</code>	remainder of the division of x by y : x if y is zero or if x/y would overflow; otherwise the number f with the same sign as x , such that $x = iy + f$ for a whole number i , and $f < y $
<code>pow (x, y)</code>	x to the power of y ; if $x < 0$, y must be a whole number
<code>random (n)</code>	generate random values from 0 to $n-1$.
<code>sin (x)</code>	sine of x in radians
<code>sinh (x)</code>	hyperbolic sine of x
<code>sqrt (x)</code>	non-negative square root of x ; $x \geq 0$
<code>tan (x)</code>	tangent of x in radians
<code>tanh (x)</code>	hyperbolic tangent of x
<code>y0 (x)</code>	Bessel function of x , second kind, order 0.
<code>y1 (x)</code>	Bessel function of x , second kind, order 1.
<code>yn (n, x)</code>	Bessel function of x , second kind, order n .

14 /INREC

The /INREC record provides a common record layout for use by the *action* phase of **sortcl** and subsequent processing. An /INREC statement is required when there are more than one input file formats being processed. A common, virtual record is required that can only contain fields that exist in all the inputs or newly named fields that can be derived from those common fields (as illustrated in the /INREC section in Example: *on page 192*).

It is not required when:

- there is only one input file
- multiple input files have the same format

/INREC can also be used to remap to shorter records where a large number of long records are being input, but shorter records are required for output. Using /INREC to reduce the amount of data that is sent to the action phase can speed the whole application. The /INREC record should contain only those fields required for /KEY, and those required to process and create the output records.

Whether specifying existing or derived field names, the /INREC section allows you to re-define the record layout, for example the positions and sizes, for processing or remapping purposes.

The syntax is similar to /INFILE and /OUTFILE:

```
/INREC[=filename]  
attributes
```

Example: Using /INREC

Consider the input file formats of the file **chicago** (see Numeric Attributes *on page 109*) and the following file, **miami**:

Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95
Sending Your	Valley Kill	130	15.75
People Please	Valley Kill	75	11.50
Map Reader	Prentice-Hall	200	14.95

The following **sortcl** script, **inrec.scl**, describes their attributes, and maps a common record layout for them:

```
/INFILE=chicago
  /FIELD= (num, POSITION=1, SIZE=5)
  /FIELD= (title, POSITION=6, SIZE=15)
  /FIELD= (price, POSITION=21, SIZE=5, NUMERIC)
  /FIELD= (publisher, POSITION=30, SIZE=15)
/INFILE=miami
  /FIELD= (title, POSITION=1, SIZE=15)
  /FIELD= (publisher, POSITION=16, SIZE=13)
  /FIELD= (quantity, POSITION=30, SIZE=3)
  /FIELD= (price, POSITION=38, SIZE=5, NUMERIC)
/INREC
  /FIELD= (title, POSITION=1, SIZE=15)
  /FIELD= (publisher, POSITION=16, SIZE=15)
  /FIELD= (price, POSITION=31, SIZE=5, NUMERIC)
/SORT
  /KEY=title
/OUTFILE=prices.out
```

Since there are no fields defined for the output file **prices.out**, the format will be the same as for the /INREC record:

Beginnings	Prentice-Hall	8.50
Going Nowhere	Valley Kill	17.95
Greater Than	Valley Kill	34.75
Married Young	Prentice-Hall	24.95
Map Reader	Prentice-Hall	14.95
Murder Plots	Harper-Row	5.90
Not On Call	Harper-Row	9.99
On Top	Harper-Row	15.95
People Please	Valley Kill	11.50
Pressure Cook	Harper-Row	9.95
Reasoning For	Prentice-Hall	10.25
Sending Your	Valley Kill	15.75
Still There	Dell	13.05
Still There	Dell	13.05

15 /DATA

/DATA statements are used to pad and format output records. /DATA statements are not named fields, so they cannot be directly mapped. They are positioned just after the previous /DATA or /FIELD statement.

There are several forms of /DATA statements:

/DATA= [{*n*}] "*literal_string*"

where *n* is the number of times to repeat the *literal_string* you specify, and the constant string can contain any combination of constants and/or control (escape) characters (see Table 6 on page 155) and conversion specifiers (see Table 7 on page 157).



The literal string can also contain syntax specific to a mark-up language, such as HTML, provided the browser (or other utility) used to read the output file accepts that syntax. For an example of producing an HTML report, see Example: on page 314.

The repetitive string syntax using {*n*} "*literal_string*" is also supported in a /CONDITION statement, and can be used for both record-level and field-level evaluation purposes (see Binary Logical Expressions on page 171).

/DATA=*field_name*

displays the value of an input field without formatting

/DATA=*internal_variable*

where the internal variable can be any **sortcl** internal variable listed in Table 5 on page 154.

You can also use a conditional data statement (see CONDITIONAL FIELD AND DATA STATEMENTS on page 180).

Example: Using /DATA

Consider the input file **miami** from Example: *on page 150*. The following script, **data.scl**, includes several instances of /DATA statements:

```
/INFILE=miami
  /FIELD=(Title,POSITION=1,SIZE=15)
  /FIELD=(Publisher,POSITION=16,SIZE=13)
/SORT
  /KEY=Publisher
  /KEY=Title
/OUTFILE=stdout
  /FIELD=(Publisher,POSITION=1,SIZE=15)
  /DATA="Book: "          #constant string
  /DATA=Title#field-name
  /DATA=" " #constant string
  /DATA=CURRENT_TIMESTAMP #internal variable
  /DATA={3}"*-*"          #repeated constant string
  /DATA="\n"              #linefeed control character
```

This produces:

Dell	Book: Still There	1999-05-24 14.34.37*-***-***-
Harper-Row	Book: Murder Plots	1999-05-24 14.34.37*-***-***-
Harper-Row	Book: Pressure Cook	1999-05-24 14.34.37*-***-***-
Prentice-Hall	Book: Map Reader	1999-05-24 14.34.37*-***-***-
Prentice-Hall	Book: Reasoning For	1999-05-24 14.34.37*-***-***-
Valley Kill	Book: People Please	1999-05-24 14.34.37*-***-***-
Valley Kill	Book: Sending Your	1999-05-24 14.34.37*-***-***-

The double line between records is produced by the statement /DATA="\n".

16 INTERNAL VARIABLES

sortcl maintains internal values that can be used in `/DATA`, `/HEADREC`, and `/FOOTREC` statements (see `/DATA` on page 152, `/HEADREC` on page 270, and `/FOOTREC` on page 272). The internal values are shown in Table 5.

Table 5: Internal Variables

Variable	Output/Example
AMERICAN_DATE EUROPEAN_DATE JAPANESE_DATE ISO_DATE CURRENT_DATE	Sep/19/2006 19.09.2006 2006-09-19 2006-09-19 2006-09-19
AMERICAN_TIME EUROPEAN_TIME JAPANESE_TIME ISO_TIME CURRENT_TIME VALUE_TIME	09:47:15 AM 21.47.15 9:47:15 PM 21:47:15 21.47.15 350:47:15
AMERICAN_TIMESTAMP EUROPEAN_TIMESTAMP JAPANESE_TIMESTAMP ISO_TIMESTAMP CURRENT_TIMESTAMP VALUE_TIMESTAMP	Sep/19/2006 09:47:15 AM 19.09.2006 21.47.15 2006-09-19 9:47:15 PM 2006-09-19 21:47:15 2006-09-19 21.47.15 247 350:47:15
SYSDATE	current date and time in the format used by Oracle and SQLbase
CURRENT_TIMEZONE	Eastern Daylight Time (Windows) EDT (Unix)
PAGE_NUMBER (/HEADREC and /FOOTREC statements only)	page number of the report
USER	the name of the user currently executing the program

For example, your script includes the following statement:

```
/HEADREC="Produced on %s",CURRENT_DATE
```

then the top of the output might contain:

```
Produced on 2006-09-19
```

See details on using format control characters, such as `%s`, see Table 12 on page 271.

17 CONTROL (ESCAPE) CHARACTERS

Statements using /DATA, /HEADREC, and /FOOTREC can contain escape characters (see /DATA on page 152, /HEADREC on page 270, and /FOOTREC on page 272). These characters, shown in Table 6, produce special effects in output records.

Table 6: Control (Escape) Characters

Character	Format within a String
\a	alert
\\	backslash
\b	backspace
\r	carriage return
\"	double quote
\f	form-feed
\t	horizontal tab
\n	newline
\0	NULL character
\'	single quote
\v	vertical tab
\\$	dollar sign

18 NON-ASCII FIELD EVALUATION AND CONVERSION SPECIFIERS

To perform a conditional evaluation on a non-ASCII field, insert the ASCII equivalent into the condition statement. For example:

```
/INFILE=test_values
/FIELD=(value,POSITION=1,SIZE=4,MF_CMP3)
/INCLUDE WHERE value > 5000000
/REPORT
/OUTFILE=values_large
```

The above job will return only those MF_CMP3 values with an ASCII equivalent of greater than 5 million. Note that the value used in the condition is 5000000, even though the input field being evaluated contains packed data with a size of four.

Use a conversion specifier to define a condition where a value's known equivalent is used in place of an input value of a different data type.



A conversion specifier can also be used to convert a known value into its equivalent in another data type (see Using a Conversion Specifier within a /DATA statement *on page 158*).

Use the EBCDIC conversion specifier to use the ASCII equivalent of an EBCDIC value within a condition, or return the EBCDIC equivalent of an ASCII value within the records of your output. Use the PACKED conversion specifier to use the ASCII-numeric equivalent of a PACKED value within a condition, or return the PACKED equivalent of an ASCII value within the records of your output.

Use the hexadecimal conversion specifier to return the hexadecimal equivalent of an ASCII value. For example, include the statement /DATA=%HEX"C134", and the two-byte hexadecimal equivalent, consisting of c1 and 34, is returned. A hex dump representation of this value therefore appears as C1 34 at the specified location within every output record.



You must use the %HEX conversion specifier for conditions involving values in multi-byte data types. See Multi-Byte Character Types *on page 127*.

The ASCII conversion specifier is not typically required because the character string you specify for conversion is already in ASCII. It is provided only for situations where the specifier itself is a variable (see Environment Variables *on page 42*).

Table 7 shows the syntax for using the conversion specifiers which are recognized by **sortcl**.

Table 7: Conversion Specifiers

Data Type	Form
EBCDIC	%EBCDIC" <i>ASCII_value</i> "
PACKED	%PACKED" <i>ASCII_value</i> "
Hexadecimal	%HEX" <i>ASCII equivalent of HEX value</i> " such as c134
ASCII	%ASCII" <i>string</i> "
UTF16	%UTF16" <i>string</i> " (see Multi-Byte Character Separators <i>on page 97</i>)
UTF32	%UTF32" <i>string</i> " (see Multi-Byte Character Separators <i>on page 97</i>)



For hexadecimal values 01 through 09, use "*n*" without the %HEX specifier, where *n* is any whole number from 1 to 9. For example, /DATA="*\4*" returns the hexadecimal value 04.

This short-hand method can also be used within conditions.

18.1 Using a Conversion Specifier within a Condition

The following is an example of a condition involving a conversion specifier when the input field's data is in EBCDIC format:

```
/CONDITION=(Senior,TEST=(Age GT %E"65"))
```

If you /INCLUDE this condition, only those records where the EBCDIC equivalent of 66 and higher for the field Age are included in the output (see CONDITIONS *on page 169*).

18.2 Using a Conversion Specifier within a /DATA statement

To return the EBCDIC equivalent of the ASCII value 65 in an output file, for example, use the following:

```
/DATA=%EBCDIC"65"
```

19 ACTION

This section describes the *actions* that can take place when moving records from input to output. Only one action statement can be designated in a **sortcl** job; they are mutually exclusive. There are five *action* statements:

- /SORT Records are passed from input into **sortcl**, and reordered on one or more *keys*. This is the default action when no action statement is specified, and encompasses report and merge functionality (see KEYS *on page 160*).
- /JOIN Fields from the records of two or more input files are joined / matched (see /JOIN *on page 224*).
- /REPORT Records are passed from input to output without reordering; this is used for selecting, reformatting, and aggregating records (see Example: *on page 295*).
- /CHECK Records are checked to see if they are already in order by a given key (see KEYS *on page 160* and Example: *on page 288*). No reformatting is performed.
- /MERGE Records from two or more input files are folded together based on the keys. The input files should already be ordered on those same keys (see KEYS *on page 160* and Example: *on page 289*).

With the exception of /CHECK, all **sortcl** job actions permit multi-target reformatting, aggregation, selection, and so on. The /JOIN action can also include sorting.

20 KEYS

With /CHECK, /SORT, and /MERGE, the order of the output records is determined by comparing one or more key fields within records until they are not equal. The /KEY statement is used for specifying each key field. Any number of /KEY statements can be given. Compares will be performed in the order given while each key is equal. If two records have keys that are all equal, the output order of the two records will be arbitrary unless the /STABILITY statement is specified.

/KEYPROCEDURE

sortcl allows the calling program to set up a custom compare using the command /KEYPROCEDURE, as described in Custom Compares *on page 534*. You can also invoke a custom compare as a function of the /KEY statement, as described in Custom Key Compares Using /KEY *on page 534*.

This section describes the components of the /KEY statement:

- Syntax *on page 161*
- Field Name Reference *on page 161*
- Unnamed Reference *on page 162*
- Alternate Data Type *on page 162*
- Alternate Collating Sequence: /ALTSEQ *on page 163*
- Direction *on page 165*
- ASCII Options *on page 166*
- Stability *on page 167*
- No Duplicates, Duplicates Only *on page 168*

20.1 Syntax

Key parameters are separated by commas. If the field name is used, it must be first, while the other parameters can appear in any order:

```
/KEY=(field [, data type or altseq] [, direction] [, ASCII options])
```

field can be one of the following:

- an input field name
- a column position and size
- a field position and separator, and optionally, a size



Specifying /SORT (the default action) with no subsequent /KEY statements indicates that a left-to-right sort will be performed, across the entire record. In this case, **sortcl** sorts records according to their sequential byte values. However, this is not necessarily the natural or defined order for non-ASCII data types, in which case you must define one or more /KEYS according to the rules of this section.

20.2 Field Name Reference

The simplest form of the /KEY statement uses a defined input field name for *field* and no other parameters. When *field* is the only parameter and the direction is ascending, the parentheses are not required. The position in the record, size of the field, and data type are known from the /FIELD description.

The following **sortcl** script defines a data file that contains first and last names. The sort order is to be by last name, followed by first name:

```
/INFILE=chiefs_sep
  /FIELD=(fname, POSITION=1, SEPARATOR='|')
  /FIELD=(lname, POSITION=2, SEPARATOR='|')
  /FIELD=(term, POSITION=3, SEPARATOR='|')
/SORT
  /KEY=lname
  /KEY=fname
/OUTFILE=chiefs.out
```

20.3 Unnamed Reference

A *key* field can contain a field statement without a name, as in:

```
/KEY=(POSITION=1,SIZE=15)
/KEY=(POSITION=3,SEPARATOR='|')
```

20.4 Alternate Data Type

A different collating sequence than a field's original data type can be specified for a compare operation. By default, the sort order is determined by the data type of the input field. If the field was not specifically typed on input, ASCII collation will occur.

For example, you might have an ASCII key field that contains random binary characters. In this case, you should specify an EBCDIC collating sequence for this key field so that the binary characters will be sorted along with the ASCII characters. Because the EBCDIC collating sequence orders characters by their hexadecimal-value equivalents, the ASCII ordering is not compromised. If the key field is called *apples*, for example, you would specify:

```
/KEY=(apples,EBCDIC)
```



When using ASCII as the collating sequence (the default), any non-printable ASCII characters in your input data are ignored for ordering purposes.

20.5 Alternate Collating Sequence: /ALTSEQ

Using the /ALTSEQ statement and the ALTSEQ data-type option, you can use a custom collating sequence in cases where the desired ordering sequence varies from the natural ordering of the data type. This feature also allows you to perform value substitutions on output, and to perform conditional evaluation on the converted field. ALTSEQ is supported for input fields that are declared as either ASCII (the default) or EBCDIC. It is not supported for use with multi-byte characters.

The syntax for specifying an alternate sequence is as follows:

```
/INFILE (S) =source (s)
/ALTSEQ= (hex_value1Ahex_value1B[,hex_value2Ahex_value2B] [,etc.] )
```

where the character represented by *hex_value1A* will be replaced with the character represented by *hex_value1B*. Optionally, you can specify additional replacements (offset with a comma) such as substituting *hex_value2A* for *hex_value2B*, and so on. See ASCII COLLATING SEQUENCE *on page 654* and EBCDIC PRINTING CHARACTERS *on page 655* for a list of the hexadecimal equivalents to ASCII and EBCDIC characters, respectively.



Use multiple /ALTSEQ statements in cases where you require multiple replacements that you do not want to include within a single entry.

Using ALTSEQ with /KEY

After you have specified the /ALTSEQ statement on input, include the data-type attribute ALTSEQ within a /KEY statement for an alternate key comparison.

For example, in addition to the syntax shown above, use:

```
/KEY= (F1, ALTSEQ)
```

to ensure that the value of *hex_value1A* will replace the value of *hex_value1B* for the field F1 for ordering purposes. Note that this usage use of ALTSEQ is for sorting precedence only. If required, use ALTSEQ at the output field level to perform value substitution for display purposes (see the following section). See *Example:* to see how ALTSEQ can be used in a /KEY statement.

Using ALTSEQ on Output

You can include the data-type attribute ALTSEQ within an output /FIELD statement to perform value replacements based on the /ALTSEQ statements specified on input. For example:

```
/OUTFILE=test.out  
/FIELD=(F1, POSITION=1, SIZE=5, ALTSEQ)
```

will convert the output of field F1 to reflect the character value substitutions specified in the earlier /ALTSEQ statement. See *Example:* to see how ALTSEQ can be used for remapping purposes.

Using ALTSEQ with Conditions

Use ALTSEQ in an /INREC section to convert a field for conditional evaluation purposes (see /INREC *on page 150*). To do this while still preserving the original field contents for output purposes, define the same input source field twice in the /INFILE section, using different field names. Therefore, if an /ALTSEQ statement is defined in the /INFILE section, you can convert one of the over-defined fields using ALTSEQ as the data-type attribute in the /INREC section, while also including the other over-defined field in the /INREC section to preserve the original field contents on output.

The output files can then use /INCLUDE or /OMIT logic against the field that was converted on /INREC, and output fields can contain IF THEN ELSE logic that also references the converted field. See Include-Omit Evaluation *on page 175* and CONDITIONAL FIELD AND DATA STATEMENTS *on page 180*.

Example: Using ALTSEQ

Consider the following input data:

```
EF GH  
IJ KL  
AB CD
```

The following job script, **altseq.scl**, includes an `/ALTSEQ` statement, and utilizes the data-type conversion in both a `/KEY` statement and at the output field level:

```
/INFILE=altseq.in
    /ALTSEQ=(4151,2040) # substitute A (hex 41) with Q (hex 51)
                        # and substitute <blank> with @
    /FIELD=(F1, POSITION=1, SIZE=5)
/SORT
    /KEY=(F1,ALTSEQ)    # utilize /ALTSEQ above to affect collation
/OUTFILE=altseq2.out
    /FIELD=(F1, POSITION=1, SIZE=5, ALTSEQ)
                        # utilize /ALTSEQ above to affect display
```

The output is as follows:

```
EF@GH
IJ@KL
QB@CD
```

As shown above, the F1 `/KEY` was affected by the substitution of AB CD with QB@CD, and that record was therefore considered last (originally first) in the sorting order. Also, the output field display reflects the conversions specified by the `/ALTSEQ` statement because of the `ALTSEQ` data-type attribute used on output.

20.6 Direction

Direction of a key's comparison is either `ASCENDING` or `DESCENDING`. If the direction is omitted, the default is `ASCENDING`. An example is:

```
/KEY=(Price,DESCENDING)
```

This will instruct **sortcl** to order the values of the Price field in reverse.

20.7 ASCII Options

These can be used when the /KEY field is ASCII (explicitly or implicitly). These options do not actually reformat the field; they only change how the comparison is made (see *Alignment on page 103* for actual field reformatting). The ASCII options are:

- alignment** Causes the key field to be shifted for comparison purposes. The options are:
- Left** Sort as if the key field characters precede trailing spaces.
 - Right** Sort as if the key field characters trail leading spaces.
 - None** The default. Sort with the key field characters compared as they currently appear in the field. Sort order can be affected by leading or trailing spaces within the field.

For example, given the following:

```
/INFILE=in.dat  
/FIELD=(f1, POSITION=1, SIZE=9)  
/KEY=(f1, alignment)
```

and assuming the record contains `__Chars__`¹, the alignments shown in *Table 8* can occur during comparisons.

Table 8: Comparisons—Alignment

Alignment	Effect
LEFT	Chars_____
RIGHT	_____Chars
NONE	__Chars__

- case** can be `CASE_ON` or `CASE_OFF` to cause the key field to become case-sensitive, as shown in *Table 9* using the previous record. Default is `CASE_ON`. With `CASE_ON`, key field characters are compared as they appear, respecting uppercase and lowercase letters. All uppercase letters collate ahead of lowercase letters.

Table 9: Comparisons—Case

Case	Effect
CASE_ON	__Chars__
CASE_OFF	__CHARS__

1. The underscore character (`_`) represents a blank space.

20.8 Stability

If stability is specified for a sort, then when records have keys that compare equally, the order of the output will be the same as the input. A stable sort will be performed with the statement:

```
/STABLE
```

Example: Using /STABLE

Given the input file:

Eisenhower, Dwight D.	134	1953-1961	REP	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Nixon, Richard M.	137	1969-1973	REP	CA
Ford, Gerald R.	138	1973-1977	REP	NE
Carter, James E.	139	1977-1981	DEM	GA
Reagan, Ronald W.	140	1981-1989	REP	IL
Bush, George H.W.	141	1989-1993	REP	TX
Clinton, William J.	142	1993-2001	DEM	AR
Bush, George W.	123	2001-2009	REP	TX
Obama, Barack H.	131	2009-2011	DEM	IL

and the script **stable.scl**:

```
/INFILE=chiefs_10
  /FIELD=(president,POSITION=1,SIZE=22)
  /FIELD=(votes,POSITION=24,SIZE=3)
  /FIELD=(service,POSITION=28,SIZE=9)
  /FIELD=(party,POSITION=40,SIZE=3)
  /FIELD=(state,POSITION=45,SIZE=2)
/SORT
  /KEY=party
  /STABLE
/OUTFILE=stable.out
```

the output will be:

Kennedy, John F.	135	1961-1963	DEM	MA
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Obama, Barack H.	131	2009-2011	DEM	IL
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Nixon, Richard M.	137	1969-1973	REP	CA
Ford, Gerald R.	138	1973-1977	REP	NE
Reagan, Ronald W.	140	1981-1989	REP	IL
Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX

Notice how the names appear in input order when the party matches.



There is overhead required to maintain stability. Do not specify `/STABLE` if your file:

- does not contain records with duplicate key fields
- contains records with duplicate key fields, but the resulting order does not matter.

Merges are inherently stable; therefore, do not specify `/STABLE` for a merge.

20.9 No Duplicates, Duplicates Only

Duplicates occur when two or more records have key fields that compare equally. Only the key fields, not the records, must compare equally for a record to be considered a duplicate.

The statement `/NODUPLICATES` results in only one of the duplicate records being processed and returned. If `/STABLE` is also specified, the earliest input duplicate is the one retained. Without `/STABLE`, the duplicate retained is arbitrary.

Using the same example, and with `/NODUPLICATES` below the `/STABLE` statement, the output will be:

Kennedy, John F.	135	1961-1963	DEM	MA
Eisenhower, Dwight D.	134	1953-1961	REP	TX

Conversely, you can specify `/DUPLICATESONLY`, where only records containing key fields that compare equally are processed and returned. Using the same example, if the input file were amended to also contain a single record with the party `WHG`, and `/DUPLICATESONLY` is specified, then all the `DEM` and `REP` records would be returned, but not the single record containing `WHG`.



To exclude duplicate records at the individual output file level (that is, after processing), see *Unary Logical Expressions (Change Test)* on page 170.

21 CONDITIONS

A condition is a *logical expression* that combines field names and/or constants with relational and/or logical operators. When the expression is evaluated, it will be either true or false.

A condition is associated with several **sortcl** statements, and can be used for both input and output. The true/false result determines how the statement works. The following statements use a condition within their definitions:

- /INCLUDE and /OMIT
- /DATA using IF-THEN-ELSE logic
- /FIELD using IF-THEN-ELSE logic
- /JOIN
- WHERE and BREAK in summary functions (MAX, MIN, SUM, AVG, COUNT)

21.1 Syntax

A condition can be one of two forms (see INCLUDE-OMIT (RECORD SELECTION) on page 175):

Named where the logical expression has a name and is defined with the statement:

```
/CONDITION=(condition_name, TEST=(logical_expression))
```

Once the *condition_name*, is defined, it can be used in one of the above statements that uses a condition, for example
/INCLUDE WHERE *condition_name*. Naming is done for documentation purposes and to build nested expressions. It is also more efficient than using unnamed conditions if the same logical expression is used for more than one statement. The named condition is analyzed only once, and the result is reusable.



If you have a named condition that pertains to an INFILE sources, and you then include an INREC section in the script (see /INREC on page 150), the named input conditions is no longer recognized beyond the input phase of the job. In all other cases, named input conditions are reusable.

Unnamed (explicit) where the logical expression is built into the statement using it:

```
/INCLUDE WHERE logical_expression  
/OMIT WHERE logical_expression
```

The *logical_expression* can involve any of the following types:

- Unary Logical Expressions (Change Test) *on page 170*
- Binary Logical Expressions *on page 171*
- Function Compares in Conditions (iscompares) *on page 182*
- Compound Logical Expressions *on page 172*

21.2 Unary Logical Expressions (Change Test)

The simplest logical expression is a field name. The condition is true when the value of the field is different from its value in the previous record. If the value of the field has not changed, the condition is false.



To remove duplicate records during processing (i.e., to affect all output files), see No Duplicates, Duplicates Only *on page 168*.

The following example uses `/INCLUDE` to show each distinct Publisher once:

```
/INFILE=Book_Store  
/FIELD= (Title, POSITION=1, SIZE=14)  
/FIELD= (Author, POSITION=15, SIZE=12)  
/FIELD= (Publisher, POSITION=27, SIZE=13)  
/SORT  
/KEY=Publisher  
/OUTFILE=Publisher_Individual  
/FIELD= (Publisher, POSITION=1, SIZE=13)  
/INCLUDE WHERE Publisher
```

Alternate ways to define the condition and `/INCLUDE` would be to replace the `/INCLUDE` line with the following:

```
/CONDITION= (newpub, TEST= (Publisher))  
/INCLUDE WHERE newpub
```

or

```
/CONDITION= (newpub, TEST= (Publisher))  
/INCLUDE= (CONDITION=newpub)
```

Note that records are first sorted by `Publisher`. As each output record is analyzed, the `Publisher` field value is compared to the previous value. When the value changes, the conditional expression becomes true and the record is included in the output file. If the value does not change, the value is false and the record is not included.

Typically in change testing, data is sorted using the named field as a key, but this is not a requirement. The most common use of change tests is for defining `BREAK` points in summary functions. See examples of this when aggregations (summary functions) are discussed in `SUMMARY FUNCTIONS (AGGREGATION)` *starting on page 234*.

21.3 Binary Logical Expressions

Another form of a logical expression involves two values and a *relational operator*. The following is the general form of the expression:

Value1 Relational_Operator Value2

The values can be field names or literals (constants), and *Value2* can contain a repetitive literal string such as `{180}"0"` to indicate 180 zeroes (see `/DATA` on *page 152*). **sortcl** recognizes both the operator and symbol forms of these relational operators, as shown in *Table 10*.

Table 10: Recognized Relational Operators

Operator	Symbol	Meaning
EQ	==	Equal
NE	!=	Not equal
GT	>	Greater than
GE	>= !<	Greater than or equal Not less than
LT	<	Less than
LE	<= !>	Less than or equal Not greater than
CT		Contains
NC		Does not contain

Examples of logical expressions using relational operators are:

```
Author EQ "Publisher"
Publisher >= "Addison-Wesley"
Price > 25.00
Value CT {10}"0"
"S" == $SOURCE
```



The final example shows how you must reverse the standard compare order when an environment variable is used. Note also the difference between `Price > 25.00` (a numeric compare) and `Publisher >= "Addison-Wesley"` (a character compare). The CT example evaluates whether the `Value` field contains 10 zeroes.

21.4 Compound Logical Expressions

The simplest form of a compound logical expression is:

Expression 1 Logical_Operator Expression 2

There are two logical operators:

AND true when both Expression 1 *and* Expression 2 are true

OR true when either Expression 1 is true *or* Expression 2 is true.

An example of a compound logical expression is:

`Publisher == "Dell" AND Price > 25.00`

where `Publisher` and `Price` are previously defined fields.

21.5 Evaluation Order

Any number of conditional expressions can be connected by logical operators, i.e.:

Expr 1 Log Op 1 Expr 2 Log Op 2 Expr 3

Evaluation proceeds left to right. For example, if *Expr 1* is TRUE and *Log Op 1* is OR, then the logical expression is TRUE. All evaluations are shown in *Table 11*.

Table 11: Evaluation of Logical Expressions

Unary/Binary Expression	Logical Operator	Result
True	None	True
False	None	False
True	AND	Continue
False	AND	False

Table 11: Evaluation of Logical Expressions

Unary/Binary Expression	Logical Operator	Result
True	OR	True
False	OR	Continue

21.6 Compound Conditions

It is possible to build conditions so that the logical expression of one condition will contain the name of a previously defined condition linked to the rest of the expression by a logical operator. Since parentheses are not recognized for grouping logical expressions, this is helpful in defining complex logical expressions. The following is an example of how to build these nested conditions:

```

/INFILE=chiefs
  /FIELD= (Name, POSITION=1, SIZE=27)
  /FIELD= (Party, POSITION=40, SIZE=3)
  /FIELD= (State, POSITION=45, SIZE=2)
  /CONDITION= (C1, TEST= (Party EQ "DEM")) # C1 defined
  /CONDITION= (C2, TEST= (Party EQ "REP")) # C2 defined
  /CONDITION= (C3, TEST= (C1 OR C2)) # C3 includes C1 and C2
  /OMIT WHERE C3 # omit DEMs and REPs
/SORT
  /KEY=Name
/OUTFILE=out

```



When defining a WHERE clause, you cannot use AND or OR logic that references multiple condition names. In the above script, for example, you can not have:

```
/OMIT WHERE C1 OR C2
```

This also applies to other instances where you can reference condition names, such as when using BREAK (see SUMMARY FUNCTIONS (AGGREGATION) *on page 234*) and when using IF THEN ELSE logic (see CONDITIONAL FIELD AND DATA STATEMENTS *on page 180*).

Instead, you must first define a new, single condition that contains the compound condition logic you require, such as C3 in the above example, and then reference that single name where required.

Alternatively, you can use explicit compound condition logic without pre-defining conditioning names earlier in the script, for example:

```
/OMIT WHERE Party EQ "DEM" OR Party EQ "REP"
```

22 INCLUDE-OMIT (RECORD SELECTION)

/INCLUDE and /OMIT statements use conditions to accept or reject entire records, respectively. Include and omit conditions use field-value conditions to determine the dispositions of entire records. This functionality can be applied to records for both input filtering and/or output purposes.

22.1 Syntax

The forms of /INCLUDE and /OMIT statements are:

```
/INCLUDE WHERE condition
/INCLUDE= (CONDITION=condition)
/OMIT WHERE condition
/OMIT= (CONDITION=condition)
```

where *condition* is a logical expression or a condition name, as discussed in CONDITIONS *starting on page 169*.

22.2 Include-Omit Evaluation

Care must be given to the order of /INCLUDE and /OMIT statements because records are tested for each /INCLUDE and /OMIT condition in the order specified. If a particular record meets a given /INCLUDE condition, it is included, *regardless* of any remaining /OMIT statements that would otherwise cause that record to be omitted. Alternatively, if a record meets a given /OMIT condition, it is omitted, regardless of any remaining /INCLUDE statements that would otherwise cause that record to be included. That is, once a record satisfies an include or omit condition, its inclusion/exclusion in the output is determined.

For better performance when there are multiple conditions, the most likely /INCLUDE and /OMIT statements should be given early. The sooner the records' dispositions are determined, the fewer conditions are required to be evaluated.

Also consider which section of the script is best for placement of /INCLUDE and /OMIT statements.

- /INFILE
- /INREC
- /OUTFILE

Any records that are not required in the output should be filtered out prior to sorting.

This makes the sort faster by keeping unnecessary records from being processed. Be sure the record will not be required for deriving fields in the output. This can be done in the /INFILE or in the /INREC sections (see /INREC *on page 150*).

22.3 Include-Omit Examples

This section provides examples of includes and omits. The following input file, **chicago_2**, is used in these examples:

5180 On Top	15.95 Harper-Row
3391 Married Young	24.95 Prentice-Hall
8835 Beginnings	8.50 Prentice-Hall
2272 Still There	13.05 Dell
1139 Greater Than	34.75 Valley Kill
1245 Not There	8.49 Dell
3928 Not On Call	9.99 Harper-Row
4877 Going Nowhere	17.95 Valley Kill

Example: Using /INCLUDE with WHERE

This script, **include.scl**, shows an /INCLUDE with an explicit condition using a WHERE clause:

```
/INFILE=chicago_2
/FIELD=(StockNum,POSITION=1,SIZE=5)
/FIELD=(Title,POSITION=6,SIZE=13)
/FIELD=(Price,POSITION=19,SIZE=10,PRECISION=2,NUMERIC)
/FIELD=(Publisher,POSITION=30,SIZE=15)
/INCLUDE WHERE Publisher CT "Dell"
/OUTFILE=chicago1.out
```

The output is:

1245 Not There	8.49 Dell
2272 Still There	13.05 Dell

Example: Using /INCLUDE with Named Conditions

The following script, **include_names.scl**, shows two /INCLUDE statements with named conditions:

```
/INFILE=chicago_2
/FIELD= (StockNum, POSITION=1, SIZE=5)
/FIELD= (Title, POSITION=6, SIZE=13)
/FIELD= (Price, POSITION=19, SIZE=10, PRECISION=2, NUMERIC)
/FIELD= (Publisher, POSITION=30, SIZE=15)
/CONDITION= (OverTen, TEST= (Price GT 10))
/CONDITION= (YesDell, TEST= (Publisher CT "Dell"))
/INCLUDE WHERE OverTen
/INCLUDE WHERE YesDell
/OUTFILE=chicago2.out
```

and produces the following output:

1139 Greater Than	34.75 Valley Kill
1245 Not There	8.49 Dell
2272 Still There	13.05 Dell
3391 Married Young	24.95 Prentice-Hall
4877 Going Nowhere	17.95 Valley Kill
5180 On Top	15.95 Harper-Row

In this case, records are included that satisfy either the `OverTen` condition or the `YesDell` condition. The rest are excluded.

Example: Using Two /OMIT Statements

This script, **omits.scl**, uses two omits. One is used with a named condition in a WHERE clause; the other is used with an explicit logical expression in a WHERE clause.

```
/INFILE=chicago_2
/FIELD= (StockNum, POSITION=1, SIZE=5)
/FIELD= (Title, POSITION=6, SIZE=13)
/FIELD= (Price, POSITION=19, SIZE=10, PRECISION=2, NUMERIC)
/FIELD= (Publisher, POSITION=30, SIZE=15)
/CONDITION= (YesDell, TEST= (Publisher CT "Dell"))
/OMIT WHERE YesDell
/OMIT WHERE Price GT 10
/OUTFILE=chicago3.out
```

The output is:

3928 Not On Call	9.99 Harper-Row
8835 Beginnings	8.50 Prentice-Hall

In this case, records that satisfy either the `YesDell` condition or the `Price GT 10` condition are omitted. The remaining two records are included.

As the previous two examples illustrate, in cases where there are only `/INCLUDE` statements and no `/OMIT` statements, all records that do not satisfy the `/INCLUDE` conditions are discarded. Alternatively, if the script has only `/OMIT` statements, all records that do not satisfy the `/OMIT` conditions are included.

When mixing two or more `/INCLUDE` and `/OMIT` statements, there is the possibility that a record will not satisfy any test. If this happens, the disposition of the record depends on the last test. If the final test was an `/INCLUDE`, the record is omitted. If the final test was an `/OMIT`, the record is included.

To illustrate this concept, the following two examples use the same conditions but in different order. The `OverTen` condition is true when `Price > 10.00`. The condition `YesDell` is satisfied when a book is published by Dell. The following two examples yield different results.

Example: Condition Ordering: `/INCLUDE` first

The following script, `include_first.scl`, will first test the record to see if the condition `OverTen` is true. If it is, the record will be included in the output. If the condition `OverTen` is not true, it is tested for the condition `YesDell`. If this test is true, the record is omitted. If the condition `YesDell` is not true, the record is included because it failed all of the tests and the last test was an `/OMIT`.

```
/INFILE=chicago_2
/FIELD=(StockNum, POSITION=1, SIZE=5)
/FIELD=(Title, POSITION=6, SIZE=13)
/FIELD=(Price, POSITION=19, SIZE=10, PRECISION=2, NUMERIC)
/FIELD=(Publisher, POSITION=30, SIZE=15)
/INCLUDE WHERE Price GT 10
/OMIT WHERE Publisher CT "Dell"
/REPORT
/OUTFILE=chicago4.out
```

The results are:

1139 Greater Than	34.75 Valley Kill
2272 Still There	13.05 Dell
3391 Married Young	24.95 Prentice-Hall
3928 Not On Call	9.99 Harper-Row
4877 Going Nowhere	17.95 Valley Kill
5180 On Top	15.95 Harper-Row
8835 Beginnings	8.50 Prentice-Hall

Example: Condition Ordering: /OMIT first

In the following script, **omit_first.scl**, a record is first tested for the condition `YesDell`, and if true, the record is omitted. If `YesDell` is not true, the next test is applied. If the record passes the condition `OverTen`, it is included in the output. But if it fails the `OverTen` test, the record is omitted because it failed all of the tests and the last test was an `/INCLUDE`.

```

/INFILE=chicago_2
/FIELD=(StockNum, POSITION=1, SIZE=5)
/FIELD=(Title, POSITION=6, SIZE=13)
/FIELD=(Price, POSITION=19, SIZE=10, PRECISION=2, NUMERIC)
/FIELD=(Publisher, POSITION=30, SIZE=15)
/OMIT WHERE Publisher CT "Dell"
/INCLUDE WHERE Price GT 10
/REPORT
/OUTFILE=chicago5.out

```

The output is different for this job because the order of the `/INCLUDE` and `/OMIT` statements is reversed.

The results are:

1139 Greater Than	34.75 Valley Kill
3391 Married Young	24.95 Prentice-Hall
4877 Going Nowhere	17.95 Valley Kill
5180 On Top	15.95 Harper-Row

23 CONDITIONAL FIELD AND DATA STATEMENTS

sortcl permits the value for `/FIELD` statements in `/INREC` and output file descriptions, and the value for `/DATA` statements in output file descriptions to be derived from `IF-THEN-ELSE` logic. When using this logic, the syntax for these statements is:

```
/FIELD=(field-name [, field attributes] [, IF-THEN-ELSE] )  
/DATA=IF-THEN-ELSE
```

The general form for `IF-THEN-ELSE` is:

```
IF condition THEN value1 ELSE value2
```

If the condition is true, the resultant value of the `/FIELD` or `/DATA` statement is *value1*; if the condition is false, the resultant value is *value2*.

A *condition* is a named condition or a logical expression as discussed in *CONDITIONS starting on page 169*. A condition can also use C-style `iscompare` criteria (see *Function Compares in Conditions (iscompares) on page 182*).

A *value* can be any of the following:

- a field name
- a literal (numeric value or character string)
- a repeated string such as `{180}"0"` to indicate 180 zeroes (see */DATA on page 152*)
- an algebraic expression
- a summary value
- value from a set reference (see *Set files on page 198*)
- another `IF-THEN-ELSE` derived value (see *Multiple IF THEN clauses on page 181*)

The following is an example with a `/DATA` and a `/FIELD` statement:

```
/CONDITION=(DT,TEST=(type == "deciduous"))  
/FIELD=(tree, POSITION=2, SIZE=4, IF DT THEN oldtree ELSE "PINE")  
/DATA=IF DT THEN instructions ELSE "none"
```

where `type`, `oldtree`, and `instructions` are previously defined input fields, and `"PINE"` and `"none"` are string literals. Notice that the string must be in double quotes. Also, *value2* can be a recurring constant string such as `{5}" - "` -- which will display five dashes. Recurring constant strings are also accepted within `/DATA` statements (see */DATA on page 152*).

Achieve the same results as above using implicit conditions:

```
/FIELD=(tree,POSITION=2,SIZE=4,IF type == "deciduous" THEN oldtree ELSE "PINE")
/DATA=IF type == "deciduous" THEN instructions ELSE "none"
```

There are two variations on the general form of IF-THEN-ELSE logic:

- IF *condition* THEN *value1*
- IF *condition* ELSE *value2*

The first implies an empty value for the ELSE clause, and the second implies an empty value for the THEN clause. Below are examples:

```
/FIELD=(tree,POSITION=2,SIZE=4,IF type == "deciduous" THEN oldtree)
/DATA=IF type == "deciduous" ELSE "none"
```

These are equivalent to:

```
/FIELD=(tree,POSITION=2,SIZE=4,IF type == "deciduous" THEN oldtree ELSE "")
/DATA=IF type == "deciduous" THEN "" ELSE "none"
```

23.1 Multiple IF THEN clauses

A next level of IF-THEN-ELSE clauses can appear in either or both of the ELSE or THEN clauses. Any number of levels can be defined to meet the degree complexity of a problem.

For example, the following is valid syntax:

```
IF C1 THEN IF C2 THEN V1 ELSE V2 ELSE IF C3 THEN V3 ELSE V4
```

but for clarity should be written:

```
IF C1 \
THEN IF C2 \
      THEN V1 \
      ELSE V2 \
ELSE IF C3 \
      THEN v3 \
      ELSE V4
```

The rule is that each THEN or ELSE clause is associated with the most recent IF that does not already have a THEN or ELSE clause associated with it. The line continuation at the end of each line of the statement is necessary for **sortcl** to evaluate the statement correctly.



When long lists are to be tested for a true condition, processing will be faster if the cases that are most likely to be true are tested first.

In cases where a job script requires multiple THEN or ELSE clauses to return literal strings (that is, known values), it is recommended that you instead use SET column look up capability to perform key-value substitutions, as described in Set files *on page 198*.

23.2 Function Compares in Conditions (*iscompares*)

Use C-style *iscompare* functions in **sortcl** to evaluate conditions at the field level, and also for record-filtering using /INCLUDE and /OMIT statements.

Two categories of iscompare tests are available, and **sortcl** takes machine locale into consideration when evaluating each compare (see /LOCALE *on page 281*):

C-library iscompare tests

The following function compares in **sortcl** are identical to those available to C programmers:

isalphadigit(<i>field</i>)	Equivalent to the C library test named isalnum. True if all characters are alphanumeric characters. This is equivalent to <code>(isalpha(<i>field</i>) isdigit(<i>field</i>))</code> .
isalpha(<i>field</i>)	True if all characters are alphabetic characters, in the current locale. This is equivalent to <code>(isupper(<i>field</i>) or islower(<i>field</i>))</code> . In some locales, there might be additional characters for which <code>isalpha(<i>field</i>)</code> is true, that is, there can be letters that are neither upper-case nor lower-case.
isascii(<i>field</i>)	True if each character is a 7-bit unsigned char value that fits into the ASCII character set.
iscntrl(<i>field</i>)	True if all characters are control characters.
isdigit(<i>field</i>)	True if all characters are digits (0 through 9).
isgraph(<i>field</i>)	True if all characters are printable (except space).
islower(<i>field</i>)	True if all characters are lower-case.
isprint(<i>field</i>)	True if all characters are printable (including space).
ispunct(<i>field</i>)	True if all printable characters, except a space or an alphanumeric.

isspace(<i>field</i>)	Checks for white-space characters, including space, form-feed (\f), newline (\n), carriage return (\r), horizontal tab (\t), and vertical tab (\v).
isupper(<i>field</i>)	True if all characters are upper-case.
isxdigit(<i>field</i>)	Checks for hexadecimal digits, that is, any of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F.



The list of printable characters on Windows operating systems differs from printable characters on Linux and Unix. Refer to your operating system documentation for details.

Additional iscompare tests

The following non-C-library function compares are also available in **sortcl**:

isempty(<i>field</i>)	Returns true for null fields or those that satisfy <code>isspace(<i>field</i>)</code> .
isnumeric(<i>field</i>)	Same as <code>isdigit(<i>field</i>)</code> , but also recognizes period (.), plus (+), and minus (-). At least one character must be a digit.
isebcalpha(<i>field</i>)	True if all characters are EBCDIC alphabetic.
isebcdigit(<i>field</i>)	True if all characters are EBCDIC digits.
isholding(<i>value1</i>,<i>value2</i>)	True if <i>value2</i> is contained within <i>value1</i> . <i>value1</i> and/or <i>value2</i> can be a literal value or a field name, for example <code>isholding(ACCOUNT, " # ")</code> .
ispattern(<i>field</i>, "<i>expression</i>")	Checks the field using Perl-compatible regular expressions such as <code>a+bc</code> .
ispacked(<i>field</i>)	Checks the field to make sure each nibble, except for the last one, contains a 0-9 value, and that the last nibble contains a hex b, c, d, or f.

Example: Using iscompares

Given this data set:

Smith, John	555-1111
Rogers, Tom	55520098
Stone, Allan	
Ross, Margaret	555-4321
Smithson, Mary	
Johnson, Bill	555-0098

The following script, **iscompares.scl**, uses iscompares within both record filter logic (/OMIT) and field-level evaluation in the output:

```
/INFILE=data
  /FIELD= (name, POSITION=1, SIZE=15)
  /FIELD= (phone, POSITION=16, SIZE=8)
  /OMIT WHERE isdigit (phone)
/REPORT
/OUTFILE=stdout
  /FIELD= (name, POSITION=1, SIZE=15)
  /FIELD= (phone, POSITION=16, SIZE=8)
  /FIELD= (flag, POSITION=30, IF isspace (phone) THEN "empty" ELSE "")
```

See also Example: *on page 316* for another example of using iscompare functions.

This produces:

Smith, John	555-1111	
Stone, Allan		empty
Ross, Margaret	555-4321	
Smithson, Mary		empty
Johnson, Bill	555-0098	

The following input record was omitted because the phone number field contains all digits (isdigit):

Rogers, Tom	55520098
-------------	----------

See INCLUDE-OMIT (RECORD SELECTION) *on page 175*.

And, the records that contain empty phone number fields (isspace) were appended with the word *empty* because of the IF THEN ELSE logic in the final output field (see CONDITIONAL FIELD AND DATA STATEMENTS *on page 180*).

23.3 Conditional Field and Data Statement Examples

This section provides examples of using conditional field and conditional data statements. The following input file, **chiefs10h**, is used in these examples:

Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Adams, John Quincy	1825-1829	D-R	MA
Jackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA

and the following metadata (data definition file) is used, **chiefs10h.ddf**:

```
/FIELD=(president, POSITION=1, SIZE=27)
/FIELD=(term, POSITION=28, SIZE=4)
/FIELD=(party, POSITION=40, SIZE=3)
/FIELD=(state, POSITION=45, SIZE=2)
```

Example: Simple /DATA Condition

The following script, **simple_data.scl**, is a simple case of using a /DATA statement to identify the presidents from Virginia.

```
/INFILE=chiefs10h
/SPECIFICATION=chiefs10h.ddf
/OUTFILE=presidents.out
/FIELD=(president, POSITION=1, SIZE=27)
/DATA=IF state == "VA" THEN " Virginian" ELSE " Non-Virginian"
```

The resulting output is:

Adams, John	Non-Virginian
Adams, John Quincy	Non-Virginian
Harrison, William Henry	Virginian
Jackson, Andrew	Non-Virginian
Jefferson, Thomas	Virginian
Madison, James	Virginian
Monroe, James	Virginian
Tyler, John	Virginian
Van Buren, Martin	Non-Virginian
Washington, George	Virginian

Example: Multi-Level Condition Based on One Field

This job script, **multi_one.scl**, uses a multi-level condition for an output field where the test is based on the value of one input field. This is the equivalent of **ELSIF** statements in some other languages.

```
/INFILE=chiefs10h
/SPECIFICATION=chiefs10h.ddf
/OUTFILE=parties.out
/SPECIFICATION=chiefs10h.ddf
/FIELD=(description,POSITION=50,\
IF party == "FED" \
THEN "Federalist" \
ELSE IF party == "WHG" \
THEN "Whig" \
ELSE IF party == "DEM" \
THEN "Democrat" \
ELSE "other")
```

The output is as follows:

Adams, John	1797	FED	MA	Federalist
Adams, John Quincy	1825	D-R	MA	other
Harrison, William Henry	1841	WHG	VA	Whig
Jackson, Andrew	1829	DEM	SC	Democrat
Jefferson, Thomas	1801	D-R	VA	other
Madison, James	1809	D-R	VA	other
Monroe, James	1817	D-R	VA	other
Tyler, John	1841	WHG	VA	Whig
Van Buren, Martin	1837	DEM	NY	Democrat
Washington, George	1789	FED	VA	Federalist

Example: Multi-Level Condition Based on Two Fields

This job script, **multi_two.scl**, uses a multi-level condition for an output field where the test is based on the value of two input fields in some instances, and one input field in other instances. Notice that both THEN IF and ELSE IF are used.

```

/INFILE=chiefs10h
/SPECIFICATION=chiefs10h.ddf
/OUTFILE=parties.out
/SPECIFICATION=chiefs10h.ddf
/FIELD=(description,POSITION=50, \
  IF party == "FED" \
  THEN IF state == "VA" \
    THEN "Virginia Federalist" \
    ELSE "Other Federalist" \
  ELSE IF party == "WHG" \
    THEN IF state == "VA" \
      THEN "Virginia Whig" \
      ELSE "Other Whig" \
    ELSE IF party == "DEM" \
      THEN "Democrat" \
      ELSE "other")

```

The output is as follows:

Adams, John	1797	FED	MA	Other Federalist
Adams, John Quincy	1825	D-R	MA	other
Harrison, William Henry	1841	WHG	VA	Virginia Whig
Jackson, Andrew	1829	DEM	SC	Democrat
Jefferson, Thomas	1801	D-R	VA	other
Madison, James	1809	D-R	VA	other
Monroe, James	1817	D-R	VA	other
Tyler, John	1841	WHG	VA	Virginia Whig
Van Buren, Martin	1837	DEM	NY	Democrat
Washington, George	1789	FED	VA	Virginia Federalist

Example: Multi-Level Condition with Empty Output

This script, **multi_empty.scl**, selects Virginians with their party affiliation. Notice the empty ELSE used for non-Virginians.

```
/INFILE=chiefs10h
/SPECIFICATION=chiefs10h.ddf
/SORT
/KEY=president
/OUTFILE=virginians.out
/SPECIFICATION=chiefs10h.ddf
/FIELD=(description,POSITION=50, \
IF party == "FED" \
THEN IF state == "VA" \
    THEN "Virginia Federalist" \
    ELSE ""\
ELSE IF party == "WHG" \
    THEN IF state == "VA" \
        THEN "Virginia Whig" \
        ELSE ""\
    ELSE IF party == "DEM" \
        THEN IF state == "VA" \
            THEN "Virginia Democrat" \
            ELSE ""\
        ELSE IF state == "VA" \
            THEN "Other Virginian" \
            ELSE "")
```

Below is the output.

Adams, John	1797	FED	MA	
Adams, John Quincy	1825	D-R	MA	
Harrison, William Henry	1841	WHG	VA	Virginia Whig
Jackson, Andrew	1829	DEM	SC	
Jefferson, Thomas	1801	D-R	VA	Other Virginian
Madison, James	1809	D-R	VA	Other Virginian
Monroe, James	1817	D-R	VA	Other Virginian
Tyler, John	1841	WHG	VA	Virginia Whig
Van Buren, Martin	1837	DEM	NY	
Washington, George	1789	FED	VA	Virginia Federalist



If the double quotes are omitted for the null ELSE clauses, the resulting output will differ slightly. If you use quotes, spaces will be padded up to the conditional field. Without the quotes, the end of the record will be at the end of the previous field. This is an issue only when the conditional field is the last field in the record. With delimited fields, the quotes will force a delimiter character prior to the conditional field, forcing a null field for the last field.

Example: Multi-Level Condition with Delimited Fields

This example is similar to the previous one, but uses delimited fields. The input file **chiefs10h_sep** is as below:

```
Washington|George|1789-1797|FED|VA
Adams|John|1797-1801|FED|MA
Jefferson|Thomas|1801-1809|D-R|VA
Madison|James|1809-1817|D-R|VA
Monroe|James|1817-1825|D-R|VA
Adams|John Quincy|1825-1829|D-R|MA
Jackson|Andrew|1829-1837|DEM|SC
Van Buren|Martin|1837-1841|DEM|NY
Harrison|William Henry|1841-1841|WHG|VA
Tyler|John|1841-1845|WHG|VA
```

and the data definition file, **chiefs10h_sep.ddf**, is:

```
/FIELD=(lname, POSITION=1, SEPARATOR='|')
/FIELD=(fname, POSITION=2, SEPARATOR='|')
/FIELD=(term, POSITION=3, SEPARATOR='|')
/FIELD=(party, POSITION=4, SEPARATOR='|')
/FIELD=(state, POSITION=5, SEPARATOR='|')
```

Using the following job script, **delim_cond.scl**:

```
/INFILE=chiefs10h_sep
/SPECIFICATION=chiefs10h_sep.ddf
/SORT
/KEY=lname
/KEY=fname
/OUTFILE=virginians.out
/SPECIFICATION=chiefs10h_sep.ddf
/FIELD=(description, POSITION=6, SEPARATOR=',', \
IF party == "FED" \
THEN IF state == "VA" \
THEN "Virginia Federalist" \
ELSE "" \
ELSE IF party == "WHG" \
THEN IF state == "VA" \
THEN "Virginia Whig" \
ELSE "" \
ELSE IF party == "DEM" \
THEN IF state == "VA" \
THEN "Virginia Democrat" \
ELSE "" \
ELSE IF state == "VA" \
THEN "Other Virginian" \
ELSE "")
```

The resulting output is:

```
Adams|John|1797-1801|FED|MA|
Adams|John Quincy|1825-1829|D-R|MA|
Harrison|William Henry|1841-1841|WHG|VA|Virginia Whig
Jackson|Andrew|1829-1837|DEM|SC|
Jefferson|Thomas|1801-1809|D-R|VA|Other Virginian
Madison|James|1809-1817|D-R|VA|Other Virginian
Monroe|James|1817-1825|D-R|VA|Other Virginian
Tyler|John|1841-1845|WHG|VA|Virginia Whig
Van Buren|Martin|1837-1841|DEM|NY|
Washington|George|1789-1797|FED|VA|Virginia Federalist
```

Notice that the IF-THEN-ELSE logic in the two previous job scripts are the same.

We can rewrite these logical clauses to process more efficiently by taking into account the fact that a true for `state == VA` is more likely. Below is the equivalent logic:

```
IF state == "VA" \
THEN IF party == "WHG" \
    THEN "Virginia Whig" \
    ELSE IF party == "FED" \
        THEN "Virginia Federalist" \
        ELSE "Other Virginian" \
ELSE ""
```

24 FIELD FUNCTIONS

To support more complex data transformation requirements, **sortel** supports the following field-level functions and options at the /INREC or output level of a job script.

Field Length Function *on page 192*

Return the byte length of a specified field.

ASCII Substrings *on page 193*

Identify substrings of an ASCII field, and reposition and recast them.

Replace characters *on page 196*

Replace all occurrences of a character in a string with another.

Format Strings *on page 196*

Provide formatted output.

Universally Unique Identifier *on page 197*

Assign unique identifiers to masked targets for obfuscating, de-identifying or auditing efforts.

URL Encoding *on page 197*

Encode or decode certain characters in a URL.

Set files *on page 198*

Perform table look ups with SET file references, for example, for pseudonymization.

Find and Replace *on page 212*

Perform search-and-replace operations based on input field values. Specify a search string for one or more input fields, and replace each occurrence of that string with a new string (or other field value) that you specify.

Date Intervals *on page 216*

Calculate the number of days between two dates.

Named Fields With a Literal (fixed) Value *on page 218*

Assign a literal fixed value.

De-identification and Re-identification *on page 220*

De-identify one or more fields with a user-specified key, then re-identify fields using the same key.

Custom Field-Level Functions (External Transformations) *on page 223*

Call an external field-level function to manipulate field data values.

For details on supported iscompare functions, see Function Compares in Conditions (iscompares) *on page 182*. For details on supported mathematical operations, see Table 4 *on page 149*.

24.1 Field Length Function

Use the `length` function to return the value of the length of a specified field. This value can be calculated as a derived field in the `/INREC` or `/OUTFILE` section of a script. It can also be used as part of a condition, but it is particularly useful when trying to ascertain the maximum or average length of a variable-length field.

This syntax for the length field function is:

```
/FIELD=(length(literal),...)
```

where `literal` can be a field name or a string value.

Example: Using the Field Length Function

Consider the following input data:

```
McKinley William
Roosevelt Theodore
Taft William H.
Wilson Woodrow
Harding Warren G.
Coolidge Calvin
```

The script, **names.scl**, incorporates several uses of the field length function, including IF THEN ELSE logic which returns overflow characters if the length is greater than 17:

```
/INFILE=names
  /FIELD=(name, POSITION=1, SEPARATOR=',')
/INREC
  /FIELD=(name, POSITION=1, SEPARATOR=',')
  /FIELD=(L=length(name)) # derived field: used for max and average
/OUTFILE=names_length.out
  /DATA="\nMaximum field length = "
  /FIELD=(max_name)
  /DATA="\nAverage field length = "
  /FIELD=(avg_name)
  /MAX max_name from L # maximum field length value
  /AVG avg_name from L # average field length value
/OUTFILE=names_length.out # display detail records
  /FIELD=(new_name, POSITION=1, SEPARATOR=',', IF length(name) > 17 THEN "****" else name)
  /FIELD=(length(name), POSITION=2, SEPARATOR=',') # function used within a field
```


This produces the following report:

```
Coolidge Calvin,15
Harding Warren G.,17
McKinley William,16
***,18
Taft William H.,15
Wilson Woodrow,14

Maximum field length = 18
Average field length = 16
```

24.2 ASCII Substrings

You can identify substrings of an ASCII field, and reposition and recast them as required on output (or in /INREC).

The syntax for using substrings is as follows:

```
/FIELD=(sub_string(field_name or "string",value1,value2),POSITION=...)
```

where:

- *field_name* is a field that was specified on input, where the substring is derived from its field contents
- "*string*" is an ASCII string from which the substring will be taken
- *value1* is the offset. It can be a positive value (the default) to indicate the number of characters from the left of the field (or string) where you want your substring to begin. Or, use a negative value to indicate the number of characters from the right of the field (or string).
- *value2* is the substring length. It is the number of bytes/characters you want to include in the string once your starting point has been determined using *value1*.



value1 and *value2* can be field names provided as shown in the third substring in the following example.

The following example incorporates some of the substring options that are available.

Example: Using ASCII Substrings

Consider the following input data:

```
McKinley,William,7,10
Roosevelt,Theodore,9,1
Taft,William H.,6,5
Wilson,Woodrow,4,2
Harding,Warren G.,5,7
```

The following script, **substring.scl**, incorporates several uses of the substring option:

```
/INFILE=chiefs.float
/FIELD=(president,POSITION=1,SEPARATOR=',')
/FIELD=(first,POSITION=2,SEPARATOR=',')
/FIELD=(term1,POSITION=3,SEPARATOR=',')
/FIELD=(term2,POSITION=4,SEPARATOR=',')
/REPORT
/OUTFILE=out
/FIELD=(president,POSITION=1,SEPARATOR=',')
/FIELD=(first,POSITION=2,SEPARATOR=',')
/FIELD=(term1,POSITION=3,SEPARATOR=',')
/FIELD=(term2,POSITION=4,SEPARATOR=',')
/FIELD=(sub_string(president,3,3),POSITION=30,SIZE=10)
/FIELD=(sub_string(president,-3,3),POSITION=40,SIZE=10)
/FIELD=(sub_string(president,term1,term2),POSITION=48,SIZE=10)
/FIELD=(sub_string("abcdefghijklmnopqrstuvwxyz",-3,3),POSITION=60,SIZE=10)
```

This produces the following:

McKinley,William,7,10	Kin	ley	ey	xyz
Roosevelt,Theodore,9,1	ose	elt	t	xyz
Taft,William H.,6,5	ft	aft		xyz
Wilson,Woodrow,4,2	lso	son	so	xyz
Harding,Warren G.,5,7	rdi	ing	ing	xyz

As shown above:

- The first substring field at position 30 includes the third character in from the left of the president and extends for three bytes.
- The second substring at position 40 uses a negative offset to show that the substring was taken by counting in three from the right-most character of the president field.
- The substring at position 48 shows how field names can be substituted for both the offset and the substring length, as long as these field values are numeric (term1 and term2 in this case).
- The final substring uses the *"string"* option and also features a negative offset.

24.2.1 INSTR

Use INSTR to return a value that is the position of a substring within a field. If the indicated substring is not found, INSTR returns 0.

The INSTR procedure has two, three, or four parameters and follows the Oracle/PLSQL guidelines. For more information, see

http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions068.htm

The syntax for using INSTR is:

```
INSTR( string1, string2 [, start [, occurrence] ] )
```

where:

- *string1* is the string to search.
- *string2* is the substring to be found in *string1*.
- *start* is the position in *string1* where the search starts. If a *start* parameter is not given, *start* defaults to 1. If *start* is negative, INSTR searches *string1* from right to left.
- *occurrence* is the number of occurrences of substring. If omitted, *occurrence* defaults to 1.

The parameters can be field names or literals. The values of the fields are extracted from the data.

Example: Using INSTR

Consider the following input data:

```
DEM|NC|Polk|James K.|1845-1849
WHG|VA|Taylor|Zachary|1849-1850
WHG|NY|Fillmore|Millard|1850-1853
```

The following script, **instring1scl**, finds the location of a delimiter in a record:

```
/INFILE=chiefs3
  /FIELD=(WholeRec, POSITION=1)
/REPORT
/OUTFILE=instring1.out
  /FIELD=(NewPos=INSTR(WholeRec,"|",1,3),POSITION=2, size=3, PRECISION=0,
NUMERIC)
  /FIELD=(WholeRec, POSITION=6)
```

This produces the following:

```
12 DEM|NC|Polk|James K.|1845-1849
14 WHG|VA|Taylor|Zachary|1849-1850
16 WHG|NY|Fillmore|Millard|1850-1853
```

The number at the beginning of each record indicates the position of the third pipe in the field `WholeRec`.

24.3 Replace characters

To define a mask, use the `replace_chars` external field transform function with the following syntax:

To replace all occurrences of a character in a string with another, use the `replace_chars` external field transform function with the following syntax

```
/FIELD=(fieldname=replace_chars(field,"char",start,length),[other attributes])
```

where:

- *fieldname* is a name that is specified on input as the name of the new masked field.
- *field* is the name of the field to be masked.
- *char* is the character to be used for replacement.
- *start* is the numerical position at which the character replacement will start.
- *length* indicates the length of the piece to be replaced.

If the offset is a positive number, the offset is at the left of the string from which to start the mask. If the offset is a negative number, it signifies the offset is from the right, or end, of the string.

24.4 Format Strings

To provide formatted output, use the `format_strings` external field transform function with the following syntax:

```
/FIELD=( format_string [, field [, field [...]]] )
```

The first argument, `format_string`, is required. The format string is copied to the resultant field, but any string placeholders (`%s`) are replaced by the optional arguments, which can be ASCII fields or literal strings. However, any literal text can be given in the format string. Note that a beneficial use of this function is to create a field from a literal value. The advantage of using the transform instead of a data statement is in the ability to name the entire field. This is important for loading database tables, and to be able to easily generate a DDF or DDL from an output section.

24.5 Universally Unique Identifier

Use the Universally unique identifiers (UUIDs) transform function to assign unique identifiers to masked targets for obfuscating, de-identifying or auditing efforts. The syntax is:

```
/FIELD= (UUID=create_uuid()) ...
```

Which produces output similar to:

```
f4046e93-a393-4a2d-8d32-c389982c197b
d1856f14-62be-46f7-8e8b-4de2eb58c140
b13d53a8-5466-43ed-bc49-585a83e02848
478e07b7-f8d9-48b7-897b-be1c3752f91b
20975a92-5516-4893-8d7e-d82738718d96
```

You can create values that look like globally unique identifiers (GUIDs) by adding an optional parameter to `create_uuid()`.

If a string literal is included, the function will use the first character as a frame, or quote, to wrap the UUID. If there are more than one characters in the string, the first and second characters will be used at the start and the end of the UUID respectively, as shown:

```
/FIELD= (UUID2=create_uuid("{ }"))
```

This produces values that look like GUIDs:

```
{84323203-577a-439b-917c-ebbe239d2ed4}
{5648171d-2c66-4d78-b7de-dfa5f8a492c8}
{0e7b144f-4ddc-461b-bc9e-b71f93b4a0fd}
{01149109-391b-4aaa-8967-77728609570b}
{c0afb113-70fd-436a-a4f2-891e636c8b15}
```

24.6 URL Encoding

`encode_url`, per RFC 3986, encodes certain non-alphanumeric characters in a URL by replacing them with one or more character triplets that consist of the percent character "%" followed by two hexadecimal digits. The two hexadecimal digits of the triplet(s) represent the numeric value of the replaced character.

For example, `http://www.iri.com/products/voracity/technical-details` is encoded as:

```
http%3A%2F%2Fwww.iri.com%2Fproducts%2Fvoracity%2Ftechnical-details
```

According to RFC 3986, the characters in a URL have to be taken from a defined set of unreserved and reserved ASCII characters. Any other characters are not allowed in a URL.

`decode_url` returns the original URL.

24.7 Set files

Sortcl fields can have values that are drawn from sets. These values are used extensively in applications involving conversion, encryption, replacement, validation, and so on.

The set uses ASCII columnar values separated by a tab character; rows as records ending with a line-feed character. A set can have one or more columns and one or more rows. Comments preceded by # symbol are allowed in all SET files at the top of the set. When the # symbol is no longer the first character, the set file content begins. Thereafter, if a # appears, it is assumed to be part of the data.



Sets with a small number of rows can be created with a text editor, with the column separator as a tab character and not spaces. Large sets are generally built using database extraction, reformatting, and no-duplicate sorting.

By convention, set files are given the file name extension *.set*. Typically, set files are application independent and can be used in different **sortcl** applications.

sortcl has a fast searching algorithm that expects the left values to be unique and in ascending order. With a two-column set and a single search argument, column one is searched, and **sortcl** returns the value from column two. With two search arguments and a three-column set, **sortcl** matches the first two columns and returns the value in column three. The same is true with additional columns.

You can generate a look up table by extracting the relevant columns from a database, for example. An example of SQL commands that will extract two columns to a text-based, tab-delimited set file is as follows:

```
SQL> spool name_code.set
SQL> select name||' '||userid from personnel;
SQL> spool off
```

where the space between the separators above (||"||) is a manually inserted tab.

You can also use **sortcl** to generate a look up table directly from database column data; for example:

```
/INFILE="personnel;DSN=oracle_tester[;UID=scott;PWD=tiger;]"
```

```

/PROCESS=ODBC
/FIELD= (name, POSITION=1, SEPARATOR=' , ' )
/FIELD= (user_id, POSITION=2, SEPARATOR=' , ' )
/SORT
/KEY=name
/NODUPPLICATES
/OUTFILE=name_code.set
/PROCESS=RECORD
/FIELD= (name, POSITION=1, SEPARATOR=' \t ' )
/FIELD= (user_id, POSITION=2, SEPARATOR=' \t ' )

```

The field look up capability can replace the need for a join if you need only to replace field values with their appropriate table values, and you do not need to match records together from disparate input files as described in */JOIN on page 224*. This capability also replaces the need for multiple IF THEN ELSE statements (see *CONDITIONAL FIELD AND DATA STATEMENTS on page 180*).

The field statement syntax for look ups is:

```
/FIELD=(FieldName, SET=<set_descript>[<control options>] [,other field attributes])
```

where:

- **FieldName** is the name of the field.
- **set_descript** is either an optional path/filename or a user defined value list (any combination of literals, ranges, and field names) enclosed in brackets; for example {5,7,(8,10)}
- **control_options** can include any of the following:
 - **DEFAULT="string"** where string represents the value to be displayed when a search yields no result. If default is not set, an error message results when there is no match for the look up.
 - **ORDER=**
 - PRE_SORTED** The default. set file records are already sorted left-to-right in alpha-numeric order. It is recommended that large set files are maintained in a sorted order so that look up performance is enhanced, and duplicates can be removed. An error message is given if you apply the **PRE_SORTED** option to a set that is not in order.

`NOT_SORTED` Records are sorted internally before processing. (The reordered set is not saved.) For larger, frequently used sets, pre-sorting is recommended.

`CHK_SORTED` The order of the input record is checked. As each input record is read, the join key field is compared against the prior record's field to check for order. If a record is out of order, the join is aborted and error message is given indicating the file name and record number. You can correct the bad record if it can be corrected, or discard or filter the record. Or, the job could be rerun using the `NOT_SORTED` option.

- `SEARCH=`

<code>=</code>	<code>EQ</code>	exact match to the argument
<code>></code>	<code>GT</code>	value in table after the argument
<code>>=</code>	<code>GE</code>	exact match if found, otherwise later value
<code><</code>	<code>LT</code>	value in table before the argument
<code><=</code>	<code>LE</code>	exact match if found, otherwise earlier value

For all non-equal operators, see Slowly Changing Dimensions Reporting *on page 208*.

- `SELECT=`

`ANY` The default. Values from the specified SET file are selected at random. Therefore, repetition of field values is possible using this default option, as well as the omission of values. The amount of repeated or omitted SET values depends on the number of rows being generated and the number of entries that exist in the SET file.

`ALL` If `ALL` is specified, selection begins at the top entry of the SET file, and continues downward through the file, in order, with each new row that is generated. Depending on the value of `/INCOLLECT` (see `/INCOLLECT` on page 61), if all SET files entries are utilized, the selection process beings again at the top, and this process is repeated.

`ONCE` If `ONCE` is specified, selection begins at the top entry of the SET file, and continues downward through the file, in order, with each new row that is generated. Depending on the value of `/INCOLLECT` (see `/INCOLLECT` on page 61), once all SET files entries are utilized, the selection process ends, and any remaining rows to be generated contain an empty value for that field.

ROW	If ROW is specified, the indicated column (determined in the argument) in a particular row or record from a SET file is retrieved.
PERMUTE	If PERMUTE is specified, all possible unique values are included when using PROCESS=RANDOM.

- **SetFile** is the set file name.
- **param list** - Bracket characters ([]) are used to contain a list of comma-separated search arguments. An argument is a literal (quoted string) or a field name. Look up values can be a literal or a value of a field from either the input or inrec.
- **def chars** is a quoted character string to be returned if the search fails.
- **other field attributes** are used to complete your field definition with other attributes, such as POSITION, SIZE, and data type.



The pre-9.5.2 version of applying these attributes without requiring the attribute= convention is still supported. For example:

```
/FIELD=(new_id,NOT_SORTED ONCE SET=c:\sets\id_list.set,POS=1,SEP='|')
```

Set File Conventions

A set file contains rows of tab-delimited entries, where values are either of an ASCII or EBCIDC data type. In a two-column set file, the value on the left is looked up, and a replacement for that value is taken from its adjacent entry. For example, consider the following two-column set file:

```
New York    Albany
California  Sacramento
```

In this case, the value Albany replaces the output (or /INREC) field value that would otherwise contain New York, and the value Sacramento replaces the output (or /INREC) field value that would otherwise contain California.

In a three-column set file, an additional level of dependency exists, where the middle column value replaces the look up value from the left, and the third column is used to replace the value from the middle column (see Example: *on page 206*).

Specify a custom string to be returned for an entry or entries in your set file that does not exist. For example, consider this excerpted set file, **roman.set**:

```
1  I
2  II
3
4  IV
```

Note that one entry is blank. Within the **sortcl** script, specify a custom entry to be returned for this empty field using the **DEFAULT** option in an **/INREC** or **/OUTFILE** field, for example:

```
/FIELD=(roman,SET=roman.set[value] default="no value",POSITION=20)
```

In this case, when the value 2 is looked up to return its Roman numeral equivalent, the string `no value`, rather than an empty field, will be displayed at position 20 in the record. Note that you can also use format characters inside the quoted string. For example, to return a tab followed by `no value`, specify:

```
default="\tno value"
```

24.7.1 Using Multi-Column Set Files

For values that depend on more than one parameter, your look up set files will have more than two columns.

Example: Substituting Sensitive Data Values

Use look up functionality to substitute the values of one or more fields in order to prevent the display of sensitive data; subsequently restore the substituted values with their original contents.



See De-identification and Re-identification *on page 220* for details on the field functions `de_identification` and `re_identification`, which are used for the random jumbling and subsequent restoring of field values based on an encryption-style key. The set file method of substituting values described in this example uses a look up table to restore the original field contents.

This example shows how to substitute sensitive field values and create a look up table in one job script, and then restore the original values by referring to this look up table in a subsequent job script.

Consider the following input data, **Sensitive**, which contains names and salary figures:

Jones, Alan	125000
Smith, Abraham	67000
Guinness, Michael	122650
Schwartz, Francis	78234
Jones, Rick	67234
Williams, Billie	87342

The following script, **substitute.scl**, produces two output files -- one which substitutes values from the name field, and another which creates a look up table for the purposes of restoring the name field in a subsequent job:

```
/INFILE=Sensitive          # input file with sensitive data
/FIELD=(name,POSITION=1,SIZE=20)
/FIELD=(salary,POSITION=21,SIZE=6)
/REPORT
/OUTFILE=Operational       # substituted data file
/FIELD=(sequencer,POSITION=1,SIZE=6,FILL='0')
/FIELD=(salary,POSITION=21,SIZE=6)
/OUTFILE=Secret_Table.set  # used to restore later
/FIELD=(sequencer,POSITION=1,SEPARATOR='\t',SIZE=6,FILL='0')
/FIELD=(name,POSITION=2,SEPARATOR='\t') # tab-delimited
```

This produces **Operational**:

000001	125000
000002	67000
000003	122650
000004	78234
000005	67234
000006	87342

It also produces the tab-delimited set file **Secret_Table.set**:

000001	Jones, Alan
000002	Smith, Abraham
000003	Guinness, Michael
000004	Schwartz, Francis
000005	Jones, Rick
000006	Williams, Billie

The output file **Operational** can be viewed and analyzed by those who do not have the authority to view real names. The sequenced values on the left represent actual names, and look up values are kept in **Secret_Table.set**.



This example shows one method for generating a look up table. DBMS users can export tab-delimited tables to be used as set files, for example.

The next script, **restore.scl**, restores the salary figures to their original names, using table look up functionality:

```
/INFILE=Operational      # the substituted values file as input
  /FIELD= (code, POSITION=1, SIZE=6)
  /FIELD= (salary, POSITION=21, SIZE=6)
/REPORT
/OUTFILE=Restored        # the restored values file as output
  /FIELD= (name, POSITION=1, SIZE=20, SET=Secret_Table.set [code])
  /FIELD= (salary, POSITION=21, SIZE=6)
```

The name field on output returns the look up equivalent to the code value found in the set file.

This produces **Restored**:

Jones, Alan	125000
Smith, Abraham	67000
Guinness, Michael	122650
Schwartz, Francis	78234
Jones, Rick	67234
Williams, Billie	87342

As shown above, the original names have been restored based on the codes found in **Secret_Table.set**.

Example: Pseudonymization

Table look up functionality can be used for the purposes of name masking.

Consider the following input data, **Sensitive**, which contains names and salary figures:

Jones, Alan	125000
Smith, Abraham	67000
Guinness, Michael	122650
Schwartz, Francis	78234
Jones, Rick	67234
Williams, Billie	87342

Consider the following set file, **pseudo.set**, which is sorted and contains pseudonyms for the real names:

Guinness, Michael	O'Connor, Sean
Jones, Alan	Brown, James
Jones, Rick	Janks, Ron
Schwartz, Francis	Fine, Macy
Smith, Abraham	Chase, Gerald
Williams, Billie	Walters, Barney

The following script, **pseudo.scl**, performs pseudonymization on the name field, according to the look up values from **pseudo.set**:

```
/INFILE=Sensitive
  /FIELD=(name, POSITION=1, SIZE=20)
  /FIELD=(salary, POSITION=21, SIZE=6)
/SORT
  /KEY=salary
/OUTFILE=pseudo_sal.out
  /FIELD=(changed_name, POSITION=1, SIZE=20, SET=pseudo.set [name] default="XX")
  /FIELD=(salary, POSITION=21, SIZE=6)
```

This produces **pseudo_sal.out**:

Chase, Gerald	67000
Janks, Ron	67234
Fine, Macy	78234
Walters, Barney	87342
O'Connor, Sean	122650
Brown, James	125000

Example: Multiple Column Set File Example

This example demonstrates how to perform look ups on a multi-column set file, where an additional level of dependency between fields is required. In this example, a small sample of US Presidents is used, and depending on which State they are from, the official state flower, bird, and tree for that state is provided on output.

Consider the following input file, **some_chiefs**:

Roosevelt, Theodore	1901-1909	REP	NY
Taft, William H.	1909-1913	REP	OH
Wilson, Woodrow	1913-1921	DEM	VA
Harding, Warren G.	1921-1923	REP	OH
Coolidge, Calvin	1923-1929	REP	VT
Hoover, Herbert C.	1929-1933	REP	IA
Roosevelt, Franklin D.	1933-1945	DEM	NY
Truman, Harry S.	1945-1953	DEM	MI

The following tab-delimited set file, **state_info.set**, is used to provide additional State-dependent information:

IA	bird	eastern goldfinch
IA	flower	wild prairie rose
IA	tree	oak
MI	bird	robin
MI	flower	apple blossom
MI	tree	eastern white pine
NY	bird	bluebird
NY	flower	rose
NY	tree	sugar maple
OH	bird	cardinal
OH	flower	scarlet carnation
OH	tree	buckeye
VT	bird	hermit thrush
VT	flower	red clover
VT	tree	sugar maple

The following script, **state_info.scl**, demonstrates multi-column dependencies, where the value of the set file's third column (which is displayed on output) is dependent on its counterparts in the preceding columns:

```

/INFILE=some_chiefs
  /FIELD=(President,POSITION=1,SIZE=27)
  /FIELD=(Term,POSITION=28,SIZE=9)
  /FIELD=(Party,POSITION=40,SIZE=3)
  /FIELD=(State,POSITION=45,SIZE=2)
/REPORT
/OUTFILE=state_info.out
  /HEADREC="President          State   Flower          Tree
Bird\n-----\n"
  /FIELD=(President,POSITION=1,SIZE=27)
  /FIELD=(State,POSITION=26,SIZE=2)
  /FIELD=(Flower,SET=state_info.set[State,"flower"] default="Unknown",\
    POSITION=33,SIZE=20)
  /FIELD=(Tree,SET=state_info.set[State,"tree"] default="Unknown",\
    POSITION=53,SIZE=20)
  /FIELD=(Bird,SET=state_info.set[State,"bird"] default="Unknown",\
    POSITION=73,SIZE=20)

```

The produces **state_info.out**, where the displayed Flower, Tree, and Bird field values are dependent on both the value of the set file's first column (State) and the literal string given ("flower", "tree", or "bird") for the second column:

President	State	Flower	Tree	Bird
Roosevelt, Theodore	NY	rose	sugar maple	bluebird
Taft, William H.	OH	scarlet carnation	buckeye	cardinal
Wilson, Woodrow	VA	Unknown	Unknown	Unknown
Harding, Warren G.	OH	scarlet carnation	buckeye	cardinal
Coolidge, Calvin	VT	red clover	sugar maple	hermit thrush
Hoover, Herbert C.	IA	wild prairie rose	oak	eastern goldfinch
Roosevelt, Franklin D.	NY	rose	sugar maple	bluebird
Truman, Harry S.	MI	apple blossom	eastern white pine	robin

The values provided for Virginia (VA) were returned as Unknown because the option `default="Unknown"` was added to the set file attribute to determine the output displayed for non-matches, and entries for VA are not present in the set file (see Set File Conventions *on page 201*).



The set file in this example contains a small amount of sorted rows. However, for unsorted set files, it is recommended that you pre-sort them from left-to-right in a prior job pass (with the `/NODUPLICATES` option, if applicable). Otherwise, you must use the `NOT_SORTED` option to force a set file sort in the look up job itself (which is more time consuming with a large set file than sorting it in advance). The syntax for using the `NOT_SORTED` option in this case would be:

```
/FIELD=(Flower,SET=NOT_SORTED \
state_info.set[State,"flower"],POSITION=33,SIZE=20)
```

24.7.2 Slowly Changing Dimensions Reporting

To report on Slowly Changing Dimensions (SCDs), you need to derive data that is dependent on a changing dimension. The independent dimension is usually time, but can be anything that is strictly increasing. The interval of change does not need to be regular.

Slowly Changing Dimensions reporting is an application of the fuzzy logic feature that finds the value for a search argument that is closest to a set entry. As an example, consider a product set file that contains non-uniform intervals of increasing dates. Each date is associated with the price change of the product. Whether or not you have an entry for a specific date, the SCD search can find the price of the product before, on, or after that date.

Example: Two-column SCD reporting

This example demonstrates how to search a table with slowly changing dimensions using a two-column set file on a single parameter.

The following tab-delimited set file, **butter.set**, provides the slowly changing dimensions of the price of butter on specific days:

2012:05:14	2.95
2012:05:21	3.05
2012:05:28	2.90

Consider the following input file, **days.in**, which contains days in `yyyymmdd` format:

```
2012:05:19
2012:05:11
2012:05:22
2012:05:28
2012:06:10
```

The following script, **butterprice.scl**, demonstrates finding the active price for a specific date:

```
/INFILE=days.in
  /FIELD=(Day, POSITION=1, SIZE=10)
/REPORT
/OUTFILE=buttercost.out
  /FIELD=(Day, POSITION=1, SIZE=10)      # gets the Day from the input file
  /FIELD=(Price, set LE butter.set [Day] default = "XX", POSITION=14, SIZE=4)
```

This produces **buttercost.out**, which lists the effective cost for the specific date that was searched.

2012:05:19	2.95	
2012:05:11	XX	(Notice that no price exists before 2012:05:14)
2012:05:22	3.05	
2012:05:28	2.90	
2012:06:10	2.90	

Example: Four-column SCD reporting

This example demonstrates how to perform look ups on slowly changing dimensions using a four-column set file and three parameters.

The following tab-delimited set file, **metals.set**, is a record of changes of the prices of the metals:

common	Gold	03/24	919
common	Gold	03/27	921
common	Gold	04/20	918
common	Jade	03/24	51
common	Jade	03/27	52
common	Jade	04/20	51
common	Topaz	03/24	21
common	Topaz	03/27	22
common	Topaz	04/20	24
common	Topaz	04/20	26
rare	Gold	03/24	948
rare	Gold	04/20	939
rare	Jade	03/24	71
rare	Jade	03/29	72
rare	Jade	04/20	71
rare	Topaz	03/24	61
rare	Topaz	03/27	62
rare	Topaz	04/20	66

Consider the following input file, **jewel_transactions**:

common	Gold	04/10
rare	Gold	04/10
common	Gold	04/15
rare	Gold	03/27
common	Gold	03/27
common	Topaz	04/10
common	Jade	04/10
rare	Jade	04/10
common	Jade	03/27
rare	Topaz	04/10
rare	Jade	04/12
rare	Jade	03/27

The following script, **metals.scl**, returns the known value for a specific date, for each metal and for its purity, based on the operator.

```
/INFILE=jewel_transactions
/FIELD=(purity,POSITION=1,SEPARATOR='\t')
/FIELD=(metal, POSITION=2,SEPARATOR='\t')
/FIELD=(date, POSITION=3,SEPARATOR='\t')
/REPORT
/OUTFILE=metals.out
/HEADREC="\npurity    metal    date    LT    LE    EQ    NE    GT    GE\n\n")
/FIELD=(purity,POSITION=1,SIZE=6)
/FIELD=(metal,POSITION=12,SIZE=5)
/FIELD=(date,POSITION=20,SIZE=5)
/FIELD=(before,POSITION=28,SIZE=3,PRECISION=0,type=NUMERIC,set LT metals.set [purity,metal,date] \
    default = "xxx")
/FIELD=(on_before,POSITION=34,SIZE=3,PRECISION=0,type=NUMERIC,set LE metals.set [purity,metal,date] \
    default = "xxx")
/FIELD=(exact,POSITION=40,SIZE=3,PRECISION=0,type=NUMERIC,set EQ metals.set [purity,metal,date] \
    default = "xxx")
/FIELD=(not_exact,POSITION=46,SIZE=3,PRECISION=0,type=NUMERIC,set NE metals.set [purity,metal,date] \
    default = "xxx")
/FIELD=(beyond,POSITION=52,SIZE=3,PRECISION=0,type=NUMERIC,set GT metals.set [purity,metal,date] \
    default = "xxx")
/FIELD=(on_beyond,POSITION=58,SIZE=3,PRECISION=0,type=NUMERIC,set GE metals.set [purity,metal,date] \
    default = "xxx")
```

This produces **metals.out**:

purity	metal	date	LT	LE	EQ	NE	GT	GE
common	Gold	04/10	921	921	xxx	xxx	918	918
rare	Gold	04/10	948	948	xxx	xxx	939	939
common	Gold	04/15	921	921	xxx	xxx	918	918
rare	Gold	03/27	948	948	xxx	xxx	939	939
common	Gold	03/27	919	921	921	xxx	918	921
common	Topaz	04/10	22	22	xxx	xxx	24	24
common	Jade	04/10	52	52	xxx	xxx	51	51
rare	Jade	04/10	72	72	xxx	xxx	71	71
common	Jade	03/27	51	52	52	xxx	51	52
rare	Topaz	04/10	62	62	xxx	xxx	66	66
rare	Jade	04/12	72	72	xxx	xxx	71	71
rare	Jade	03/27	71	71	xxx	xxx	72	72

Note that for a particular date, for each metal, and for its purity, the known value was returned based on the operator.

The default string 'xxx' is shown because the option default string was placed in the field statement to give value in the case of an unsuccessful table search.

24.8 Find and Replace

sortcl's find and replace feature produces a derived output field value in the /INREC or output record by examining the source field and replacing every occurrence of a specified search string with the specified replace string. Both the search and replace string can be a field value or a literal string, and you can use find and replace statements for one or more output (or /INREC) fields.

The syntax is:

```
/FIELD=(find_and_replace(source_field,search_string,replace_string or  
field_name),other_attributes)
```

For example, consider an input field that is defined as follows:

```
/FIELD=(Pres, POSITION=1, SIZE=25)
```

If the value of Pres is:

```
Kennedy
```

then the value produced to /OUTPUT (or /INREC) is as follows, depending on what you specify:

```
FIND_AND_REPLACE(Pres, "en", "EN") will produce KENnedy
```

```
FIND_AND_REPLACE(Pres, "en", "1234") will produce K1234nedy
```

```
FIND_AND_REPLACE(Pres, "e", "") will produce Knndy
```

If the value of the field Pres2 is Johnson while the value of Pres is Kennedy, for example, you could specify:

```
FIND_AND_REPLACE(Pres, "K", Pres2) which will produce Johnsonennedy.
```



If the intended find or replace value is a double quote ("), then you must escape the string itself with a backslash, for example:

```
/FIELD=(find_and_replace(f2, "\"", "\"*\"), POSITION=1,  
SEPARATOR=',', FRAME='\"')  

```

Find and Replace: 2string Functions

As described in Function Compares in Conditions (iscompares) on page 182, evaluate field contents with C-language iscompare functions -- to determine if field contents conform to particular criteria (such as being comprised of all digits). With 2string functions, identify one or more characters within a field that does not conform to a specified criterion, and replace each of the non-conforming characters with a user-defined string, or with the value of another field from the record.

The 2string function is applied in the /INREC or output record. The find string is one of several iscompare-style conditions, and the replace string can be a literal string or a field value. The available 2string functions are:

```
non_print2s(field1,"string" or field2)
    Replaces all non-printable characters in field1 with the specified
    string or value from field2.
```

```
non_alpha2s(field1,"string" or field2)
    Replaces all non-alphabetic characters (in the current locale) in
    field1 with the specified string or value from field2.
```

```
non_alnum2s(field1,"string" or field2)
    Replaces all non-alphabetic characters (in the current locale) and
    non-digits in field1 with the specified string or value from field2.
```

```
non_ascii2s(field1,"string" or field2)
    Replaces all non-7-bit unsigned char values (from the ASCII
    character set) in field1 with the specified string or value from
    field2.
```

```
non_digit2s(field1,"string" or field2)
    Replaces all non-digits in field1 with the specified string or value
    from field2.
```

```
non_asc_let2s(field1,"string" or field2)
    Replaces all non-ASCII characters in field1 with the specified
    string or value from field2.
```

```
non_ebc_let2s(field1,"string" or field2)
    Replaces all non-EBCDIC characters in field1 with the specified
    string or value from field2.
```

The syntax is, for example:

```
/FIELD=(non_print2s(source_field,replace_string or field_name),POSITION=...)
```



The list of printable characters on Windows operating systems differs from printable characters on Linux and Unix. Refer to your operating system documentation for details.

Be sure to size the 2string field appropriately to accommodate a larger field if one or more characters may be replaced by a string of more than one byte. Using a 2string Function

Consider the following data which contains some non-printable characters in the second field, salary:

Jones, Alan	12~000	130-95-8765
Smith, Abraham	67000	211-28-3698
Guinness, Michael	1226~0	123-34-8484
Schwartz, Francis	78234	456-56-3456
Jones, Rick	67234	222-22-4444
Williams, Billie	8734~	123-22-2222

The following script, **2string.scl**, converts all non-conforming characters in the salary field with an asterisk (*):

```
/INFILE=2string_data      # file containing non-conforming characters
/FIELD=(name,POSITION=1,SIZE=20)
/FIELD=(salary,POSITION=21,SIZE=6)
/FIELD=(ssn,POSITION=30,SIZE=11)
/REPORT
/OUTFILE=salaries_fixed  # output file with converted field data
/FIELD=(name,POSITION=1,SIZE=20)
/FIELD=(non_print2s(salary,"*"),POSITION=21)
# replaces non-printable characters with *
/FIELD=(ssn,POSITION=30,SIZE=11)
```

This produces **salaries_fixed**, which replaces the non-printable characters from the **2string_data** file with an asterisk:

Jones, Alan	12*000	130-95-8765
Smith, Abraham	67000	211-28-3698
Guinness, Michael	1226*0	123-34-8484
Schwartz, Francis	78234	456-56-3456
Jones, Rick	67234	222-22-4444
Williams, Billie	8734*	123-22-2222

Find and Replace: Case Transfer Functions

The following case transfer functions operate on the alphabetic characters of an ASCII string. The string can be either a field value or a quoted literal. The incoming case is not considered. The case transfer functions are as follows:

- | | |
|---------------------------|--|
| <code>toupper (S)</code> | The resultant string will contain all upper-case letters. |
| <code>tolower (S)</code> | The resultant string will contain all lower-case letters. |
| <code>toproper (S)</code> | The resultant string will contain both upper- and lower-case letters. The following are converted to upper-case: <ul style="list-style-type: none"> • the first character of <i>S</i> • any character that follows an apostrophe (') • any character that follows Mc • any character that follows a tab or other white space character |

All other characters will be converted to lower case.



Characters that are not letters will not be converted.

The syntax is, for example:

```
/FIELD=(toupper(field or "string"),POSITION=...)
```

where *field* is the field name whose contents will be converted, and "*string*" is a literal string value to be converted.

Example: Using All Case Transfer Functions

Consider the following input data:

```
Jones, Alan
Smith, Abraham
Guinness, Michael
Schwartz, Francis
D'angelo, Rick
Mcwilliams, Billie
```

The following script, **case.scl**, converts the names to upper-case and lower-case characters, and also applies the `toproper()` function. Results are displayed in three separate columns:

```
/INFILE=names_data
  /FIELD=(name, POSITION=1, SIZE=20)
/REPORT
/OUTFILE=changed_case      # output file with converted cases
  /FIELD=(toupper(name), POSITION=1, SIZE=20) # all upper-case
  /FIELD=(tolower(name), POSITION=21, SIZE=20) # all lower-case
  /FIELD=(toproper(name), POSITION=41, SIZE=20) # proper case
```

This produces **changed_case**:

JONES, ALAN	jones, alan	Jones, Alan
SMITH, ABRAHAM	smith, abraham	Smith, Abraham
GUINNESS, MICHAEL	guinness, michael	Guinness, Michael
SCHWARTZ, FRANCIS	schwartz, francis	Schwartz, Francis
D'ANGELO, RICK	d'angelo, rick	D'Angelo, Rick
MCWILLIAMS, BILLIE	mcwilliams, billie	McWilliams, Billie

24.9 Date Intervals

Calculate the number of days between any two dates (the interval), provided that the dates are in a supported DATESTAMP format either on input, or converted to a supported DATESTAMP type in the `/INREC` section of the job script. You can also use a mathematical expression to derive a new date value from an existing one.

Example: Calculating Date Intervals

For example, consider these date pairs as input:

```
1999-12-31,2000-12-31
2004-07-01,2004-01-01
1997-06-14,2004-06-14
2004-05-21,2003-02-28
2004-02-28,2001-02-28
1905-11-29,1992-05-31
1987-10-29,1911-09-10
```

The following script, **date_interval.scl**, calculates the interval between the date pairs shown above, and also derives a new date value:

```
/INFILE=dates
  /FIELD=(date1, POSITION=1, SEPARATOR=', ', ISO_DATE)
  /FIELD=(date2, POSITION=2, SEPARATOR=', ', ISO_DATE)
/REPORT
/OUTFILE=date_out
  /FIELD=(date1, POSITION=1, SEPARATOR=', ', ISO_DATE)
  /FIELD=(date2, POSITION=2, SEPARATOR=', ', ISO_DATE)
  /FIELD=(abs(date1 - date2), POSITION=3, SEPARATOR=', ') # interval
  /FIELD=(date1 + 10, POSITION=3, SEPARATOR=', ', ISO_DATE) # new date
```

This produces the following:

```
1999-12-31,2000-12-31,366,2000-01-10
2004-07-01,2004-01-01,182,2004-07-11
1997-06-14,2004-06-14,2557,1997-06-24
2004-05-21,2003-02-28,448,2004-05-31
2004-02-28,2001-02-28,1095,2004-03-09
1905-11-29,1992-05-31,31595,1905-12-09
1987-10-29,1911-09-10,27808,1987-11-08
```

As shown above, absolute value (abs) was used to return a positive integer as the date interval, regardless of whether the earlier date in the pair appears first in the input record. The final field reflects the addition of 10 days added to the date1 field.

24.10 Named Fields With a Literal (fixed) Value

You can assign a literal string or functional value to a named field in input, inrec, or output.

A literal value is a numeric value or a quoted string; a numeric value can also be an expression enclosed in parenthesis. The field is evaluated once and is not re-assignable.

The syntax for assigning a value to a named field is:

```
/FIELD= (string=fieldname)  
/FIELD= (string=expression)  
/FIELD= (animal="dog")  
/FIELD= (amount=123)
```

Example: Using a Literal Value

Use a literal statement to overwrite part of a field. For example, consider the following input:

Replace an existing value with a fixed, literal string value.

With **chiefs** as the input file, the following script, **chiefs_lstrings.scl**, replaces an existing field value with a fixed literal string value:

```
/INFILE=chiefs  
/FIELD= (president, POSITION=1, SIZE=26)  
/FIELD= (service, POSITION=28, SIZE=9)  
/FIELD= (party, POSITION=40, SIZE=3)  
/FIELD= (state, POSITION=45, SIZE=2)  
/INCLUDE WHERE service GT 1900  
/SORT  
/KEY=party  
/KEY=president  
/OUTFILE=chiefs_ls.out  
/HEADREC="Presidents after 1900. \n"  
/FIELD= (president, POSITION=1, SIZE=26)  
/FIELD= (party="N/A", POSITION=28, SIZE=3)  
/FIELD= (state, POSITION=33, SIZE=2)
```

This produces **chiefs_ls.out**:

Presidents after 1900.		
Carter, James E.	N/A	GA
Clinton, William J.	N/A	AR
Johnson, Lyndon B.	N/A	TX
Kennedy, John F.	N/A	MA
Obama, Barack H.	N/A	IL
Roosevelt, Franklin D.	N/A	NY
Truman, Harry S.	N/A	MI
Wilson, Woodrow	N/A	VA
Bush, George H.W.	N/A	TX
Bush, George W.	N/A	TX
Coolidge, Calvin	N/A	VT
Eisenhower, Dwight D.	N/A	TX
Ford, Gerald R.	N/A	NB
Harding, Warren G.	N/A	OH
Hoover, Herbert C.	N/A	IA
Nixon, Richard M.	N/A	CA
Reagan, Ronald W.	N/A	IL
Roosevelt, Theodore	N/A	NY
Taft, William H.	N/A	OH

Note that the / INCLUDE statement accepts only presidents serving after 1900, and the fixed literal string replaces the party field on each record.

24.11 De-identification and Re-identification

De-identify the contents of one or more output fields -- using an encryption-style *key* -- in cases where field data is of a sensitive nature and must be obscured. In a subsequent **sortcl** job, re-identify the field values to restore their original contents.



The algorithm for the **sortcl** de-identification function is based on bit manipulation. It is not intended for encryption purposes. To utilize Advanced Encryption Standard (AES) algorithms on your data, see ASCII/ALPHANUM ENCRYPTION AND DECRYPTION *on page 334*.

The de-identification key used by **sortcl** is a user-specified string, where the same string is used for subsequent re-identification, if required. The hexadecimal range of the data contents to be de-identified must fall within the ASCII printable character range (see ASCII COLLATING SEQUENCE *on page 654*).

The syntax for de-identification is:

```
/FIELD=(de_identify(field_name, "string") , other_attributes)
```

where *field_name* is the field to be de-identified, and *string* is any string that functions as the de-identification key.

The syntax for re-identification is:

```
/FIELD=(re_identify(field_name, "string") , other_attributes)
```

where *field_name* is the field to be re-identified, and *string* is the same string that was used for de-identification.

Example: De-identifying Data

Consider the file **sensitive_names**, which contains names, salaries and social security numbers, where the name field is to be de-identified:

Jones, Alan	125000	130-95-8765
Smith, Abraham	67000	211-28-3698
Guinness, Michael	122650	123-34-8484
Schwartz, Francis	78234	456-56-3456
Jones, Rick	67234	222-22-4444
Williams, Billie	87342	123-22-2222

The following script, **de_ident.scl**, sorts the data by salary, and de-identifies the name field (remapping it to the last column):

```
/INFILE=sensitive_names    # file containing a field to de-identify
  /FIELD=(name, POSITION=1, SIZE=20)
  /FIELD=(salary, POSITION=21, SIZE=6)
  /FIELD=(ssn, POSITION=30, SIZE=11)
/SORT
  /KEY=salary
/OUTFILE=de_identified    # output file with de-identified field
data
  /FIELD=(salary, POSITION=1, SIZE=6)
  /FIELD=(ssn, POSITION=10, SIZE=11)
  /FIELD=(de_identify(name, "12345678"), POSITION=25, SIZE=20)
    # key is 12345678
```

This produces the file **de_identified**:

67000	211-28-3698	Oy2Fd(K=^n*d*yKKKKKKK
67234	222-22-4444	xD}q; (K 2+3KKKKKKKKKK
78234	456-56-3456	O+dL*nFv(Krn*}+2;KKK
87342	123-22-2222	,2>>2*y; (Kp2>>2qKKKK
122650	123-34-8484	QU2}q;; (KY2+d*q>KKKK
125000	130-95-8765	xD}q; (K=>*}KKKKKKKKKK

The name field has been de-identified. It is unreadable and the salary field is sorted in ascending order.

Example: Re-identifying Data

The following script, **re-ident.scl**, reads the sorted **de_identified** file (created in the previous example) and returns the original name field values on output:

```
/INFILE=de_identified # file with de-identified field data
  /FIELD=(salary, POSITION=1, SIZE=6)
  /FIELD=(ssn, POSITION=10, SIZE=11)
  /FIELD=(name, POSITION=25, SIZE=20)
/REPORT
/OUTFILE=re_identified # output file with re-identified field data
  /FIELD=(re_identify(name, "12345678"), POSITION=1, SIZE=20) # same
key
  /FIELD=(salary, POSITION=21, SIZE=6)
  /FIELD=(ssn, POSITION=30, SIZE=11)
```

This produces the following:

Smith, Abraham	67000	211-28-3698
Jones, Rick	67234	222-22-4444
Schwartz, Francis	78234	456-56-3456
Williams, Billie	87342	123-22-2222
Guinness, Michael	122650	123-34-8484
Jones, Alan	125000	130-95-8765

As shown above, the original name field is restored because it was re-identified with the original de-identification key.

24.12 Custom Field-Level Functions (External Transformations)

Invoke a custom field-level function that can be loaded into **sortcl** at runtime from a **.dll** (Windows) or a **.so** (Unix or Linux) to modify field values in ways that are not natively supported by **CoSort**.



For complete details on writing a custom procedure that can be invoked by **sortcl**, see CUSTOM TRANSFORMS chapter on page 345. That chapter also contains details on using pre-written routines that are delivered with **CoSort**.

To ensure that the correct libraries are loaded, copy the appropriate library files from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

Field-level routines can be invoked in the **/INREC** or output section of a script (or both, to receive the benefit of a pre- and post-processing transformation). The syntax is:

```
/FIELD=(field2=custom_function(arg1,arg2,arg3,...))
```

where *field2* is the name of the derived field to contain the custom-manipulated values. *custom_function* is the name of a user-written function in the user-specified **.dll** or **.so** that sets up the arguments to be specified.

custom_function can have any number of arguments. An argument can be any of:

- another **sortcl** field
- a literal string or a constant number
- a **sortcl**-supported function or another custom function
- an arithmetic operation whose operands can be a field, a literal, or a function

The following are examples of valid custom field function specifications:

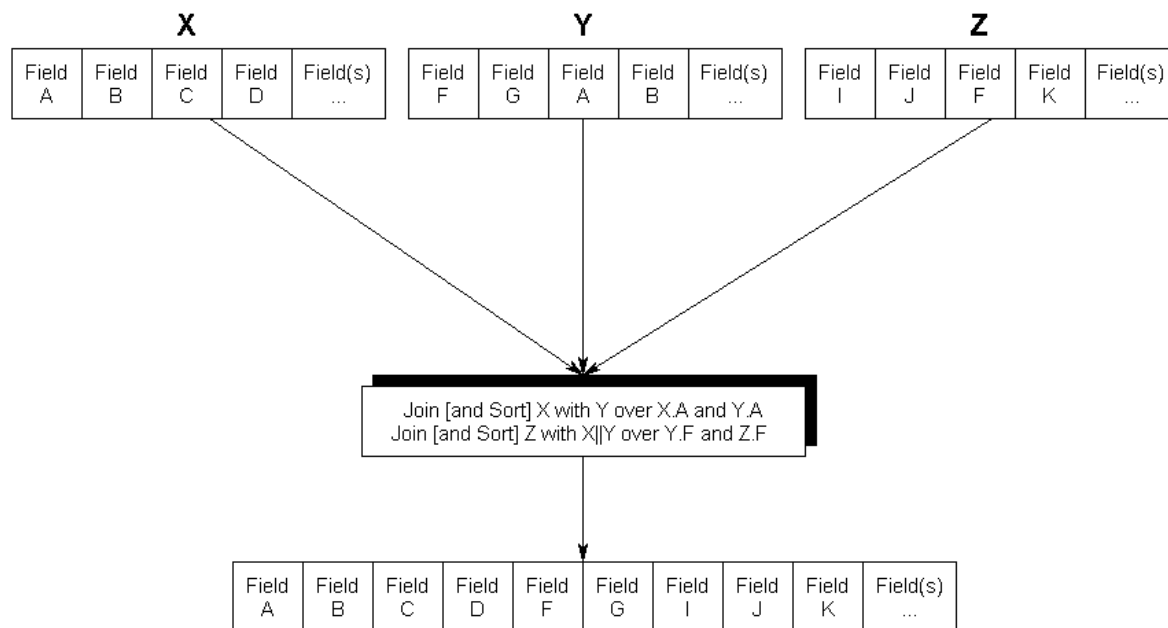
- `/FIELD=(field2=custom_function(fieldx,fieldy,120,"xyz"))`
- `/FIELD=(field2=custom_function1 \`
`(fieldx,custom_function2(fieldy,"abc"))`
- `/FIELD=(field2=custom_function1 \`
`(fieldx + custom_function2("xyz"))`
- `/FIELD=(field2=custom_function1(custom_function2 \`
`(custom_function3(sub_string("xyz",1,3),fieldx)))`

25 /JOIN

The `/JOIN` action in **sortcl** combines fields of two or more input sources into a single output record. With the exception of unordered joins, the join, or match, is based on common join keys. Input sources and output targets can be physical files, pipes, procedure calls, and/or ODBC-connected database tables.

The **sortcl** join works by matching two tables over a common key and condition. If multi-table joins are specified, additional joins are performed against the resultant set from the previous join, and so on. Therefore, each subsequent join in a multi-table join requires its own join key and match condition.

The following schematic illustrates a three-table join using **sortcl**:



Join X with Y to make X||Y and Join Z to X||Y

The `/JOIN` statement used to perform the above three-table join might be:

```
/JOIN X Y WHERE X.A == Y.A
Z WHERE Y.F == Z.F
```

See Syntax on page 225 for a full description of the `/JOIN` syntax and its components.

25.1 Syntax

The `/JOIN` statement allows you to specify multiple sources, and the join keys and conditions over which they are matched. The following syntax demonstrates the joining of four tables (note that the first line gives the complete syntax for a two-table join):

```
/JOIN [join_type [ONLY]] [not_sorted] source1 [not_sorted] source2 [WHERE condition] \
      [join_type [ONLY]] [not_sorted] source3 [WHERE condition] \
      [join_type [ONLY]] [not_sorted] source4 [WHERE condition]
```

where *source#* is the file name or alias (see */ALIAS on page 89*) for each join input source.

join_type is an option that is used for each join being performed, and can be any of:

- | | |
|--------------------|--|
| INNER | The default. Produces only those records which satisfy the conditions. That is, no non-matched fields from either the left- or right-side ¹ input source are displayed. See Example: <i>on page 229</i> . |
| LEFT_OUTER | Shows both the matches that satisfy the conditions, and the field contents from the left-side input source that are not satisfied by the conditions. See Example: <i>on page 229</i> . |
| RIGHT_OUTER | Shows both the matches that satisfy the condition(s), and the field contents from the right-side input source that are not satisfied by the conditions. See Example: <i>on page 229</i>). |
| FULL_OUTER | Shows both the matches that satisfy the condition(s), and the field contents from both the left-side and right-side input sources that are not satisfied by the conditions. See Example: <i>on page 231</i> . |
| UNORDERED | Every combination from the input sources is produced (no <code>WHERE</code> condition). See Unordered (Unconditional) Joins <i>on page 233</i> . |

For OUTER-type joins, blanks (or nulls) are inserted for the fields of the table that are not matched against the other table source.

-
1. The input sources are described in terms of a left and right side, even in the context of a multi-table join, because **sortel** always joins two tables together internally based on a `WHERE` condition. Subsequent tables can be joined to the results from the previous join, using a new `WHERE` condition. You can specify different join types for each internal phase of a multi-table join operation. See Example: *on page 309* and Example: *on page 311* for multi-table join examples.

The **ONLY** option can be included with **OUTER**-type joins, and is used to display *only* the non-matched fields (that is, no **INNER** matches are returned). See **JOIN ONLY** on page 232 for an example.

You must use the **NOT_SORTED** option for each input source that is not pre-sorted over the join key specified in its respective **WHERE** clause. The default option is **PRE_SORTED**, which is used when the file is already sorted over its join key. You can also use the **CHK_SORTED** option if you are not sure whether or not an input source is sorted.



Failure to use the **NOT_SORTED** option, when required, can produce unpredictable results.

Multi-table joins can be heavy consumers of CPU, memory, and disk space. If there are insufficient machine resources to execute your multi-table join, consider pre-sorting over the intended join keys, rather than using the **NOT_SORTED** option. Also, you can break out the multi-table join into multiple steps of two-table joins.

It is also advised that, prior to running a join script, you filter out records and fields that are not be required in the output.

For details on configuring memory setting for intensive sorting and joining, see **WHERE Condition**

When doing multi-table joins with delimited input files, the delimiters must be the same in each file.

The **WHERE** conditions in the **/JOIN** statement must be an equi join that specifies a single or multiple condition evaluation. An equi join uses **==** or **EQ** in defining the join condition. See Binary Logical Expressions on page 171 and Compound Logical Expressions on page 172.

For example, to join records that match the cost field from **file1** with the charge field from **file2**, the following **WHERE** condition is used:

```
WHERE cost EQ charge or
WHERE cost == charge
```

If other condition types are required, apply **/INCLUDE** and **/OMIT** statements to the resultant output files (see **INCLUDE-OMIT (RECORD SELECTION)** on page 175). So, a complete two-table **/JOIN** statement for the supported match criterion above might look like this:

```
/JOIN file1 file2 WHERE cost == charge
```

Note that the alternate notation *file.field* must be used when the a field name is the same in both input sources so that there is no ambiguity regarding the source file. If the field names are the same in both inputs, then references in the join condition such as:

```
file1.cost EQ file2.cost
```

are necessary for specifying output fields with shared input names.



It is recommended that you use the `/ALIAS` feature to reference join file sources, especially in cases where input file names include a reference to the path. This makes it easier to refer to the files throughout the script, especially when using *file.field* notation in `WHERE` conditions. See */ALIAS on page 89* for details.

File (or file alias) names should not be a join semantic, such as `LEFT`, `INNER`, or `ONLY`.

The supplemental formatting options `HEADREAD/HEADWRITE` and `TAILREAD/TAILEWRITE` are currently not supported in a join script (see */HEADREAD on page 261*, */HEADWRITE on page 269*, */TAILREAD on page 263*, and */TAILEWRITE on page 270*).



WARNING!

In a join script with three or more input files, you cannot use the `POSITION` attribute alone, that is, without an accompanying `SIZE` or `SEPARATOR` attribute. An error will be returned in these cases.

Either provide a `SIZE` statement (for fixed-position fields) and/or a separator character. For example, either of the following options would be valid to define an input field for one or more sources in a multi-table join:

```
/FIELD=(wholerec, POSITION=1, SEPARATOR='^') or  
/FIELD=(value, POSITION=8, SIZE=13)
```

25.2 /JOIN Examples

This section contains examples of the basic **sortcl** join types. For additional join options, see **JOIN ONLY** *on page 232* and **Unordered (Unconditional) Joins** *on page 233*. For a complete multi-table join example, see **Example: *on page 309***.

The following examples relate to a hypothetical credit account system. The system has a **Customer** file and a **Purchases** file. At some point, the files contain:

---- Purchases ----			----- Customers -----		
12212	Skis	140.25	11010	Sedaka, Neal	5.00
12345	Shoes	40.25	12212	Diamond, Neil	11.24
13322	Hammer	112.99	12214	Mendez, Sergio	6.66
13323	Nails	11.97	12332	Ho, Don	-55.00
13342	Rocks	8.45	12345	Jones, Tom	123.45
23372	Whiskey	28.67	13322	Mercer, Johnny	-22.22
23373	Wine	18.57	13332	Torme, Mel	8.00
			13333	Martin, Dean	12.67
			13342	Sinatra, Frank	34.67

Note that both files are sorted on their **Acct_nb** field.

File and data descriptions are found in the data definition files **Purchases.ddf** and **Customers.ddf**:

```
# metadata for Purchases
/FIELD= (Acct_nb, POSITION=1, SIZE=5)
/FIELD= (Item, POSITION=7, SIZE=8)
/FIELD= (Charge, POSITION=15, SIZE=6)

# metadata for Customers
/FIELD= (Acct_nb, POSITION=1, SIZE=5)
/FIELD= (Name, POSITION=9, SIZE=15)
/FIELD= (Balance, POSITION=26, SIZE=11, NUMERIC)
```

Example: INNER Join

To find only the matches, use **inner.scl**:

```
/INFILE=Purchases
  /SPECIFICATION=Purchases.ddf
/INFILE=Customers
  /SPECIFICATION=Customers.ddf
/JOIN INNER Purchases Customers WHERE \
Purchases.Acct_nb == Customers.Acct_nb
/OUTFILE=Bills
/HEADREC="Item   Charge P_Ac#   C_Ac#   Customer       Old Balance New Bal\n\n"
  /FIELD=(Item,POSITION=1,SIZE=7)
  /FIELD=(Charge,POSITION=8,SIZE=6)
  /FIELD=(Purchases.Acct_nb,POSITION=15,SIZE=5)
  /FIELD=(Customers.Acct_nb,POSITION=22,SIZE=5)
  /FIELD=(Name,POSITION=29,SIZE=15)
  /FIELD=(Balance,POSITION=44,SIZE=8,NUMERIC)
  /FIELD=((Bills.Balance + Bills.Charge),POSITION=52,SIZE=9,NUMERIC)
```

Bills output is:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
Skis	140.25	12212	12212	Diamond, Neil	11.24	151.49
Shoes	40.25	12345	12345	Jones, Tom	123.45	163.70
Hammer	112.99	13322	13322	Mercer, Johny	-22.22	90.77
Rocks	8.45	13342	13342	Sinatra, Frank	34.67	43.12

Example: LEFT_OUTER and RIGHT_OUTER Join

To find the non-matches from the left file, plus the matches, use **left_outer.scl**:

```
/INFILE=Purchases
  /SPECIFICATION=Purchases.ddf
/INFILE=Customers
  /SPECIFICATION=Customers.ddf
/JOIN LEFT_OUTER Purchases Customers WHERE \
Purchases.Acct_nb == Customers.Acct_nb
/OUTFILE=Bills2
/HEADREC="Item   Charge P_Ac#   C_Ac#   Customer       Old Balance New Bal\n\n"
  /FIELD=(Item,POSITION1,SIZE=7)
  /FIELD=(Charge,POSITION=8,SIZE=6)
  /FIELD=(Purchases.Acct_nb,POSITION=15,SIZE=5)
  /FIELD=(Customers.Acct_nb,POSITION=22,SIZE=5)
  /FIELD=(Name,POSITION=29,SIZE=15)
  /FIELD=(Balance,POSITION=44,SIZE=8,NUMERIC)
  /FIELD=((Bills2.Balance + Bills2.Charge),POSITION=52,SIZE=9,NUMERIC)
```

Bills2 output is:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
Skis	140.25	12212	12212	Diamond, Neil	11.24	151.49
Shoes	40.25	12345	12345	Jones, Tom	123.45	163.70
Hammer	112.99	13322	13322	Mercer, Johny	-22.22	90.77
Nails	11.97	13323			0.00	11.97
Rocks	8.45	13342	13342	Sinatra, Frank	34.67	43.12
Whiskey	28.67	23372			0.00	28.67
Wine	18.57	23373			0.00	18.57

To find the non-matches from the right file, plus the matches, use **right_outer.scl**:

```
/INFILE=Purchases
/SPECIFICATION=Purchases.ddf
/INFILE=Customers
/SPECIFICATION=Customers.ddf
/JOIN RIGHT_OUTER Purchases Customers WHERE \
      Purchases.Acct_nb == Customers.Acct_nb
/OUTFILE=Bills3
/HEADREC="Item    Charge P_Ac#  C_Ac#  Customer      Old Balance New Bal\n\n"
/FIELD=(Item, POSITION=1, SIZE=7)
/FIELD=(Charge, POSITION=8, SIZE=6)
/FIELD=(Purchases.Acct_nb, POSITION=15, SIZE=5)
/FIELD=(Customers.Acct_nb, POSITION=22, SIZE=5)
/FIELD=(Name, POSITION=29, SIZE=15)
/FIELD=(Balance, POSITION=44, SIZE=8, NUMERIC)
/FIELD=((Bills3.Balance + Bills3.Charge), POSITION=52, SIZE=9, NUMERIC)
```

Bills3 output is:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
			11010	Sedaka, Neal	5.00	5.00
Skis	140.25	12212	12212	Diamond, Neil	11.24	151.49
			12214	Mendez, Sergio	6.66	6.66
			12332	Ho, Don	-55.00	-55.00
Shoes	40.25	12345	12345	Jones, Tom	123.45	163.70
Hammer	112.99	13322	13322	Mercer, Johny	-22.22	90.77
			13332	Torme, Mel	8.00	8.00
			13333	Martin, Dean	12.67	12.67
Rocks	8.45	13342	13342	Sinatra, Frank	34.67	43.12

Example: FULL OUTER Join

A typical data processing function is to find the matches and non-matches from two input streams. This can be done using a full outer join. A full outer join is a left outer, inner, and right outer join at the same time. The following job, **full_outer.scl**, demonstrates a full outer join where Acct_nb in the file **Purchases** is matched to Acct_nb in the file **Customers**:

```
/INFILE=Purchases
  /SPECIFICATION=Purchases.ddf
/INFILE=Customers
  /SPECIFICATION=Customers.ddf
/JOIN FULL_OUTER Purchases Customers WHERE \
      Purchases.Acct_nb == Customers.Acct_nb
/OUTFILE=Bills4
/HEADREC="Item    Charge P_Ac#  C_Ac#  Customer      Old Balance New Bal\n\n"
  /FIELD=(Item, POSITION=1, SIZE=7)
  /FIELD=(Charge, POSITION=8, SIZE=6)
  /FIELD=(Purchases.Acct_nb, POSITION=15, SIZE=5)
  /FIELD=(Customers.Acct_nb, POSITION=22, SIZE=5)
  /FIELD=(Name, POSITION=29, SIZE=15)
  /FIELD=(Balance, POSITION=44, SIZE=8, NUMERIC)
  /FIELD=((Bills4.Balance + Bills4.Charge), POSITION=52, SIZE=9, NUMERIC)
```

The output file **Bills4**:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
			11010	Sedaka, Neal	5.00	5.00
Skis	140.25	12212	12212	Diamond, Neil	11.24	151.49
			12214	Mendez, Sergio	6.66	6.66
			12332	Ho, Don	-55.00	-55.00
Shoes	40.25	12345	12345	Jones, Tom	123.45	163.70
Hammer	112.99	13322	13322	Mercer, Johny	-22.22	90.77
Nails	11.97	13323			0.00	11.97
			13332	Torme, Mel	8.00	8.00
			13333	Martin, Dean	12.67	12.67
Rocks	8.45	13342	13342	Sinatra, Frank	34.67	43.12
Whiskey	28.67	23372			0.00	28.67
Wine	18.57	23373			0.00	18.57

Records 1, 3, 4, 8, and 9 do not have a Purchase match to Customer for Account_nb -- this shows non-active customers. These output records contain blanks in the Item, Charge, and Purchase.Account_nb fields.

Records 7, 11 and 12 do not have a Customer match to Purchase for Account_nb -- this might indicate fraud. These output records contain blanks in the Customer.Account_nb, Customer, and Old Balance fields.

Records 2, 5, 6 and 10 have matching Purchase and customer Account_nb values so that no fields mentioned are blank.

Because **sortcl** permits multiple output files, it is possible to generate three different output files using unique include and omit logic on the blank fields (see INCLUDE-OMIT (RECORD SELECTION) *on page 175*). Each output file is formattable, and you can derive down- and cross-calculated values as shown in the New Bal field.

Note the use of the *file.field* reference to remove any ambiguity over the source.

For all ordered joins, the input files must be in order over the join field. Output will be ordered on that same key field. For sources that are not pre-sorted, use the NOT_SORTED option in the /JOIN statement, as described in Syntax *on page 225*.

25.3 JOIN ONLY

When you add the ONLY option to LEFT, RIGHT, or FULL OUTER joins, the matching (inner) records are removed from the resulting output, that is, only the non-matching records are returned.

A) /JOIN LEFT_OUTER ONLY:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
Nails	11.97	13323				11.97
Whiskey	28.67	23372				28.67
Wine	18.57	23373				18.57

B) /JOIN RIGHT_OUTER ONLY:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
			11010	Sedaka, Neal	5.00	5.00
			12214	Mendez, Sergio	6.66	6.66
			12332	Ho, Don	-55.00	-55.00
			13332	Torme, Mel	8.00	8.00
			13333	Martin, Dean	12.67	12.67

C) /JOIN FULL_OUTER ONLY:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
			11010	Sedaka, Neal	5.00	5.00
			12214	Mendez, Sergio	6.66	6.66
			12332	Ho, Don	-55.00	-55.00
Nails	11.97	13323				11.97
			13332	Torme, Mel	8.00	8.00
			13333	Martin, Dean	12.67	12.67
Whiskey	28.67	23372				28.67
Wine	18.57	23373				18.57

25.4 Unordered (Unconditional) Joins

The unordered join command in this case is:

```
/JOIN UNORDERED Purchases Customers
```

Note that a WHERE condition is not used.

From the input data in the previous examples, this join creates 63 output records, and these contain every combination of Purchase and Customer records:

Item	Charge	P_Ac#	C_Ac#	Customer	Old Balance	New Bal
Skis	140.25	12212	11010	Sedaka, Neal	5.00	145.25
Skis	140.25	12212	12212	Diamond, Neil	11.24	151.49
Skis	140.25	12212	12214	Mendez, Sergio	6.66	146.91
Skis	140.25	12212	12332	Ho, Don	-55.00	85.25
Skis	140.25	12212	12345	Jones, Tom	123.45	263.70
Skis	140.25	12212	13322	Mercer, Johny	-22.22	118.03
Skis	140.25	12212	13332	Torme, Mel	8.00	148.25
Skis	140.25	12212	13333	Martin, Dean	12.67	152.92
Skis	140.25	12212	13342	Sinatra, Frank	34.67	174.92
Shoes	40.25	12345	11010	Sedaka, Neal	5.00	45.25
Shoes	40.25	12345	12212	Diamond, Neil	11.24	51.49
Shoes	40.25	12345	12214	Mendez, Sergio	6.66	46.91
Shoes	40.25	12345	12332	Ho, Don	-55.00	-14.75

and so on...



If the input files are of significant size, the output file can be quite large; therefore, care must be used to ensure that the system resources are not unduly consumed by an unordered join.

26 SUMMARY FUNCTIONS (AGGREGATION)

sortcl can produce output records containing summary fields derived from accumulated detail records. Multiple levels of summary records can be created in the same pass. Depending on the different levels of **BREAKs** you specify, multiple levels of subtotals can be provided, along with a grand total.

One or more summary fields can be derived using the following **sortcl** features:

- Summary, Average, and Standard Deviation *on page 235*
- Maximum and Minimum *on page 239*
- Counting *on page 241*
- Ranking *on page 242*
- Creating New Files Based on **BREAK (/NEWFILE)** *on page 245*

Summary records can be produced on a given **BREAK** condition at the end of processing. Any number of intermediate **BREAK** levels can also be specified. This is particularly useful in data-warehousing environments where complex drill-down aggregation and grouping are required.

There are two steps required to define a summary field within a summary or detail record:

- 1) Use a **/FIELD** statement to describe its position and format in the record.
- 2) Use one of the above functions to determine how the value of the field is derived.

Summary records can be formatted differently at each level. Each of these levels can be written to a separate file, or merged into one file to produce a structured report.

Create **RUNNING** summary fields in the detail records. These running, or accumulating, summary fields are updated at each record.



It is possible to use the **/ROUNDING** statement to change the way in which numeric values with several decimal places are rounded after an arithmetic **sortcl** operation (see **/ROUNDING** *on page 281*).

26.1 Summary, Average, and Standard Deviation

The syntax for /SUM and /AVERAGE is the same:

```
/FUNCTION=FieldX [FROM field or expression] [RUNNING [lag_count]] \
    [WHERE condition1] [BREAK condition2]
```

A sum is accumulated until each break. With /AVERAGE, the accumulating sum is divided by the number of records before each break. A grand total or file-wide average is produced if there is no grouping via breaks.

The difference in the syntax for standard deviation is that RUNNING and lag count are not optional.

The syntax for /STD is:

```
/FUNCTION=FieldX [FROM field or expression] RUNNING lag_count \
    [WHERE condition1] [BREAK condition2]
```

The /STD function calculates the population standard deviation, which measures the disbursement from the average.

If *condition1* in a WHERE clause is an expression (such as `price GT 11`), rather than simply a field name, and is followed by a BREAK statement, then *condition1* must be enclosed in parentheses.

Where:

- FUNCTION is SUMMARY (SUM) , AVERAGE (AVG) , or STD
- *FieldX* is the new field name being created by this aggregation
- FROM indicates the source of values; it is either an input field name or simple expression which references one or more input field names. If an expression is used, it should be enclosed with parentheses.



When using an expression to define the FROM clause, enclose the expression in parentheses. Only simple expressions can be used in the FROM of a summary definition. Do not use parentheses for grouping. For example, you can have:

```
/SUM=Exp_B FROM (A * B - 5)
```

but not

```
/SUM=Exp_B FROM (A * (B - 5))
```

- RUNNING calculates, based on the FUNCTION, a new value for each input record. Refer to Running (Accumulating) Aggregates *on page 251*.

- *lag_count* specifies the number of records used in the RUNNING calculation. This number includes current and prior records up to the BREAK. This allows a windowing feature for aggregation.
- *condition1* determines if the sum field value of a particular record is to be added to the sum. If *Condition1* is true, the value is added, otherwise it is not. If there is no WHERE portion in the statement, then the addition is unconditional.
- BREAK *Condition2* controls when the /SUMMARY or /AVERAGE record is output and the values are reset.

Again, there is a natural BREAK at the end of the job; therefore, if the BREAK portion is not used, or *Condition2* never becomes true, results for the whole job display.

An example is:

```
/INFILE=bookstock
/FIELD=(title,POSITION=1,SIZE=14)
/FIELD=(publisher,POSITION=16,SIZE=13)
/FIELD=(quantity,POSITION=30,SIZE=3,NUMERIC)
/FIELD=(price,POSITION=36,SIZE=7,NUMERIC)
/CONDITION=(expr,TEST=(price GT 11))
/SORT
/KEY=publisher
/OUTFILE=bookstock.out
/FIELD=(publisher,POSITION=1,SIZE=13)
/FIELD=(avgprice,POSITION=15,SIZE=7,PRECISION=2,NUMERIC)
/FIELD=(cost,POSITION=24,SIZE=8,PRECISION=2,MILL,NUMERIC)
/AVERAGE=avgprice from price BREAK publisher
/SUM=cost FROM (quantity * price) WHERE expr BREAK publisher
```

Note that the output fields *avgprice* and *cost* are specified with regard to position, size, and type using /FIELD statements. The MILL parameter specified for *cost* places commas in the appropriate places of the output for this field (see MILL on page 106). The /SUM and /AVERAGE produce the values for *avgprice* and *cost*.

The FROM portion of /AVERAGE is an input field name. The FROM portion of /SUM is an expression.

```
/AVERAGE=avgprice FROM price
/SUM=cost from (quantity * price)
```

Two different methods for defining conditions were used. The BREAK clause uses a condition name that was previously defined in a /CONDITION statement. You can define the condition once and use the condition name in multiple statements. The WHERE clause defines the condition explicitly within the summary definition.

Using the below input file **bookstock**:

Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95
Sending Your	Valley Kill	130	15.75
People Please	Valley Kill	75	11.50
Map Reader	Prentice-Hall	200	14.95

The output file **bookstock.out** from the above job script is:

Dell	13.05	1,044.00
Harper-Row	7.92	0.00
Prentice-Hall	12.60	2,990.00
Valley Kill	13.62	2,910.00

Example: Using the running lag count for windowing

Using the input file **sales.in**:

01	100.00
01	200.00
01	300.00
01	400.00
01	500.00
01	600.00
01	700.00
02	10.00
02	20.00
02	30.00
02	40.00
02	50.00
02	60.00
02	70.00
03	1.00
03	2.00
03	3.00
03	4.00
03	5.00
03	6.00
03	7.00

Consider the following script, **lagfunction.scl**:

```
/INFILE=sales.in
/FIELD=(store, POSITION=1, SIZE=2)
/FIELD=(amount, POSITION=7, SIZE=7, PRECISION=2, NUMERIC)

/REPORT

/OUTFILE=lagreport.out
/HEADREC="#      Amt      ct [2]      sum [3]      avg [4]      std [5]      min [6]      max [7] \n\n"
/FIELD=(store, POSITION=1, SIZE=2)
/FIELD=(amount, POSITION=4, SIZE=9, PRECISION=2, NUMERIC)
/FIELD=(cnt, POSITION=15, SIZE=4, PRECISION=0, NUMERIC)
/FIELD=(sum, POSITION=25, SIZE=9, PRECISION=2, NUMERIC)
/FIELD=(avg, POSITION=35, SIZE=9, PRECISION=2, NUMERIC)
/FIELD=(std, POSITION=45, SIZE=9, PRECISION=2, NUMERIC)
/FIELD=(min, POSITION=55, SIZE=9, PRECISION=2, NUMERIC)
/FIELD=(max, POSITION=65, SIZE=9, PRECISION=2, NUMERIC)

/COUNT      cnt RUNNING 2 break store
/sum        sum FROM amount RUNNING 3 break store
/average    avg FROM amount RUNNING 4 break store
/standard   std FROM amount RUNNING 5 break store
/minimum    min FROM amount RUNNING 6 break store
/maximum    max FROM amount RUNNING 7 break store
```

Note that in the output file, **lagreport.out**, the aggregation values results are from the number of records specified by the lag count.

#	Amt	ct [2]	sum [3]	avg [4]	std [5]	min [6]	max [7]
01	100.00	1	100.00	100.00	0.00	100.00	100.00
01	200.00	2	300.00	150.00	50.00	100.00	200.00
01	300.00	1	600.00	200.00	81.65	100.00	300.00
01	400.00	2	900.00	250.00	111.80	100.00	400.00
01	500.00	1	1200.00	350.00	141.42	100.00	500.00
01	600.00	2	1500.00	450.00	141.42	100.00	600.00
01	700.00	1	1800.00	550.00	141.42	200.00	700.00
02	10.00	1	10.00	10.00	0.00	10.00	10.00
02	20.00	2	30.00	15.00	5.00	10.00	20.00
02	30.00	1	60.00	20.00	8.16	10.00	30.00
02	40.00	2	90.00	25.00	11.18	10.00	40.00
02	50.00	1	120.00	35.00	14.14	10.00	50.00
02	60.00	2	150.00	45.00	14.14	10.00	60.00
02	70.00	1	180.00	55.00	14.14	20.00	70.00
03	1.00	1	1.00	1.00	0.00	1.00	1.00
03	2.00	2	3.00	1.50	0.50	1.00	2.00
03	3.00	1	6.00	2.00	0.82	1.00	3.00
03	4.00	2	9.00	2.50	1.12	1.00	4.00
03	5.00	1	12.00	3.50	1.41	1.00	5.00
03	6.00	2	15.00	4.50	1.41	1.00	6.00
03	7.00	1	18.00	5.50	1.41	2.00	7.00

26.2 Maximum and Minimum

/MAXIMUM and /MINIMUM are used to calculate the maximum and minimum values, respectively, of a field.

/MAX and /MIN are used the same way as /SUM or /AVERAGE. The syntax is:

```
/FUNCTION=FieldX [FROM field or expression] [RUNNING [lag_count]] \
    [ WHERE condition1] [BREAK condition2]
```

Where

- FUNCTION is MAXIMUM or MAX, or MINIMUM or MIN
- *FieldX* is the new field name being created by this aggregation
- FROM indicates the source of values; it is either an input field name or simple expression which references one or more input field names. If an expression is used, it should be enclosed with parentheses.



When using an expression to define the FROM clause, enclose the expression in parentheses. Only simple expressions can be used in the FROM of a summary definition. Do not use parentheses for grouping. For example, you can have:

```
/SUM=Exp_B FROM (A * B - 5)
```

but not

```
/SUM=Exp_B FROM (A * (B - 5))
```

- RUNNING calculates, based on the *FUNCTION*, a new value for each input record. Refer to Running (Accumulating) Aggregates on page 251.
- *lag_count* limits the number of records used in the RUNNING calculation. This allows a windowing feature for aggregation.
- *condition1* determines if the sum field value of a particular record is to be added to the sum. If *Condition1* is true, the value is added, otherwise it is not. If there is no WHERE portion in the statement, then the addition is unconditional.
- BREAK *Condition2* controls when the /SUMMARY or /AVERAGE record is output and the values are reset.

If two or more records share the same /MAX or /MIN field value, **sortcl** will output the first sorted record that satisfies the test.

Example: Calculating Maximum and Minimum Values

Consider the following input data:

jewelry	103.25
clothing	54.25
clothing	9.45
linens	40.95
clothing	235.74
jewelry	1245.00
toys	21.98
clothing	121.05

This script, **max_min.scl**, produces maximum and minimum values for **sales.dat**, by dept:

```
/INFILE=sales.dat
/FIELD=(dept, POSITION=1, SIZE=9)
/FIELD=(sales, POSPOSITION=10, SIZE=8, NUMERIC)
/CONDITION=(new_dept, TEST=(dept))
/SORT
/KEY=dept
/OUTFILE=sales.out
/HEADREC="  DEPT                MIN                MAX\n"
/FIELD=(dept, POSPOSITION=3, SIZE=9)
/FIELD=(minsales, POSPOSITION=15, SIZE=9, MILL, NUMERIC)
/FIELD=(maxsales, POSPOSITION=27, SIZE=10, MILL, NUMERIC)
/MIN minsales FROM sales BREAK new_dept
/MAX maxsales FROM sales BREAK new_dept
```

It produces the following output:

DEPT	MIN	MAX
clothing	9.45	235.74
jewelry	103.25	1245.00
linens	40.95	40.95
toys	21.98	21.98

To also display the minimum and maximum for all departments, add the following lines to the bottom of the script:

```
/OUTFILE=sales.out
/HEADREC="  -----\n"
/FIELD=(minsales, POSITION=15, SIZE=9, PRECISION=2, NUMERIC)
/FIELD=(maxsales, POSITION=27, SIZE=10, PRECISION=2, NUMERIC)
/MIN minsales FROM sales
/MAX maxsales FROM sales
```


to obtain these results:

DEPT	MIN	MAX
clothing	9.45	235.74
jewelry	103.25	1245.00
linens	40.95	40.95
toys	21.98	21.98

	9.45	1245.00

Use of the `RUNNING` option will display the current maximum or minimum value (within each reset `BREAK` group) in detailed output records.

In addition to displaying the maximum and/or minimum value of numeric fields, you can choose to display the maximum and/or minimum value of character fields. Using the previous input file and the following job script:

```
/INFILE=sales.dat
  /FIELD=(dept, POSITION=1, SIZE=9)
  /FIELD=(sales, POSITION=10, SIZE=8, NUMERIC)
/SORT
  /KEY=dept
/OUTFILE=sales.out
/HEADREC="  MIN          MAX\n"
  /FIELD=(mindept, POSITION=3, SIZE=9)
  /FIELD=(maxdept, POSITION=15, SIZE=9)
  /MIN mindept FROM dept
  /MAX maxdept FROM dept
```

will result in the following output file:

MIN	MAX
clothing	toys

26.3 Counting

The syntax of a `/COUNT` statement is:

```
/COUNT = FieldX [RUNNING] [WHERE Condition1] /
          [BREAK Condition2]
```

FieldX will contain the count of records that satisfy *Condition1*. The record containing *FieldX* is displayed and reset to 0 when *Condition2* occurs. If no `WHERE` condition is specified, the records are counted until the `BREAK` occurs. If no `BREAK` is given, all records that satisfy *Condition1* are counted and displayed at the end of the job.

Example: Using /COUNT

Using the example given for maximum and minimum (Example: *on page 240*), the following job script, **count.scl**, additionally displays the number of sales from which the sales amounts are taken:

```
/INFILE=sales.dat
  /FIELD=(dept, POSITION=1, SIZE=9)
  /FIELD=(sales, POSITION=10, SIZE=8, NUMERIC)
  /CONDITION=(new_dept, TEST=(dept))
/SORT
  /KEY=dept
/OUTFILE=sales.out
  /HEADREC="  DEPT      QTY      MIN      MAX\n"
  /FIELD=(dept, POSITION=3, SIZE=9)
  /FIELD=(qty, POSITION=13, SIZE=2, PRECISION=0, NUMERIC)
  /FIELD=(minsales, POSITION=16, SIZE=9, MILL, NUMERIC)
  /FIELD=(maxsales, POSITION=28, SIZE=10, MILL, NUMERIC)
  /COUNT qty BREAK new_dept
  /MIN minsales FROM sales BREAK new_dept
  /MAX maxsales FROM sales BREAK new_dept
/OUTFILE=sales.out
  /HEADREC=" -----\n"
  /FIELD=(qty, POSITION=13, SIZE=2, PRECISION=0, NUMERIC)
  /FIELD=(minsales, POSITION=16, SIZE=9, PRECISION=2, NUMERIC)
  /FIELD=(maxsales, POSITION=28, SIZE=10, PRECISION=2, NUMERIC)
  /MIN minsales FROM sales
  /MAX maxsales FROM sales
  /COUNT qty
```

The output is:

DEPT	QTY	MIN	MAX
clothing	4	9.45	235.74
jewelry	2	103.25	1,245.00
linens	1	40.95	40.95
toys	1	21.98	21.98

	8	9.45	1245.00

Use of the **RUNNING** option will display the current count for each **BREAK** group (see *Running (Accumulating) Aggregates on page 251*).

26.4 Ranking

One type of statistical analysis involves assigning a sequential rank to a set of data values. The following **sortcl** syntax can be used as a template to perform ranking:

```
/INFILE=...
/FIELD=(data_value, ...)
```

```

...
/OUTFILE=...
/FIELD=(rank, SIZE=n, PRECISION=0, NUMERIC)
/FIELD=(data_value, ...)
...
/COUNT=rank RUNNING WHERE data_value

```

The *data_value* is the input field on which ranking is to be performed, and this must be the primary /KEY field for sorting. The WHERE clause is required for instances when there are equal values in *data_value*.

Example: Ranking Values

Suppose you want to rank salespeople by the value of sales they have made, with the highest sales ranking first. The following is a data file containing names of salespeople and their associated sales (in thousand dollar amounts).

Mary	23
Robert	129
John	345
Vanessa	31
Donald	345
Sarah	54
Laura	45
Henry	98
Richard	31
Tom	29
Nancy	18
Barbara	32

The following script, **rank.scl**, ranks the salespeople in order of sales made:

```

/INFILE=salesbyname
/FIELD=(person, POSITION=1, SIZE=11)
/FIELD=(sales, POSITION=13, SIZE=3)
/SORT
/KEY=(sales, DESCENDING)
/OUTFILE=sale_rank.out
/FIELD=(rank, POSITION=1, SIZE=2, PRECISION=0, NUMERIC)
/FIELD=(person, POSITION=4, SIZE=11)
/FIELD=(sales, POSITION=15, SIZE=3)
/COUNT rank RUNNING WHERE sales

```

Notice that the sort is in descending order so that highest sales values are returned first. Also, the rank field is declared as NUMERIC so that the values will be right-justified.

The preceding script produces the following output:

1	John	345
1	Donald	345
2	Robert	129
3	Henry	98
4	Sarah	54
5	Laura	45
6	Barbara	32
7	Richard	31
7	Vanessa	31
8	Tom	29
9	Mary	23
10	Nancy	18

Now that a rank has been assigned, suppose that the salespeople with a ranking of 3 or better are due to receive a bonus, and that the company wants to produce two files: one containing the salespeople due for a bonus, and the other containing a list of every salesperson, but with an indicator showing each person who is due to receive a bonus.

The following script uses the above output file to achieve both objectives:

```
/INFILE=sale_rank.out
  /FIELD= (rank, POSITION=1, SIZE=2, num)
  /FIELD= (name, POSITION=4, SIZE=11)
  /FIELD= (sales, POSITION=15, SIZE=3)
  /CONDITION= (C1, TEST= (rank GT 3) )
/REPORT
/OUTFILE=bonus1.out
  /OMIT WHERE C1
  /FIELD= (rank, POSITION=1, SIZE=2, PRECISION=0, num)
  /FIELD= (name, POSITION=4, SIZE=11)
  /FIELD= (sales, POSITION=15, SIZE=3)
/OUTFILE=bonus2.out
  /FIELD= (rank, POSITION=1, SIZE=2, PRECISION=0, num)
  /FIELD= (name, POSITION=4, SIZE=11)
  /FIELD= (sales, POSITION=15, SIZE=3)
  /FIELD= (bonus, POSITION=20, SIZE=8, IF C1 THEN "No Bonus" ELSE "Bonus")
```

This produces the following two output files:

bonus1.out

1	John	345
1	Donald	345
2	Robert	129
3	Henry	98

bonus2.out

1	John	345	Bonus
1	Donald	345	Bonus
2	Robert	129	Bonus
3	Henry	98	Bonus
4	Sarah	54	No Bonus
5	Laura	45	No Bonus
6	Barbara	32	No Bonus
7	Richard	31	No Bonus
7	Vanessa	31	No Bonus
8	Tom	29	No Bonus
9	Mary	23	No Bonus
10	Nancy	18	No Bonus

26.5 Creating New Files Based on BREAK (/NEWFILE)

Use the /NEWFILE command to create separate output files that group records together for each unique value of a specified field. /NEWFILE can be used to store these records into separate output files (see Example: *on page 246*), or to perform aggregation (sum, min, max, average, and count) on the grouped records within each output file (see Example: *on page 247*).

When you use /NEWFILE, specify an output file name suffix (using the /OUTFILE command) that will apply to all output files that are generated, and **sortcl** will add the unique data value from the BREAK file prior to each file name suffix, thereby giving a clear indication of the contents of each output file.

Syntax

The /NEWFILE command is used in the output section of a **sortcl** job script, and works in conjunction with the /OUTFILE statement. The syntax is as follows:

```
/NEWFILE BREAK fieldname
```

where *fieldname* is the field that determines the record groups to be sent to individual output files. That is, for each unique value of the specified field, a new output file will be created to store records with that value. Note that records must either be pre-sorted over this field, or you must include a /KEY statement in the script to sort that field (see KEYS *on page 160*).

The /OUTFILE statement allows you to specify a file name suffix that will be shared by all output files, as follows:

```
/OUTFILE=suffix.extension
```

For example, using /OUTFILE=id.dat would result in output file names such as **1000id.dat**, **2000id.dat**, **3000id.dat**, etc., in a case where the BREAK field, id, contains the unique values 1000, 2000, 3000, etc.

Alternatively, if you do not want a file name suffix, use:

```
/OUTFILE=.extension
```

For example, using /OUTFILE=.dat would result in output file names such as **1000.dat**, **2000.dat**, **3000.dat**, etc.

Example: Using /NEWFILE for Multiple Output File Creation

Consider the following input file, **areasales.dat**:

```
20 jewelry      103.25
10 clothing     54.25
30 clothing      9.45
10 linens       40.95
10 clothing    235.74
30 jewelry    1245.00
30 toys        21.98
20 clothing    121.05
30 jewelry     203.25
10 clothing     64.25
30 clothing     19.45
10 linens       43.95
20 clothing     35.74
30 jewelry    1745.00
10 toys        121.98
20 clothing    181.05
```

The following script, **newfiles_by_store.scl**, creates separate output files containing records for each unique value of the store field:

```
/INFILE=areasales.dat
/FIELD=(store, POSITION=1, SIZE=2)
/FIELD=(dept, POSITION=4, SIZE=9)
/FIELD=(amount, POSITION=14, SIZE=7, PRECISION=2, NUMERIC)
/SORT
/KEY=store # required when BREAK field is not pre-sorted
/KEY=dept # sort by dept within each output file
/OUTFILE=store.out
/FIELD=(dept, POSITION=1, SIZE=9)
/FIELD=(amount, POSITION=14, SIZE=7, PRECISION=2, NUMERIC)
/NEWFILE BREAK store
```

This produces three output files for each of the unique values of the store field.

10store.out contains records where the department value was 10 in the input file:

clothing	54.25
clothing	235.74
clothing	64.25
linens	40.95
linens	43.95
toys	121.98

20store.out contains records where the store value was 20 in the input file:

clothing	121.05
clothing	181.05
clothing	35.74
jewelry	103.25

30store.out contains records where the store value was 30 in the input file:

clothing	19.45
clothing	9.45
jewelry	1745.00
jewelry	203.25
jewelry	1245.00
toys	21.98

Example: Using /NEWFILE with Aggregation

This example uses the input file **areales.dat** from Example: *on page 246*.

The following script, **newfile_by_store_sums.scl**, creates separate output files for each unique value of the store field, each containing summary records for each department within that store:

```
/INFILE=areasales.dat
/FIELD=(store,POSITION=1,SIZE=2)
/FIELD=(dept,POSITION=4,SIZE=9)
/FIELD=(amount,POSITION=14,SIZE=7,PRECISION=2,NUMERIC)
/SORT
/KEY=store # required when BREAK field is not pre-sorted
/KEY=dept # needed to sum by dept within each output file
/OUTFILE=store_totals.out
/FIELD=(store,POSITION=1,SIZE=2)
/FIELD=(dept,POSITION=4,SIZE=9)
/FIELD=(tamount,POSITION=14,SIZE=7, NUMERIC)
/SUM tamount FROM amount BREAK store or dept
# Sum records by dept within each output file
/NEWFILE BREAK store
```

10store_totals.out contains summary records, by department, where the store value was 10 in the input file:

10 clothing	354.24
10 linens	84.90
10 toys	121.98

20store_totals.out contains summary records, by department, where the store value was 20 in the input file:

20 clothing	337.84
20 jewelry	103.25

30store_totals.out contains summary records, by department, where the store value was 30 in the input file:

30 clothing	28.90
30 jewelry	3193.25
30 toys	21.98

26.6 Reports with Summaries

To produce a report that has detail records, any number of subtotals, and a final total in the same output file, use the same output file name to define each type of record. The following is an example which displays detail records and two levels of summaries in a single output file named **bookstock.out**.

This script, **bookstock.scl**, sorts the data and creates a report:

```
# bookstock.scl
/INFILE=bookstock
  /FIELD=(title, POSITION=1, SIZE=15)
  /FIELD=(publisher, POSITION=16, SIZE=13)
  /FIELD=(quantity, POSITION=30, SIZE=3, NUMERIC)
  /FIELD=(price, POSITION=38, SIZE=5, NUMERIC)
  /CONDITION=(newpub, TEST=(publisher))
/SORT
  /KEY=publisher
  /KEY=title
/OUTFILE=bookstock.out # Sub-total or BREAK Records
  /RECSPPERPAGE=1
  /HEADREC="-----\n"
  /FOOTREC="\n"
  /FIELD=(CountBook, POSITION=1, SIZE=2)
  /DATA=":"
  /FIELD=(Publisher, POSITION=5, SIZE=15)
  /DATA="    Total:"
  /FIELD=(SumQuantity, POSITION=30, SIZE=4)
  /DATA="  Av Price:"
  /FIELD=(AvgPrice, POSITION=46, SIZE=6, MONEY)
  /COUNT CountBook BREAK NewPub
  /AVERAGE AvgPrice FROM Price BREAK NewPub
  /SUM SumQuantity FROM Quantity BREAK NewPub
/OUTFILE=bookstock.out # Grand Total Records
  /HEADREC="=====\n"
  /FIELD=(CountBook, POSITION=1, SIZE=2)
  /DATA=": Titles"
  /FIELD=(SumQuantity, POSITION=30, SIZE=4)
  /FIELD=(AvgPrice, POSITION=43, SIZE=9, MONEY)
  /COUNT CountBook
  /AVERAGE AvgPrice FROM Price
  /SUM SumQuantity FROM Quantity
/OUTFILE=bookstock.out # Detail Records
  /HEADREC="Titles by Publisher      Quantity      Price\n\n"
  /FIELD=(title, POSITION=1, SIZE=15)
  /FIELD=(quantity, POSITION=31, SIZE=3, NUMERIC)
  /FIELD=(price, POSITION=47, SIZE=5, NUMERIC)
```

The BREAK record is defined, followed by the total record, followed by the detail record.

The preceding **sortcl** script uses the following input file, **bookstock**:

Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95
Sending Your	Valley Kill	130	15.75
People Please	Valley Kill	75	11.50
Map Reader	Prentice-Hall	200	14.95

to produce detail, summary, and final values in one report:

Titles by Publisher	Quantity	Price
Still There	80	13.05

1: Dell	Total: 80	Av Price: \$13.05
Murder Plots	160	5.90
Pressure Cook	228	9.95

2: Harper-Row	Total: 388	Av Price: \$7.92
Map Reader	200	14.95
Reasoning For	150	10.25

2: Prentice-Hall	Total: 350	Av Price: \$12.60
People Please	75	11.50
Sending Your	130	15.75

2: Valley Kill	Total: 205	Av Price: \$13.62
=====		
7: Titles	1023	\$11.62

Because the **BREAK** occurs on the field **publisher**, it is the first key. This will allow the calculation by **publisher** for the count of titles, the average price, and the quantity of books. The second key is **title** because the detail records are ordered by title within the publishers.

Notice that there is a **/HEADREC** for each type of record (sub-totals and grand totals, plus detail) in the overall report (see **/HEADREC** on page 270 and **/RECSPPERPAGE** on page 272 for more information).

26.7 Running (Accumulating) Aggregates

For each aggregation function, you can display RUNNING aggregate values in the detail record. To define a running summary, for example, add the parameter RUNNING to the summary definition. Following are examples of job scripts that contain both aggregate functions and RUNNING functions. These use the input file **sales**:

```
01 25 2103.54 1999-01-13
12 03 589.32 1999-01-04
01 10 8712.45 1999-01-15
05 25 498.56 1999-01-05
12 17 54.23 1999-01-12
12 25 867.32 1999-01-12
05 17 3495.56 1999-01-10
01 25 98.63 1999-01-17
01 03 239.39 1999-01-18
12 10 4098.34 1999-01-19
```

Example: Running Summary

The script below, **running.scl**, outputs detail records that are in order by date and then by department number. There is a transaction amount associated with each record and a running sum of the transaction amount. There is also a running count of the records.

```
/INFILE=sales
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)
/SORT
  /KEY=date
  /KEY=dept
/OUTFILE=run1.out
  /HEADREC="Count  Date          Dept  Store    Amount  Running Amt\n\n"
  /FIELD=(runcount, POSITION=2, SIZE=3, PRECISION=0, NUMERIC)
  /FIELD=(date, POSITION=8, SIZE=10, JAPANESE_DATE)
  /FIELD=(dept, POSITION=21, SIZE=2)
  /FIELD=(store, POSITION=27, SIZE=2)
  /FIELD=(amount, POSITION=34, SIZE=7, PRECISION=2, NUMERIC)
  /FIELD=(run_amt, POSITION=45, SIZE=9, PRECISION=2, NUMERIC)
  /SUM run_amt FROM amount RUNNING
  /COUNT runcount RUNNING
```

This will produce the following output records:

Count	Date	Dept	Store	Amount	Running Amt
1	1999-01-04	03	12	589.32	589.32
2	1999-01-05	25	05	498.56	1087.88
3	1999-01-10	17	05	3495.56	4583.44
4	1999-01-12	17	12	54.23	4637.67
5	1999-01-12	25	12	867.32	5504.99
6	1999-01-13	25	01	2103.54	7608.53
7	1999-01-15	10	01	8712.45	16320.98
8	1999-01-17	25	01	98.63	16419.61
9	1999-01-18	03	01	239.39	16659.00
10	1999-01-19	10	12	4098.34	20757.34

Example: Running Summary with Subtotals

Suppose that in the previous example you want to group the records with respect to store number and have subtotals for amount. You still want the running totals. The following script, **running_sub.scl**, can be used for this:

```
/INFILE=sales
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)
/SORT
  /KEY=store
  /KEY=dept
/OUTFILE=run2.out
  /RECSPPERPAGE=1
  /HEADREC="          -----\n"
  /DATA="    STORE:"
  /FIELD=(store, POSITION=11, SIZE=2)
  /DATA="    COUNT:"
  /FIELD=(counter, POSITION=21, SIZE=3, PRECISION=0, NUMERIC)
  /FIELD=(sub_amt, POSITION=26, SIZE=9, PRECISION=2)
  /DATA="\n"
  /SUM sub_amt FROM amount BREAK store
  /COUNT counter BREAK store
/OUTFILE=run2.out
  /HEADREC="Count   Date           Dept       Amount   Running Amt\n\n"
  /FIELD=(runcount, POSITION=1, SIZE=3, PRECISION=0, NUMERIC)
  /FIELD=(date, POSITION=8, SIZE=10, JAPANESE_DATE)
  /FIELD=(dept, POSITION=21, SIZE=2)
  /FIELD=(amount, POSITION=28, SIZE=7, PRECISION=2, NUMERIC)
  /FIELD=(run_amt, POSITION=39, SIZE=9, PRECISION=2)
  /SUM run_amt FROM amount RUNNING
  /COUNT runcount RUNNING
```

Following is the output:

Count	Date	Dept	Amount	Running Amt
1	1999-01-18	03	239.39	239.39
2	1999-01-15	10	8712.45	8951.84
3	1999-01-17	25	98.63	9050.47
4	1999-01-13	25	2103.54	11154.01

STORE: 01	COUNT	4	11154.01	
5	1999-01-10	17	3495.56	14649.57
6	1999-01-05	25	498.56	15148.13

STORE: 05	COUNT	2	3994.12	
7	1999-01-04	03	589.32	15737.45
8	1999-01-19	10	4098.34	19835.79
9	1999-01-12	17	54.23	19890.02
10	1999-01-12	25	867.32	20757.34

STORE: 12	COUNT	4	5609.21	

Example: Running Using a Lag Count

A windowing feature showing running aggregation values over a selected number of previous records can be produced using a lag count with `RUNNING`.

Consider the following input file, **sales.in**:

01	100.00
01	200.00
01	300.00
01	400.00
01	500.00
01	600.00
01	700.00
02	10.00
02	20.00
02	30.00
02	40.00
02	50.00
02	60.00
02	70.00
03	1.00
03	2.00
03	3.00
03	4.00
03	5.00
03	6.00
03	7.00

The following script, **lagfunction.scl**, demonstrates usage of the windowing feature on aggregate functions.

```
/INFILE=sales.in
/FIELD=(store,POSITION=1,SIZE=2)
/FIELD=(amount, POSITION=7, SIZE=7,PRECISION=2,NUMERIC)

/REPORT

/OUTFILE=lagreport.out
/HEADREC="#      Amt      ct [2]      sum [3]      avg [4]      std [5]      min [6]      max [7] \n\n"
/FIELD=(store,POSITION=1,SIZE=2)
/FIELD=(amount,POSITION=4,SIZE=9,PRECISION=2,NUMERIC)
/FIELD=(cnt,POSITION=15,SIZE=4,PRECISION=0,NUMERIC)
/FIELD=(sum,POSITION=25,SIZE=9,PRECISION=2,NUMERIC)
/FIELD=(avg,POSITION=35,SIZE=9,PRECISION=2,NUMERIC)
/FIELD=(std,POSITION=45,SIZE=9,PRECISION=2,NUMERIC)
/FIELD=(min,POSITION=55,SIZE=9,PRECISION=2,NUMERIC)
/FIELD=(max,POSITION=65,SIZE=9,PRECISION=2,NUMERIC)

/COUNT      cnt RUNNING 2 break store
/sum        sum FROM amount RUNNING 3 break store
/average    avg FROM amount RUNNING 4 break store
/standard   std FROM amount RUNNING 5 break store
/minimum    min FROM amount RUNNING 6 break store
/maximum    max FROM amount RUNNING 7 break store
```

The results are shown below:

#	Amt	ct [2]	sum [3]	avg [4]	std [5]	min [6]	max [7]
01	100.00	1	100.00	100.00	0.00	100.00	100.00
01	200.00	2	300.00	150.00	50.00	100.00	200.00
01	300.00	1	600.00	200.00	81.65	100.00	300.00
01	400.00	2	900.00	250.00	111.80	100.00	400.00
01	500.00	1	1200.00	350.00	141.42	100.00	500.00
01	600.00	2	1500.00	450.00	141.42	100.00	600.00
01	700.00	1	1800.00	550.00	141.42	200.00	700.00
02	10.00	1	10.00	10.00	0.00	10.00	10.00
02	20.00	2	30.00	15.00	5.00	10.00	20.00
02	30.00	1	60.00	20.00	8.16	10.00	30.00
02	40.00	2	90.00	25.00	11.18	10.00	40.00
02	50.00	1	120.00	35.00	14.14	10.00	50.00
02	60.00	2	150.00	45.00	14.14	10.00	60.00
02	70.00	1	180.00	55.00	14.14	20.00	70.00
03	1.00	1	1.00	1.00	0.00	1.00	1.00
03	2.00	2	3.00	1.50	0.50	1.00	2.00
03	3.00	1	6.00	2.00	0.82	1.00	3.00
03	4.00	2	9.00	2.50	1.12	1.00	4.00
03	5.00	1	12.00	3.50	1.41	1.00	5.00
03	6.00	2	15.00	4.50	1.41	1.00	6.00
03	7.00	1	18.00	5.50	1.41	2.00	7.00

27 SEQUENCER

SEQUENCER is an internal field that keeps a running count of records up to a BREAK. The SEQUENCER field can be positioned and sized, as required, and is particularly useful in database indexing and re-loading work. It can be used in the /INREC or /OUTFILE section of a **sortel** job script.

The field named SEQUENCER can appear one time in an /INREC section, or one time in an output file layout (or one time for each separate /OUTFILE definition, if applicable) with the following syntax:

```
/FIELD=(SEQUENCER[ = [+ / -]n], [field attributes])
```

where *n* is a whole number and you choose either "+" or "-" to indicate whether the initial value is positive or negative. If no initial value is given, then the initial value is 1. If no sign is given, then the sign is assumed to be positive.



Because of different memory and disk allocations on the same machine, or on different machines, the sequence of common key records returning from a sort is not always equal. Adding a SEQUENCER field on /INREC to be used as a /KEY field will ensure that the sequence of returning records will be unique. In this way, a sort will produce the same order across all environments.

Also, you can add a SEQUENCER field on /INREC to place an ID tag in the output records to identify which original records were included, and which records were eliminated, in a sort with /STABLE and /NODUPPLICATES (see *Stability on page 167* and *No Duplicates, Duplicates Only on page 168*). Otherwise, it would not be possible to ascertain which records were kept or deleted.

Below are possible usages:

```
/FIELD=(SEQUENCER, POSITION=5, SIZE=4)
/FIELD=(SEQUENCER = -10, POSITION=5, SIZE=4)
/FIELD=(SEQUENCER = +100, POSITION=5, SIZE=4)
```

Example: Using Sequencer

If you have the following input data:

Roosevelt, Theodore	1901-1909	REP	NY
Taft, William H.	1909-1913	REP	OH
Harding, Warren G.	1921-1923	REP	OH
Coolidge, Calvin	1923-1929	REP	VT
Hoover, Herbert C.	1929-1933	REP	IA
Roosevelt, Franklin D.	1933-1945	DEM	NY
Eisenhower, Dwight D.	1953-1961	REP	TX
Kennedy, John F.	1961-1963	DEM	MA
Johnson, Lyndon B.	1963-1969	DEM	TX
Nixon, Richard M.	1969-1973	REP	CA
Ford, Gerald R.	1973-1977	REP	NE
Carter, James E.	1977-1981	DEM	GA
Reagan, Ronald W.	1981-1989	REP	IL
Bush, George H.W.	1989-1993	REP	TX
Clinton, William J.	1993-2001	DEM	AR

and sort using the following script file, **sequencer.scl**:

```
/INFILE=chiefs_1900
/FIELD=(President,POSITION=1,SIZE=27)
/FIELD=(Term,POSITION=28,SIZE=9)
/FIELD=(Party,POSITION=40,SIZE=3)
/FIELD=(State,POSITION=45,SIZE=2)
/SORT
/KEY=President
/OUTFILE=out
/FIELD=(President,POSITION=1,SIZE=22)
/FIELD=(SEQUENCER,POSITION=25,SIZE=3)
/FIELD=(Party,POSITION=30,SIZE=3)
```

the output records will be as follows:

Bush, George H.W.	1	REP
Carter, James E.	2	DEM
Clinton, William J.	3	DEM
Coolidge, Calvin	4	REP
Eisenhower, Dwight D.	5	REP
Ford, Gerald R.	6	REP
Harding, Warren G.	7	REP
Hoover, Herbert C.	8	REP
Johnson, Lyndon B.	9	DEM
Kennedy, John F.	10	DEM
Nixon, Richard M.	11	REP
Reagan, Ronald W.	12	REP
Roosevelt, Franklin D.	13	DEM
Roosevelt, Theodore	14	REP
Taft, William H.	15	REP

Since there are 27 presidents prior to 1900, we can start the SEQUENCER field at 28 by changing the SEQUENCER field as in the following:

```
/Field=(SEQUENCER = +28,POSITION=25,SIZE=3)
```

Then the output is:

Bush, George H.W.	28	REP
Carter, James E.	29	DEM
Clinton, William J.	30	DEM
Coolidge, Calvin	31	REP
Eisenhower, Dwight D.	32	REP
Ford, Gerald R.	33	REP
Harding, Warren G.	34	REP
Hoover, Herbert C.	35	REP
Johnson, Lyndon B.	36	DEM
Kennedy, John F.	37	DEM
Nixon, Richard M.	38	REP
Reagan, Ronald W.	39	REP
Roosevelt, Franklin D.	40	DEM
Roosevelt, Theodore	41	REP
Taft, William H.	42	REP

When generating an output file that has both summary BREAK records and detail records, SEQUENCER can be used in both types of records. Every time there is a BREAK record, the counter for the detail records gets reset. Also, SEQUENCER gives a running count for the BREAK records. The following script file and output data demonstrate this:

```
/INFILE=chiefs_1900
/Field=(President,POSITION=1,SIZE=27)
/Field=(Term,POSITION=28,SIZE=4)
/Field=(Party,POSITION=40,SIZE=3)
/SORT
/KEY=Party
/KEY=President
/OUTFILE=out # Summary BREAK record
/DATA=" Party"
/Field=(Sequencer,POSITION=18,SIZE=3)
/DATA=" Count"
/Field=(How_many,POSITION=30,SIZE=3)
/Field=(Party,POSITION=40,SIZE=3)
/CONDITION=(NP,TEST=(Party))
/COUNT How_many BREAK NP
/OUTFILE=out # Detail record
/Field=(President,POSITION=1,SIZE=22)
/Field=(Sequencer,POSITION=25,SIZE=3)
```

The output is as follows:

Carter, James E.	1		
Clinton, William J.	2		
Johnson, Lyndon B.	3		
Kennedy, John F.	4		
Roosevelt, Franklin D.	5		
Party 1	Count 5		DEM
Bush, George H.W.	1		
Coolidge, Calvin	2		
Eisenhower, Dwight D.	3		
Ford, Gerald R.	4		
Harding, Warren G.	5		
Hoover, Herbert C.	6		
Nixon, Richard M.	7		
Reagan, Ronald W.	8		
Roosevelt, Theodore	9		
Taft, William H.	10		
Party 2	Count 10		REP

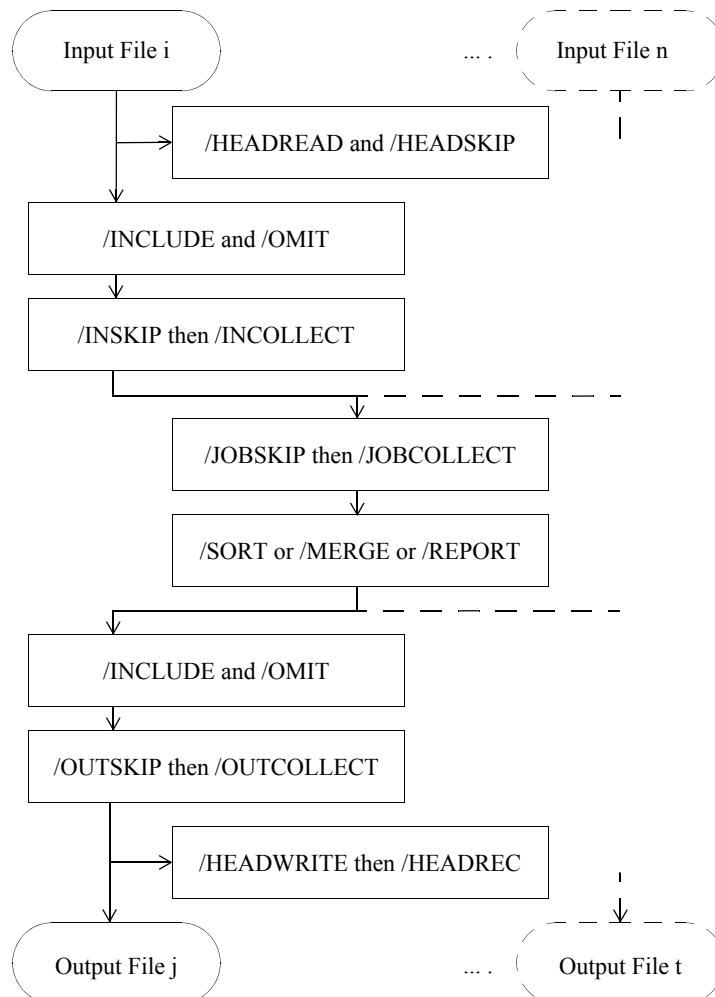
28 RECORD FILTERS

sortcl provides several horizontal record filtering statements for managing the size, number, and flow of records:

- Input Options *on page 260*
- Output Options *on page 268*

The following diagram shows the order in which the commands are applied:

Record Management (Logic Filters)



28.1 Input Options

This section describes the statements relating to the input phase of **sortcl** record management:

- */HEADSKIP*
- */TAILSKIP on page 261*
- */HEADREAD on page 261*
- */TAILREAD on page 263*
- */INSKIP on page 264*
- */INCOLLECT on page 265*
- */JOBSKIP on page 265*
- */JOBCOLLECT on page 266*
- */QUERY on page 266*

/HEADSKIP

This statement causes **sortcl** to skip the first n bytes of an input file before applying the file format used to define the records. The syntax is:

`/HEADSKIP= n`

For example, if **bookstock** begins with the header record:

Title	Publisher	Qty	Price
-------	-----------	-----	-------

it can be skipped over before sorting begins by using:

```
/INFILE=bookstock
  /HEADSKIP=43      # length of header
/OUTFILE=noTop
```

The output of **noTop** contains the sorted file **bookstock**, minus the header.

/TAILSKIP

This statement causes **sortcl** to skip the last n bytes of an input file before applying the file format used to define the records. The syntax is:

`/TAILSKIP= n`

For example, if **bookstock** ends with the record:

End of File

it can be skipped over before sorting begins by using:

```
/INFILE=bookstock
/TAILSKIP=12          # length of tail record
/OUTFILE=noBottom
```

The output of **noBottom** contains the sorted file **bookstock**, minus the final record.

/HEADREAD

This statement reads and saves, but does not process, the first n bytes of an input file. These bytes will be used in the corresponding `/HEADWRITE` statement for an output file (see `/HEADWRITE` on page 269).

`/HEADREAD` is useful for output reporting where a special header record or formatted title needs to appear in the output report, but should not be sorted with the file's records.

The syntax for `/HEADREAD` is:

`/HEADREAD= n`

where n is the number of bytes held for output.

Example: Using /HEADREAD and /HEADWRITE

For this example, a 46-byte header record including a CRLF, with an additional CRLF added to create a blank line, were added to **bookstock**:

Title	Publisher	Qty	Price
Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95
Sending Your	Valley Kill	130	15.75
People Please	Valley Kill	75	11.50
Map Reader	Prentice-Hall	200	14.95

To create reports sorted on price while preserving header data, the specification file **hi-lo.scl** contains:

```
/INFILE=bookstock
/HEADREAD=46          # length of header data
/FIELD=(title,POSITION=1,SIZE=15)
/FIELD=(publisher,POSITION=16,SIZE=13)
/FIELD=(quantity,POSITION=30,SIZE=3,NUMERIC)
/FIELD=(price,POSITION=38,SIZE=5,NUMERIC)
/SORT
/KEY=Price
/OUTFILE=hiprice.out
/INCLUDE WHERE price GT 10
/HEADWRITE=bookstock # header data
/OUTFILE=loprice.out
/OMIT where price GT 10
/HEADWRITE=bookstock # header data
```

The resulting output of **hiprice.out** is:

Title	Publisher	Qty	Price
Reasoning For	Prentice-Hall	150	10.25
People Please	Valley Kill	75	11.50
Still There	Dell	80	13.05
Map Reader	Prentice-Hall	200	14.95
Sending Your	Valley Kill	130	15.75

and the output of **loprice.out** is:

Title	Publisher	Qty	Price
Murder Plots	Harper-Row	160	5.90
Pressure Cook	Harper-Row	228	9.95

/TAILREAD

This statement reads and saves, but does not process, the last n bytes of an input file. These bytes will be used in the corresponding `/TAILWRITE` statement for an output file (see `/TAILWRITE` on page 270).

`/TAILREAD` is useful for output reporting where a special tail record needs to appear in the output report, but should not be sorted with the file's records.

The syntax for `/TAILREAD` is:

`/TAILREAD= n`

where n is the number of bytes held for output.

Example: Using /TAILREAD and /TAILWRITE

For this example, two 43-byte header lines were added to the end of **bookstock**:

Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95
Sending Your	Valley Kill	130	15.75
People Please	Valley Kill	75	11.50
Map Reader	Prentice-Hall	200	14.95
Title	Publisher	Qty	Price

To create reports sorted on price while preserving tail data, the specification file **hi.scl** contains:

```

/INFILE=bookstock
/TAILREAD=46           # length of tail data
/FIELD=(title,POSITION=1,SIZE=15)
/FIELD=(publisher,POSITION=16,SIZE=13)
/FIELD=(quantity,POSITION=30,SIZE=3,NUMERIC)
/FIELD=(price,POSITION=38,SIZE=5,NUMERIC)
/SORT
/KEY=Price
/OUTFILE=hiprice.out
/INCLUDE WHERE price GT 10
/TAIWRITE=bookstock # tail data

```

The resulting output of **hiprice.out** is:

Reasoning For	Prentice-Hall	150	10.25
People Please	Valley Kill	75	11.50
Still There	Dell	80	13.05
Map Reader	Prentice-Hall	200	14.95
Sending Your	Valley Kill	130	15.75
Title	Publisher	Qty	Price

/INSKIP

This statement causes **sortcl** to skip the first *n* number of records from the input source after any /INCLUDE or /OMIT conditions are met.

The syntax for /INSKIP is:

/INSKIP=*n*

where *n* is the number of remaining records to be skipped.

Example: Using /INSKIP

For example, using **bookstock** as the input (see Example: *on page 263*), the following script, **inskip.scl**:

```
/INFILE=bookstock
/FIELD=(title,POSITION=1,SIZE=15)
/FIELD=(publisher,POSITION=16,SIZE=13)
/FIELD=(quantity,POSITION=30,SIZE=3,NUMERIC)
/FIELD=(price,POSITION=38,SIZE=5,NUMERIC)
/INCLUDE WHERE price GT 10
/INSKIP=2
/OUTFILE=inskip.out
```

produces this output:

Map Reader	Prentice-Hall	200	14.95
People Please	Valley Kill	75	11.50
Sending Your	Valley Kill	130	15.75

In this case, the records with the titles Reasoning For and Still There were skipped because they were the first two input records that satisfied the condition.

/INCOLLECT

This statement determines the maximum number of records to be accepted and processed from an input source. This statement is placed in the script after any other input filters for the input file are specified. The syntax is:

```
/INCOLLECT=n
or
/INCOLLECT=PERMUTE
```

where *n* is the number of records to be processed and PERMUTE is all possible permutations after all /INCLUDE, /OMIT, and /INSKIP filters are satisfied. PERMUTE is used only with PROCESS=RANDOM.

Example: Using /INSKIP and /INCOLLECT

The following specification file, **incollect.scl**, demonstrates the use of /INSKIP and /INCOLLECT on the file **bookstock**:

```
/INFILE=bookstock
/FIELD=(title, POSITION=1, SIZE=15)
/FIELD=(publisher, POSITION=16, SIZE=13)
/FIELD=(quantity, POSITION=30, SIZE=3, NUMERIC)
/FIELD=(price, POSITION=38, SIZE=5, NUMERIC)
/INCLUDE WHERE price GT 10
/INSKIP=2
/INCOLLECT=2
/OUTFILE=incollect.out
```

Upon execution, the /INCLUDE condition was tested first, so that only records where the price exceeded \$10.00 were accepted. Then, the /INSKIP statement skipped the next two records. The /INCOLLECT statement accepted the next two records only, and sorted from left to right. These records are output to the file **incollect.out**:

People Please	Valley Kill	75	11.50
Sending Your	Valley Kill	130	15.75

/JOBSKIP

This statement causes **sortcl** to skip the first *n* number of records from your input data prior to processing. It is typically used when you have multiple input sources.

/JOBSKIP operates on records that are still remaining after any /INCLUDE or /OMIT logic is satisfied, and after any /INSKIP and /INCOLLECT filters have been applied to individual inputs.

The syntax for `/JOBSKIP` is:

`/JOBSKIP=n`

where *n* is the number of remaining records to be skipped.



The order that you specify your input files within your script is important because a `/JOBSKIP` statement is applied to your input data in a sequential fashion. For example, if you have 4 input files consisting of 25 records each, and `/JOBSKIP` is set to 50, then only records from the third and fourth input files will be processed, provided that no other filter logic has excluded them.

`/JOBCOLLECT`

This statement determines the maximum number of records accepted from your input data. It is typically used to limit total processing volume when there are multiple input sources. `/JOBCOLLECT` operates on records that are still remaining after any `/INCLUDE` or `/OMIT` logic is satisfied, and, if specified, after a `/JOBSKIP` statement has been applied (see `/JOBSKIP` on page 265).

The syntax is:

`/JOBCOLLECT=n`

where *n* is the number of records to be processed.



The order in which you specify your input sources within your script is important because a `/JOBCOLLECT` statement is applied to your input data in a sequential fashion. For example, if you have 4 input files consisting of 25 records each, and `/JOBCOLLECT` is set to 50 (with no `/JOBSKIP` statement), then only records from the first and second inputs will be processed, provided that no other filter logic has excluded them.

`/QUERY`

This statement introduces a standard Java-style quoted string. The string must contain a valid SQL `SELECT` statement. All columns returned from the query will be mapped to the declared fields in the section by position. The field or column names will be ignored.

The syntax is:

```
/QUERY=SELECT statement
```

where *statement* is a valid SQL statement.

For example:

```
/QUERY="SELECT (SALARIO + 100),SEG_SOC FROM ANULLS WHERE SALARIO > 10"  
/FIELD=(SALARIO,TYPE=NUMERIC,POSITION=1, SEPARATOR=",")  
/FIELD=(SOC_NO,TYPE=ASCII,POSITION=2,SEPARATOR=",", FRAME='\')
```

The query returns two columns per row. The first column value is mapped to the SALARIO field, the second column value is mapped to SOC_NO, even though the names do not match.

28.2 Output Options

This section describes the statements relating to the output phase of **sortcl** record management:

- */CREATE*
- */APPEND*
- */UPDATE on page 269*
- */HEADWRITE on page 269*
- */TAILWRITE on page 270*
- */HEADREC on page 270*
- */FOOTREC on page 272*
- */RECSPPERPAGE on page 272*
- */OUTSKIP on page 275*
- */OUTCOLLECT on page 275*

/CREATE

This is the default specification for an output target. It indicates that a new output file will be created or an existing table will be truncated. If the file name already exists, all previous data in the file will be lost, even if nothing is written by this job.

The syntax is:

```
/OUTFILE=output filename  
/CREATE
```

/APPEND

Associate an */APPEND* with any target to cause output data to be placed after the existing data in a file or table. If the file does not exist or is empty, */CREATE* will be invoked.

The syntax is:

```
/OUTFILE=output_filename  
/APPEND
```

/UPDATE

The /UPDATE statement introduces a third option to the choice of /APPEND and /CREATE. The /UPDATE= command must include one or more key fields to use in the WHERE clause of the resulting UPDATE command in SQL. Note that the key fields must exist somewhere in the INPUT.

The syntax is:

```
/OUTFILE=output_filename
/PROCESS=ODBC
/UPDATE=(field)
```

```
/INFILE="acards;DSN=mytwister"
/PROCESS=ODBC
/FIELD=(TOKEN,TYPE=NUMERIC,POSITION=1,SEPARATOR="\t",PRECISION=0,EXT_FIELD="token")
/FIELD=(CARD_NO,TYPE=ASCII,POSITION=2,SEPARATOR="\t",EXT_FIELD="card_no")

/OUTFILE="acards;DSN=mytwister"
/PROCESS=ODBC
/UPDATE=(TOKEN)
/FIELD=(SUB_CARD_NO=sub_string(CARD_NO, -4, 4),TYPE=ASCII,\
POSITION=2,SEPARATOR="\t",EXT_FIELD="card_no")
```

Here, token is extracted because it is the key. The token value will not change or be written back to the table, but it is needed to uniquely identify the row into which the modified card_no value will be put. The resulting SQL statement used to update the target table will be:

```
UPDATE acards SET card_no=SUB_CARD_NO WHERE token=TOKEN;
```

/HEADWRITE

This statement causes a header record (the first record) that was scanned from an input file using /HEADREAD to be written in an output file. The header record is an exact copy of the header from a corresponding /HEADREAD statement associated with an input file (see /HEADREAD on page 261).

The syntax is:

```
/HEADWRITE=input_filename
```

The /HEADREAD and /HEADWRITE statements can be used to transfer binary data as well as ASCII text.

See the definition in */HEADREAD on page 261* for an example of how */HEADWRITE* is used.



If the */HEADREC* statement is used in addition to */HEADWRITE*, the */HEADREC* string is displayed second.

/TAILWRITE

This statement causes a tail record (the last record) that was scanned from an input file using */TAILREAD* to be written in an output file. The tail record will be an exact copy of the tail record from the corresponding */TAILREAD* statement associated with an input file (see */TAILREAD on page 263*).

The syntax is:

/TAILWRITE=input filename

The */TAILREAD* and */TAILWRITE* statements can be used to transfer binary data as well as ASCII text.

See the definition in */TAILREAD on page 263* for an example of how */TAILWRITE* is used.



If the */FOOTREC* statement is used in addition to */TAILWRITE*, the */FOOTREC* string is displayed second.

/HEADREC

This statement creates a new customized header record in the output file (report). See */RECSPPERPAGE on page 272* for details on making the header record appear on every page of output.

The syntax is:

*/HEADREC="character string with format for each embedded *
variable (format control characters) ... "[, var1, var2, ...]

The *character string* can be a constant that contains any combination of internal variables, control (escape) characters, and conversion-specifier characters recognized by **sortcl** (see *Table 5 on page 154*, *Table 6 on page 155*, and *Table 7 on page 157*).



A constant string can also contain syntax specific to a mark-up language, such as HTML, provided the browser (or other utility) used to read the output file accepts that syntax. For an example of producing an HTML report, see Example: *on page 314*.

Some examples of using the /HEADREC statement are:

```
/HEADREC="The Monthly Report\n"
/HEADREC="%s                Sales Report\n",CURRENT_DATE
```

where \n indicates a new line, and %s is the format for the variable CURRENT_DATE. See Table 5 *on page 154* for a list of accepted variables.



The \n is needed to cause a line-feed. Without it, the first record will display immediately after the header on the same line. Also, the variables must be listed in the order in which they will appear in the string.

Table 12 lists the display formats known to **sortcl**.

Table 12: Format Control Characters for Variables

Character	Printed Result
%c	character
%d, %i	decimal integer
%u	unsigned decimal integer
%o	unsigned octal integer
%x, %X	unsigned hexadecimal integer
%e	floating point number, such as 4.321e+00
%E	floating point number, such as 4.321E+00
%f	floating point number, such as 4.321
%g	either e-format or f-format, whichever is shorter
%G	either E-format or f-format, whichever is shorter
%s	as a string
%%	writes a single % to the output stream; no argument is converted

/FOOTREC

This statement uses the same syntax as /HEADREC but its string values appear at the bottom of the output data as a footer (see /HEADREC on page 270). See /RECSPPERPAGE on page 272 for details on making the footer record appear on every page of output.

The syntax is:

```
/FOOTREC="character string with format for each embedded variable . . ." [,  
var1, var2, ...]
```

The *character string* can be a constant that contains any combination of internal variables, control (escape) characters, and conversion-specifier characters recognized by **sortcl** (see Table 5 on page 154, Table 6 on page 155, and Table 7 on page 157).



A constant string can also contain syntax specific to a mark-up language, such as HTML, provided the browser (or other utility) used to read the output file accepts that syntax. For an example of producing an HTML report, see Example: on page 314.

An example of using the /FOOTREC statement is:

```
/FOOTREC="generated by %s [%d] -----\n\f", \ USER, PAGE_NUMBER
```

/RECSPPERPAGE

This statement sets the number of records displayed on each page of a **sortcl** output report. It is used in conjunction with /FOOTREC and/or /HEADREC. If /RECSPPERPAGE is specified, the /HEADREC header and/or /FOOTREC footer will output on each page. If the /RECSPPERPAGE statement is not given, the header and/or footer will only be displayed once (at the start and end of the file, respectively).

Using an explicit value, for example:

```
/RECSPPERPAGE=10
```

After every 10 records have been displayed or written, the header and/or footer will be repeated if /HEADREC and/or /FOOTREC are defined.



Using `/RECSPPERPAGE` does not actually cause a page break; it designates the number of records to be printed before printing the footer and header records again. In order to force a pagebreak, a "`\f`" should be included in the `/FOOTREC` statement.



The statement used alone reads the user's environment variable, `LINES`, and uses it for the value of `RECSPPERPAGE`. Therefore, if `LINES` is set to 25, the header and/or footer will output every twenty-five records. It would appear in the script as `/RECSPPERPAGE`. ◆

Example: Using `/RECSPPERPAGE`

Using the input file **bookstock** and the following script, **recsperpage.scl**:

```
/INFILE=bookstock
  /FIELD= (title, POSITION=1, SIZE=15)
  /FIELD= (publisher, POSITION=16, SIZE=13)
  /FIELD= (quantity, POSITION=30, SIZE=3, NUMERIC)
  /FIELD= (price, POSITION=38, SIZE=5, NUMERIC)
/OUTFILE=bookstock.out # Detail Records
  /RECSPPERPAGE=5
  /HEADREC="%s          INVENTORY REPORT\nTitles
Qty          Price\n\n", AMERICAN_DATE
  /FOOTREC="\nPage %d\n\n", PAGE_NUMBER
  /FIELD= (title, POSITION=1, SIZE=15)
  /FIELD= (quantity, POSITION=31, SIZE=3, NUMERIC)
  /FIELD= (price, POSITION=47, SIZE=5, NUMERIC)
```

this two-page output is generated:

May/30/2012	INVENTORY REPORT	
Titles	Qty	Price
Map Reader	200	14.95
Murder Plots	160	5.90
People Please	75	11.50
Pressure Cook	228	9.95
Reasoning For	150	10.25

Page 1

May/30/2012	INVENTORY REPORT	
Titles	Qty	Price
Sending Your	130	15.75

Still There

80

13.05

Page 2

/OUTSKIP

This statement skips a given number of sorted or processed records that satisfy any previous `/INCLUDE` or `/OMIT` selection criteria. The syntax is:

```
/OUTSKIP=n
```

where *n* is the number of records to be excluded from the output file to which the `/OUTSKIP` statement applies.

/OUTCOLLECT

This statement sets the number of records sent to an output file after the `/INCLUDE`, `/OMIT`, and `/OUTSKIP` filters are applied. The syntax is:

```
/OUTCOLLECT=n
```

where *n* is the maximum number of records sent to the output file.

Example: Using /HEADREC, /OUTSKIP, and /OUTCOLLECT

Use the input file **miami** in a specification file, **book.scl**:

```
/INFILE=miami
/FIELD=(title,POSITION=1,SIZE=15)
/FIELD=(publisher,POSITION=16,SIZE=15)
/FIELD=(qty,POSITION=30,SIZE=3)
/FIELD=(price,POSITION=38,SIZE=5, numeric)
/CONDITION=(OverTen,TEST=(price GT 10))
/OUTFILE=miami.rpt
/HEADREC="Title           Publisher      Qty      Price\n\n"
/INCLUDE=(COND=OverTen)
/OUTSKIP=2
/OUTCOLLECT=3
```

The output is:

Title	Publisher	Qty	Price
Reasoning For	Prentice-Hall	150	10.25
Sending Your	Valley Kill	130	15.75
Still There	Dell	80	13.05

The `/HEADREC` was written first to the output report **miami.rpt**, and is followed by a line-feed character ("`\n`").

After the sort, the first two records with prices over \$10.00 in the file (Map Reader and People Please) were discarded by `/OUTSKIP`. Then, `/OUTCOLLECT` accepted the next three records with prices over \$10.00, as shown above.

29 MISCELLANEOUS OPTIONS

This section describes the miscellaneous statements relating to neither input nor output:

- `/RC`
- `/EXECUTE` *on page 277*
- `/MONITOR` *on page 277*
- Runtime Warnings (`/WARNINGSON` and `/WARNINGSOFF`) *on page 280*
- `/ERRORCOUNT` *on page 280*
- `/ROUNDING` *on page 281*
- `/LOCALE` *on page 281*

29.1 /RC

This statement bypasses **sortcl** data processing so that you can:

- display all tuner settings, including visible and non-visible default settings, that will be applied to **sortcl** jobs (see *Using Customized Resource Control Files on page 615*):

```
sortcl /rc
```

- test for errors (and warnings, if `/WARNINGSON` is set in the job script) to assist in debugging a job specification file:

```
sortcl /specification=filename.scl /rc
```

You can also add an `/RC` statement anywhere in a job specification file.



The `/RC` statement was named `/DEBUG` in versions of **CoSort** prior to v8.2.2. The `/DEBUG` statement is still supported.

29.2 /EXECUTE

This statement causes the operating system shell to execute the specified command. It can be placed anywhere in the job script (see Example: *on page 316*). The syntax is:

```
/EXECUTE="command_statement"
```

An example is:

```
/EXECUTE "echo 'SORTCL began  '`date`'"
/INFILE=.....
```

When the above was executed, the following was echoed to the console:

```
SORTCL began  Thu May 19 18:52:33 EST 2011
```

29.3 /MONITOR

You can monitor the progress of your job by setting a level of runtime verbosity that will report through **stderr**. Important events such as job start and stop, file opens and closes, and record throughput can each be reported with a timestamp. The syntax of the statement is:

```
/MONITOR=n
```

where *n* is a whole number greater than or equal to 0 and less than 16. The statement can appear anywhere in the job script. The `MONITOR_LEVEL` can also be set in the **cosortrc** file or Windows registry (see `MONITOR_LEVEL` level number *on page 624*).

Below is a chart describing the levels:

Level	Description
0	no monitoring
1	Show job initiation and job completion. This includes for the job itself, the sort processes, and the merge processes.
2	Includes Level 1 plus the opening and closing of input and output files.
3-9	Includes Level 2 plus the opening and closing of temporary files. Each progressive level will show number of records processed with an increasing degree of frequency.
3	every 1,000,000 records
4	every 100,000 records
5	every 10,000 records

Level	Description
6	every 1,000 records
7	every 100 records
8	every 10 records
9	every 1 record
10-14	undefined
15	Use the monitor value in the cosortrc file.



Setting the monitor verbosity level higher than 2 can degrade job performance. Overhead is required to monitor and report the progress of individual (groups) of records. The higher the level, the greater the impact.

If you have a job using the input file **t.in**, and output file **t.out**, then the following are examples of the reporting by /MONITOR:

Level 1

```
CoSort Version 9.5.1 R91110204-1205 © 1978-2011 IRI, Inc. www.cosort.com
EDT 03:42:41 PM Friday, February 26 2011
<00:00:00.00> event (57): sortcl /infile=t.in /outfile=t.out /monitor=1 initiated
<00:00:00.02> event (66): cosort() process begins
<00:00:02.03> event (67): cosort() process ends
<00:00:02.03> event (58): sortcl /infile=t.in /outfile=t.out /monitor=1 completed
EDT 03:42:43 CoSort serial # 11027.9518 6 CPUs Non-expiring
```

Level 2

```
CoSort Version 9.5.1 R91090204-1205 © 1978-2011 IRI, Inc. www.cosort.com
EDT 03:42:41 PM Friday, February 26 2011
<00:00:00.00> event (57): sortcl /infile=t.in /outfile=t.out /monitor=2 initiated
<00:00:00.07> event (66): cosort() process begins
<00:00:00.07> event (59): t.in infile opened
<00:00:00.05> event (60): t.in infile closed
<00:00:01.22> event (61): t.out outfile opened
<00:00:02.01> event (62): t.out outfile closed
<00:00:02.02> event (67): cosort() process ends
<00:00:02.02> event (58): sortcl /infile=t.in /outfile=t.out /monitor=2 completed
EDT 03:42:59 CoSort serial # 11027.9518 6 CPUs Non-expiring
```

Level 3

Because this sort requires the creation of temporary files, the lines pertaining to *workfiles* are displayed, which shows that the temporary files are being deleted as they are merged together. Below is the final output that /MONITOR will display at this level:

```
CoSort Version 9.5.1 R91090204-1205 © 1978-2011 IRI, Inc. www.cosort.com
EDT 03:42:41 PM Friday, August 26 2011
<00:00:00.00> event (57): sortcl /infile=t.in /outfile=t.out /monitor=3 initiated
<00:00:00.04> event (66): cosort() process begins
<00:00:00.03> event (59): t.in infile opened
<00:00:00.01> event (60): CS00001400 workfile opened
<00:00:00.36> event (61): t.in infile closed
<00:00:01.07> event (62): t.out outfile opened
<00:00:01.89> event (62): t.out outfile closed
<00:00:01.90> event (62): CS00001400 workfile deleted
<00:00:01.90> event (62): cosort() process ends
<00:00:01.90> event (62): sortcl /infile=t.in /outfile=t.out /monitor=3 completed
EDT 03:43:15 CoSort serial # 13245.0905 6 CPUs Non-expiring
```

Level 5

When using a large input file, for example **s.in**, you will see, at this level, when every 10,000 records are processed as the job is running:

```
CoSort Version 9.5.1 R91090204-1205 © 1978-2011 IRI, Inc. www.cosort.com
EDT 03:42:41 PM Friday, August 26 2011
<00:00:00.00> event (57): sortcl /infile=s.in /outfile=s.out /monitor=5 initiated
<00:00:00.02> event (66): cosort() process begins
<00:00:00.23> event (59): s.in infile opened
<00:06:08.96> event (65): µ1, 50000 processed
. . .
```

If there are criteria for accepting or rejecting records, you will periodically see the accepted and rejected line as the job is running:

```
CoSort Version 9.5.1 R91090204-1205 © 1978-2011 IRI, Inc. www.cosort.com
EDT 03:42:41 PM Friday, August 26 2011
<00:00:00.00> event (57): sortcl /infile=s.in /outfile=s.out /monitor=5 initiated
<00:00:00.02> event (66): cosort() process begins
<00:00:00.23> event (59): s.in infile opened
<00:00:14.08> event (69): accepted 78571 rejected 141429
. . .
```

29.4 Runtime Warnings (/WARNINGSON and /WARNINGSOFF)

The command /WARNINGSON captures warning messages that do not stop execution, but do indicate where some steps might have been omitted. /WARNINGSOFF is the default and turns off the output of the warning messages.

By including /WARNINGSON, **sortcl** displays warning messages via **stderr**.

The two commands used together allow the user to select when the warning messages will be generated. This can pertain to number of input and output files, for example:

```
/SPECIFICATION=stores.ddf
/WARNINGSON           # warnings on for chicago
/INFILE=chicago
/WARNINGSOFF          # warnings off
/INFILE=miami
/INREC
    /FIELD=(Title,POSITION=1,SIZE=15)
    /FIELD=(Publisher,POSITION=16,SIZE=16)
    /FIELD=(Price,POSITION=32,SIZE=10,NUMERIC)
/SORT
    /KEY=(Price,DES)
    /NODUPPLICATES
/WARNINGSON           # warnings on for output
/OUTFILE=pubs
/OUTFILE=titles
/WARNINGSOFF          # warnings off
```

29.5 /ERRORCOUNT

If any errors are present in your job script that would prevent normal execution, **sortcl** displays these errors to **stderr** and the job is not executed. The default limit to the number of errors displayed in this case is eight. Add the /ERRORCOUNT option to a job script to override this default. The syntax is as follows:

```
/ERRORCOUNT=n
```

where *n* is the total number of errors to be displayed.

29.6 /ROUNDING

Different hardware systems produce different results as a result of rounding arithmetic operations. These differences occur in the 12th or 13th decimal digit, but can become apparent after truncation.

The `/ROUNDING` statement, which should be placed at or near the top of your **sortcl** script, is used to control how rounding is to be performed throughout the execution of the job. One of two options are available:

<code>/ROUNDING=SYSTEM</code>	The default. This produces results in the manner determined by your operating system.
<code>/ROUNDING=NEAREST</code>	For display/output purposes (only) for the results of an arithmetic operation, this option causes half a decimal digit to be added to the least significant digit. For example, if an actual result is .794999, the <code>NEAREST</code> option will cause a display of .80 when precision is specified as 2.



The `ROUNDING=NEAREST` option slows numeric output. If your calculations do not require this level of accuracy, or if your system rounds in your preferred manner by default, then `/ROUNDING=NEAREST` should not be specified.

29.7 /LOCALE

sortcl allows the user to set another language environment. This will affect the following processes:

- collation
- character classification and case conversion
- numeric formatting
- some message language
- currency formatting
- date and time formatting.

To set the **sortcl** locale, use the following command:

```
/LOCALE=[language[_territory[.codeset]]]
```

Refer to your system's International Language Support documentation.

If your operating system does not support your desired language, contact IRI for further assistance.

If the `/LOCALE` command is used without options, the locale is the current environment. Therefore, the `/LOCALE` command by itself is needed when switching back to the current locale.

Example: Using /LOCALE with Currency

The values in the input file **values.dat** will be converted to British currency:

```
5
16
138
0
227
```

The following file, **Gvalues.scl**, uses a `/LOCALE` command (with language option) to perform the conversion:

```
/LOCALE=en_UK # sets the language
/INFILE=values.dat
  /FIELD=(value,POSITION=1,SIZE=3,NUMERIC)
/REPORT      # no sorting
/OUTFILE=Gvalues.out
  /FIELD=(value,POSITION=1,SIZE=7,CURRENCY)
  # resized for extra characters
```

On execution, **Gvalues.out** contains:

```
£5.00
£16.00
£138.00
£0.00
£227.00
```

Notice that the value given for `/LOCALE` in this example is different than the value in the next example. This is because these values are operating-system-specific. `CURRENCY` values are sorted in the order defined by the given locale. Using **Gvalues.out** as the input data, create **Gsort.scl** as follows:

```

/LOCALE=de
/STATISTICS
/INFILE=Gvalues.out
  /FIELD=(newvalue, POSITION=1, SIZE=9, CURRENCY)
/SORT
  /KEY=newvalue
/OUTFILE=Gsort.out
  /FIELD=(newvalue, POSITION=1, SIZE=9, CURRENCY)

```

Gsort.scl produces the following statistics:

```

SORTCL 9.5.1          R91090204-1205      sortcl STATISTICS
(c) 1978-2011 Innovative Routines International

```

```

Lic.: CoSort / IRI, Inc. Ser#: 07011.0915
Loc.: de User: unknown
Cmd.: sortcl /specification=Gsort.spec
Eastern Daylight Time 10:57:42 AM Freitag, F  v 8 2008

```

```

Input Files          Record Length
(001) Gvalues.out          0

```

Common Sort Record Length 0 bytes (variable)

```

Key   Dir  Location  Pos  Size  Data Type
(001) Asc   Fixed    1    9   External Numeric

```

```

Output Files          Record Length
(001) Gsort.out          0

```

```

Blocksize  MemorySize  ExtMaxMemory  RecsInMemory  Buffers
    320kb      120mb      120mb          512kb          2

```

```

Work Areas  Directory
(01)        ./

```

```

Records processed:      5 sorted
Began:  18:14:15
Ended:  18:14:16
Total:  00:00:01

```

Example: Using /LOCALE with Dates

Convert dates into a new language environment (to change their form, see */LOCALE on page 281*). Using **date.dat** (with English months):

```
Jul/01/1969
Nov/21/1966
Apr/07/1995
Jan/12/1975
Apr/18/1990
Jun/05/1962
Feb/09/1980
Mar/16/1962
May/04/1948
Dec/18/1995
Oct/12/1978
```

create a specification file, **Fdate.scl**:

```
/LOCALE=french # will change language of date
/INFILE=date.dat
  /FIELD=(when, POSITION=1, SIZE=11, AMERICAN_DATE)
/SORT
  /KEY=when
/OUTFILE=stdout
  /FIELD=(when, POSITION=1, SIZE=12, AMERICAN_DATE)
  /FIELD=(when, POSITION=16, SIZE=12, EUROPEAN_DATE)
```

Upon execution, **Fdate.scl** produces:

mai/04/1948	04.05.1948
mar/16/1962	16.03.1962
jui/05/1962	05.06.1962
nov/21/1966	21.11.1966
jui/01/1969	01.07.1969
jan/12/1975	12.01.1975
oct/12/1978	12.10.1978
fÅv/09/1980	09.02.1980
avr/18/1990	18.04.1990
avr/07/1995	07.04.1995
dÅc/18/1995	18.12.1995

Refer to your operating system documentation for supported system-specific */LOCALE* options.

30 SORTCL JOB SCRIPTING EXAMPLES

This section provides examples of **sortcl** script files and the output generated by them. It will start with simple examples and gradually build to more complex examples. For many examples, the input file **chiefs_10** will be used:

Eisenhower, Dwight D.	134	1953-1961	REP	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Nixon, Richard M.	137	1969-1973	REP	CA
Ford, Gerald R.	138	1973-1977	REP	NE
Carter, James E.	139	1977-1981	DEM	GA
Reagan, Ronald W.	140	1981-1989	REP	IL
Bush, George H.W.	141	1989-1993	REP	TX
Clinton, William J.	142	1993-2001	DEM	AR
Bush, George W.	123	2001-2009	REP	TX

Example: Simple Script Form

A simple **sortcl** script can contain only the name of the input file. The default action performed is **/SORT**, the entire record is used for the key, and it is compared from left to right. The output records will have the same format as the input records, and be sent to standard out.

For example, **simplest.scl**:

```
/INFILE=chiefs_10
```

produces this **stdout** display:

Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX
Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Ford, Gerald R.	138	1973-1977	REP	NE
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Nixon, Richard M.	137	1969-1973	REP	CA
Reagan, Ronald W.	140	1981-1989	REP	IL

Example: Two-Key Sort

The following script, **two_keys.scl**, adds input fields, two keys, and an output file:

```
/INFILE=chiefs_10
  /FIELD=(president,POSITION=1,SIZ=22)
  /FIELD=(votes,POSITION=24,SIZ=3)
  /FIELD=(service,POSITION=28,SIZ=9)
  /FIELD=(party,POSITION=40,SIZ=3)
  /FIELD=(state,POSITION=45,SIZ=2)
/SORT
  /KEY=party
  /KEY=president
/OUTFILE=chiefs.out
```

Example: Less Verbose

It is necessary to define only those fields to be used for the keys and for the output mapping.



If the exact same fields are defined for input as for output, with the same definitions, the whole record will be written to output.

You would, therefore, get the same results with **two_keys_simpler.scl**:

```
/INFILE=chiefs_10
  /FIELD=(president,POSITION=1,SIZE=22)
  /FIELD=(party,POSITION=40,SIZE=3)
/SORT
  /KEY=party      # SORT is the default action
  /KEY=president
/OUTFILE=chiefs.out
```

The records will be sorted by party. If party compares equally, those records will be ordered by president.

Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Ford, Gerald R.	138	1973-1977	REP	NE
Nixon, Richard M.	137	1969-1973	REP	CA
Reagan, Ronald W.	140	1981-1989	REP	IL

Example: Two-Key Sort with Delimited Fields

If the input file has fields delimited by, or separated by, a pipe (|), the field positions are ordinal:, as shown in **two_keys_delim.scl**:

```
/INFILE=chiefs_10_sep
/FIELD=(president, POSITION=1, SEPARATOR='|')
/FIELD=(votes, POSITION=2, SEPARATOR='|')
/FIELD=(service, POSITION=3, SEPARATOR='|')
/FIELD=(party, POSITION=4, SEPARATOR='|')
/FIELD=(state, POSITION=5, SEPARATOR='|')
/SORT
/KEY=party
/KEY=president
/OUTFILE=chiefs_sep.out
```

The output is as follows:

```
Carter, James E.|139|1977-1981|DEM|GA
Clinton, William J.|142|1993-2001|DEM|AR
Johnson, Lyndon B.|136|1963-1969|DEM|TX
Kennedy, John F.|135|1961-1963|DEM|MA
Bush, George H.W.|141|1989-1993|REP|TX
Bush, George W.|123|2001-2009|REP|TX
Eisenhower, Dwight D.|134|1953-1961|REP|TX
Ford, Gerald R.|138|1973-1977|REP|NE
Nixon, Richard M.|137|1969-1973|REP|CA
Reagan, Ronald W.|140|1981-1989|REP|IL
```

Example: Delimited-to-Fixed Format Change

To make the output in *Example:* match the output produced by the scripts with non-delimited fields in *Example:* and *Example:*, map the output fields as shown in **delim_to_fixed.scl**

```
/INFILE=chiefs_10_sep
/FIELD=(president, POSITION=1, SEPARATOR='|')
/FIELD=(votes, POSITION=2, SEPARATOR='|')
/FIELD=(service, POSITION=3, SEPARATOR='|')
/FIELD=(party, POSITION=4, SEPARATOR='|')
/FIELD=(state, POSITION=5, SEPARATOR='|')
/SORT
/KEY=party
/KEY=president
/OUTFILE=chiefs.out
/FIELD=(president, POSITION=1, SIZE=22)
/FIELD=(votes, POSITION=24, SIZE=3)
/FIELD=(service, POSITION=28, SIZE=9)
/FIELD=(party, POSITION=40, SIZE=3)
/FIELD=(state, POSITION=45, SIZE=2)
```

Example: Using /CHECK

This script, **check.scl**, performs a check to determine if the file **chiefs_10** is in order by party

```
/INFILE=chiefs_10
/FIELD=(president, POSITION=1, SIZE=22)
/FIELD=(votes, POSITION=24, SIZE=3)
/FIELD=(service, POSITION=28, SIZE=9)
/FIELD=(party, POSITION=40, SIZE=3)
/FIELD=(state, POSITION=45, SIZE=2)
/CHECK
/KEY=party
```

If the records are in order, the prompt will return with the following message

```
event (118): records are in order
```

If the records are not in order, the following message will display:

```
error (111): chiefs_10 @ 2; records not in order
```

where 2 is the record number within the file **chiefs_10**.

Example: Using /MERGE

This example folds together (merges) records from two input files, **dems** and **reps**, based on a shared key.



The /MERGE action works properly only if the records in the input files are already sorted by their shared key field (presidents' names in this case) on which you are merging.

Given the input files **reps**:

Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Ford, Gerald R.	138	1973-1977	REP	NE
Nixon, Richard M.	137	1969-1973	REP	CA
Reagan, Ronald W.	140	1981-1989	REP	IL

and **dems**:

Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Truman, Harry S.	133	1945-1953	DEM	MI

use the following script, **merge.scl**, to produce an output file that merges the two input files together by their shared key, presidents' names:

```

/INFILES= (reps,dems)
/FIELD= (president, POSITION=1, SIZE=22)
/FIELD= (votes, POSITION=24, SIZE=3, NUMERIC)
/FIELD= (service, POSITION=28, SIZE=9)
/FIELD= (party, POSITION=40, SIZE=3)
/FIELD= (state, POSITION=45, SIZE=2)
/MERGE
/KEY=president
/OUTFILE=presmerge.out

```

presmerge.out contains:

Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX
Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Ford, Gerald R.	138	1973-1977	REP	NE
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Nixon, Richard M.	137	1969-1973	REP	CA
Reagan, Ronald W.	140	1981-1989	REP	IL
Truman, Harry S.	133	1945-1953	DEM	MI

Example: Using a Data Definition File

Commonly used data definitions should be placed in a data definition file to be referenced by **sortcl** job specification scripts. Since **chiefs_10** and **chiefs_10_sep** are used in many examples, their descriptions can be placed in the corresponding files **chiefs_10.ddf** and **chiefs_10_sep.ddf**.

```
# chiefs_10.ddf
/FIELD=(president, POSITION=1, SIZE=22)
/FIELD=(votes, POSITION=24, SIZE=3, NUMERIC)
/FIELD=(service, POSITION=28, SIZE=9)
/FIELD=(party, POSITION=40, SIZE=3)
/FIELD=(state, POSITION=45, SIZE=2)

# chiefs_10_sep.ddf
/FIELD=(president, POSITION=1, SEPARATOR=" | ")
/FIELD=(votes, POSITION=2, SEPARATOR=" | ")
/FIELD=(service, POSITION=3, SEPARATOR=" | ")
/FIELD=(party, POSITION=4, SEPARATOR=" | ")
/FIELD=(state, POSITION=5, SEPARATOR=" | ")
```

The **sortcl** job scripts can omit the **/FIELD** statements for the infiles, and then include a specification statement after **/INFILE**. For example:

```
/SPECIFICATION=echiefs_10.ddf
```

The script for example Example: *on page 286* can therefore be **two_keys_meta.scl**:

```
/INFILE=chiefs_10
  /SPECIFICATION=chiefs_10.ddf
/SORT
  /KEY=party
  /KEY=president
/OUTFILE=chiefs.out
```

The script for example Example: *on page 287* will be **two_keys_delim_meta.scl**:

```
/INFILE=chiefs_10_sep
  /SPECIFICATION=chiefs_10_sep.ddf
/SORT
  /KEY=party
  /KEY=president
/OUTFILE=chiefs_sep.out
```

Example: Multiple Output Files and Formats

This example produces two output files. One will contain president and state. The other will contain president and party, in order by president. Use the following script, **multiple_out.scl**:

```
/INFILE=chiefs_10
  /SPECIFICATION=chiefs_10.ddf
/SORT
  /KEY=president
/OUTFILE=party.out
  /FIELD=(president,POSITION=5,SIZE=22)
  /FIELD=(party,POSITION=30,SIZE=3)
/OUTFILE=state.out
  /FIELD=(president,POSITION=5,SIZE=22)
  /FIELD=(state,POSITION=30,SIZE=2)
```

The output file **party.out** contains:

Bush, George H.W.	REP
Bush, George W.	REP
Carter, James E.	DEM
Clinton, William J.	DEM
Eisenhower, Dwight D.	REP
Ford, Gerald R.	REP
Johnson, Lyndon B.	DEM
Kennedy, John F.	DEM
Nixon, Richard M.	REP
Reagan, Ronald W.	REP

The output file **state.out** contains:

Bush, George H.W.	TX
Bush, George W.	TX
Carter, James E.	GA
Clinton, William J.	AR
Eisenhower, Dwight D.	TX
Ford, Gerald R.	NE
Johnson, Lyndon B.	TX
Kennedy, John F.	MA
Nixon, Richard M.	CA
Reagan, Ronald W.	IL

Example: Conditional Output Selection

The following two scripts, **include_dems.scl** and **exclude_reps.scl**, produce an output file that contains only DEMS. Since it is more efficient to do the filtering before sorting, the selection statement (**/INCLUDE** or **/OMIT**) is in the input section. The first script filters by including; the second filters by omitting:

```
/INFILE=chiefs_10
/SPECIFICATION=chiefs_10.ddf
/INCLUDE WHERE party EQ "DEM"
/SORT
/KEY=president
/OUTFILE=dems.out
```

```
/INFILE=chiefs_10
/SPECIFICATION=chiefs_10.ddf
/OMIT WHERE party EQ "REP"
/SORT
/KEY=president
/OUTFILE=dems.out
```

Both scripts produce this same output:

Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA

Example: Multiple Output Files with Conditional Selection

This script, **two_filters.scl**, will produce two output files, one with DEMs and one with REPS. Because each is filtered differently, the `/INCLUDE` will be in the output file:

```

/INFILE=chiefs_10
/SPECIFICATION=chiefs_10.ddf
/SORT
/KEY=president
/OUTFILE=reps.out
/INCLUDE WHERE party EQ "REP"
/OUTFILE=dems.out
/INCLUDE WHERE party EQ "DEM"

```

reps.out contains:

Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Ford, Gerald R.	138	1973-1977	REP	NE
Nixon, Richard M.	137	1969-1973	REP	CA
Reagan, Ronald W.	140	1981-1989	REP	IL

dems.out contains:

Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA

Example: Carrying a Condition Forward

This script, **cond_forward.scl**, is similar to *Example:*, but the output files should list only the field `president`. Because the condition field does not exist in the output files, the `/CONDITION` statements must be in the input file:

```
/INFILE=chiefs_10
/SPECIFICATION=chiefs_10.ddf
/CONDITION=(republican,TEST=(party EQ "REP"))
/SORT
/KEY=president
/OUTFILE=reps.out
/INCLUDE WHERE republican
/FIELD=(president,POSITION=1,SIZE=22)
/OUTFILE=dems.out
/OMIT WHERE republican
/FIELD=(president,POSITION=1,SIZE=22)
```

Note how the one condition named on input is used for both output files.

reps.out contains:

```
Bush, George H.W.
Bush, George W.
Eisenhower, Dwight D.
Ford, Gerald R.
Nixon, Richard M.
Reagan, Ronald W.
```

dems.out contains:

```
Carter, James E.
Clinton, William J.
Johnson, Lyndon B.
Kennedy, John F.
```

Example: Report with Complex Selection

The examples in this section show a more complex problem for an /INCLUDE, and two ways to set up the conditions. The output should include presidents whose terms of service started between 1950 and 1975. Those results must have a party of REP and the state cannot be TX. Additionally, presidents whose votes are greater than 140 are also included. Records are processed without sorting or merging (/REPORT).

complex.scl is:

```
/INFILE=chiefs_10
/SPECIFICATION=chiefs_10.ddf
/CONDITION=(range,TEST=(service > "1950" AND service < "1975"))
/CONDITION=(republican,TEST=(party EQ "REP"))
/CONDITION=(texas,TEST=(state NE "TX"))
/CONDITION=(over_140,TEST=(votes GT 140))
/CONDITION=(winner,TEST=(range AND republican AND texas))
/CONDITION=(all_winners,TEST=(winner OR over_140))
/INCLUDE WHERE all_winners
/REPORT
/OUTFILE=chiefs.out
```

The output will be:

Nixon, Richard M.	137	1969-1973	REP	CA
Ford, Gerald R.	138	1973-1977	REP	NE
Bush, George H.W.	141	1989-1993	REP	TX
Clinton, William J.	142	1993-	DEM	AR

Example: Alternate Report with Complex Selection

The resulting output from **complex_alt.scl** is the same as in *Example:* above:

```
/INFILE=chiefs_10
/SPECIFICATION=chiefs_10.ddf
/CONDITION=(range,TEST=(service > "1950" AND service < "1975"))
/CONDITION=(republican,TEST=(party NE "REP"))
/CONDITION=(texas,TEST=(state EQ "TX"))
/CONDITION=(over_140,TEST=(votes GT 140))
/INCLUDE WHERE over_140
/OMIT WHERE texas
/OMIT WHERE republican
/INCLUDE WHERE range
/REPORT
/OUTFILE=chiefs.out
```

Example: Multiple Input Files with Same Formats

Given the input files **reps**:

Bush, George H.W.	141	1989-1993	REP	TX
Bush, George W.	123	2001-2009	REP	TX
Eisenhower, Dwight D.	134	1953-1961	REP	TX
Ford, Gerald R.	138	1973-1977	REP	NE
Nixon, Richard M.	137	1969-1973	REP	CA
Reagan, Ronald W.	140	1981-1989	REP	IL

and **dems**:

Carter, James E.	139	1977-1981	DEM	GA
Clinton, William J.	142	1993-2001	DEM	AR
Johnson, Lyndon B.	136	1963-1969	DEM	TX
Kennedy, John F.	135	1961-1963	DEM	MA
Truman, Harry S.	133	1945-1953	DEM	MI

use the following script, **multi_in.scl**, to combine and order them by name, listing only name and state in the output:

```
/INFILES=(reps,dems)
/SPECIFICATION=chiefs_10.ddf
/SORT
/KEY=president
/OUTFILE=pres.out
/FIELD=(president,POSITION=1,SIZE=22)
/FIELD=(state,POSITION=25,SIZE=2)
```


pres.out contains:

Bush, George H.W.	TX
Bush, George W.	TX
Carter, James E.	GA
Clinton, William J.	AR
Eisenhower, Dwight D.	TX
Ford, Gerald R.	NE
Johnson, Lyndon B.	TX
Kennedy, John F.	MA
Nixon, Richard M.	CA
Reagan, Ronald W.	IL
Truman, Harry S.	MI

Example: Multiple Input Files with Different Formats (/INREC)

Given the input files **chiefs_5p**:

Ford, Gerald R.	REP	NE
Carter, James E.	DEM	GA
Reagan, Ronald W.	REP	IL
Bush, George H.W.	REP	TX
Clinton, William J.	DEM	AR

and **chiefs_5s**:

Truman, Harry S.	MI
Eisenhower, Dwight D.	TX
Kennedy, John F.	MA
Johnson, Lyndon B.	TX
Nixon, Richard M.	CA

Use the following script, **multi_in_inrec.scl**, to combine the names in alphabetical order:

```
/INFILE=chiefs_5p
  /FIELD=(pres,POSITION=1,SIZE=19)
  /FIELD=(party,POSITION=21,SIZE=3)
  /FIELD=(state,POSITION=26,SIZE=2)
/INFILE=chiefs_5s
  /FIELD=(pres,POSITION=1,SIZE=21)
  /FIELD=(state,POSITION=24,SIZE=2)
/INREC
  /FIELD=(pres,POSITION=1,SIZE=21)
/SORT
  /KEY=pres
/OUTFILE=presidents
```

Since the input files do not have all the same fields, /INREC is used to map a common input record to pass to the sort. The output file is:

```
Bush, George H.W.  
Carter, James E.  
Clinton, William J.  
Eisenhower, Dwight  
Ford, Gerald R.  
Johnson, Lyndon B.  
Kennedy, John F.  
Nixon, Richard M.  
Reagan, Ronald W.  
Truman, Harry S.
```

Example: Removing Duplicate Records

Using the same input files from *Example:*, this script, **nodups.scl** will report each state represented:

```
/INFILE=chiefs_5p  
  /FIELD= (name, POSITION=1, SIZE=19)  
  /FIELD= (party, POSITION=21, SIZE=3)  
  /FIELD= (state, POSITION=26, SIZE=2)  
/INFILE=chiefs_5s  
  /FIELD= (name, POSITION=1, SIZE=21)  
  /FIELD= (state, POSITION=24, SIZE=2)  
/INREC  
  /FIELD= (state, POSITION=1, SIZE=2)  
/SORT  
  /KEY=state  
  /NODUPPLICATES  
/OUTFILE=states
```

The output file is:

```
AR  
CA  
GA  
IL  
MA  
MI  
NB  
TX
```

Example: Summary Fields with Breaks

The following input file, **sales**, lists sales amounts with store number, department number, and date:

```
01 25 2103.54 1999-01-13
12 03  589.32 1999-01-04
01 10 8712.45 1999-01-15
05 25  498.56 1999-01-05
12 17   54.23 1999-01-12
12 25  867.32 1999-01-12
05 17 3495.56 1999-01-10
01 25   98.63 1999-01-17
01 03  239.39 1999-01-18
12 10 4098.34 1999-01-19
```

The following **sortcl** script, **sum_break.scl**, will produce total sales, average sales, and the maximum and minimum sale by department. Notice that the key is dept, so the BREAKS will have meaning:

```
/INFILE=sales
/FIELD=(store, POSITION=1, SIZE=2)
/FIELD=(dept, POSITION=4, SIZE=2)
/FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
/FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)
/CONDITION=(newdept, TEST=(dept))
/SORT
/KEY=dept
/OUTFILE=sales.out
/HEADREC="Dept      Total Average Maximum Minimum\n"
/FIELD=(dept, POSITION=2)
/FIELD=(sum_amt, POSITION=6, SIZE=8, PRECISION=2)
/FIELD=(avg_amt, POSITION=15, SIZE=7, PRECISION=2)
/FIELD=(max_amt, POSITION=23, SIZE=7, PRECISION=2)
/FIELD=(min_amt, POSITION=31, SIZE=7, PRECISION=2)
/SUM sum_amt FROM amount BREAK newdept
/AVERAGE avg_amt FROM amount BREAK newdept
/MAX max_amt FROM amount BREAK newdept
/MIN min_amt FROM amount BREAK newdept
```

The output is:

Dept	Total	Average	Maximum	Minimum
03	828.71	414.36	589.32	239.39
10	12810.79	6405.40	8712.45	4098.34
17	3549.79	1774.89	3495.56	54.23
25	3568.05	892.01	2103.54	98.63

Example: Summary Functions with Detail, Break, and Total Records

This following script, **sum_totals.scl**, produces total, subtotal by department, and detail records:

```
/INFILE=sales
/FIELD=(store,POSITION=1,SIZE=2)
/FIELD=(dept,POSITION=4,SIZE=2)
/FIELD=(amount,POSITION=7,SIZE=7,NUMERIC)
/FIELD=(date,POSITION=15,SIZE=10,JAPANESE_DATE)
/CONDITION=(newdept,TEST=(dept))
/SORT
/KEY=dept
/KEY=date
/OUTFILE=sales.out
/RECSPPERPAGE=1
/HEADREC="Dept ----- Average Maximum Minimum\n"
/FOOTREC="\n"
/FIELD=(dept,POSITION=2)
/FIELD=(sum_amt,POSITION=6,SIZE=8,PRECISION=2)
/FIELD=(avg_amt,POSITION=15,SIZE=7,PRECISION=2)
/FIELD=(max_amt,POSITION=22,SIZE=8,PRECISION=2)
/FIELD=(min_amt,POSITION=30,SIZE=8,PRECISION=2)
/SUM sum_amt FROM amount BREAK newdept
/AVERAGE avg_amt FROM amount BREAK newdept
/MAX max_amt FROM amount BREAK newdept
/MIN min_amt FROM amount BREAK newdept
/OUTFILE=sales.out
/HEADREC="=====\n"
/FIELD=(sum_amt,POSITION=5,SIZE=9,PRECISION=2)
/FIELD=(avg_amt,POSITION=15,SIZE=7,PRECISION=2)
/FIELD=(max_amt,POSITION=22,SIZE=8,PRECISION=2)
/FIELD=(min_amt,POSITION=30,SIZE=8,PRECISION=2)
/SUM sum_amt FROM amount
/AVERAGE avg_amt FROM amount
/MAX max_amt FROM amount
/MIN min_amt FROM amount
/OUTFILE=sales.out
/HEADREC="STORE  AMOUNT  DATE\n"
/FIELD=(store,POSITION=3,SIZE=2)
/FIELD=(amount,POSITION=7,SIZE=7,NUMERIC)
/FIELD=(date,POSITION=15,SIZE=10,JAPANESE_DATE)
```

The output is:

```

STORE  AMOUNT  DATE
   12   589.32 1999-01-04
   01   239.39 1999-01-18
Dept  ----- Average Maximum Minimum
   03   828.71  414.36  589.32  239.39

   01  8712.45 1999-01-15
   12  4098.34 1999-01-19
Dept  ----- Average Maximum Minimum
   10 12810.79 6405.40 8712.45 4098.34

   05  3495.56 1999-01-10
   12   54.23 1999-01-12
Dept  ----- Average Maximum Minimum
   17  3549.79  774.89 3495.56   54.23

   05   498.56 1999-01-05
   12   867.32 1999-01-12
   01  2103.54 1999-01-13
   01   98.63 1999-01-17
Dept  ----- Average Maximum Minimum
   25  3568.05  892.01 2103.54   98.63

=====
      20757.34 2075.73 8712.45   54.23

```

Example: Using WHERE with Sums, Cross-Calcs on Sums

This example uses the following data set, **sales2**, which lists retail and wholesale amounts with store number, department number, and date:

```

01 25 2103.54 526.50 1999-01-13
12 03  589.32  23.90 1999-01-04
01 10 8712.45 240.65 1999-01-15
05 25  498.56  34.23 1999-01-05
12 17   54.23  11.25 1999-01-12
12 25  867.32 590.43 1999-01-12
05 17 3495.56 980.34 1999-01-10
01 25   98.63  25.60 1999-01-17
01 03  239.39  89.34 1999-01-18
12 10 4098.34 920.25 1999-01-19

```

The following script, **cross_calc_sums.scl**, produces sums for the profit for each store within each department, where the profit is a cross-calculated value (the difference between the retail and wholesale amounts). It also produces a sum total of all store profits within each department:

```
/INFILE=sales2
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(retail, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(wholesale, POSITION=15, SIZE=6, NUMERIC)
  /FIELD=(date, POSITION=22, SIZE=10, JAPANESE_DATE)
  /CONDITION=(st_01, TEST=(store == "01"))
  /CONDITION=(st_05, TEST=(store == "05"))
  /CONDITION=(st_12, TEST=(store == "12"))
/SORT
  /KEY=dept
/OUTFILE=sales.out
  /HEADREC="Dept StoreProfit_01  StoreProfit_05  StoreProfit_12
DeptProfit\n\n"
  /FIELD=(dept, POSITION=2, SIZE=2)
  /FIELD=(t01, POSITION=12, SIZE=8, PRECISION=2)
  /FIELD=(t05, POSITION=28, SIZE=8, PRECISION=2)
  /FIELD=(t12, POSITION=44, SIZE=8, PRECISION=2)
  /FIELD=(sum_profit, POSITION=57, SIZE=8, PRECISION=2, NUMERIC)
  /SUM t01 FROM (retail - wholesale) WHERE st_01 BREAK dept
  /SUM t05 FROM (retail - wholesale) WHERE st_05 BREAK dept
  /SUM t12 FROM (retail - wholesale) WHERE st_12 BREAK dept
  /SUM sum_profit FROM (retail - wholesale) BREAK dept
```

The output is:

Dept	StoreProfit_01	StoreProfit_05	StoreProfit_12	DeptProfit
03	150.05	0.00	565.42	715.47
10	8471.80	0.00	3178.09	11649.89
17	0.00	2515.22	42.98	2558.20
25	1650.07	464.33	276.89	2391.29

Example: Multiple Output Files with Summaries

Using the same input file from *Example:*, the following script, **multi_out_sums.scl**, produces an output file for each store number showing detail records and total sales:

```

/INFILE=sales
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)
  /CONDITION=(st_01, TEST=(store == "01"))
  /CONDITION=(st_05, TEST=(store == "05"))
  /CONDITION=(st_12, TEST=(store == "12"))
/SORT
  /KEY=dept
/OUTFILE=sales.01
  /HEADREC="      -----\n"
  /INCLUDE WHERE st_01
  /FIELD=(t01, POSITION=6, SIZE=8, NUMERIC)
  /SUM t01 FROM amount
/OUTFILE=sales.01
  /INCLUDE WHERE st_01
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)
/OUTFILE=sales.05
  /HEADREC="      -----\n"
  /INCLUDE WHERE st_05
  /FIELD=(t01, POSITION=6, SIZE=8, NUMERIC)
  /SUM t01 FROM amount
/OUTFILE=sales.05
  /INCLUDE WHERE st_05
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)
/OUTFILE=sales.12
  /HEADREC="      -----\n"
  /INCLUDE WHERE st_12
  /FIELD=(t01, POSITION=6, SIZE=8, NUMERIC)
  /SUM t01 FROM amount
/OUTFILE=sales.12
  /INCLUDE WHERE st_12
  /FIELD=(store, POSITION=1, SIZE=2)
  /FIELD=(dept, POSITION=4, SIZE=2)
  /FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
  /FIELD=(date, POSITION=15, SIZE=10, JAPANESE_DATE)

```

Notice that the `/INCLUDE` statement is required for both the detail and summary record definitions.

sales.01 contains:

```
01 03 239.39 1999-01-18
01 10 8712.45 1999-01-15
01 25 2103.54 1999-01-13
01 25 98.63 1999-01-17
-----
11154.01
```

sales.05 contains:

```
05 17 3495.56 1999-01-10
05 25 498.56 1999-01-05
-----
3994.12
```

sales.12 contains:

```
12 03 589.32 1999-01-04
12 10 4098.34 1999-01-19
12 17 54.23 1999-01-12
12 25 867.32 1999-01-12
-----
5609.21
```

Example: /OUTSKIP and /OUTCOLLECT - Mixed Record Types

In this example, two phone lists are combined in alphabetical order and then split for three people to do the calling. The phone number is to be listed first, followed by the name. There is a total of 15 records. The following are the two input files, **list1** and **list2**:

Smith, John	555-1111	Jones, Jennifer	555-3434
Stone, Allan	555-5678	Milton, Sam	555-0022
Ross, Margaret	555-4321	Clark, Patricia	555-3412
Smithson, Mary	555-5847	Aaron, Henry	555-1092
Jones, Thomas	555-0098	Smith, Mary	555-3412
Ellis, Wayne	555-9375	Johnson, Linda	555-6509
Beck, Tim	555-2298		
Wright, Bill	555-1029		
Hartman, Jill	555-2376		

Notice that this script, **outskip_outcollect.scl**, defines a field for the full name, and separate fields for first and last name. The full-name field is used for mapping to the output, while the fields for first and last name are used for sorting:

```

/INFILE=(list1,list2)
/FIELD=(name,POSITION=1,SIZE=15)
/FIELD=(lname,POSITION=1,SEPARATOR=' ')
/FIELD=(fname,POSITION=2,SEPARATOR=' ')
/FIELD=(phone,POSITION=16,SIZE=9)
/SORT
/KEY=lname
/KEY=fname
/OUTFILE=calls.1
/OUTCOLLECT=5
/FIELD=(phone,POSITION=1,SIZE=9)
/FIELD=(name,POSITION=15,SIZE=15)
/OUTFILE=calls.2
/OUTSKIP=5
/OUTCOLLECT=5
/FIELD=(phone,POSITION=1,SIZE=9)
/FIELD=(name,POSITION=15,SIZE=15)
/OUTFILE=calls.3
/OUTSKIP=10
/FIELD=(phone,POSITION=1,SIZE=9)
/FIELD=(name,POSITION=15,SIZE=15)

```

calls.1 contains:

555-1092	Aaron, Henry
555-2298	Beck, Tim
555-3412	Clark, Patricia
555-9375	Ellis, Wayne
555-2376	Hartman, Jill

calls.2 contains:

555-6509	Johnson, Linda
555-3434	Jones, Jennifer
555-0098	Jones, Thomas
555-0022	Milton, Sam
555-4321	Ross, Margaret

calls.3 contains:

555-1111	Smith, John
555-3412	Smith, Mary
555-5847	Smithson, Mary
555-5678	Stone, Allan
555-1029	Wright, Bill

Example: Piping from a Sort to an Inner Join; Sort with Cross-Calcs

The following is a table of order requests, ordered by customer. Each product item in the **requests** table needs to be matched to a price in the **prices** table. Line-item totals then need to be calculated. The final output is to be in order by customer, followed by item, and contains only those customers whose names begin with the letters A-L.

The first data file is **requests**:

Acme Tool and Die	Boards	10
Acme Tool and Die	Needles	6
Acme Tool and Die	Pins	12
Acme Tool and Die	Snaps	3
American Pipes	Needles	12
American Pipes	Pins	5
American Pipes	Shoes	2
Brazzini Auto Works	Boards	3
Brazzini Auto Works	Chairs	2
Brazzini Auto Works	Tables	5
United Boats	Needles	4
United Boats	Snaps	7
United Boats	Thimbles	2

The second data file is **prices**:

Boards	9.95
Bushing	15.75
Caps	4.90
Chairs	22.60
Cushions	5.75
Guards	6.55
Harnesses	24.90
Machines	26.50
Needles	12.50
Pins	11.25
Shoes	8.50
Snaps	5.55
Sofas	47.50
Tables	12.45
Thimbles	8.00
Watches	23.50
Widgets	8.00
Zippers	7.55

The script **requests.scl** is used to put **requests** in order by item:

```
# requests.scl
/INFILE=requests      # name the input file
  /FIELD=(name,POSITION=1,SIZE=20)
  /FIELD=(item,POSITION=23,SIZE=10)
  /FIELD=(qty,POSITION=33,SIZE=4,NUMERIC)
/SORT
  /KEY=item            # order by product requested
/OUTFILE=stdout        # name the output file
```

This will produce:

Acme Tool and Die	Boards	10
Brazzini Auto Works	Boards	3
Brazzini Auto Works	Chairs	2
Acme Tool and Die	Needles	6
American Pipes	Needles	12
United Boats	Needles	4
Acme Tool and Die	Pins	12
American Pipes	Pins	5
American Pipes	Shoes	2
Acme Tool and Die	Snaps	3
United Boats	Snaps	7
Brazzini Auto Works	Tables	5
United Boats	Thimbles	2

The above output is piped to **get_prices.scl**, where customer names are selected and the **/JOIN** matches prices with line items. Output is sent to **stdout** for further processing:

```
# get_prices.scl
/INFILE=stdin          # left join file
  /FIELD=(name,POSITION=1,SIZE=20)
  /FIELD=(item,POSITION=23,SIZE=10)
  /FIELD=(qty,POSITION=33,SIZE=4,NUMERIC)
  /INCLUDE WHERE name LT "M"
/INFILE=prices         # right join file
  /FIELD=(item,POSITION=1,SIZE=10)
  /FIELD=(price,POSITION=12,SIZE=6,NUMERIC)
/JOIN stdin prices WHERE stdin.item EQ prices.item
/OUTFILE=stdout
  /FIELD=(stdin.name,POSITION=1,SIZE=20)
  /FIELD=(stdin.item,POSITION=23,SIZE=10)
  /FIELD=(stdin.qty,POSITION=33,SIZE=4,NUMERIC)
  /FIELD=(prices.price,POSITION=41,SIZE=6,NUMERIC)
```

The resulting output is:

Acme Tool and Die	Boards	10	9.95
Brazzini Auto Works	Boards	3	9.95
Brazzini Auto Works	Chairs	2	22.60
Acme Tool and Die	Needles	6	12.50
American Pipes	Needles	12	12.50
Acme Tool and Die	Pins	12	11.25
American Pipes	Pins	5	11.25
American Pipes	Shoes	2	8.50
Acme Tool and Die	Snaps	3	5.55
Brazzini Auto Works	Tables	5	12.45

The above data are piped into **customer_ar.scl**, where line-item totals are calculated, and the records are put into customer and item order.

```
/INFILE=stdin
/FIELD=(name, POSITION=1, SIZE=20)
/FIELD=(item, POSITION=23, SIZE=10)
/FIELD=(qty, POSITION=33, SIZE=4, PRECISION=0, NUMERIC)
/FIELD=(price, POSITION=41, SIZE=6, PRECISION=2, NUMERIC)
/SORT
/KEY=name
/KEY=item
/OUTFILE=customers.inv
/HEADREC="CUSTOMER          PRODUCT      QTY  PRICE      TOTAL\n"
/FIELD=(name, POSITION=1, SIZE=20)
/FIELD=(item, POSITION=23, SIZE=10)
/FIELD=(qty, POSITION=33, SIZE=4, PRECISION=0, NUMERIC)
/FIELD=(price, POSITION=41, SIZE=6, PRECISION=2, NUMERIC)
/FIELD=(line_price=qty * price, POSITION=50, SIZE=8, PRECISION=2, NUMERIC)
```

The final output in file **customers.inv** is:

CUSTOMER	PRODUCT	QTY	PRICE	TOTAL
Acme Tool and Die	Boards	10	9.95	99.50
Acme Tool and Die	Needles	6	12.50	75.00
Acme Tool and Die	Pins	12	11.25	135.00
Acme Tool and Die	Snaps	3	5.55	16.65
American Pipes	Needles	12	12.50	150.00
American Pipes	Pins	5	11.25	56.25
American Pipes	Shoes	2	8.50	17.00
Brazzini Auto Works	Boards	3	9.95	29.85
Brazzini Auto Works	Chairs	2	22.60	45.20
Brazzini Auto Works	Tables	5	12.45	62.25

The entire operation is performed with a single command:

```
sortcl /specification=requests.scl|sortcl /spec=get_prices.scl| \
sortcl /specification=customer_ar.scl
```

Example: Multi-Table Join

In this three-table join example, a publishing warehouse is conducting a telephone promotion scheme, whereby phone calls are made to customers to pitch magazine subscriptions that best suit their buying habits. There are three input files:

customers contains:

11010	Smith, John	555-1111
12212	Stone, Allan	555-5678
12214	Ross, Margaret	555-4321
12332	Smithson, Mary	555-5847
12345	Jones, Thomas	555-0098
13322	Ellis, Wayne	555-9375
13332	Beck, Tim	555-2298
13333	Wright, Bill	555-1029
13342	Hartman, Jill	555-2376

purchases contains:

Nails	Hardware	10102
Skis	Sporting	12212
Shoes	Footwear	12345
Football	Sporting	13322
Whiskey	Alcohol	13332
Boots	Footwear	13342
Hammer	Hardware	13456
Wine	Alcohol	23373

The account number (11010, 12212, etc) is required to match customers' contact information with the specific purchases they have made. Note that both **customers** and **purchases** are already sorted by account number.

mags contains:

Sporting	Athletes Digest
Hardware	Home Fixers
Alcohol	Booze Weekly
Footwear	On your toes

The **mags** file is a look up table that shares the category field (Alcohol, Footwear, etc.) with the **purchases** file so that specific magazine titles can be pitched based on customer purchases. Note that the category field is not pre-sorted.

The following script, **multijoin.scl**, joins the **customers** and **magazines** files over the account number field, the results of which are joined with the **mags** file over the category field. Note that an INNER join is followed by a RIGHT_OUTER join:

```
/INFILE=customers      # join table 1
/  FIELD= (Acct_nb, POSITION=1, SIZE=5)
/  FIELD= (Name, POSITION=9, SIZE=18)
/  FIELD= (Phone, POSITION=27, SIZE=8)
/INFILE=purchases      # join table 2
/  FIELD= (Item, POSITION=1, SIZE=10)
/  FIELD= (Category, POSITION=11, SIZE=11)
/  FIELD= (Acct_nb, POSITION=22, SIZE=5)
/INFILE=mags           # join table 3
/  FIELD= (Category, POSITION=1, SIZE=11)
/  FIELD= (Magazine, POSITION=13, SIZE=17)
/JOIN INNER customers purchases WHERE \
customers.Acct_nb == purchases.Acct_nb \
RIGHT_OUTER not_sorted mags WHERE purchases.Category == mags.Category
/OUTFILE=cold_call
/HEADREC="Name           Phone           Item           Magazine\n"
/  FIELD= (Name, POSITION=1, SIZE=18)
/  FIELD= (Phone, POSITION=19, SIZE=8)
/  FIELD= (Item, POSITION=30, SIZE=10)
/  FIELD= (Magazine, POSITION=41, SIZE=17)
```

Internally, **sortcl** first produces an interim file -- the results of an inner join where **customers** with **purchases** are matched over the account number field. These interim results are *not* seen on output:

12212	Stone, Allan	555-5678	Skis	Sporting	12212
12345	Jones, Thomas	555-0098	Shoes	Footwear	12345
13322	Ellis, Wayne	555-9375	Football	Sporting	13322
13332	Beck, Tim	555-2298	Whiskey	Alcohol	13332
13342	Hartman, Jill	555-2376	Boots	Footwear	13342

The final output file that is produced, **cold_call**, joins the above interim results with the **mags** file over the category key. This file can now be used for the calling campaign:

Name	Phone	Item	Magazine
Beck, Tim	555-2298	Whiskey	Booze Weekly
Hartman, Jill	555-2376	Boots	On your toes
Jones, Thomas	555-0098	Shoes	On your toes
			Home Fixers
Ellis, Wayne	555-9375	Football	Athletes Digest
Stone, Allan	555-5678	Skis	Athletes Digest

Note that the **mags** input file was sorted over the category key before joining by way of the NOT_SORTED option. **customers** and **purchases** did not require an additional sort step option because they were pre-sorted on their common account number key.

Example: Joining a Fourth Table

Using the data and **sortcl** job script from Multi-Table Join *on page 309*, this example introduces a fourth join in script to further customize the result set.

Consider the following input file, **incomes**, which identifies the household salary level of the customers involved (note that it is not sorted fully):

11010	Smith, John	75,000-100,000
12212	Stone, Allan	25,000-50000
12214	Ross, Margaret	75,000-100,000
12332	Smithson, Mary	50,000-75,000
12345	Jones, Thomas	<25,000
13322	Ellis, Wayne	>100,000
13332	Beck, Tim	50,000-75,000
13333	Wright, Bill	25,000-50,000
13342	Hartman, Jill	<25,000

Amend the script from Multi-Table Join *on page 309* to include the above input source and to add it to the join:

```
/INFILE=customers      # join table 1
  /FIELD=(Acct_nb,POSITION=1,SIZE=5)
  /FIELD=(Name,POSITION=9,SIZE=18)
  /FIELD=(Phone,POSITION=27,SIZE=8)
/INFILE=purchases      # join table 2
  /FIELD=(Item,POSITION=1,SIZE=10)
  /FIELD=(Category,POSITION=11,SIZE=11)
  /FIELD=(Acct_nb,POSITION=22,SIZE=5)
/INFILE=mags           # join table 3
  /FIELD=(Category,POSITION=1,SIZE=11)
  /FIELD=(Magazine,POSITION=13,SIZE=17)
/INFILE=incomes        # join table 4
  /FIELD=(Acct_nb,POSITION=1,SIZE=5)
  /FIELD=(Name,POSITION=9,SIZE=18)
  /FIELD=(income_level,POSITION=27)
/JOIN INNER customers purchases WHERE \
customers.Acct_nb == purchases.Acct_nb \
RIGHT OUTER not_sorted mags WHERE \
purchases.Category == mags.Category \
INNER NOT_SORTED incomes WHERE customers.Name == incomes.Name
/OUTFILE=cold_call
/HEADREC="Name           Phone           Income           Item
Magazine\n\n"
  /FIELD=(Name,POSITION=1,SIZE=18)
  /FIELD=(Phone,POSITION=19,SIZE=8)
  /FIELD=(income_level,POSITION=30,SIZE=20)
  /FIELD=(Item,POSITION=50,SIZE=10)
  /FIELD=(Magazine,POSITION=60,SIZE=17)
```

The final result set, **cold_call**, is as follows:

Name	Phone	Income	Item	Magazine
Beck, Tim	555-2298	50,000-75,000	Whiskey	Booze Weekly
Ellis, Wayne	555-9375	>100,000	Football	Athletes Digest
Hartman, Jill	555-2376	<25,000	Boots	On your toes
Jones, Thomas	555-0098	<25,000	Shoes	On your toes
Stone, Allan	555-5678	25,000-50000	Skis	Athletes Digest

As shown in the above example, the fourth table, **incomes**, was sorted internally (using the NOT_SORTED option), and matched with the names from the **customers** file.

Example: Using ALTSEQ with Includes

This example demonstrates how an alternate collating sequence can be applied to a field for use in an /INCLUDE condition. See Alternate Collating Sequence: /ALTSEQ on page 163 for complete details on ALTSEQ.

sales.dat contains:

A book	6.98
C blouse	23.45
1 tablet	2.45
B yarn	10.78
3 skirt	78.98
C coat	235.97
2 thread	4.25
A pen,blk	2.98

The script, **altseq_include.scl**, produces three output files, where the /INCLUDE is performed on the field whose values were converted using ALTSEQ value substitutions:


```

/INFILE=sales.dat
  /ALTSEQ=(4131,4232,4333)  # replace A with 1, B with 2, C with 3
  /FIELD=(dept,POSITION=1,SIZE=1)
  /FIELD=(alt_dept,POSITION=1,SIZE=1)
  /FIELD=(item,POSITION=3,SIZE=10)
  /FIELD=(amount,POSITION=14,SIZE=8,NUMERIC)
/INREC
  /FIELD=(dept,POSITION=1,SIZE=1)           # the original field
  /FIELD=(alt_dept,POSITION=2,SIZE=1,ALTSEQ) # the converted field
  /FIELD=(item,POSITION=3,SIZE=10)
  /FIELD=(amount,POSITION=14,SIZE=8,NUMERIC)
/SORT
  /KEY=(dept,ALTSEQ)           # sort over the converted field
  /KEY=item
/OUTFILE=dept1
  /INCLUDE WHERE alt_dept == "1"  # evaluate the converted field
  /FIELD=(alt_dept,POSITION=1,SIZE=1) # display the converted field
  /FIELD=(item,POSITION=3,SIZE=10)
  /FIELD=(amount,POSITION=14,SIZE=8,NUMERIC)
/OUTFILE=dept2
  /INCLUDE WHERE alt_dept == "2"  # evaluate the converted field
  /FIELD=(dept,POSITION=1,SIZE=1)  # display the original field
  /FIELD=(item,POSITION=3,SIZE=10)
  /FIELD=(amount,POSITION=14,SIZE=8,NUMERIC)
/OUTFILE=dept3
  /INCLUDE WHERE alt_dept == "3"  # evaluate the converted field
  /FIELD=(dept,POSITION=1,SIZE=1)  # display the original field
  /FIELD=(item,POSITION=3,SIZE=10)
  /FIELD=(amount,POSITION=14,SIZE=8,NUMERIC)

```

This produces three output files.

dept1 contains:

1 book	6.98
1 pen,blk	2.98
1 tablet	2.45

dept2 contains:

2 thread	4.25
B yarn	10.78

dept3 contains:

C blouse	23.45
C coat	235.97
3 skirt	78.98

Note that all output files reflect the alternate sorting order specified by /ALTSEQ, regardless of whether the original or converted field values are displayed.

Example: Creating an HTML-formatted output file

This example produces an HTML file showing, as a table, the summary of sales by geographic region.

sales2.dat contains:

```
USA, Texas, 13432444
EUROPE, France, 834422
AFRICA, Congo, 153453
EUROPE, Spain, 434422
AFRICA, Egypt, 23453
USA, Florida, 31332444
USA, Georgia, 93444
EUROPE, Germany, 234422
```

The script, **by_region.scl**, is used to create a summary report, using HTML syntax within /DATA, /HEADREC, and /FOOTREC statements so that output is readable by a web browser:

```
# by_region.scl
/INFILE=sales2.dat
  /FIELD= (Region, SEPARATOR=', ', POSITION=1)
  /FIELD= (State, SEPARATOR=', ', POSITION=2)
  /FIELD= (Sales, SEPARATOR=', ', POSITION=3, NUMERIC)
/SORT
  /KEY=Region
/OUTFILE=sales_report.htm # details lines
  /HEADREC="<HTML><HEAD>\n<TITLE>HTML produced by SORTCL\
    </TITLE>\n</HEAD>\n<BODY><H2>Summary of Sales by\
    Region</H2>\nSales under \$100,000 are shown in \
    italics.\n<TABLE CELLSPACING=4 CELLPADDING=1 \
    BORDER COLS=5>\n"
  /DATA="<TR>\n<TD><B>"
  /FIELD= (State)
  /DATA="</B></TD>\n<TD align=right>"
  /DATA=(IF Sales LT 100000 THEN "<em>")
  /FIELD= (Sales, SIZE=15, CURRENCY)
  /DATA=(IF Sales LT 100000 THEN "</FONT>")
  /DATA="</TD>\n</TR>\n"
/OUTFILE=sales_report.htm # summary lines
  /DATA="<TR>\n<TD><B><FONT SIZE=+2>"
  /FIELD= (Region)
  /DATA="</FONT><B></TD>\n<TD align=right><B><U><FONT SIZE=+2>"
  /FIELD= (Sum_Sales, SIZE=15, CURRENCY)
  /DATA="</FONT></U></B></TD>\n</TR>\n"
  /SUM Sum_Sales FROM Sales BREAK Region
  /FOOTREC="</TABLE><BR>\nCreated on </B>%s.\
    <HR></BODY>\n</HTML>", AMERICAN_DATE
```

This produces **sales_report.htm**, which you can open with a web browser

Summary of Sales by Region

Sales under \$100,000 are shown in italics.

Congo	\$153,453.00
Egypt	<i>\$23,453.00</i>
Germany	\$234,422.00
AFRICA	<u>\$176,906.00</u>
Spain	\$434,422.00
France	\$834,422.00
Texas	\$13,432,444.00
EUROPE	<u>\$1,503,266.00</u>
Georgia	<i>\$93,444.00</i>
Florida	\$31,332,444.00
USA	<u>\$44,858,332.00</u>

You can use any HTML syntax, such as commands to modify text and background color, to enhance a web-ready report. You can also include commands specific to other markup languages, such as XML and SGML.



The mark-up language syntax you use is dependent on the version of the browser, or other utility, you use to open and read the output file(s).

Example: Calling sortcl from a Batch Script, Using /EXECUTE

This example uses a looping batch processing script (written in Bourne shell) to invoke a **sortcl** job script that validates records in a series of input files. It also demonstrates the use of /EXECUTE to launch command line statements from within a **sortcl** job script (see /EXECUTE *on page 277*). The use of environment variables is also demonstrated (see Environment Variables *on page 42*), so be sure to export all environment variables used by **sortcl**, and rename any temporary ones (\$1, for example) so that they can be exported. Finally, this example uses *iscompare* functions to verify field data (see Function Compares in Conditions (iscompares) *on page 182*).



This example cannot be performed on Windows.

In the batch script on the next page, **exec.sh**, note that most information and error messages are written to a file defined by the environment variable \$JOBLOG. The expression 2>> \$JOBLOG indicates that any **stderr** output from the current statement will be appended to the file defined by \$JOBLOG. If there is no 2, then output from **stdout** will append to \$JOBLOG.

```

FILEDIR=$1 # Set the directory for the job files from the
            # temporary environment variable $1
LENGTH=$2  # Set the record length from the temporary
            # environment variable $2
COSORT_TUNER="${FILEDIR}/exec.rc"
JOBLOG=${FILEDIR}/exec.log
export JOBLOG LENGTH COSORT_TUNER # Export environment
                                # variables used by sortcl

FNUM=0 # initialize file count
echo "**** Start of exec Batch Process ****\n" > $JOBLOG
echo "**** Start of exec Batch Process ****\n"
echo "Verify records contain printable characters" >> $JOBLOG
echo "and that the field acct contains only digits," >> $JOBLOG
echo "the field amount contains numeric data," >> $JOBLOG
echo "and the field type contains only an A, B, or C." >> $JOBLOG
#
../sortcl /rc 2>> $JOBLOG # Write tuner settings being used
                        # Note that they are redirected
                        # from stderr

# start processing the input files
for IN in $FILEDIR/*.in # Define each input file as the job loops
do
    FNUM=`expr $FNUM + 1` # Calculate the number of
                        # the input file being processed
    export FNUM           # Export for use in sortcl job script
    OUT=${FILEDIR}/exec.out # Define the output file
    export IN OUT # Export the environment variables used for the
                # input and output files in the sortcl job script
    # Below invoke sortcl to run the job
    ../sortcl /specification=exec.scl 2>> $JOBLOG
    exstat=$? # save the exit status to a variable
    date >> $JOBLOG
    printf "End of file $FNUM process\n" >> $JOBLOG
    printf "Exit status $exstat\n" >> $JOBLOG
done
printf "\nEnd of Batch Processing\n" >> $JOBLOG
printf " **** End of Batch Processing ****\n"

```

The batch script is invoked as follows:

```
sh exec.sh execfiles 0
```

execfiles is the name of the directory defined by **FILEDIR**, and 0 is the length (**LENGTH**) used for the input records. The 0 indicates that the records are linefeed-terminated and can be variable-length.

The **sortcl** script on the following page, **exec.scl**, is the job invoked by the batch script. Note the conditions that are used to validate fields in the records. All valid records are appended to the output file defined by **OUT**. Invalid records are written to error files whose names are based on the input source. Counts are done of valid records, invalid records, and total records processed, and then appended to **JOBLOG**.

```
/EXECUTE="echo '\nSortcl began processing file $FNUM of \
      batch process\n' `date` >> $JOBLOG"
/EXECUTE="echo 'Output file is ${OUT}' >> $JOBLOG"
/EXECUTE="printf 'Number of input records and input file \
      are: \n' >> $JOBLOG"
/EXECUTE="wc -l $IN >> $JOBLOG" # do a line count of current \
      input file
/INFILE=$IN
  /LENGTH=$LENGTH
  /FIELD=(store,POSITION=1,SIZE=2)
  /FIELD=(acct,POSITION=4,SIZE=5)
  /FIELD=(type,POSITION=10,SIZE=1)
  /FIELD=(amount,POSITION=12,SIZE=6,NUM)
  /FIELD=(dept,POSITION=19,SIZE=3)
  /FIELD=(wholerec,POSITION=1,SIZE=22) # define a field for the
      # entire record
  /CONDITION=(printable,TEST=(ISPRINT(wholerec)))
  /CONDITION=(amt_true,TEST=(ISNUMERIC(amount)))
  /CONDITION=(acct_true,TEST=(ISDIGIT(acct)))
  /CONDITION=(type_true,TEST=(type EQ "A" OR type EQ "B" \
      OR type EQ "C"))
  /CONDITION=(validrec,TEST=(printable AND amt_true \
      AND acct_true AND type_true))
/REPORT
/OUTFILE=$OUT
  /APPEND
  /INCLUDE where validrec
/OUTFILE=${IN}.err
  /OMIT where validrec
# count the valid records and send the result to the log file
/OUTFILE=$JOBLOG
  /APPEND
  /INCLUDE where validrec
  /DATA="Number of valid records: "
  /FIELD=(validct,SIZE=2)
  /COUNT=validct
# count the invalid records and send the result to the log file
/OUTFILE=$JOBLOG
  /APPEND
  /OMIT where validrec
  /DATA="Number of invalid records: "
  /FIELD=(invalidct,SIZE=2)
  /COUNT=invalidct
# count all the processed records and send the result
# to the log file
/OUTFILE=$JOBLOG
  /APPEND
  /DATA="Total number of records processed: "
  /FIELD=(recordct,SIZE=2)
  /COUNT=recordct
```

There are four input files in the directory defined by the batch script.

exec_store01.in contains:

```
01 21045 D 123.45 B24
01 34062 B 121.50 A02
01 45963 A 12.20 ^B24
01 45974 A 312.20 B24
```

As shown above:

- record 1 is invalid because of the D in the field type
- record 2 is invalid because of the letter l and not number 1 in the field amount
- record 3 is invalid because of the non-printable character ^B
- record 4 is valid.

exec_store02.in contains:

```
02 34O62 B 12.50 A02
02 45963 A 252.20 B24
04 30486 A 15.35 A14
```

As shown above:

- record 1 is invalid because of the field acct contains a O and not an 0
- record 2 is valid
- record 3 is invalid because it contains a hidden character (hex 00) in column 19 of the record.

The following shows the column number, the hex representation, and the text representation of record 3. Note that hex 0a is the linefeed at the end of the record:

```

1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 22 23 24 < column
30 34 20 33 30 34 38 36 20 41 20 20 31 35 2e 33 35 20 00 41 31 34 0a < hex
0  4    3  0  4  8  6    A    1  5  .  3  5    A  1  4    < ASCII
```

exec_store03.in is an empty input file that produces zeroes for the counts. Note that this is the case only when the following **cosortrc** parameter is set:

ON_EMPTY_INPUT PROCESS_WITH_ZEROS

exec_store04.in contains:

```
04 21045 A 123.45 B24
04 34062 B 12.50 A02
04 45963 A 252.20 B24
04 59405 C 0.00 F11
04 30486 A 15.35 A14
```

All records are valid.

The files produced by this batch process are as follows:

exec_store01.in.err contains:

```
01 21045 D 123.45 B24
01 34062 B 121.50 A02
01 45963 A 12.20 B24
```

exec_store02.in.err contains:

```
02 34062 B 12.50 A02
04 30486 A 15.35 A14
```

The files **exec_store03.in.err** and **exec_store04.in.err** are empty.

The following is the beginning section of the JOBLLOG file that is created, **exec.log**, up to and including the processing of the first input file:

```

**** Start of exec Batch Process ****

Verify records contain printable characters
and that the field acct contains only digits,
the field amount contains numeric data, and the
field type contains only an A, B, or C.

CoSort Version 9.5.1 R91090204-1205  © 1978-2011 IRI, Inc.
www.cosort.com
EST 11:07:32 AM Friday, February 26 2011.  #13245.0905 6 CPUs sun4u
<00:00:00.00> event (112): Errors 0
Filename: execfiles /exec.rc
    MEMORY__MAX                50%
    THREAD__MIN                 0
    THREAD__MAX                 2
    BLOCKSIZE                   1228800
    MONITOR__LEVEL              0
    MIN__YEAR                   86
    ON_EMPTY__INPUT             PROCESS_WITH_ZEROS
    LOG                         /export/home/csuser/etc/cosort.log
    WORK__AREA                  1
    -- /export/data1/csuser/ : 0
Filename: /export/home/csuser/etc/cosortrc
Filename: /usr/local/cosort/etc/cosortrc
        cosortrc report
Environment Information:
    User: csuser
    COSORT_HOME: /export/home/csuser
Process Limits from ULIMIT:
    Memory: unlimited
    Open Files: 256
User: csuser
COSORT_HOME: /export/home/csuser

Sortcl began processing file 1 of batch process
Tue Feb 26 11:07:32 EST 2011
Output file is exexfiles/exec.out
Number of input records and input file are:
    4 execfiles/exec_store01.in
Number of valid records: 1
Number of invalid records: 3
Total number of records processed: 4
Tue Feb 26 11:07:32 EST 2011
End of file 1 process
Exit status 0

```

Example: Pivoting Columns to Rows, with Summaries

The input file below, **transactions.dat**, contains three-column rows: a store code, department code, and transaction amount for that department. The object of this transformation is to create one record for each store containing a column for each store,

```
11, 2, 41.45
11, 3, 100.00
11, 3, 54.99
11, 1, 12.78
54, 3, 201.59
11, 2, 124.78
11, 3, 69.67
41, 2, 150.39
41, 1, 2.95
41, 2, 146.98
54, 3, 53.65
41, 3, 24.78
41, 3, 39.99
```

In the script below, **pivot.scl**:

- The INPUT section defines the layouts of the source file, transactions.dat
- The INREC section reformats the input records so that the transaction amount is in the correct column for the department, and puts a zero where there is no transaction amount for a given department
- The SORT order on StoreCode allows us to group the transaction amounts by store code
- The OUTPUT section summarizes the transaction amounts in each department, grouped by store code, and defines the target layout.

```

/INFILE=transactions.dat
/FIELD=(StoreCode,POSITION=1,SEPARATOR=',')
/FIELD=(DeptCode,POSITION=2,SEPARATOR=',')
/FIELD=(TransAmount,POSITION=3,SEPARATOR=',',NUMERIC)
/INREC
# change the record so that the TransAmount is in the correct column
# for the DeptCode. Place zeros in the other columns
/FIELD=(StoreCode,POSITION=1,SEPARATOR=',')
/FIELD=(DeptCode1,POSITION=2,SEPARATOR=',',NUMERIC,IF DeptCode == "1"
THEN TransAmount ELSE 0)
/FIELD=(DeptCode2,POSITION=3,SEPARATOR=',',NUMERIC,IF DeptCode == "2"
THEN TransAmount ELSE 0)
/FIELD=(DeptCode3,POSITION=4,SEPARATOR=',',NUMERIC,IF DeptCode == "3"
THEN TransAmount ELSE 0)
/SORT
/KEY=StoreCode
/OUTFILE=StoreSums.out
# Collapse the columns using SUM statements so there is a unique StoreCode
# record with a Transaction Amount for each DeptCode
/FIELD=(StoreCode,POSITION=1,SEPARATOR=',')
/FIELD=(DC1_sum,POSITION=2,SEPARATOR=',',NUMERIC)
/FIELD=(DC2_sum,POSITION=3,SEPARATOR=',',NUMERIC)
/FIELD=(DC3_sum,POSITION=4,SEPARATOR=',',NUMERIC)
/SUM DC1_sum FROM DeptCode1 BREAK StoreCode
/SUM DC2_sum FROM DeptCode2 BREAK StoreCode
/SUM DC3_sum FROM DeptCode3 BREAK StoreCode

```

The target, **StoreSums.out**, contains:

```

11,12.78,166.23,224.66
41,2.95,297.37,64.77
54,0.00,0.00,255.24

```

The original input format was flattened to rows based on the StoreCode field, which was also used as the 'unary change' break key when summing. For those familiar with the Microsoft SSIS platform, DeptCode is the Pivot Key (top row value), StoreCode is the Set Key (left column value), and TransAmount is the Pivot Value (table values).

Example: Updating Data, separate current and history records

In this example, the current record and the historical records are kept in the same file, the Master file. The key or identifier for a record is the field ProductCode. The file tracks cost changes for the ProductCodes. The master file has the following fields:

- ProductCode: This is the identifier field
- Vendor: This has the vendor that supplies the product
- Cost: cost of the product
- StartDate: date that the cost in this record became effective
- EndDate: date that the cost in this record was no longer effective. If the cost is still in effect, then the year for the EndDate is set to 99991231
- Current: Y if the cost is still in effect, N if it is not

All dates are in the form YYYYMMDD

The Update file contains product codes that need to have the cost updated. The fields in the Update file are:

- ProductCode: This is the identifier field
- Cost: new cost for the product

The product codes in the Master and Update files are matched using a left outer join so that all the records in the Master file are kept, plus new records are created using the matches to the Update file.

The input files are:

product2.dat

```
C123, ACME, 125.50, 20110228, 99991231, Y
F112, LANE, 2365.00, 20120101, 99991231, Y
G101, MONROE, 21.25, 20110930, 20120515, N
G101, MONROE, 19.25, 20110515, 99991231, Y
J245, HONORA, 395.00, 20100430, 20110430, N
J245, HONORA, 425.00, 20110430, 99991231, Y
S022, SAS, 98.75, 20110515, 99991231, Y
```

change2.dat

```
J245, 450.00
S022, 101.75
```

The script is **scd2.scl**:

```

/INFILE=product2.dat
  /ALIAS=Master
  /FIELD=(ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(Vendor, POSITION=2, SEPARATOR=',')
  /FIELD=(Cost, POSITION=3, SEPARATOR=',',NUMERIC)
  /FIELD=(StartDate, POSITION=4, SEPARATOR=',')
  /FIELD=(EndDate, POSITION=5, SEPARATOR=',')
  /FIELD=(Current, POSITION=6, SEPARATOR=',')
/INFILE=change2.dat
  /ALIAS=Update
  /FIELD=(ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(Cost, POSITION=2, SEPARATOR=',',NUMERIC)
/JOIN LEFT_OUTER Master Update WHERE Master.ProductCode == Update.ProductCode
/OUTFILE=newmaster2.dat
  # Change any current records that are being updated to history records
  # by giving the EndDate as today's date and changing the field Current to N
  /CONDITION=(Match, TEST=(Master.ProductCode == Update.ProductCode AND Master.Current == "Y"))
  /FIELD=(Master.ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(Master.Vendor, POSITION=2, SEPARATOR=',')
  /FIELD=(Master.Cost, POSITION=3, SEPARATOR=',',NUMERIC)
  /FIELD=(Master.StartDate, POSITION=4, SEPARATOR=',')
  /FIELD=(NewEndDate, POSITION=5, SEPARATOR=',', IF Match THEN "20120621" ELSE Master.EndDate)
  /FIELD=(NewCurrent, POSITION=6, SEPARATOR=',', IF Match THEN "N" ELSE Master.Current)
/OUTFILE=newmaster2.dat
  # Include only records that are being updated
  # Create the new current Master record by changing the startDate to
  # today's date
  # Use the Cost from the Update file
  /INCLUDE WHERE Master.ProductCode == Update.ProductCode AND Master.Current == "Y"
  /FIELD=(Master.ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(Master.Vendor, POSITION=2, SEPARATOR=',')
  /FIELD=(Update.Cost, POSITION=3, SEPARATOR=',',NUMERIC)
  /FIELD=(NewStartDate="20120621", POSITION=4, SEPARATOR=',')
  /FIELD=(Master.EndDate, POSITION=5, SEPARATOR=',')
  /FIELD=(Master.Current, POSITION=6, SEPARATOR=',')

```

Which produces **newmaster2.dat**:

```

C123,ACME,125.50,20110228,99991231,Y
F112,LANE,2365.00,20120101,99991231,Y
G101,MONROE,21.25,20110930,20120515,N
G101,MONROE,19.25,20110515,99991231,Y
J245,HONORA,395.00,201004301,20110430,N
J245,HONORA,425.00,20110430,20120621,N
J245,HONORA,450.00,20120621,99991231,Y
S022,SAS,98.75,20110515,20120621,N
S022,SAS,101.75,20120621,99991231,Y

```

Example: Updating Data, current and historical values in a single record

This example includes a master file and an update file. Each record in the master file also contains history. The fields in the master file are:

- ProductCode: this is the identifier field
- Vendor: This has the vendor that supplies the product
- Cost1: Current cost for the product
- EffectiveDate1: date that Cost1 in this record became effective
- Cost2: Previous cost for the product
- EffectiveDate2: date that Cost2 in this record became effective
- Cost3: Effective cost prior to Cost2
- EffectiveDate3: date that Cost3 in this record became effective

The fields in the update file are:

- ProductCode: this is the identifier field
- Cost: new effective cost for the product
- EffectiveDate: new effective date for the cost

All dates are in the form YYYYMMDD

The product codes in the Master and Update files are matched using a left outer join. The first output file updates records where the cost has a change. The second output file outputs the records that are not changed. The use of 2 output files with the same name is for processing different types of records.

The input files are:

product3.dat

```
C123,ACME,125.50,20110228,100.75,20111014,,  
F112,LANE,2365.00,20120101,2295.00,20111014,2155.50,20110104  
G101,MONROE,19.25,20110515,,,,  
J245,HONORA,425.00,20110430,,,,  
S022,SAS,98.75,20110515,95.40,20110305,,
```

change3.dat

```
F112,2425.00,20120701  
J245,450.00,20120701  
S022,101.75,20120701
```

The script is **scd3.scl**:

```
/INFILE=product3.dat
  /ALIAS=master
  /FIELD=(ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(Vendor, POSITION=2, SEPARATOR=',')
  /FIELD=(Cost1, POSITION=3, SEPARATOR=',')
  /FIELD=(EffectiveDate1, POSITION=4, SEPARATOR=',')
  /FIELD=(Cost2, POSITION=5, SEPARATOR=',')
  /FIELD=(EffectiveDate2, POSITION=6, SEPARATOR=',')
  /FIELD=(Cost3, POSITION=7, SEPARATOR=',')
  /FIELD=(EffectiveDate3, POSITION=8, SEPARATOR=',')
/INFILE=change3.dat
  /ALIAS=update
  /FIELD=(ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(Cost, POSITION=2, SEPARATOR=',')
  /FIELD=(EffectiveDate, POSITION=3, SEPARATOR=',')
/JOIN LEFT_OUTER master update WHERE master.ProductCode EQ update.ProductCode
/OUTFILE=newmaster3.dat
  # Include those records where the ProductCode matches for
  # the master and update files
  # Put the cost and effective date for the update record where
  # Cost1 and EffectiveDate1 are normally.
  # Move Cost1 and EffectiveDate1 to the place for Cost2 and EffectiveDate
  # Move Cost2 and EffectiveDate2 to the place for Cost3 and EffectiveDate3
  /FIELD=(master.ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(master.Vendor, POSITION=2, SEPARATOR=',')
  /FIELD=(update.Cost, POSITION=3, SEPARATOR=',')
  /FIELD=(update.EffectiveDate, POSITION=4, SEPARATOR=',')
  /FIELD=(master.Cost1, POSITION=5, SEPARATOR=',')
  /FIELD=(master.EffectiveDate1, POSITION=6, SEPARATOR=',')
  /FIELD=(master.Cost2, POSITION=7, SEPARATOR=',')
  /FIELD=(master.EffectiveDate2, POSITION=8, SEPARATOR=',')
/OUTFILE=newmaster3.dat
  # Include those records where the ProductCode does not match for
  # the master and update files
  /INCLUDE WHERE master.ProductCode NE update.ProductCode
  /FIELD=(master.ProductCode, POSITION=1, SEPARATOR=',')
  /FIELD=(master.Vendor, POSITION=2, SEPARATOR=',')
  /FIELD=(master.Cost1, POSITION=3, SEPARATOR=',')
  /FIELD=(master.EffectiveDate1, POSITION=4, SEPARATOR=',')
  /FIELD=(master.Cost2, POSITION=5, SEPARATOR=',')
  /FIELD=(master.EffectiveDate2, POSITION=6, SEPARATOR=',')
  /FIELD=(master.Cost3, POSITION=7, SEPARATOR=',')
  /FIELD=(master.EffectiveDate3, POSITION=8, SEPARATOR=',')
```

Which produces **newmaster3.dat**:

```
C123,ACME,, ,125.50,20110228,100.75,20111014
C123,ACME,125.50,20110228,100.75,20111014,,
F112,LANE,2425.00,20120701,2365.00,20120101,2295.00,20111014
G101,MONROE,, ,19.25,20110515,,
G101,MONROE,19.25,20110515,,,
J245,HONORA,450.00,20120701,425.00,20110430,,
S022,SAS,101.75,20120701,98.75,20110515,95.40,20110305
```

31 SORTCL STATEMENTS

This section contains a list of all SORTCL statements, grouped into usage-related areas. The usage and ordering of the statements appear as they would in a typical **sortcl** script. The order of statements within the script is important.



Some of these statements would never occur together in the same script, and many run by default, so they are optional.

The *n* shown for some of the parameter values represents a whole number.

Job Controls Section

```
/MONITOR=n
/MEMORY-WORK=" [path] filename"
/LOCALE= [language[_territory[_codeset]]]
/JOBCOLLECT=n
/ROUNDING=type
/EXECUTE="command statement"
/SPECIFICATION= [path] filename
/STATISTICS [= [path] filename]
/RC
/WARNINGSON
/WARNINGSOFF
```

Input Section

```
/INFILE= [path] filename
/PROCESS=file_format
/CHARSET=encoding
/ENDIAN=parameter
/ALIAS=name
/LENGTH=n
/HEADREAD=n
/TAILREAD=n
/HEADSKIP=n
/TAILSKIP=n
/FIELD=(field_name, attributes)
/CONDITION=(condition_name, TEST=(logical expression))
/INSKIP=n
/INCOLLECT=n
/INCLUDE=condition_name or
/INCLUDE WHERE condition
```



```

/INCLUDE=condition_name or
/INCLUDE WHERE condition
/INFILES=( [path] filename1, [path] filename2, ...)
... same as for /INFILE
/INPROCEDURE [= procedure]
... same as for /INFILE
/INREC
  /LENGTH=n
  /FIELD=(field name/expression/function, attributes)
  /FIELD=(field name, IF condition THEN value ELSE value, attributes)
  /CONDITION=(condition_name, TEST=(logical expression))
  /INCLUDE=/INCLUDE WHERE condition OR (CONDITION=condition_name)
  /OMIT=/OMIT WHERE condition OR (CONDITION=condition_name)

```

Action Section

```

/REPORT
/SORT   OR   /MERGE   OR   /CHECK
  /KEY=(field [, ASCENDING OR DESCENDING] [, LEFT OR RIGHT OR NONE]
    [, CASE_ON OR CASE_OFF] [, data type] [, ALTSEQ])
/JOIN [join_type [ONLY]] [not_sorted] source1 [not_sorted] source2
[WHERE condition]
[join_type [ONLY]] [not_sorted] source3 [WHERE condition]
[join_type [ONLY]] [not_sorted] source4 [WHERE condition]
  /KEYPROCEDURE
  /STABLE
  /NODUPPLICATES
  /DUPLICATESONLY
  /JOBSKIP=n
  /JOBCOLLECT=n

```

Output Section

```

/OUTFILE= [path] filename
  /APPEND   OR   /CREATE
  /PROCESS=file_format
  /CHARSET=encoding
  /ENDIAN=parameter
  /LENGTH=n
  /RECSPPERPAGE=n
  /HEADWRITE=input filename where /HEADREAD occurred
  /TAILWRITE=input filename where /TAILREAD occurred
  /HEADREC="string with variable formats if variables used", variable1, ...
  /FOOTREC="... same as /HEADREC", ...
  /CONDITION=(condition_name, TEST=(logical expression))

```

```
/INCLUDE=condition_name or
/INCLUDE WHERE condition
/INCLUDE=condition_name or
/INCLUDE WHERE condition
/OUTSKIP=n
/OUTCOLLECT=n
/FIELD=(field name/expression/function, attributes)
/FIELD=(field name, IF condition THEN value ELSE value, attributes)
/DATA=[{n}] "literal string"
/DATA=(IF condition_name THEN value1 [ELSE value2]
/COUNT=count_field [RUNNING] [WHERE condition_name] [BREAK
        condition]
/SUM=sum_field [FROM field] [RUNNING] [WHERE condition]
        [BREAK condition]
/AVERAGE
        ... same as for /SUM
/MINIMUM
        ... same as for /SUM
/MAXIMUM
        ... same as for /SUM
/OUTPROCEDURE
        /FIELD=(field name [, POSITION=n, SIZE=n,
        SEPARATOR='char', data type])
```

Symbols

!< 171
 != 171
 !> 171
 .n 492
 /ALIAS 89
 /ALTSEQ 163
 /APPEND 66, 268
 /AUDIT 653
 /AVERAGE 235, 236
 /CHARSET 78
 /CHECK 159
 example 288
 /CONDITION 170, 294
 /COUNT 241
 /CREATE 66, 268
 /DATA 155
 defined 152
 /DUPLICATESONLY 168
 /ENDIAN 79
 /ERRORCOUNT 280
 /EXECUTE 277
 /FIELD 92
 /FIELD_PREDICATE 143
 /FOOTREAD 270
 defined 263
 /FOOTREC 155
 defined 272
 /HEADREAD 269
 defined 261
 /HEADREC 155, 250
 defined 270
 /HEADSKIP
 defined 260
 /HEADWRITE 261
 defined 269
 /INCLUDE 170, 226, 293
 defined 175
 /INCOLLECT 265
 /INFILE 43, 48, 439
 /INFILES 48
 /INREC
 defined 150
 example 297
 /INSKIP 264
 /JOBCOLLECT 266
 /JOBSKIP 265
 /JOIN 159, 307
 defined 224
 /KEY 160
 /LENGTH 54, 439
 /LOCALE 281
 /MAXIMUM 239
 /MEMORY-WORK 44
 /MERGE 159
 example 289
 /MINIMUM 239
 /NEWFILE 245
 /NODUPPLICATES
 defined 168
 example 298
 /OMIT 226
 /OUTCOLLECT 275
 example 304
 /OUTFILE 51
 /OUTSKIP 275
 example 304
 /PROCESS 53
 /QUERY 266
 /RC 276
 /RECSPPERPAGE 272
 /REPORT 159
 example 295, 296
 /ROUNDING 281
 /SORT 159
 /SPECIFICATION 45
 /SPECIFICATIONS 428, 435, 447
 /STABILITY 160
 /STABLE 168, 439
 defined 167
 /STATISTICS 81
 /SUM 235
 /TAILREAD
 defined 263
 /TAILSKIP
 defined 261
 /TAILWRITE
 defined 270
 /UPDATE 66, 269
 /VERSION 40
 /WARNINGSOFF 280
 /WARNINGSON 280
 \ 155

\ " 155
\\ 155
\0 155
\a 155
\b 155
\f 155, 273
\n 155, 272
\r 155, 272
\t 155
\v 155
43
%% 271
%c 271
%d 271
%E 271
%e 271
%f 271
%G 271
%g 271
%i 271
%o 271
%s 271
%u 271
%X 271
%x 271
+m.n 490, 491
< 171
<= 171
== 171
> 171
>= 171
>>see append
\$ 42, 155
\$variable 509

Numerics

2string functions 213
 non_alnum2s 213
 non_alpha2s 213
 non_asc_let2s 213
 non_ascii2s 213
 non_digit2s 213
 non_ebc_let2s 213
 non_print2s 213

A

abbreviations 42, 509, 520
abs (x) 149
accepting records 265
accumulating aggregates 251
acos (x) 149
Action 509, 513, 516
action 530
action statements 159
 /CHECK 159
 /JOIN 159
 /MERGE 159
 /REPORT 159
 /SORT 159
ACUCOBOL-GT file format 58
addition 146
aggregation. See summary functions.
alert 155
alias 89
alignment 513, 514, 530
 left 523
 none 523, 530
 right 523
alignment in keys 166
alignment of fields 103
ALPHA 522
alphabetic 576
alternate collating sequence 163
ALTSEQ 163
AMERICAN 523
AMERICAN_DATE 154
AMERICAN_TIME 154
AMERICAN_TIMESTAMP 154
appending to an output file 66, 268
application program 535, 536
arithmetic symbols
 -, subtraction 146
 (), parentheses 146
 *, multiplication 146
 +, addition 146
 unary 146
arithmetic value 488
ASCENDING 165
ascending 489, 519, 576
ASCII 157, 436, 448, 522
ASCII options

- alignment 166
 - case 166
 - in keys 166
- asin (x) 149
- atan (x) 149
- atan2 (x,y) 149
- auditing 82, 653

B

- backslash 155, 520
- backspace 155
- banner 516, 534
- batch 507, 510, 526, 534, 535, 540
- batch script execution 40
- Bessel functions 149
- binary data 517
- binary logical expressions 171
- binary NULL 106
- BIT 125
- bit pattern 517
- blank 488, 489, 521
- BLOCKED file format 68
- BREAK 169, 234
 - with /COUNT 241
 - with /MAXIMUM and /MINIMUM 239
 - with /SUM and /AVERAGE 236, 239
 - with SEQUENCER 255
 - with summary reports 250, 299, 300
- buffer 509, 582
- buffer pointer 582
- building conditions 173
- byte count 526
- byte order mark 79
- byte position 517

C

- C 599
- calling program 542, 574, 583
- carriage return 155
- case fold 488, 513, 514, 523, 530
 - no 523
 - yes 523
- case transfer functions
 - tolower 215
 - toproper 215

- toupper 215
- case-sensitive 41, 166
- cat 486
- ceil (x) 149
- change test 170
- character array 543, 570
- character data 436, 448
- character pointer 573
- check 487
- CLF (NCSA Common Log Format) 77
- COBOL 522, 577
 - Micro Focus 437, 449
 - Ryan McFarland 437, 449
- collating sequence 487, 488, 489
- columns. See fields, POSITION.
- comma 521
- command line 397, 431, 442, 451, 516
- command-line execution 40
- commas 95, 106
- comments 43, 509, 535, 537, 539
- common input record 150
- Common Log Format. See CLF.
- COMP 437, 449
- compares 574, 575
- comparison 522
- compound conditions 173
- compound logical expressions 172
- compressed input files 49
- compressed output files 52
- condition carrying example 294
- conditional /DATA statements 180
 - IF 181
 - IF-THEN-ELSE logic 181
 - THEN 181
- conditional /FIELD statements 180
 - IF 181
 - IF-THEN-ELSE logic 181
 - THEN 181
- conditional output selection example 292
- conditions 169–179
 - syntax 169
 - /CONDITION 170
 - example 294
 - /INCLUDE 170
 - /OMIT 226
 - binary 171
 - compound 172, 173

- explicit (named) 169, 177
- function compares 182
- implicit 169
- unary 170
- WHERE 170, 175
 - with /COUNT 241
 - with /MAXIMUM and /MINIMUM 239
 - with /SUM and /AVERAGE 236
 - with include-omit 175
- continuation character 42
- control characters 155, 517, 520, 536
 - \', single quote 155
 - \", double quote 155
 - \\, backslash 155
 - \0, NULL 155
 - \a, alert 155
 - \b, backspace 155
 - \f, form feed 155
 - \n, newline 155
 - \r, carriage return 155
 - \t, horizontal tab 155
 - \v, vertical tab 155
 - \$, dollar sign 155
- conventions 41
- conversion specifiers 156
 - ASCII 157
 - EBCDIC 156
 - hexadecimal 156
 - within a /DATA statement 158
 - within a condition 157
- conversion. See data-type conversion.
- coroutine 574
- cos (x) 149
- cosh (x) 149
- COSORT_TUNER 44, 496, 497
- cosort.rc 44
- cosort() 507, 519, 573, 574, 580, 582, 583, 584
- cosortrc 44
- count 241
- creating an output file 66, 268
- cross-calculation 143
- CS_BOTH 581
- cs_compare() 575
- CS_FONLY 581
- CS_INIT 572

- cs_input() 518, 535
- CS_OUTPUT 582
- CS_PROGRAM 581
- CS_UONLY 581
- CSV file format 60
- csv2ddf 60
- CT 171
- CURRENCY 106, 623
- CURRENCY, MONEY 106
- CURRENT_DATE 154
- CURRENT_TIME 154
- CURRENT_TIMESTAMP 154
- CURRENT_TIMEZONE 154
- custom functions 223

D

- data definition file 46
 - example 290
- data files 48
- data structure 577
- data type
 - CURRENCY 623
 - IP_ADDRESS 623
 - NUMERIC 623
 - WHOLE_NUMBER 623
- data types
 - BIT 125
 - CURRENCY 106
 - CURRENCY, MONEY 106
 - EBCDIC 162
 - multi-byte
 - multi-byte data types 122
 - WHOLE_NUMBER 124
- data warehouse 121
- database column name
 - EXT_FIELD 68
- database extraction 65
- database loading 66
- data-type conversion 121
- date 576
- decimal point 99, 488
- decimal precision of numeric fields 100
- default 489, 509, 516, 524, 537
- definition calls 570
- definition phase 572, 582, 583
- de-identify 220

delimited-to-fixed example 288
 delimiter 95, 520, 521, 576
 DESCENDING 165
 descending 489, 519, 576
 device 495
 devices 525
 dictionary order 488
 different separators 98
 direction in keys 165
 ASCENDING 165
 DESCENDING 165
 directory 509, 535
 DISP 437, 449
 DISPLAY 510, 511, 531
 dollar sign 155
 double quote 155
 DSN 63
 duplicates only 168

E

EBCDIC 156, 162, 436, 448, 522
 ELF file format 77
 elf2ddf 77
 ELSE 180
 endian 120
 endianness 120
 sorting based on 121
 ending 491
 environment variables 42, 572
 EQ 171
 equal keys 487, 575
 EQUALS 439
 EQUIVALENCE 577
 error 507, 534, 540, 571
 errors 276, 280
 escape characters. *See* control characters.
 ETL 37
 EUROPEAN 523
 EUROPEAN_DATE 154
 EUROPEAN_TIME 154
 EUROPEAN_TIMESTAMP 154
 evaluation of include-omit 175
 evaluation order of expressions 172
 examples 285–328
 /OUTSKIP and /OUTCOLLECT 304
 alternate report with complex selection

296
 carrying a condition forward 294
 conditional output selection 292
 creating an HTML-formatted output file
 314
 delimited-to-fixed format change 288
 joining an additional table 309
 less verbose 286
 multiple input files with different format
 297
 multiple input files with same format 296
 multiple output files and formats 291
 multiple output files with conditional se-
 lection 293
 piping from a sort to an inner join 306
 removing duplicate records 298
 report with complex section 295
 simple script form 285
 summary fields with breaks 299
 summary functions with detail, break,
 and total records 300
 two-key sort 286
 two-key sort with delimited fields 287
 using /CHECK 288
 using /MERGE 289
 using a data definition file 290
 execute 514, 526, 530
 execution 40, 428, 434, 446, 582
 execution calls 570
 execution phase 572, 583
 execution time 526
 exit status 487
 exp (x) 149
 explicit conditions 169, 177
 EXPONENT 109
 expressions
 binary logical 171
 compound logical 172
 unary logical 170
 EXT_FIELD 68
 Extended Log Format. *See* ELF.
 external function 575
 external transformations 223
 External values 620
 external values 517, 620
 extraction 65
 Extraction, Transformation, and Loading.

See ETL.

F

Fast Access Table (FAT) 41

field 436, 448, 512

 position 520

 variable position 520

field endianness 120

field expressions 143

field length function 192

field name references in keys 161

field separator 489

field specifier 490

fields 92–125

 syntax 92

 alignment 103

 FILL 106

 FRAME 101

 MILL 106

 naming 93

 NULL 107

 NULL_HIGH 107

 NULL_LOW 107

 padding 104

 POSITION 93

 PRECISION 100

 reducing 104

 SEPARATOR 95

 SIZE 99

 substrings 99, 100

 trimming 104

file formats 53

 ACUCOBOL-GT 58

 BLOCKED 68

 CSV 60

 ELF 77

 LDIF 61

 LINE_SEQUENTIAL 57

 MFISAM 59

 MFVL_LARGE 56

 MFVL_SMALL 56

 ODBC 62, 375

 RANDOM 73

 RECORD 54

 RECORD_SEQUENTIAL 54

 UNIBF 59

 UNIVBF 60

 V 58

 VARIABLE_SEQUENTIAL 57

 VSAM 58

 XML 68

FILE FORMATS (/PROCESS) on page 53
477

file names 573

files 44–81

 data 48

 data definition 46

 example 290

 input

 example 296, 297

 job specification 45

 output 51, 291, 293

 resource control 44

 specification 45

 statistics 81

FILL 106

fill characters in fields 106

find and replace 208

 2string functions 213

 case transfer functions 215

FIXED 520

fixed length 574, 583

fixed-length records 54

float 436, 448

floating-point precision 147

floor (x) 149

footer record 272

format 513, 530

 514

 alpha 530

format control characters 271

 %%, to write % to output 271

 %c, character 271

 %d, %i, decimal integer 271

 %e, %E, %f, floating point number 271

 %G, E-format or F-format 271

 %g, e-format or f-format 271

 %o, unsigned octal integer 271

 %s, as a string 271

 %u, unsigned decimal integer 271

 %x, %X, unsigned hexadecimal integer

 271

format_strings 196

- form-feed 155
- FORTRAN 542, 577
- FRAME 101
- framed characters in fields 101
- framing 101
- FROM
 - with /MAXIMUM and /MINIMUM 239
- FULL_OUTER join 231
- function compares in conditions 182

- G**

- gamma (x) 149
- GE 171
- globally unique identifiers. See GUID.
- GT 171
- GUID 197
- gzip 49, 52

- H**

- head 528
- header record 260, 261, 269, 270
- help 507, 509, 511, 516
- hexadecimal 156
- horizontal selection 597
- horizontal tab 155
- HTML 152, 271, 272, 314
- HTML-format output file example 314
- hypot (x,y) 149

- I**

- I/O 509
- I/O buffer 497
- IBM V 58
- IF 181
- IF-THEN-ELSE logic 180, 181
- ignore 488
- implicit conditions 169
- IMPLIED_DECIMAL 110
- INCLUDE-OMIT 432
- include-omit 175
 - evaluation 175
 - examples 176
- inequality 519
- initialization 572
- INNER join 229
 - example 306
- input 573, 575, 581
- input file 484, 487, 495, 511, 518, 527, 530
- input file #x 513
- input files 574
 - example 296, 297
- input procedure 518
- input record 571
- input record count 584
- input records 574, 584
- input statements 260
- integer 436, 448
- integer values 563, 579
- integers 620
- internal values 517
- internal variables 154
 - AMERICAN_DATE 154
 - AMERICAN_TIME 154
 - AMERICAN_TIMESTAMP 154
 - CURRENT_DATE 154
 - CURRENT_TIME 154
 - CURRENT_TIMESTAMP 154
 - CURRENT_TIMEZONE 154
 - EUROPEAN_DATE 154
 - EUROPEAN_TIME 154
 - EUROPEAN_TIMESTAMP 154
 - ISO_DATE 154
 - ISO_TIME 154
 - ISO_TIMESTAMP 154
 - JAPANESE_DATE 154
 - JAPANESE_TIME 154
 - JAPANESE_TIMESTAMP 154
 - PAGE_NUMBER 154
 - SYSDATE 154
 - USER 154
 - VALUE_TIME 154
 - VALUE_TIMESTAMP 154
- IP_ADDRESS 623
- iscompares 182
 - isalpha 182
 - isalphadigit 182
 - isascii 182
 - isctrl 182
 - isdigit 182
 - isbcalpha 183

- isebcdigit 183
- isempty 183
- isgraph 182
- isholding 183
- islower 182
- isnumeric 183
- ispacked 183
- ispattern 183
- isprint 182
- ispunct 182
- isspace 183
- isupper 183
- isxdigit 183

ISO 523

ISO_DATE 154

ISO_TIME 154

ISO_TIMESTAMP 154

J

JAPANESE 523

JAPANESE_DATE 154

JAPANESE_TIME 154

JAPANESE_TIMESTAMP 154

job specification file 45

joining additional tables example 309

joins

- syntax 225

- examples

- FULL_OUTER 231

- INNER 229

- LEFT_OUTER 229

- ONLY 232

- RIGHT_OUTER 229

- UNORDERED 233

- FULL_OUTER 231

- INNER 229

- example 306

- LEFT_OUTER 229

- ONLY 232

- RIGHT_OUTER 229

- UNORDERED 233

K

Key 519

key 442, 443, 575

- length 577

- position 577

- structure 576, 577

key comparison 519

key direction 513, 530

- ascending 519, 530

- descending 519

key field 522

key length 514

keys 160

- syntax 161

- ASCII options

- alignment 166

- case 166

- direction 165

- field name reference 161

- no duplicates 168

- stability 167

- unnamed reference 162

keys unique 513, 519

L

language environment 281

LDIF file format 61

LE 171

leading blanks 489

leading space 620

left align 103

LEFT_OUTER join 229

length 54

length of field 192

less verbose example 286

letter case transfer 215

line continuation 42

line length 482

LINE_SEQUENTIAL file format 57

line-feed 482, 489, 512, 517

LINES 273

loading 66

locale 281, 487, 488

locale(M) 485

location 513, 530

log (x) 149

log10 (x) 149

logical expressions 169

- binary 171

- compound 172
- evaluation order 172
- unary 170
- logical field 522
- logical record 439
- lower case 537
- lowercase 41, 166, 488
- LT 171

M

- m. 491
- man page 482
- mapping 150, 305
- mark-up language 152, 314
- matches. See joins.
- mathematical functions
 - abs (x) 149
 - acos (x) 149
 - asin (x) 149
 - atan (x) 149
 - atan2 (x,y) 149
 - Bessel functions 149
 - ceil (x) 149
 - cos (x) 149
 - cosh (x) 149
 - exp (x) 149
 - floor (x) 149
 - gamma (x) 149
 - hypot (x,y) 149
 - log (x) 149
 - log10 (x) 149
 - mod (x,y) 149
 - pow (x,y) 149
 - sin (x) 149
 - sinh (x) 149
 - sqrt (x) 149
 - tan (x) 149
 - tanh (x) 149
- maximum 239
- MaxMemory 509
- memory 481, 497, 583
- memory allocation 497
- merge 487, 497, 525, 527, 528, 530, 571, 584
- MESSAGE 571, 574, 575, 581, 582
- MFISAM file format 59
- MFVL_LARGE file format 56

- MFVL_SMALL file format 56
- MILL 106, 236
- mill option in fields 106
- MIN_DIGITS 109
- minimum 239
- MINUS_CHAR 109
- miscellaneous options 276–??
- mod (x,y) 149
- MONITOR_LEVEL 277
- MONTH_DAY 522
- multi-byte character types 127
- multiple input 485
- multiple input files with different formats example 297
- multiple input files with same formats example 296
- multiple output files and formats example 291
- multiple output files with condition example 293
- multiplication 146
- multi-table join 309
- MVS 433, 436, 445, 448

N

- named conditions 169, 177
- named keys 161
- naming conventions 41
- naming fields 93
- NC 171
- NCSA Common Log Format 77
- NE 171
- newline 155
- no duplicates 168
- NT File System. See NTFS.
- NTFS 41
- NULL 106, 107, 155
- null assignment in fields 107
- NULL_HIGH 107
- NULL_LOW 107
- number of input files 513
- NUMERIC 623
- Numeric 624
- numeric 576
- numeric data 99
- numeric field attributes 109

- EXPONENT 109
- IMPLIED_DECIMAL 110
- MIN_DIGITS 109
- MINUS_CHAR 109
- PLUS_CHAR 109
- SIGN_CONTROL 109
- numeric sort 488, 492

O

- ODBC file format 62, 375
- ODS 37
- OMIT
 - defined 175
- ONLY option in joins 232
- Operational Data Store. See ODS.
- optional statements 42
- optional values 42
- options 484, 487
- output 495, 514, 524, 530, 581
- output file 495, 514, 525, 527, 530, 540, 581, 583
- output files 51, 291, 293
- output records 524, 583
- overflow 496, 497, 509, 527, 540

P

- padding fields 104
- PAGE_NUMBER 154, 272
- pagebreak 273
- parameters 570
- parentheses in arithmetic 146
- Pascal 542
- path 535
- pattern matching 183
- performance 496, 508
- permission 535, 540
- PERMUTE 265
- phone directory 488
- pipe 486, 495
- pipng from a sort to an inner join with selection example 306
- PLUS_CHAR 109
- pointers 543, 570
- PORTA 572, 575, 581, 582
- PORTB 572, 575, 581, 582, 583

- POSITION 93
- position 576
- pow (x,y) 149
- PRECISION 100
- priority of resource control settings 645
- procedure 507
 - compare 519
 - input 518
- processing records 265
- program 524, 534, 535
- pseudonymization 204

R

- RAM 497, 509
- RANDOM file format 73
- ranking 242
- reals 620
- record
 - fixed length 517
- RECORD file format 54
- record filters 259–275
- record length 482, 512, 513, 517, 530
- record size 573
- RECORD_SEQUENTIAL file format 54
- records
 - common input 150
 - fixed-length 54
 - footer 272
 - format control characters 271
 - header 261, 269, 270
 - length 54
 - selection 175
 - variable-length 55
- records per page 272
- reducing fields 104
- registry 44, 277
- re-identify 220
- relational operators
 - CT 171
 - EQ, == 171
 - GE, >=, !=< 171
 - GT, > 171
 - LE, <=, !=> 171
 - list of 171
 - LT, < 171
 - NC 171

- NE, != 171
 - removing duplicate records example 298
 - replace characters
 - replace_chars 196
 - replace string 208
 - replace_chars 196
 - report with complex selection example 295, 296
 - reports with summaries 249
 - resource control file 44
 - resource controls
 - search order 645
 - Responses 509
 - restricted sort key 489, 494
 - right align 103
 - RIGHT_OUTER join 229
 - rows. See records.
 - RUNNING
 - with /COUNT 242
 - with /COUNT. See also SEQUENCER.
 - with /MAXIMUM and /MINIMUM 241
 - with aggregates 234, 251
 - running aggregates 251
 - runtime statistics 81
 - runtime warnings 280
- S**
- script 45
 - search order for resource controls 645
 - search string 208
 - security 535
 - SEPARATOR 95
 - separator character 490, 520
 - separator characters in fields 95
 - SEQUENCER 255
 - SEQUENCER. See also RUNNING with /COUNT.
 - SET files 198
 - SGML 315
 - shell 510
 - SIGN_CONTROL 109
 - simple script form example 285
 - sin (x) 149
 - single quote 155
 - sinh (x) 149
 - SIZE 99
 - size of fields 99
 - substrings 99, 100
 - skip bytes 518
 - skipping records 264
 - SMP 37
 - SORT 516
 - sort 431, 439, 442, 450, 451, 571, 584
 - sort key 490, 491
 - SORTCL 37
 - sortcl tools
 - csv2ddf 60
 - elf2ddf 77
 - specification file 45, 429, 431, 439, 442, 450, 451
 - specifications 507, 510, 526, 530, 536, 571
 - specifications file 510, 514, 526, 530, 535, 536, 537, 540
 - sqrt (x) 149
 - stability 167
 - start position 490, 513, 521, 530
 - start-up 508, 532
 - statements 328–330
 - /ALIAS 89
 - /ALTSEQ 163
 - /APPEND 66, 268
 - /AUDIT 653
 - /AVERAGE 235, 236
 - /CHARSET 78
 - /CHECK 159
 - example 288
 - /CONDITION 170, 294
 - /COUNT 241
 - /CREATE 66, 268
 - /DATA 155
 - defined 152
 - /DUPLICATESONLY 168
 - /ENDIAN 79
 - /ERRORCOUNT 280
 - /EXECUTE 277
 - /FIELD 92
 - /FIELD_PREDICATE 143
 - /FOOTREAD 270
 - defined 263
 - /FOOTREC 155
 - defined 272
 - /HEADREAD 269
 - defined 261

- /HEADREC 155, 250
 - defined 270
- /HEADSKIP
 - defined 260
- /HEADWRITE 261
 - defined 269
- /INCLUDE 170, 226, 293
 - defined 175
- /INCOLLECT 265
- /INFILE 43, 48
- /INFILES 48
- /INREC
 - defined 150
 - example 297
- /INSKIP 264
- /JOB COLLECT 266
- /JOB SKIP 265
- /JOIN 159, 307
 - defined 224
- /KEY 160
- /LENGTH 54
- /LOCALE 281
- /MAXIMUM 239
- /MERGE 159
 - example 289
- /MINIMUM 239
- /NEWFILE 245
- /NODUPPLICATES
 - defined 168
 - example 298
- /OMIT 226
 - defined 175
- /OUTCOLLECT 275
 - example 304
- /OUTFILE 51
- /OUTSKIP 275
 - example 304
- /PROCESS 53
- /QUERY 266
- /RC 276
- /RECS PER PAGE 272
- /REPORT 159
 - example 295, 296
- /ROUNDING 281
- /SORT 159
- /SPECIFICATION 45
- /STABILITY 160
- /STABLE 168
 - defined 167
- /STATISTICS 81
- /SUM 235
- /TAIL READ
 - defined 263
- /TAIL SKIP
 - defined 261
- /TAIL WRITE
 - defined 270
- /UPDATE 66, 269
- /VERSION 40
- /WARNINGSOFF 280
- /WARNINGSON 280
- statistical analysis
 - ranking 242
- statistics file 81
- stderr 277, 534, 540
- stdin 41, 484, 486, 535
- stdout 41, 280, 485, 495, 524, 525
- stop 509, 511, 516, 531
- structured data types 577
- substrings 99, 100
- subtraction 146
- sum 432, 442, 443
- summary fields with breaks example 299
- summary functions 234–253
 - /AVERAGE 235, 236
 - /COUNT 241
 - /MAXIMUM 239
 - /MINIMUM 239
 - /SUM 235
 - BREAK 169, 234
 - with /COUNT 241
 - with /MAXIMUM and /MINIMUM 239
 - with /SUM and /AVERAGE 236, 239
 - with SEQUENCER 255
 - with summary reports 250, 299, 300
 - example 300
 - FROM 239
 - RUNNING 234, 241, 242, 251
- summary output 430, 441
- summary records 432
- summary reports 249
- SWITCH 571, 572, 573, 574, 575, 580, 582, 583

Symmetric MultiProcessing. See SMP.
 syntax 481, 484
 SYSDATE 154

T

table look ups 198
 tables 309
 tail 511, 528
 tail record 261
 tan (x) 149
 tanh (x) 149
 temporary file 496, 527
 terminal 507, 508, 524
 text processor 509, 534
 THEN 181
 thrashing 481
 time 576
 timestamp 576
 tmpdir 496
 tolower 215
 toproper 215
 toupper 215
 trailing space 620
 trimming fields 104
 tuning 44
 two-key sort example 286
 two-key with delimited fields example 287

U

unary logical expressions 170
 unary operators 146
 unconditional join 233
 undefined variable 509
 UNIBF file format 59
 unique 487, 519, 530
 UNIVBF file format 60
 universally unique identifiers. See UUID.
 unnamed references in keys 162
 UNORDERED join 233
 updating an output file 66, 269
 upper case 537
 uppercase 41, 166
 USER 154, 272
 UUID 197

V

V file format 58
 VALUE_TIME 154
 VALUE_TIMESTAMP 154
 variable length 481, 517, 537, 583
 VARIABLE_SEQUENTIAL file format 57
 variable-length records 55
 verbose
 example 286
 vertical selection 597
 vertical tab 155
 Vision (ACUCOBOL-GT) 58
 VSAM file format 58

W

warnings 276, 280, 535
 web-ready report. See HTML.
 WHERE 170, 175
 with /COUNT 241
 with /MAXIMUM and /MINIMUM 239
 with /SUM and /AVERAGE 236
 with include-omit 175
 white space 523, 537
 who 486
 WHOLE_NUMBER 124, 623
 Windows registry 44, 277
 WorkAreas 509
 workspace 496, 540

X

XDEF 68
 XML 315
 XML file format 68

Z

-zrecsz 497

CUSTOM TRANSFORMS

This chapter describes how to write libraries used for custom field-level transformations in **sortcl**. It also provides details on the field transformation routines currently provided with **CoSort**:

ASCII/ALPHANUM ENCRYPTION AND DECRYPTION *on page 358*
Provided with **CoSort**.

MELISSA DATA CLEANSING ROUTINE *on page 383*
Requires Melissa Data's Address Object product to be installed on your system.

TRILLIUM CLEANSING ROUTINE *on page 387*
Requires the Trillium product to be installed on your system.



To ensure that the correct libraries are loaded, copy the appropriate library files from **/lib** in the home directory, to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

1 WRITING LIBRARIES FOR CUSTOM DATA PROCESSING

As described in Custom Field-Level Functions (External Transformations) *on page 223*, you can invoke custom field-level functions that can be loaded at runtime via a dynamic linked library on Windows (**.dll**), or a shared object or shared library on Unix or Linux (**.so** or **.sl**). This allows you to modify field values in ways that are not natively supported by **CoSort**. And because the custom function is at the field level, once initialized, it is called for every field -- for every record -- that exists in the input stream.

sortcl is thread-safe, so custom procedures can be called from multiple threads, depending on your **THREAD_MAX** value (see **THREAD_MAX** count *on page 647*). Custom field-level functions must be written so that they are thread-safe.

1.1 Syntax

To ensure that the correct libraries are loaded, copy the appropriate library files from **/lib** in the home directory, to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

For details on calling a custom function from a **sortcl** job script, Custom Field-Level Functions (External Transformations) *on page 223*.

The following sections describe the implementation process, in the order in which it is performed:

- 1) Custom Procedure Declaration *on page 347*
- 2) Custom Procedure Arguments *on page 348*
- 3) Sequence of Custom Procedure Calls *on page 352*

1.2 Custom Procedure Declaration

This section describes the structures and the interface for writing custom procedures.

The procedure declaration for a custom routine is as follows:

```
int procedure_name( cs_ext_source_t *source, cs_ext_result_t *result );
```

For Windows users, the function needs to be exported either in a **.def** file, or by using `__declspec (dllexport)`.

The first argument is a pointer to the structure `cs_ext_source_t`, which contains the source arguments. The source arguments are passed into the custom procedure using the **sortcl** script (see Custom Field-Level Functions (External Transformations) *on page 223*). These are then internally set up in the `cs_ext_source_t` structure to be passed to the custom procedure.

The second argument is a pointer to the structure `cs_ext_result_t`, which holds the result of the field-level transformation. This needs to be set up by the custom procedure after performing the transformation, and then passed back to **sortcl**.

The `cs_ext_result_t` structure also makes provisions for sending back error codes and error messages.

The return value is an integer. The return codes are defined in the header file **sortcl_routine.h**.

The two structures `cs_ext_source_t` and `cs_ext_result_t`, and their elements, are explained in further detail below, and their declarations can be found in the header file **sortcl_routine.h**

1.3 Custom Procedure Arguments

This section describes the following:

- *Source Arguments*
- *Result Argument on page 350*
- *Return Value on page 351*

1.3.1 Source Arguments

The source arguments are of the type `cs_ext_source_t`:

Table 1: Source Arguments Structure

```
typedef struct cs_ext_source_s {  
    void*          pPrivate;  
    cs_ext_mode     iMode;  
    int             iNumArgs;  
    cs_ext_arg_t*   SrcArgs;  
} cs_ext_source_t;
```

`pPrivate`

`pPrivate` is a void pointer that can be used by the custom procedure to store its own structures or data, so that the values can be retained within function calls.

`iMode`

`iMode` can have the following values, as also indicated in the header file :

Table 2: List of Possible Modes

```
typedef enum {  
    CS_EXT_BEGIN, /* a signal to initialize, before any data is sent */  
    CS_EXT_VALUE, /* used on every call to return a field or value */  
    CS_EXT_BREAK, /* used on break conditions, not implemented yet */  
    CS_EXT_END    /* used to signal the end of processing, clean up */  
} cs_ext_mode;
```

`CS_EXT_BEGIN` indicates an initialization call to the custom procedure.

`CS_EXT_VALUE` indicates that **sortcl** is now passing in the function arguments to the custom procedure. If any of the arguments are field names, arithmetic operations or other function calls, they are resolved before this, so that the literal values can be passed in.

CS_EXT_END is a finish call to the custom procedure, indicating that the records have now been processed.

CS_EXT_BREAK is reserved for future use.

The iMode values must not be modified by the custom procedure.

iNumArgs

iNumArgs is the total number of arguments passed to the custom procedure, for example:

```
/FIELD= (encodeHex(field1,":"))
```

In this case, the custom procedure encodeHex is being called with two arguments, and therefore iNumArgs will be set to 2 (when the iMode is CS_EXT_VALUE) to indicate the number to arguments passed.

SrcArgs

SrcArgs is an array that holds the source arguments and their properties. These are set up and passed into the custom procedure from **sortcl**, for every record, when iMode is CS_EXT_VALUE.

The structure containing the source arguments has the following elements:

Table 3: External Arguments Structure

```
/* This contains the definitions associated with the data that is
passed into the external function. */
typedef struct cs_ext_arg_s {
    int          iValueForm;
    int          iValueType;
    int          iValueLen;
    cs_ext_value_t* arg;
} cs_ext_arg_t;
```

The first two elements, iValueForm and iValueType, point to the form and the data type of the argument being passed in (see **cosort.h** for a list of all possible values).

The element iValueLen holds the length of the argument being passed in.

The final element, cs_ext_value_t* arg, holds the literal value. Depending on the data type, **sortcl** populates the correct element of the cs_ext_value_t union:

Table 4: External Arguments Value Structure

```
typedef union cs_ext_value_s {
    char          *szValue;
    cs_longdouble_t ldValue;
    double        dValue;
    int            iValue;
    long           lValue;
    short          shValue;
} cs_ext_value_t;
```

1.3.2 Result Argument

The result arguments are of the type `cs_ext_result_t`:

Table 5: Result Argument Structure

```
typedef struct cs_ext_result_s {
    int            iErrorNum;
    char          *szErrorMsg;
    cs_ext_arg_t  *ResArg;
} cs_ext_result_t;
```

ResArg

The `ResArg` element, also of the type `cs_ext_arg_t`, stores the result value that needs to be passed back to **sortcl** in the event of successful processing.

The structure containing the result argument has the following elements:

Table 6: External Arguments Structure

```
/* This contains the definitions associated with the data that is
   passed into the external function. */
typedef struct cs_ext_arg_s {
    int            iValueForm;
    int            iValueType;
    int            iValueLen;
    cs_ext_value_t* arg;
} cs_ext_arg_t;
```

The first two elements, `iValueForm` and `iValueType`, point to the form and the data type of the argument being passed back to **sortcl** (see **cosort.h** for a list of all possible values).

The element `iValueLen` holds the length of the argument being passed back.

The final element, `cs_ext_value_t* arg`, holds the actual result value. Depending on the return data type, the custom procedure needs to insert the value into the correct element of the `cs_ext_value_t` union, and then return it to **sortcl**:

Table 7: External Arguments Value Structure

```
typedef union cs_ext_value_s {
    char          *szValue;
    cs_longdouble_t ldValue;
    double        dValue;
    int           iValue;
    long          lValue;
    short         shValue;
} cs_ext_value_t;
```

Error Handling

The first two elements of the result structure, `iErrorNum` and `szErrorMsg`, can be used to pass back an error code and an error message string to be displayed by **sortcl**. `iErrorNum` can be any integer, and the element `szErrorMsg` must point to a null-terminated string.

1.3.3 Return Value

The following table lists the return values expected by **sortcl**:

Table 8: Return Values

```
#define CE_EXT_OK          0
#define CE_EXT_BADTYPE    1
#define CE_EXT_BADVALUE   2
#define CE_EXT_BADNUM     3
#define CE_EXT_ABORT      -1
```

The return value `CE_EXT_OK` indicates success.

`CE_EXT_BADTYPE` indicates that the data type of an argument being passed was incorrect. In this case, a warning is generated, and any further calls for the remaining records to the custom procedure are bypassed.

`CE_EXT_BADVALUE` and `CE_EXT_BADNUM` indicate an incorrect string or numeric value, and both generate warning messages for that particular record. No further calls to the custom procedure are bypassed.

On receiving a return value of `CE_EXT_ABORT` from the custom procedure, **sortcl** will cleanup its resources and exit the program.

1.4 Sequence of Custom Procedure Calls

sortcl makes three primary sets of calls to the custom routine in the following order:

- 1) *Initialization Call*
- 2) *Processing field level transformations for each record*
- 3) *Termination on page 353*

1.4.1 Initialization Call

At first, the custom function is called with `iMode` set to `CS_EXT_BEGIN` (see *Table 2* on page 348 for a list of all the modes). If you are executing with `THREADMAX > 1` (see `THREAD_MAX` count on page 647), then this call is made once for each thread.

At this point, the custom procedure can set up its own resources and initialization parameters. It can then set the `pPrivate` pointer to point to any allocated resources, so that it is passed in, from this point on, for all the records that will be processed. The `pPrivate` pointer is for use with the custom procedure only and is not altered by **sortcl**.

1.4.2 Processing field level transformations for each record

Once the initialization is successful, **sortcl** will set up the source arguments structure for each record, and call the custom procedure with `iMode` set to `CS_EXT_VALUE`.

At this point, **sortcl** has populated the source arguments structure (*Table 1* on page 348) with all the necessary values.

For example, assume that the source argument variable is named `cs_source`.

`cs_source->pPrivate` is the pointer set up by the custom procedure during the initialization call.

The element `cs_source->iNumArgs` contains the total number of arguments that are being passed in.

These arguments are stored in the array `cs_source->SrcArgs`, and can be accessed using the array index. For example, the first argument passed can be accessed using `cs_source->SrcArgs[0]`, and so on.

Each source argument has its form, data type, length and actual value passed in. For example, the length of the first argument can be accessed using `cs_source->SrcArgs[0].iValueLen`.

The actual argument values are stored in the argument structure (see *Table 7* on page 351) for that particular source element. For example, if the first source argument is a string, its value can be obtained from `cs_source->SrcArgs[0].arg->szValue`.

Once the custom procedure makes sure it has the correct arguments, it can finish the processing and set up the result argument (see *Table 8* on page 351) to pass back the transformed values to **sortcl**.

For example, assume here that the result argument is named `cs_result`.

The final result argument value is stored in `cs_result->ResArg`. The custom procedure needs to set up the form, data type, and length in order for the result to be passed back to **sortcl**.

The actual computed value is stored in `cs_result->ResArg->arg`. For example, if the custom procedure is passing back a string result to **sortcl**, the final value should be stored in `cs_result->ResArg->arg->szValue`.

1.4.3 Termination

After all the records have been sent for processing to the custom procedure, **sortcl** makes a final call to the custom procedure with `iMode` set to `CS_EXT_END` (see *Table 2* on page 348 for a list of all the modes).

At this point, the custom procedure should free any allocated resources. If you are executing with `THREADMAX > 1` (see `THREAD_MAX` count on page 647), then this call is made once for each thread.

This procedure is illustrated in Example on page 354.

1.4.4 Example

This example is divided into two parts. The first is the **sortcl** script containing the call to the custom procedure. The second is the C program that explains how the custom procedure is implemented.

Sortcl Script Using a Custom Field Level Function

```
/INFILE=data.in
  /FIELD=(raw_data, POSITION=1, SIZE=10)
/REPORT
/OUTFILE=data.out
  /FIELD=(encoded_data=encodeHex(raw_data, ":"), POSITION=1, SIZE=30)
```

In the output file, the derived field `encoded_data` is generated by calling the function `encodeHex` with two arguments: the first is the input field `raw_data`, and the second is the literal string `:"`. To use a library file, move or copy the file from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If there are files in **/modules** that you are not using, it is recommended that you move the files back to **/lib**.

The following code is an implementation of the custom routine `encodeHex` in C, which can be compiled into **ext_func.dll** on Windows and **ext_func.so (.sl)** on Unix.

Note the function:

```
_LIBSPEC int encodeHex( cs_ext_source_t *source, cs_ext_result_t *result )
```

in order to understand how arguments are resolved and used to perform the transformation, where results sent back to the **sortcl** program.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* ext_func.c - example plugin for field level transformations
*
* Copyright (c) 2008, Innovative Routines International, Inc.
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* Defines an external function that encodes a possibly
* binary field into ASCII hex, replacing every byte with two
* characters 0-f.
*
* Also demonstrates using optional parameters in an external
* function. If called with a character string as the second
* parameter, the first character of the string will be used
* as a separator.
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /
/* Define macros to allow exported functions under both
   Windows or Unix
*/
#ifdef (_WIN32)
#define _LIBSPEC __declspec (dllexport)
#include <windows.h>
#else
#define _LIBSPEC
#endif /* _WIN32 */

#include <memory.h>

#include "cosort.h"
#include "sortcl_routine.h"
/*
* Handle field lengths up to 256 bytes.
* Encoding replaces every byte with two characters
* (or three with optional separator)
* so we need up to three times the space for transformed fields.
*/
#define MAX_FIELD_IN 256
#define MAX_FIELD_OUT (MAX_FIELD_IN * 3)

static const char chex[] = "0123456789abcdef";

/*
* This helper function is called locally and need not be
* exported. Returns the number of characters in the encoded
* string.
*/
int hex_encode( unsigned char *inb, char *outb, const int len, const char
sep )
{

```

continued on next page...

```
int n, iI, iO;

iO = 0;
iI = 0;

while (iI < len) {
    n = ((inb[iI] >> 4) & 0x0f);
    outb[iO++] = chex[n];
    n = (inb[iI++] & 0x0f);
    outb[iO++] = chex[n];
    /* do not put separator at end of string */
    if (('\'0' != sep) && (iI < len)) {
        outb[iO++] = sep;
    }
}
return (iO);
}

/*
 * This is the exported function that will be referenced in
 * the sortcl FIELD statement. It receives a field value in
 * the source structure and returns the hex encoded field in
 * the result structure.
 */
_LIBSPEC int encodeHex( cs_ext_source_t *source,
                        cs_ext_result_t *result )
{
    int n;
    int iRet = CE_EXT_OK;
    char *p;
    char c = '\0';

    if (CS_EXT_BEGIN == source->iMode) {
        /* first call to external function, allocate
         memory for field out */
        p = (char *)malloc( MAX_FIELD_OUT );
        if (NULL == p) {
            iRet = CE_EXT_ABORT;
        }
        else {
            source->pPrivate = p;
        }
    }
    else if (CS_EXT_END == source->iMode) {
        /* last call to function, clean up */
        free( source->pPrivate );
        source->pPrivate = NULL;
    }
}
```

continued on next page...

```
else {
    p = source->pPrivate;
    if (NULL != p) {
        if (1 < source->iNumArgs) {
            c = source->SrcArgs[1].arg->szValue[0];
        }
        n = hex_encode( (unsigned char *)
                        source->SrcArgs[0].arg->szValue,
                        p,
                        source->SrcArgs[0].iValueLen,
                        (const)c );

        result->ResArg[0].iValueLen = n;
        result->ResArg[0].iValueType = CS_ASCII;
        result->ResArg[0].arg->szValue = p;
    }
    else {
        /* we have lost our private data pointer... abort! */
        iRet = CE_EXT_ABORT;
    }
}
return (iRet);
}
```

2 ASCII/ALPHANUM ENCRYPTION AND DECRYPTION

This section describes the various encryption, decryption, and hashing routines supplied with **CoSort** to protect one or more ASCII or ALPHANUM fields or columns in your input source.

Advanced Encryption Standard (AES) *on page 362*

Original field widths and field formatting are not preserved.

GPG Encryption and Decryption *on page 366*

Compatible with GPG key ring management. Original field widths and original field formatting are not preserved.

Format-Preserving Encryption and Decryption *on page 369*

Original field widths are preserved, but not any original formatting.

Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256) *on page 372*

Original field sizes and any original formatting are preserved.

Triple Data Encryption Standard (3DES) *on page 378*

All the original field widths and field formatting are not preserved.

Encoding and Decoding *on page 380*

Converts each byte into its base64 or hexadecimal equivalent.

SHA-2 Hashing *on page 381*

All the field widths and field formatting are not preserved.

All of the above encryption methods, except Format Preserving Encryption, produce printable ASCII characters (base64) as ciphertext.

All routines except the GPG routines are provided in the library file **libscrypt.so** or **libscrypt.dll** (Windows). The GPG routines are provided in the file **libsgpg.so** (UNIX/Linux) and **libsgpg.dll** (Windows). The encode and decode hex functions are provided in **libsutil.so** (UNIX/Linux) and **libsutil.dll** (Windows). The default location for all **sortcl** library files is the **\$COSORT_HOME/lib** directory (UNIX/Linux) or **install_dir\lib** (Windows).

To use a library file, move or copy the file from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If there are files in **/modules** that you are not using, it is recommended that you move the files back to **/lib**.

The **CoSort** GPG library files have a dependency file, **cl32.so** (UNIX/Linux) or **cl32.dll** (Windows), which is provided with the product **cryptlib** (see GPG Encryption and Decryption *on page 366* for usage details).



You can use one or more external encryption routines instead of, or in addition to, the supplied routines. For complete details on writing and invoking your own custom field-level functions, see WRITING LIBRARIES FOR CUSTOM DATA PROCESSING *on page 346*.

You can use multiple routines and multiple passphrases in the same **sortcl** job script to vary protection among different input fields in order to improve security and target multiple recipients.

2.1 Usage: ASCII Encryption and Decryption

Encryption and decryption using the supplied **CoSort** routines are supported only on fields declared as ASCII in the **sortcl** job script.



To preserve a field's *original* data type for output display purposes (if other than ASCII), you can over-define the same field of data in the input section. This requires that you supply two distinct `/FIELD` statements for the same field of data in the `/INFILE` section of a job script -- for example, define one field named `idnum_orig` and declare its original data type, then define another field named `idnum_to_encrypt` (with the same `SIZE` and `POSITION`), where `idnum_to_encrypt` is declared as ASCII so it can be encrypted.

In the output section for display purposes, specify the field `idnum_orig` with the desired output size and position, and declare its original data type (see Data Types (Single-Byte) *on page 121*).

Routines are invoked within a `/FIELD` statement either within the `/INREC` or `/OUTFILE` section of a **sortcl** job script.

2.2 Syntax: ASCII/ALPHANUM Encryption

The syntax for invoking an ASCII/ALPHANUM encryption routine is:

```
encryption_routine_name(field_name,passphrase_option[,public_key_path]
optional escape character )
```

where `encryption_routine_name` can be any of:

- `enc_3des_ebc`
- `enc_3des_ssl`
- `enc_aes128`
- `enc_aes128_ssl`
- `enc_aes256`
- `enc_aes256_ssl`
- `enc_gpg`
- `enc_fp_aes256_alphanum`
- `enc_fp_aes256_alphanum_ssl`
- `enc_fp_aes256_ascii`
- `enc_fp_aes256_ascii_ssl`
- `hash_sha1`
- `hash_sha2`
- `hash_sha2_ssl`

Where `field_name` is the ASCII or ALPHANUM source field to be encrypted.

The `passphrase_option` can be any of the following:

"*literal_string*" A user-supplied passphrase. Alternatively, you can specify this using `"pass:literal_string"`.

"file:[path/]filename" For additional protection. A reference to a (restricted-access) file whose first line contains the passphrase. Any subsequent lines in that file are ignored. If the first line of that file is longer than 512 bytes, only the first 512 bytes are used. (Any trailing carriage returns or linefeeds are stripped from the passphrase line.)



Environment variable substitution is supported for the literal string and `file:` options described above (see Environment Variables *on page 42*). However, you should use this option with caution because certain platforms allow you to view the environment of other processes (for example, when running certain UNIX operating systems).

- No Argument** Applies only to `enc_aes256()`. If you do not supply a *passphrase_option*, a secret, internal passphrase is used. This passphrase is embedded inside the plug-in binary library, but is scrambled before use.
- "Key ID"** Applies only to `enc_gpg()`. Supply the key ID that corresponds to the key inside the public key ring.

public_kr_path is a filename (with optional path) that contains the GPG public key ring (see *Example 99* on page 367). Used only with `enc_gpg()`.



UNIX / Linux users can also apply the `setuid` file permission option to the **sortcl** executable to help keep the passphrase secret, including from someone who has rights to execute the **sortcl** script. This enables users to obtain access to files for which they do not have read permission, but certain fields will be encrypted.

The `file:` option described above is not required to accomplish this. It can be done using a literal string, but the `file:` methods provide more overall flexibility. Moreover, if the **sortcl** job script itself is not protected, users can write their own scripts to decrypt the data.

2.3 Syntax: ASCII Decryption

After you have encrypted the values in one or more fields, the syntax for invoking the decryption routine in a subsequent job is:

```
encryption_routine_name(field_name,passphrase_option[,secret_kr_path])
```

where `decryption_routine_name` can be any of:

- `dec_3des_ebc`
- `dec_3des_ssl`
- `dec_aes128`
- `dec_aes128_ssl`
- `dec_aes256`
- `dec_aes256_ssl`
- `dec_fp_aes256_alphanum`
- `dec_fp_aes256_alphanum_ssl`
- `dec_fp_aes256_ascii`
- `dec_fp_aes256_ascii_ssl`

- `dec_gpg`

Where *field_name* is the source field to be decrypted, which must be the same *field_name* used in the previous encryption operation.

For decryption purposes, the *passphrase_option* is required as follows:

When using the `dec_gpg()` routine

Supply the password, in quotes, that corresponds to the key inside the secret key ring.

For all other decryption routines

The passphrase that you specified or referenced in the previous field-level encryption operation must be the same passphrase used for decryption. You can, however, use an alternative passphrase option. For example, you can use the literal string method for specifying the passphrase when encrypting, but use a `file:` option when decrypting, provided that the passphrase stored in the first line of the filename is identical to the literal passphrase given for the encryption operation. Similarly, when using `dec_aes256()`, if you used no argument for the *passphrase_option* for encryption, you must also use no argument for decryption.

secret_kr_path is a filename (with optional path) that contains the secret GPG key ring (see *Example 100* on page 369). Used only with `dec_gpg()`.

2.4 Advanced Encryption Standard (AES)

Standard encryption should be used when there is no requirement to preserve original field widths or any original field formatting.

The routines used for this method are `enc_aes256()` (for encryption) and `dec_aes256()` (for decryption).



AES encryption and decryption routines are also available using a 128-bit key algorithm. These routines are `enc_aes128` (for encryption) and `dec_aes128` (for decryption). All the rules applied to `aes256` will also apply to `aes128` routines.

Example 97 Standard Encryption

Consider the following input file with three fields: date, credit card number, and amount:

```

10/12/2006 4111222233334444 21.30
10/12/2006 4555666677778888 19.99
10/13/2006 5999000011112222 256.00
10/15/2006 5222333344445555 8.77
10/15/2006 4111333355557777 3723.50

```

The following **sortcl** script, **purchases_encrypt.scl**, encrypts the credit card number field:

```

/INFILE=purchases.in
  /FIELD= (date, POSITION=1, SIZE=10, AMERICAN_DATE)
  /FIELD= (cc, POSITION=12, SIZE=16)
  /FIELD= (total, POSITION=30, SIZE=7, NUMERIC)
/REPORT
/OUTFILE=purchases.encrypted
  /FIELD= (date, POSITION=1, SIZE=10, AMERICAN_DATE)
  /FIELD= (encc=enc_aes256(cc, "myPassword123"), POSITION=12, SIZE=24)
  /FIELD= (total, POSITION=37, SIZE=7, NUMERIC)

```

The arguments passed to the `enc_aes256()` routine were `cc`, the field to be encrypted, and `myPassword123`, the encryption passphrase enclosed in quotes. This must be the same passphrase used for any subsequent decryption.



As described in Syntax: ASCII/ALPHANUM Encryption *on page 359*, you can substitute the above literal passphrase "myPassword123" with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains `myPassword123` on the top line.

This produces **purchases.encrypted**:

```

10/12/2006 6jtwsg37heXMHQ1X7hbwiQ== 21.30
10/12/2006 2tpC7B4TSjoickODQ2MNPg== 19.99
10/13/2006 oftLzB4iNdmg7SDMZCsp5Q== 256.00
10/15/2006 XyCbogPOwSTNdOAl3N1qzMw== 8.77
10/15/2006 jLDqEg3s7Ct6R9kq1haQAw== 3723.50

```

The credit card number field has been obscured with printable ASCII characters and protected with 256-bit encryption.

Standard Encryption Field Size Considerations

In the previous example, note the size difference between the original credit card number field and the ciphertext output from encryption. Fields are encrypted in blocks of 16 bytes (characters). If a block is less than 16 bytes, it will be padded to the right with zero value bytes (0x00 or null).

Because the result of encryption is a binary value (may contain bytes in the range of 0 to 255), it is automatically encoded using the Base64 algorithm. Base64 replaces every 3 binary bytes with 4 printable characters. This results in a further 3:4 expansion of the field size. The steps below can be used to determine the size that **sortcl** requires for the encrypted output field:

- 1) Round up the field size to the next even multiple of 16.
- 2) Divide by 3.
- 3) Round up to the next whole number.
- 4) Multiply by 4.

In the previous example, the original field size is 16. Because 16 is a multiple of 16, rounding up is not required. Therefore, 16 is divided by 3, and the result is rounded up to 6. 6 is then multiplied by 4 to arrive at the encrypted field size of 24.



It is suggested that you specify the original field size when fields are to be decrypted (as shown in the output section of the script in the next example). An 8-character field will be padded with 8 null bytes to reach the 16-byte boundary. If the resultant 24-byte encrypted field is to be wholly decrypted, it will result in a 16-byte field that is padded with nulls.

Example 98 Standard Decryption

The next script, **purchases_decrypt.scl**, decrypts the credit card numbers from the encrypted file **purchases.encrypted**, which was produced in the previous example:

```
/INFILE=purchases.encrypted
  /FIELD=(date, POSITION=1, SIZE=10, AMERICAN_DATE)
  /FIELD=(encc, POSITION=12, SIZE=24)
  /FIELD=(total, POSITION=37, NUMERIC)
/OUTFILE=purchases.restored
  /FIELD=(date, POSITION=1, SIZE=10, AMERICAN_DATE)
  /FIELD=(cc=dec_aes256(encc, "myPassword123"), POSITION=12, SIZE=16)
  /FIELD=(total, POSITION=29, SIZE=7, NUMERIC)
```

The arguments passed to the `dec_aes256()` routine were `encc`, the field to be decrypted, and `myPassword123`, which is the same passphrase used in the previous encryption operation, enclosed in quotes.



As described in Syntax: ASCII Decryption *on page 361*, you can substitute the above literal passphrase "`myPassword123`" with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains `myPassword123` on the top line.

This produces **purchases.restored**, which restores the original credit card numbers:

```
10/12/2006 4111222233334444 21.30
10/12/2006 4555666677778888 19.99
10/13/2006 5999000011112222 256.00
10/15/2006 5222333344445555 8.77
10/15/2006 4111333355557777 3723.50
```

The `dec_aes256()` routine was applied to the credit card field, and the original numbers are restored.

2.5 GPG Encryption and Decryption

GPG encryption and decryption routines can be used only when GPG is installed on your machine. You must use these routines if you want to apply GPG's key management facility to **sortcl**'s field-level encryption and decryption operations.



The GPG routines provided with **CoSort** are also compatible with Pretty Good Privacy (PGP). The only difference between GPG and PGP is that GPG is a free product using a GNU General Public License (GPL), whereas PGP provides commercial support and it can be used for commercial purposes. Please refer to your product license terms before generating keys with **CoSort**'s GPG routines.

Contact IRI if you need GPG/SSL (Windows only).

When using the GPG routines, you will be required to supply a public key ring for encryption, and a corresponding private / secret key ring for decryption (see Syntax: ASCII/ALPHANUM Encryption *on page 359* and Syntax: ASCII Decryption *on page 361*).

With GPG encryption, original field widths and any original field formatting are not preserved.

The routines used for this method are `enc_gpg()` and `dec_gpg()`. They are contained in the library file **libcs`gpg`.so** (UNIX/Linux) and **libcs`gpg`.dll** (Windows). By default, these are found in **\$COSORT_HOME/lib** (UNIX/Linux) or **install_dir\lib** (Windows). The **CoSort** GPG library files have a dependency file, **cl32.so** (UNIX/Linux) or **cl32.dll** (Windows), which is provided with the product **cryptlib**. You must move or copy the files from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The files can also be referenced from the current working directory.

Example 99 GPG Encryption

Consider the following tab-delimited input file, **personal_info**, which contains credit card numbers, driver's license numbers, and names:

9654-4338-8732-8128	W389-324-33-473-Q	Jessica Steffani
2312-7218-4829-0111	H583-832-87-178-P	Cody Blagg
8940-8391-9147-8291	E372-273-92-893-G	Jacob Blagg
6438-8932-2284-6262	L556-731-91-842-J	Just Rushlo
8291-7381-8291-7489	G803-389-53-934-J	Maria Sheldon
7828-8391-7737-0822	K991-892-02-578-O	Keenan Ross
7834-5445-7823-7843	F894-895-10-215-N	Francesca Leonie
8383-9745-1230-4820	M352-811-49-765-N	Nadia Elyse
3129-3648-3589-0848	S891-915-48-653-E	Gordon Cade
0583-7290-7492-8375	Z538-482-61-543-M	Hanna Fay

The following **sortcl** script, **GPG_encrypt.scl** encrypts the name field:

```
/INFILE=personal_info
  /FIELD=(credit_card, POSITION=1, SEPARATOR='\t')
  /FIELD=(driv_lic, POSITION=2, SEPARATOR='\t')
  /FIELD=(name, POSITION=3, SEPARATOR='\t')
/REPORT
/OUTFILE=personal_info_gpgName
  /FIELD=(credit_card, POSITION=1, SEPARATOR='\t')
  /FIELD=(driv_lic, POSITION=2, SEPARATOR='\t')
  /FIELD=(name=enc_gpg(name, "public1", \ # line continuation
    "/admin/fsuser/pubring.gpg"), POSITION=3, SEPARATOR='\t')
```

Three arguments were passed to the `enc_gpg()` routine. `name` is the field to be encrypted. And, the key ID used was `public1`, which corresponds to the key held in the public key ring inside **pubring.gpg**, which is located in the directory **C:\admin\fsuser** (see Syntax: ASCII/ALPHANUM Encryption *on page 359*).

This produces **personal_info_gpgName**:

9654-4338-8732-8128	W389-324-33-473-Q	wYwDFu5KtyLrpicBBAAgnEpDK...
2312-7218-4829-0111	H583-832-87-178-P	wYwDFu5KtyLrpicBBADbdXznf...
8940-8391-9147-8291	E372-273-92-893-G	wYwDFu5KtyLrpicBBAA7qBKyH...
6438-8932-2284-6262	L556-731-91-842-J	wYwDFu5KtyLrpicBBABybdKe1...
8291-7381-8291-7489	G803-389-53-934-J	wYwDFu5KtyLrpicBBADCZpKOH...
7828-8391-7737-0822	K991-892-02-578-O	wYwDFu5KtyLrpicBBAB4eMT33...
7834-5445-7823-7843	F894-895-10-215-N	wYwDFu5KtyLrpicBBAB3hWvTe...
8383-9745-1230-4820	M352-811-49-765-N	wYwDFu5KtyLrpicBBACXwL3wo...
3129-3648-3589-0848	S891-915-48-653-E	wYwDFu5KtyLrpicBBAC2M7lfQ...
0583-7290-7492-8375	Z538-482-61-543-M	wYwDFu5KtyLrpicBBACScCZYM...

As shown, the name field is encrypted using the GPG public key ring located in **pubring.gpg**. The width of the name column has been truncated for documentation purposes, because GPG-encrypted field widths are significantly longer than the original widths. Therefore, be sure to accommodate the increased length if applying a **SIZE** attribute to any GPG-encrypted fields (see **SIZE** *on page 99*). If you require width-preserving encryption, use **Format-Preserving Encryption and Decryption** *on page 369*.



It is suggested that you specify the original field size when fields are to be decrypted (as shown in the output section of the script in the next example). An 8-character field will be padded with 8 null bytes to reach the 16-byte boundary. If the resultant 24-byte encrypted field is to be wholly decrypted, it will result in a 16-byte field that is padded with nulls.

Example 100 GPG Decryption

The next script, **GPG_decrypt.scl** decrypts the name field from the encrypted file **personal_info_gpgName**, which was produced in the previous example:

```
/INFILE=personal_info_gpgName
/FIELD=(credit_card,POSITION=1,SEPARATOR='\t')
/FIELD=(driv_lic,POSITION=2,SEPARATOR='\t')
/FIELD=(name,POSITION=3,SEPARATOR='\t')
/REPORT
/OUTFILE=personal_info_restored
/FIELD=(credit_card,POSITION=1,SEPARATOR='\t')
/FIELD=(driv_lic,POSITION=2,SEPARATOR='\t')
/FIELD=(name=dec_gpg(name,"sec554", \
"/admin/fsuser/secring.gpg"),POSITION=3,SEPARATOR='\t')
```

Three arguments were passed to the `dec_gpg()` routine. `name` is the field to be decrypted. And, the key ID used was `sec554`, which corresponds to the key held in the secret key ring inside **secring.gpg**, which is located in the directory **C:\admin\fsuser** (see Syntax: ASCII Decryption on page 361).

This produces **personal_info_restored**, which restores the original names:

9654-4338-8732-8128	W389-324-33-473-Q	Jessica Steffani
2312-7218-4829-0111	H583-832-87-178-P	Cody Blagg
8940-8391-9147-8291	E372-273-92-893-G	Jacob Blagg
6438-8932-2284-6262	L556-731-91-842-J	Just Rushlo
8291-7381-8291-7489	G803-389-53-934-J	Maria Sheldon
7828-8391-7737-0822	K991-892-02-578-O	Keenan Ross
7834-5445-7823-7843	F894-895-10-215-N	Francesca Leonie
8383-9745-1230-4820	M352-811-49-765-N	Nadia Elyse
3129-3648-3589-0848	S891-915-48-653-E	Gordon Cade
0583-7290-7492-8375	Z538-482-61-543-M	Hanna Fay

The `dec_gpg()` routine was applied to the name field, and the original names are restored.

2.6 Format-Preserving Encryption and Decryption

sortcl supports the preservation of original field widths when encrypting ASCII fields. This is useful when you do not want to preserve custom format characteristics of a field, but need to retain original field widths.

The routines used for this method are `enc_fp_aes256_ascii()` (for encryption) and `dec_fp_aes256_ascii()` (for decryption).

Example 101 Width-Preserving Encryption

Consider the tab-delimited input file, **personal_info**, as shown in *Example 99* on page 367.

The following **sortcl** script, **PII_ASC_enc.scl**, encrypts the name field, while preserving its field widths:

```
/INFILE=personal_info
  /FIELD=(credit_card, POSITION=1, SEPARATOR='\t')
  /FIELD=(driv_lic, POSITION=2, SEPARATOR='\t')
  /FIELD=(name, POSITION=3, SEPARATOR='\t')
/REPORT
/OUTFILE=personal_info_name_encrypted
  /FIELD=(credit_card, POSITION=1, SEPARATOR='\t')
  /FIELD=(driv_lic, POSITION=2, SEPARATOR='\t')
  /FIELD=(name1=enc_fp_aes256_ascii(name, "pass"), POSITION=3, SEPARATOR="\t")
```

The arguments passed to the `enc_fp_aes256_ascii()` routine were the field name to be encrypted (`name`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption.



The `enc_fp_aes256_ascii()` routine can be used with the ASCII. For the ASCII encryption you need to provide `type=ASCII` or, by default, `ASCII`.

`enc_fp_aes256_ascii()` routine is capable of taking three arguments where the first argument is a field name which is a mandatory argument for the function. This argument should not require quotes (") around the field name. The second argument is a passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption. This second argument is not mandatory. If you do not use a passphrase, then the function will use an internal salt as a passphrase argument. The third argument to be called is an escape character(s) which take a single character or a string as an argument. These characters will be ignored in the encryption process. For example, if you have separators, such as a tab (`/t`) or comma (`,`), in your input data, then by mentioning (`"/t,"`) this will ignore both the separator and the comma in the encryption process. This is how you preserve your separator characters from your input data.



As described in Syntax: ASCII/ALPHANUM Encryption *on page 359*, you can substitute the above literal passphrase "pass" with a reference to a file-name (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains `pass` on the top line.

This produces **personal_info_name_encrypted**:

9654-4338-8732-8128	W389-324-33-473-Q	5 dGuN/a"Cp: "/I>
2312-7218-4829-0111	H583-832-87-178-P	j<K^m] ~G.>
8940-8391-9147-8291	E372-273-92-893-G	HG. HF0{*yq
6438-8932-2284-6262	L556-731-91-842-J	[xIcfp) #eg0
8291-7381-8291-7489	G803-389-53-934-J	D*9 I-G\o38w>
7828-8391-7737-0822	K991-892-02-578-O	SX0] ZAGfa7]
7834-5445-7823-7843	F894-895-10-215-N	PXdpCW0 ["a`,S8H
8383-9745-1230-4820	M352-811-49-765-N	n:gAG<fnN.1
3129-3648-3589-0848	S891-915-48-653-E	d=b\>" s\OB
0583-7290-7492-8375	Z538-482-61-543-M	:nMF^Q<) f

As demonstrated, the name field is encrypted, and the original field widths have been retained.

Example 102 Width-Preserving Decryption

The following script, **PII_ASC_dec.scl**, decrypts the name field from the encrypted file **personal_info_name_encrypted**, which was produced in the previous example:

```
/INFILE=personal_info_encrypted
/FIELD=(credit_card,POSITION=1,SEPARATOR="\t")
/FIELD=(driv_lic,POSITION=2,SEPARATOR="\t")
/FIELD=(name,POSITION=3,SEPARATOR="\t")
/REPORT
/OUTFILE=PII_orig
/FIELD=(credit_card,POSITION=1,SEPARATOR="\t")
/FIELD=(driv_lic,POSITION=2,SEPARATOR="\t")
/FIELD=(name1=dec_fp_aes256_ascii()(name,"pass"),POSITION=3,SEPARATOR="\t")
```

The arguments passed to the `dec_fp_aes256_ascii()` routine were the field name to be decrypted (`name`), followed by the decryption passphrase enclosed in quotes (`"pass"`), which must be the same passphrase used in the previous encryption operation.



As described in Syntax: ASCII Decryption *on page 361*, you can substitute the above literal passphrase "pass" with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains *pass* on the top line.

2.7 Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256)

sortcl supports the preservation of both numeric and alphabetic characters (including upper- and lower-case letters) when encrypting fields, as well as their original field widths. Other characters, such as - or *, are left unaltered so that field formats such as those found in social security numbers and phone numbers are retained. This type of encryption is useful for fields with Personally Identifiable Information (PII), where the field formats must be preserved, but the values themselves must be encrypted.

The routines used for this method are `enc_fp_aes256_alphanum()` (for encryption) and `dec_fp_aes256_alphanum()` (for decryption).

Example 103 Alpha-Numeric Format-Preserving Encryption

Consider the tab-delimited input file, **personal_info**, as shown in *Example 99* on page 367.

The following **sortcl** script, **PII_enc.scl**, encrypts the credit card and driver's license number fields, while preserving the field formats:

```
/INFILE=personal_info
  /FIELD=(credit_card, POSITION=1, SEPARATOR='\t')
  /FIELD=(driv_lic, POSITION=2, SEPARATOR='\t')
  /FIELD=(name, POSITION=3, SEPARATOR='\t')
/REPORT
/OUTFILE=personal_info_encrypted
  /FIELD=(credit_card1=enc_fp_aes256_alphanum(credit_card, "pass"), POSITION=1, SEPARATOR='\t')
  /FIELD=(driv_lic1=enc_fp_aes256_alphanum(driv_lic, "pass"), POSITION=2, SEPARATOR='\t')
  /FIELD=(name, POSITION=3, SEPARATOR='\t')
```

The arguments passed to the `enc_fp_aes256_alphanum()` routines were the field names to be encrypted (`credit_card` or `driv_lic`), followed by the encryption passphrase enclosed in quotes ("pass"). This must be the same passphrase used for any subsequent decryption.



As described in Syntax: ASCII/ALPHANUM Encryption *on page 359*, you can substitute the above literal passphrase "pass" with a reference to a file-name (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains `pass` on the top line.

This produces **personal_info_encrypted**:

0832-9678-1911-0645	R784-107-86-619-Q	Jessica Steffani
0835-7171-0577-5699	G156-454-45-303-O	Cody Blagg
0789-2128-0461-5374	Q305-118-71-384-Q	Jacob Blagg
1591-0561-0417-5772	D344-156-20-555-G	Just Rushlo
9296-9613-4710-5436	U751-860-67-075-Y	Maria Sheldon
9881-4436-0773-0973	X878-716-85-252-C	Keenan Ross
4594-9802-2566-4840	T273-579-67-063-M	Francesca Leonie
6514-3079-6147-6828	A617-849-83-864-X	Nadia Elyse
9221-6125-6496-9606	S039-406-12-369-U	Gordon Cade
1404-8512-8389-2619	K379-587-05-591-C	Hanna Fay

As shown above:

- The numeric values from the credit card number fields have been encrypted, but the field format remains intact. The encrypted values remain as digits, and the dashes have been retained.
- The alphabetic and numeric values from the driver's license fields have been encrypted, but the field format remains intact. The encrypted values remain as letters (upper-case preservation in this case) and digits where appropriate, and the dashes have been retained.

Example 104 Alpha-Numeric Format-Preserving Decryption

The next script, **PII_dec.scl**, decrypts the credit card and driver's license number fields from the encrypted file **personal_info_encrypted**, which was produced in the previous example:

```
/INFILE=personal_info_encrypted
  /FIELD=(credit_card,POSITION=1,SEPARATOR="\t")
  /FIELD=(driv_lic,POSITION=2,SEPARATOR="\t")
  /FIELD=(name,POSITION=3,SEPARATOR="\t")
/REPORT
/OUTFILE=PII_orig
  /FIELD=(credit_card1=FPD_ALPHANUM(credit_card,"pass"),POSITION=1,SEPARATOR="\t")
  /FIELD=(driv_lic1=dec_fp_aes256_alphanum()(driv_lic,"pass"),POSITION=2,SEPARATOR="\t")
  /FIELD=(name,POSITION=3,SEPARATOR="\t")
```

The arguments passed to the `dec_fp_aes256_alphanum()` routines were the field names to be decrypted (`credit_card` or `driv_lic`), followed by the decryption passphrase enclosed in quotes ("`pass`"), which must be the same passphrase used in the previous encryption operation.



As described in Syntax: ASCII Decryption *on page 361*, you can substitute the above literal passphrase "`pass`" with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains `pass` on the top line.

Example 105 Using ASCII Encryption on Database Columns

As described in ODBC *on page 62*, you can use `/PROCESS=ODBC` in a **sortcl** job script to process (on input) and populate (on output) table data in databases supported by Open Database Connectivity (ODBC).

This example demonstrates how the ASCII encryption and decryption routines can be applied to database data.



To perform encryption with `/PROCESS=ODBC`, you must move or copy the library file **libsodbc.so** (UNIX/Linux) or **libsodbc.dll** (Windows) from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If **/modules** contains files that you are not using, it is recommended that you move the files back to **/lib**. The file can also be referenced from the current working directory.

Consider the following Oracle table, **EMPLOYEE_INFO**, the contents of which are displayed with a SQL **SELECT** query:

```
SELECT * FROM EMPLOYEE_INFO;
CREDIT_CARD          DRIV_LIC          NAME
-----
9654-4338-8732-8128 W389-324-33-473-Q Jessica Steffani
2312-7218-4829-0111 H583-832-87-178-P Cody Blagg
8940-8391-9147-8291 E372-273-92-893-G Jacob Blagg
6438-8932-2284-6262 L556-731-91-842-J Just Rushlo
8291-7381-8291-7489 G803-389-53-934-J Maria Sheldon
7828-8391-7737-0822 K991-892-02-578-O Keenan Ross
7834-5445-7823-7843 F894-895-10-215-N Francesca Leonie
8383-9745-1230-4820 M352-811-49-765-N Nadia Elyse
3129-3648-3589-0848 S891-915-48-653-E Gordon Cade
0583-7290-7492-8375 Z538-482-61-543-M Hanna Fay

10 rows selected.
SQL>
```

The following script, **odbc_encrypt.scl**, uses alpha-numeric format-preserving encryption for the credit card and driver's license number fields, and width-preserving encryption on the name field:

```
/INFILE="EMPLOYEE_INFO;DSN=oracle11g"           # identify source table and DSN
/PROCESS=ODBC                                   # required for extracting RDBMS data
  /FIELD=(CREDIT_CARD, POSITION=1, SEPARATOR='\t')
  /FIELD=(DRIV_LIC, POSITION=2, SEPARATOR='\t')
  /FIELD=(NAME, POSITION=3, SEPARATOR='\t')
/REPORT
/OUTFILE="EMPLOYEE_INFO;DSN=oracle11g"         # identify target table and DSN
/PROCESS=ODBC                                   # required for loading RDBMS data
/CREATE                                         # clears existing rows first
  /FIELD=(CREDIT_CARD=enc_fp_aes256_alphanum(CREDIT_CARD, "pass"), EXT_FIELD="CREDIT_CARD", \
    POSITION=1, SEPARATOR='\t')
  /FIELD=(DRIV_LIC=enc_fp_aes256_alphanum(DRIV_LIC, "pass"), EXT_FIELD=DRIV_LIC, POSITION=2, SEPARATOR='\t')
  /FIELD=(NAME=enc_fp_aes256_ascii()(NAME, "pass"), EXT_FIELD=NAME, POSITION=3, SEPARATOR='\t')
```

The arguments passed to the `enc_fp_aes256_alphanum()` routines were the field names to be encrypted (`credit_card` or `driv_lic`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption. The arguments passed to the `enc_fp_aes256_ascii()` routine were the field name to be encrypted (`name`), followed by the encryption passphrase enclosed in quotes (`"pass"`). This must be the same passphrase used for any subsequent decryption.

The use of `/PROCESS=ODBC` in the input section, along with the DSN and table name in the `/INFILE` statement ensure that data is accessed directly from the `EMPLOYEE_INFO` table in the `oracle11g` DSN. The use of `/PROCESS=ODBC` in the output section, along with the DSN and table name in the `/OUTFILE` statement ensure that the encrypted column data is loaded directly back into the `EMPLOYEE_INFO` table of the same DSN in this case. The `/CREATE` statement ensures that the original data rows are cleared before the new rows are inserted. For complete details on using `/PROCESS=ODBC`, see *ODBC on page 62*.



When using `/PROCESS=ODBC` on output, you must use the `EXT_FIELD` option to name the original database source column name for each `/FIELD` that invokes a routine (see *EXT_FIELD on page 68*).

After execution, the database table, EMPLOYEE_INFO, contains the following encrypted data:

```

SELECT * FROM EMPLOYEE_INFO
CREDIT_CARD      DRIV_LIC      NAME
-----
0832-9678-1911-0645 R784-107-86-619-Q 5 dGuN/a"Cp: "/I>
0835-7171-0577-5699 G156-454-45-303-O j<K^m] ~G.>
0789-2128-0461-5374 Q305-118-71-384-Q HG. |HF0{*yq
1591-0561-0417-5772 D344-156-20-555-G [xIcfp)#eg0
9296-9613-4710-5436 U751-860-67-075-Y D*9| I-G\o38w>
9881-4436-0773-0973 X878-716-85-252-C SX0] ZAGfa7]
4594-9802-2566-4840 T273-579-67-063-M PXdpCW0 ["a`,S8H
6514-3079-6147-6828 A617-849-83-864-X n:gAG<fnN.1
9221-6125-6496-9606 S039-406-12-369-U d=b\>" s\OB
1404-8512-8389-2619 K379-587-05-591-C :nMF^Q<) f

10 rows selected.

```

As shown above:

- The numeric values from the credit card number fields have been encrypted, but the field format remains intact. The encrypted values remain as digits, and the dashes were retained (see Format and Type Preserving Encryption and Decryption for ALPHNUMERIC (using AES 256) *on page 372*).
- The alphabetic and numeric values from the driver's license number fields were encrypted, but the field format was preserved. The encrypted values remain as letters (upper-case preservation in this case) and digits where appropriate, and the dashes were retained.
- The name field is encrypted, but not with format preservation, and the original field widths were retained (see Format-Preserving Encryption and Decryption *on page 369*).

2.8 Triple Data Encryption Standard (3DES)

Use Triple DES encryption when there is no requirement to preserve original field widths or original field formatting.

The routines used for this method are `enc_3des_ebc ()` for encryption and `dec_3des_ebc ()` for decryption.

Example 106 Triple DES encryption

Consider the following input file, **credit_card.in**, which contains credit card numbers.

```
8932-4338-8732-8128
2312-7218-4829-0111
8940-8391-9147-8128
6438-8932-2284-8291
8291-7381-8291-6262
9782-8391-7737-7489
7834-5445-7823-0822
8383-9745-1230-4820
3129-3648-3589-8128
0583-7290-7492-8375
```

The following script, **credit_card_enc.scl**, encrypts the numbers using 3DES encryption.

```
/INFILE=credit_card.in
  /FIELD=(credit_card, POSITION=1, SIZE=19, ASCII)
/REPORT
/OUTFILE=credit_card_enc.out
  /FIELD=(credit_card_enc=enc_3des_ebc(credit_card), POSITION=1, ASCII)
```

The first argument passed to `enc_3des_ebc ()` is the field name, and the second argument is a passphrase.



As described in Syntax: ASCII/ALPHANUM Encryption *on page 359*, you can substitute the above literal passphrase "myPassword123" with a reference to a filename (and optionally a path) that contains the passphrase as its first entry, for example,

```
"file:/home/usr/admin/Privatekey.txt"
```

where the file **Privatekey.txt** contains myPassword123 on the top line.

This produces **credit_card_enc.out**:

```
X90Muc0mh4fAhf8XCM85kBY/O33G5lzn
4BYA/K8dqpDc5cjwYpOlXipU/gS4gN2c
D7qC8d3IhYX3ECXVV6jG1eubi3lNEnf8
ryPJZ5gGjOF6fpqJbVHYnnbEk8pJbT90
8GuPlLzoTrTwKnhMi4nGwoKS1Gqns+tH
Ne75m9ROny4VRKIhTYgrhDXN1gF5s7UH
+Ax0U5p2tuCQcZbmhnDYn2KmTrwDbQKM
Izv7B8sdqLVeEUaRjsr5S6cjj1kWCw9h
+NeCVzSmGmYGRlqM2y9ZOE7ul8R1788k
vTElamB1Mtl9zABNyVn5zDRoDECietSW
```

As shown above, the credit_card field has been obscured with printable ASCII characters and protected with 3DES encryption.

3DES Encryption Field Size Considerations

In the above example, note the size difference between the original input data and the encrypted output data. If the block size is less than 8 bytes, it will pad to the right with nulls. Because the result of encryption is a binary value, it is automatically enclosed using the base64 format. Every 3 bytes will be replaced with the 4 printable characters. This results in a further 3:4 expansion of the field size. The steps below can be used to determine the size that **sortel** requires for the 3DES output field:

- 1) Round up the field size to the next even multiple of 8.
- 2) Divide by 3.
- 3) Round up to the next whole number.
- 4) Multiply by 4.

Example 107 Triple DES decryption

Consider the output file from the above example, **credit_card_enc.out**, as the input file.

The following script, **credit_card_dec.scl**, decrypts this input file. The results are the same as the original file, **credit_card.in**, that was encrypted in *Example 106*

```
/INFILE=credit_card_enc.out
  /FIELD=(credit_card_enc, POSITION=1, SIZE=19, ASCII)
/REPORT
/OUTFILE=credit_card_dec.out
  /FIELD=(credit_card_dec=dec_3des_ebc(credit_card_enc), POSITION=1, ASCII)
```

The first argument passed to `dec_3des_ebc ()` is the field name, and the second argument is a passphrase.

This produces **credit_card_dec.out**:

```
8932-4338-8732-8128
2312-7218-4829-0111
8940-8391-9147-8128
6438-8932-2284-8291
8291-7381-8291-6262
9782-8391-7737-7489
7834-5445-7823-0822
8383-9745-1230-4820
3129-3648-3589-8128
0583-7290-7492-8375
```

2.9 Encoding and Decoding

Use encoding routines when you need to encode byte data to be stored and transferred over media that are designed to deal with text data, such as email systems.

The routines used for this method are `encode_base64()`, `encode_base64_ssl()`, and `encode_hex()` for encryption.

encode_base64()

Converts the data in an ASCII string format into its base64 equivalent value.

`encode_base64()` uses the encoding scheme as defined in RFC1113. Encoding causes an expansion of data. Every 3 bytes is replaced with 4 printable characters. This results in a further 3:4 expansion of the field size.

encode_base64_ssl()

Converts the data in an ASCII string format into its base64 equivalent value.

`encode_base64_ssl()` uses the open ssl implementation of base 64 encoding.

Encoding causes an expansion of data. Every 3 bytes is replaced with the 4 printable characters. This results in a further 3:4 expansion of the field size.

encode_hex()

Returns the hexadecimal representation of the byte input. Hex encoding converts 8 bit data to 2 hex characters. The hex characters are then stored as the two byte string representation of the characters. Hex encoding is probably a better choice than base64 encoding for human readability.

Use `decode_hex()`, `decode_base64()`, and `decode_base64_ssl()` to decode the above routines.

The encode and decode base64 routines are provided in the library file **libscrypt.so** or **libscrypt.dll** (Windows). The encode and decode for hex functions are provided in **libsutil.so** (UNIX/Linux) and **libsutil.dll** (Windows). The default location for all CoSort library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows).

2.10 SHA-2 Hashing

The purpose of the (Secure Hash Algorithm) SHA-2 field-level routine is to return a hash of the data string in a given field or column. This is also known as a cryptographic hashing function. For any number of input bytes, the routine always returns a 32-byte hash code (in binary) that internally translates to a 44-byte, base64 value. Hashing is a useful technique for integrity checking.

Example 108 Using SHA-2

Consider the following input file, **account_number.in**, which contains account numbers.

```
AWE-399732947327302
DKF-492376343732919
VMN-399732947327302
OIS-399732947327302
KAK-399732947327302
USK-399732947327302
KLS-399732947327302
DSJ-399732947327302
KWP-399732947327302
LSA-399732947327302
```

The following script, **account_number.scl**, takes the hash off the account numbers; the resulting hash will be in base64 format. The length of the base64 output will be 44 bytes of ASCII-readable characters.

```
/INFILE=account_number.set
  /FIELD= (Membership_Number, POSITION=1, SIZE=19, ASCII
/REPORT
/OUTFILE=account_number.out
  /FIELD= (Membership_Number_hash=sha256 (Membership_Number) , POSITION=1, ASCII)
```

The argument passed to `hash_sha2` is the field name. The resulting data will be always a hash of the input field data.

This produces **account_number.out**:

```
MjIhEPbKMfh9/tv7KsqLAd/cXiIKac4KdEWVZBCNFEE=
FCKh0SUlq2PXvzdYy4ZKa9ZyRnfZ3obGx2kH+yybw/o=
UkX5lwU+4DMajxr02qb92Wl6kqVWybFhky7KVDm3zL8=
tVXPjejWNJL4m5sRjjYNzhnqAV8MDBRF2SjXRd8QiR0=
CiqalJbZ4Vd6+uZfwWToYKIzV6kexioy9uX8Zg9cKw0=
BB3AmhSyn4Ya0jk3F7Gx3j3gK91cieZNVpyMK5405CY=
gjxv2Cuah6HSD4pEch8fUiv8sEBFRrDLy2atBxgYlW0=
uDx/P5A5DT1xEIKWkFo1JQWPwFI7TTCemIsIamz5PdW=
Pa29aDcM+1ODXCzaiVeV/Y7IGJ9qcCaSIUo7ccLHsaE=
g6vHVOO3LmJEUJ29qTD/mB9NMtW4Ax0q+UKQaC1WP2s=
```

3 MELISSA DATA CLEANSING ROUTINE

This example demonstrates how you can apply an address standardization routine from Melissa Data in a **sortel** script.



This examples requires Melissa Data's Address Object product to be installed on your system. For more information on this tool, see:

<http://www.melissadata.com/addressobject/addressobject.htm>.

Example 109 on page 385 references a Windows library, and is written for Windows users. If you are a Linux or UNIX user, contact IRI for instructions on building the requisite library.

The routine `melissadataAddressStandardize()` is provided by **CoSort** in the Windows library **libmelissadataplugin.dll**. Its default location is the **install_dir\lib** directory. The source code for the plug-in is available in the **install_dir\src** directory. The function `melissadataAddressStandardize()` can accept from one to 12 arguments. It returns a string which is the final standardized address.

To use a library file, move or copy the file from **/lib** in the home directory to **/lib/modules**. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or **install_dir\lib\modules** (Windows). If there are files in **/modules** that you are not using, it is recommended that you move the files back to **/lib**.

3.1 Input Arguments

The address field to be standardized contains two major parts, and the value can be divided into 11 components of any combination. When passed into the function as arguments, the components need to be separated into the two parts using the pipe (|) character so that Melissa Data parse the value correctly (parsing occurs before standardization).

For example, consider this address record:

2194 North Hwy A1A Suite 303, Melbourne, FL - 32937

The first part comprises the street address, and its components are as follows

Range 2194

PreDirection North

StreetName Hwy A1A

PostDirection

Suffix

SuiteName Suite

SuiteNumber 303

The second part comprises the city, state and zip:

City Melbourne

State FL

Zip 32937

Plus4

For complete details on address components, refer to the Address Object documentation supplied by Melissa Data.

The following are example combinations of components, and the correct way of specifying them as arguments in `melissadataAddressStandardize()`:

```
melissadataAddressStandardize("2194 Hwy A1A","Suite 303","|",  
                               "Melbourne Florida 32937")
```

```
melissadataAddressStandardize("2194","Hwy A1A","Suite 303","|",  
                               "Melbourne Florida","32937")
```

The components from the street address and the city, state, and zip can be combined or split in any manner as long as the street address and the city, state, and zip parts are separated by a pipe (`|`). Both field names and literal strings are acceptable as arguments, but literal strings were used for the purposes of this example.

3.2 Syntax

The syntax for invoking this routine is:

```
melissadataAddressStandardize(field_name or  
  "string" [field_name or "string" [,and so on] ,"|",field_name or  
  "string" [field_name or "string" [,and so on] )
```

where the `field_name` is an address component of an existing source field. Or, you can use a literal "`string`" for this value. All field names and strings to the left of the pipe (`|`)

comprise the first part of the address, and the field names and strings to the right of the pipe comprise the second part.

Example 109 Address Standardization Using Melissa Data

For example, consider the following input data, **address.txt**:

```
1320 Wisteria Drive, # 4721 *** Ann Arbor Michigan 48104
7950 Jones Branch Drive *** McLean VA 22107
2194 Hwy A1A, Suite 303 *** Melbourne FL 32937
8600 Rockville Pike *** Bethesda MD 20209
10301 Baltimore Avenue *** Beltsville MD 20705
425 Molton St. *** Montgomery AL 36104
200 East Van Buren St. *** Phoenix AZ 85004
4850 South Park Ave. *** Tucson AZ 85714
16 West Sixth St. *** Mountain Home AR 72653
123 West Alisal St. *** Salinas CA 93901
750 North Gene Autry Trail *** Palm Springs CA 92262
330 North West St. *** Visalia CA 93291
```

The following script, **address_clean.scl**, standardizes the disparate address types within the different records, provides a uniform presentation for the complete address, and also provides *zip code plus 4* values based on the street addresses:

```
/INFILE=address.txt
  /FIELD=(streetaddress,POSITION=1,SEPARATOR='***')
  /FIELD=(citystatezip,POSITION=2,SEPARATOR='***')
/REPORT
/OUTFILE=addresses.out
  /FIELD=(address=melissadatapluginAddressStandardize \
    (streetaddress,"|",citystatezip),POSITION=1,SEPARATOR='|')
```

The output file, **addresses.out**, reflects the application of the address standardization routine:

```
1320 Wisteria Dr # 4721 Ann Arbor MI 48104-4676
7950 Jones Branch Dr Mc Lean VA 22107-0001
2194 Highway Ala Ste 303 Indian Harbour Beach FL 32937-4932
10301 Baltimore Ave Beltsville MD 20705-2326
425 Molton St Montgomery AL 36104-3523
200 E Van Buren St Phoenix AZ 85004-2238
4850 S Park Ave Tucson AZ 85714-1637
123 W Alisal St Salinas CA 93901-2644
750 N Gene Autry Trl Palm Springs CA 92262-5463
330 N West St Visalia CA 93291-6010
```

As shown above, blanks are generated when standardization cannot be performed due to an incorrect or invalid character value or representation. In this case, **sortel** displays warning messages.

4 TRILLIUM CLEANSING ROUTINE

This example demonstrates how you can apply an address standardization routine from Trillium Software in a **sortcl** script.



To execute this example, you must have the Trillium product installed on your system. For more information, see www.trilliumsoftware.com.

Example 110 on page 391 references a Windows library, and is written for Windows users. If you are a Linux and UNIX user, contact IRI for instructions on building the requisite library.

This example utilizes a standardization routine that can be used to cleanse an address field. Using this example as a reference, you can develop more specific custom functions, depending on your requirements.

The routine `trilliumAddressStandardize()` is provided by **CoSort** in the library **libtrilliumplugin.dll**. It is located by default in *install_dir\lib*. The source code for the plug-in is located in *install_dir\src*.

To use a library file, move or copy the file from */lib* in the home directory to */lib/modules*. The default location for all **CoSort** library files is the **\$COSORT_HOME/lib/modules** directory (UNIX/Linux) or *install_dir\lib\modules* (Windows). If there are files in */modules* that you are not using, it is recommended that you move the files back to */lib*.

`trilliumAddressStandardize()` supports a varying number of arguments from 1 to a maximum of 12. Its return value is a string that contains the standardized address.

4.1 Input Arguments

The input arguments, which are components of an address, can be divided into two sets:

- arguments for the street component
- arguments for the geographical component.

These sets must be separated by a pipe (|) character so that the values can be parsed correctly and passed to the Trillium Cleanser.

The breakdown of a basic address field and the components supported by this function can be illustrated with the following example:

2194 North Highway A1A Suite 303 Melbourne Florida 32937

The first part, the street component, is comprised of the following:

House Number	2194
Street pre-direction	North
Street name	Highway A1A
Street post-direction	
Street suffix	
Secondary type	Suite
Secondary Number	303

The second part, the geographical component, is comprised of the city, state and zip:

City name	Melbourne
State/Region name	Florida
Zip code	32937
Plus4	

4.2 Syntax

The syntax for invoking this routine is:

```
trilliumAddressStandardize(field_name or "string" [field_name or "string"  
[,and so on],"|",field_name or "string" [field_name or "string" [,and so on])
```

where `field_name` is an address component in an existing source field. You can also use a literal `"string"` for this value.

All field names and/or strings to the left of the pipe (|) comprise the street component of the address. Field names and/or strings to the right of the pipe comprise the geographical component of the address.



For complete details on basic and extended address components, refer to the Trillium documentation.

The following are valid examples of the syntax for this combination of components:

```
trilliumAddressStandardize("2194 Hwy A1A", "Suite 303", "|",  
"Melbourne Florida 32937")
```

or

```
trilliumAddressStandardize("2194", "Hwy A1A", "Suite 303", "|",  
"Melbourne Florida", "32937")
```

The components from the street address and the city, state, and zip can be combined or split in any manner as long as the street address and the city, state, and zip parts are separated by a pipe (|). Both field names and literal strings are acceptable as arguments, but literal strings were used for the purposes of this example.



This plugin has been tested with the US postal data tables. No major changes should be needed in order to use, or extend, the plugin for other postal tables.

If the address contains more buckets, for example `FLOOR`, the information will need to be combined into one of the existing buckets before the values can be passed into the Trillium cleansing service.

The Trillium cleanser has a 100-byte constraint for each virtual line that can be passed to Trillium.

4.3 Table of Errors

The following table lists the supported errors that can be returned by the **CoSort** Trillium plugin:

Table 9: CoSort Trillium Plug-in Errors

Error Number	Name	Description
0	CE_EXT_OK	Return value is OK.
1	CE_EXT_BADTYPE	Indicates incorrect data type passed to the external function. The error causes CoSort to display a warning message and not call the external function again.
2	CE_EXT_BADVALUE	Indicates possible bad data. The error causes CoSort to display a warning message regarding this particular field data.
3	CE_EXT_BADNUM	Indicates possible bad data. The error causes CoSort to display a warning message regarding this particular field data.
-1	CE_EXT_ABORT	Indicates an error from which the custom function cannot recover, and causes the calling program CoSort to release all resources and abort.
0	CS_TRILL_OK	Return value is OK.
100	CS_TRILL_ERR_HANDLE	Indicates an error trying to obtain a handle to the Trillium cleanser service.
101	CS_TRILL_ERR_MEMORY	Insufficient memory error.
102	CS_TRILL_ERR_RECLENGTH	Maximum record length supported was exceeded.
103	CS_TRILL_ERR_CLEANSER	Indicates an error returned by the Trillium cleanse function. The variable iTrillError contains the error message number returned by Trillium.

Example 110 Address Standardization Using Trillium

For example, consider the following input data, **address.txt**:

```
1320 Wisteria Drive, Apt 4721 *** Ann Arbor Michigan 48104
7950 Jones Branch Drive *** McLean VA 22107
2194 Hwy A1A, Suite 303 *** Melbourne FL 32937
8600 Rockville Pike *** Bethesda MD 20209
10301 Baltimore Avenue *** Beltsville MD 20705
425 Molton St. *** Montgomery AL 36104
200 East Van Buren St. *** Phoenix AZ 85004
4850 South Park Ave. *** Tucson AZ 85714
16 West Sixth St. *** Mountain Home AR 72653
123 West Alisal St. *** Salinas CA 93901
750 North Gene Autry Trail *** Palm Springs CA 92262
330 North West St. *** Visalia CA 93291
```

The following script, **address_clean.scl**, standardizes the disparate address types within the different records, provides a uniform presentation for the complete address, and also provides *zip code plus 4* values based on the street addresses:

```
/INFILE=address.txt # input data file
  /FIELD=(streetaddress,POSITION=1,SEPARATOR='***') # street address field
  /FIELD=(citystatezip,POSITION=2,SEPARATOR='***') # geographic location field
/REPORT
/OUTFILE=address.out # output data file
# call the address standardization function:
  /FIELD=(address=trilliumStandardizeAddress(streetaddress,"|" \
    ,citystatezip),POSITION=1,SEPARATOR='|')
```

The output file, **addresses.out**, reflects the application of the address standardization routine:

```
1320 WISTERIA DR APT 4721 ANN ARBOR MI 48104-4676
7950 JONES BRANCH DR MCLEAN VA 22102-3302
2194 HIGHWAY A1A STE 303 MELBOURNE FL 32937-4932
8600 ROCKVILLE PIKE BETHESDA MD 20894-0002
10301 BALTIMORE AVE BELTSVILLE MD 20705-2326
425 MOLTON ST MONTGOMERY AL 36104-3523
200 E VAN BUREN ST PHOENIX AZ 85004-2238
4850 S PARK AVE TUCSON AZ 85714-1637
16 W 6TH ST MOUNTAIN HOME AR 72653-3508
123 W ALISAL ST SALINAS CA 93901-2644
750 N GENE AUTRY TRL PALM SPRINGS CA 92262-5463
330 N WEST ST VISALIA CA 93291-6010
```

Blanks or empty strings are generated when standardization cannot be performed due to an incorrect or invalid character value or representation. In this case, **sortcl** will display warning messages.

sortcl TOOLS

This chapter discusses the tools that **CoSort** provides to aid in the use of the **sortcl** program. It contains the following sub-chapters:

- *cob2ddf on page 395*
- *csv2ddf on page 401*
- *ctl2ddf on page 399*
- *elf2ddf on page 405*
- *ldif2ddf on page 409*
- *odbc2ddf on page 411*
- *xml2ddf on page 417*

These command line programs parse COBOL copybooks, comma-separated values files, extended web logs, SQL*Loader control files, XML files, and LDIF files, and certain database tables (respectively) to generate SORTCL data definitions for use within **sortcl** jobs.



Meta Integration Technology, Inc. (MITI) has also created a Meta Integration Model Bridge (MMB) that automatically converts flat file layouts formatted in third-party application metadata structures (e.g., DataStage .dsx, Informatica .xml, and XMI) into sortcl data definition files (.ddf). For more information on available metadata conversions, see:

<http://metaintegration.net/Products/MIMB>.

- *CLF TEMPLATES on page 421*
Three **.ddf** files are provided as metadata support for the three NCSA separate (common) log file types.
- *mvs2scl on page 433*
- *sorti2scl on page 427*
- *vse2scl on page 445*

These command line programs convert existing sort parameters into SORTCL job specifications for immediate (seamless) or future **sortcl** execution.

cob2ddf

1 PURPOSE

cob2ddf (COBOL-to-**sortcl**) is a translation program for users with input data from a COBOL application who want to convert the record (copybook) layouts into **sortcl** data definition files. The **cob2ddf** program, located in the directory **\$COSORT_HOME/bin** on Unix (**\install_dir\bin** on Windows), produces descriptive file name and field-layout text that can be referenced by, or pasted directly into, a **sortcl** job.

SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. **sortcl** is the **CoSort** user interface that uses the **SORTCL** syntax and the **cosort()** library routine. The **SORTCL** language supports multiple, differently-formatted input and output files, structured reports, and has features that encompass and surpass COBOL and legacy sort capabilities.

Currently, **cob2ddf** does not convert the entire range of COBOL data-definition functionality, but it provides a convenient way to convert field descriptions. See the Micro Focus COBOL Language Reference for documentation on the data description portion of COBOL programs.

2 USAGE

The syntax is:

```
cob2ddf [-warningsoff] [-use-wordstorage] infile [ddf_file]
```

To execute **cob2cl** and create a **sortcl** data definition file, enter:

```
cob2ddf filename.cbl ddf_file
```

where *filename.cbl* is the name of the copybook or file description file, and *ddf_file* is the resultant **sortcl** data definition file. For example, the command:

```
cob2ddf cpybk.cbl cpybk.ddf
```

converts the file and field descriptions in **cpybk.cbl** to a **sortcl** data definition file named **cpybk.ddf**. For details on how to reference a **.ddf** file from within a **sortcl** job script, see *Data Definition Files* on page 46.

You can turn off any runtime warning messages using the **-warningsoff** flag, for example:

```
cob2ddf -warningsoff cpybk.cbl cpybk.ddf
```

By default, runtime warning messages are enabled.

You can add a **wordstorage** flag on the command line to ensure that **sortcl** field **SIZE** attributes are computed in the word storage mode, for example:

```
cob2ddf -use-wordstorage copybk.cbl cpybk.ddf
```

For details on word storage mode sizes, see <http://docs.hp.com/cgi-bin/doc3k/BB243390008.13027/10>.

3 EXAMPLE

The following is a COBOL copybook file, **cob.app**:

```
01 REG
   05 PLANT PIC X(08) .
   05 FREE PIC 9(10) .
   05 CLIENT PIC X(09) .
   05 CARRIER-18 PIC 9(12) .
   05 CARRIER-23 PIC 9(12) .
   05 INTERESTPIC S9(11) sign is leading
                           separate character.
   05 CARGOESPIC S9(12) sign is leading
                           separate character.
```

To create the **sortcl** equivalent, enter the following on the command line:

```
cob2ddf cob.app cob.ddf
```

The resultant SORTCL data definition file, **cob.ddf**, is:

```
/FIELD=(FREE, POSITION=9, SIZE=10, NUMERIC)
/FIELD=(CLIENT, POSITION=19, SIZE=9)
/FIELD=(CARRIER_18, POSITION=28, SIZE=12, NUMERIC)
/FIELD=(CARRIER_23, POSITION=40, SIZE=12, NUMERIC)
/FIELD=(INTEREST, POSITION=52, SIZE=12, MF_DISPSLS)
/FIELD=(CARGOES, POSITION=64, SIZE=13, MF_DISPSLS)
```

This file gives the record layout (field definitions) for a file that can be used for input or for output. If this is used with a specific file name or will be referenced by an environment variable, you can add a `/FILE` statement at the top of the data definition file as follows:

```
/FILE=$REG
```

where `$REG` will reference the name of the file to be used in the job. The defined layout can be used for either input or output and more than one layout can be defined in the data definition file as long as each `/FILE` statement is unique.

If using the data definition file in this manner, then the following statement needs to be placed at the top of your **sortcl** job specification file:

```
/SPECIFICATION=cob.ddf
```

You can also copy the field definitions into the job specification file.

ctl2ddf

1 PURPOSE

The program **ctl2ddf** allows **CoSort** users to convert the column layouts specified in an Oracle SQL*Loader control file (**.ctl**) into a data definition file (**.ddf**) containing /
FIELD layout descriptions. This **.ddf** can then be used in **sortcl** job scripts (see *Data Definition Files* on page 46).

2 USAGE

You must have permission to access both the **ctl2ddf** and **sortcl** programs. These are standalone programs, i.e., they are not interdependent and do not require any other modules. They can be found in the **\$COSORT_HOME/bin** directory on Unix, or in **\install_dir\bin** on Windows.

To execute **ctl2ddf**, enter:

```
ctl2ddf control_file
```

where *control_file* is the SQL*Loader control file (typically **.ctl**) containing the column layout specifications you want to convert. The output is sent to **filename.ddf**, where **filename** is the table name specified in the .ctl file into which the data will be loaded.).

3 EXAMPLE

Given the following SQL*Loader control file, **test1.ctl**:

```
# SQL*Loader Control File specifications (user)
# Fri Mar 30 17:LOAD DATA
INFILE 'out.dat'
INTO TABLE emp_sorted
TRAILING NULLCOLS
(EMPNO position(0001:0006) DECIMAL EXTERNAL NULLIF (EMPNO = BLANKS),
ENAME position(0007:0016) char,
JOB position(0017:0026) char,
MGR position(0027:0032) DECIMAL EXTERNAL NULLIF (MGR = BLANKS),
SAL position(0033:0043) DECIMAL EXTERNAL NULLIF (SAL = BLANKS),
COMM position(0044:0053) DECIMAL EXTERNAL NULLIF (COMM = BLANKS),
DEPTNO position(0054:0056) DECIMAL EXTERNAL NULLIF (DEPTNO = BLANKS))
```

The command to convert it to a **sortcl** data definition file might be:

```
ctl2ddf test1.ctl
```

emp_sorted.ddf is generated:

```
/FILE=out.dat
/FIELD=(EMPNO, POSITION=1, SIZE=6, DOUBLE)
/FIELD=(ENAME, POSITION=7, SIZE=10)
/FIELD=(JOB, POSITION=17, SIZE=10)
/FIELD=(MGR, POSITION=27, SIZE=6, DOUBLE)
/FIELD=(SAL, POSITION=33, SIZE=11, DOUBLE)
/FIELD=(COMM, POSITION=44, SIZE=10, DOUBLE)
/FIELD=(DEPTNO, POSITION=54, SIZE=3, DOUBLE)
```

Note that the INFILE source for the **.ctl** file, **out.dat**, is used as the **/FILE** specification in the **.ddf** (see in *Data Definition Files* on page 46).

This **.ddf** can be now invoked from within a **sortcl** job script, for example:

```
/SPEC=test1.ddf
/INFILE=out.dat
/SORT
  /KEY=(JOB)
  /KEY=(SAL, DESCENDING)
/OUTFILE=stdout.dat
```

In this example, **stdout.dat** could be a named pipe for directly feeding output data into another SQL*Loader operation.

csv2ddf

1 PURPOSE

csv2ddf (comma separated values-to-**sortcl**) is a translation program for converting Microsoft CSV file header descriptions to **sortcl** data definition files. The **csv2ddf** program, found in the **\$COSORT_HOME/bin** directory (*install_dir\bin* on Windows systems), scans CSV files to produce descriptive file name and input field layout text from the header that can be referenced by, or pasted directly into, a **sortcl** job specification file.



For the purposes of **csv2ddf**, it is expected that the first record of a Microsoft **.csv** file data is preceded by header descriptions. Therefore, if your data file does not have a header, you cannot use the **csv2ddf** program. See *SEPARATOR* on page 95 for details on how to use **sortcl** to define input file fields for comma-delimited records.

For details on using the `/PROCESS=CSV` command in a **sortcl** specification file to instruct **sortcl** to treat a file as a **.csv** file, see *CSV* on page 60 in the *sortcl PROGRAM* chapter.

2 USAGE

The syntax of **csv2ddf** is:

```
csv2ddf filename.csv filename.ddf
```

where *filename.csv* is the name of the comma-separated-values file, and *filename.ddf* is the resulting **sortcl** data definition file. For example, the command:

```
csv2ddf spreadsheet.csv spreadsheet.ddf
```

converts the field layout descriptions in *spreadsheet.csv* to a **sortcl** data definition file called **spreadsheet.ddf**. For details on how to reference a **.ddf** file from within a **sortcl** job script, see *Data Definition Files* on page 46.

3 **EXAMPLE**

Using the following CSV input file, **test.csv**:

```
Element_Name,Windows_NT,Windows,Windows_CE,Win32s,Component,
Component_Version,Header_File,Import_Library,Unicode,Element_Type
ADsBuildEnumerator,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsBuildVarArrayInt,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsBuildVarArrayStr,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsEnumerateNext,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsFreeEnumerator,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsGetLastError,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsGetObject,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsOpenObject,4.0 or later,,,,ADSI,,adshlp.h,,,function
ADsSetLastError,4.0 or later,,,,ADSI,,adshlp.h,,,function
IAds::Get,4.0 or later,,,,ADSI,,iads.h,,,interface method
```

Execute the following command to generate the **.ddf**:

```
csv2ddf test.csv csv.ddf
```

You can then modify the resultant **csv.ddf** to create the following sort script, **csv.scl**:

```
/INFILE=e:\test.csv
/PROCESS=CSV
  /LENGTH=0
  /FIELD=(Element_Name,POSITION=1,SEPARATOR=',')
  /FIELD=(Windows_NT,POSITION=2,SEPARATOR=',')
  /FIELD=(Windows,POSITION=3,SEPARATOR=',')
  /FIELD=(Windows_CE,POSITION=4,SEPARATOR=',')
  /FIELD=(Win32s,POSITION=5,SEPARATOR=',')
  /FIELD=(Component,POSITION=6,SEPARATOR=',')
  /FIELD=(Component_Version,POSITION=7,SEPARATOR=',')
  /FIELD=(Header_File,POSITION=8,SEPARATOR=',')
  /FIELD=(Import_Library,POSITION=9,SEPARATOR=',')
  /FIELD=(Unicode,POSITION=10,SEPARATOR=',')
  /FIELD=(Element_Type,POSITION=11,SEPARATOR=',')
  /KEY=(Element_Type)
  /KEY=(Windows)
  /KEY=(Windows_NT)
  /KEY=(Windows_CE)
/OUTFILE=e:\csv.out
/PROCESS=RECORD
  /LENGTH=0
```

Because the file type is converted from /PROCESS=CSV to /PROCESS=RECORD, the header record is removed from the sorted output.

You can also specify `/PROCESS=CSV` on output, in which case **sortcl** adds a CSV-style header record based on the field names specified on output (or the field names specified on input if there is no remapping).



When specifying `/PROCESS=CSV` on output, **sortcl** will enclose each output field with double-quotes if you specify comma-separated fields on output, which is recommended. Otherwise, the input field layout is used by default (which can not be comma-separated), and the only change on output will be the addition of the header.

elf2ddf

1 PURPOSE

elf2ddf (extended log format-to-**sortcl**) is a translation program for converting W3C web data descriptions to SORTCL data definition files. The **elf2ddf** program, found in the **\$COSORT_HOME/bin** directory (*\install_dir\bin* on Windows), scans web transaction files in ELF to produce descriptive file name and field-layout text from the header that can be referenced by, or pasted directly into, a **sortcl** job specification file.

SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. **sortcl** is the **CoSort** user interface that uses the **SORTCL** syntax and the `cosort()` library routine. The SORTCL language supports multiple, differently-formatted input and output files, and structured reports.

For details on using the `/PROCESS=ELF` command in the `/INFILE` and `/OUTFILE` sections of a **sortcl** specification file, see *ELF (W3C Extended Log Format)* on page 77 in the *sortcl PROGRAM* chapter.

sortcl also handles web logs in CLF format. The sample **sortcl** data definition files **CLF_Referrer.ddf**, **CLF_Agent.ddf**, and **CLF_Access.ddf** are provided in the **examples/sortcl** directory.

2 USAGE

elf2ddf requires that your ELF file is in the W3C convention format described on this page.

An extended log file contains a sequence of lines containing ASCII characters terminated by either the sequence LF or CRLF. Log file generators should follow the line termination convention for the platform on which they are executed. Analyzers should accept either form. Each line can contain either a *directive* or an *entry*.

Entries consist of a sequence of fields relating to a single HTTP transaction. Fields are separated by white space, the use of tab characters for this purpose is encouraged. If a field is unused in a particular entry dash "-" marks the omitted field. Directives record information about the logging process itself.

Lines beginning with the # character contain directives. The following directives are defined:

- **Version:** *integer.integer*
The version of the extended log file format used.
- **Fields:** [*specifier...*]
Specifies the fields recorded in the log.
- **Software:** *string*
Identifies the software which generated the log.
- **Start-Date:** *date_time*
The date and time at which the log was started.
- **End-Date:** *date_time*
The date and time at which the log was finished.
- **Date:** *date_time*
The date and time at which the entry was added.
- **Remark:** *text*
Specifies comment information. Data recorded in this field should be ignored by analysis tools.

The directives **Version** and **Fields** are required and should precede all entries in the log. The **Fields** directive specifies the data recorded in the fields of each entry.

The syntax of **elf2ddf** is:

```
elf2ddf filename.elf filename.ddf
```

To execute **elf2cl** and create a **sortcl** data definition file, enter:

```
elf2ddf filename.elf sortcl_script
```

where *filename.elf* is the name of the extended log format file, and *sortcl_script* is the resulting **sortcl** data definition file. For example, the command:

```
elf2ddf clickstream.elf clickstream.ddf
```

converts the field layout descriptions in the **clickstream.elf** file header to a **sortcl** data definition file named **clickstream.ddf**. For details on how to reference a **.ddf** file from within a **sortcl** job script, see *Data Definition Files* on page 46.

3 EXAMPLE

The following is a header from an ELF file:

```
#Version 1.0
#Date: 12-Jan-2000 00:00:00
#Fields: time cs-method cs-url
00:24:23 GET /tak/far.html
12:21:16 GET /tak/far.html
12:45:52 GET /tak/far.html
12:57:34 GET /tak/far.html
```

The following is the **sortcl .ddf** generated by **elf2ddf** based on the above header:

```
# SORTCL data definition file for ELF data
# Generated by elf2ddf.exe based on ELF header
# in "data.elf".
/file=data.elf
/process=ELF
/length=0
/field=(time, position=1, separator=' ', ASCII)
/field=(cs-method, position=2, separator=' ', ASCII)
/field=(cs-url, position=3, separator=' ', ASCII)
```



If you specify **/PROCESS=ELF** on output in a **sortcl** job script, an ELF-style header will be generated.

ldif2ddf

1 PURPOSE

The program **ldif2ddf** allows **CoSort** users to convert the column layouts specified in an LDIF (Lightweight Directory Interchange) format into a data definition file (**.ddf**) containing `/FIELD` layout descriptions. This **.ddf** can then be used in **sortcl** job scripts (see *Data Definition Files* on page 46). LDIF is the format of data exported from an LDAP database.

The **ldif2ddf** program is located in the directory **\$COSORT_HOME/bin** on Unix (**\install_dir\bin** on Windows).

2 USAGE

The syntax of **ldif2ddf** is:

```
ldif2ddf [-s separator] source_filename [target_filename]
```

where *source_filename* is the name of the LDIF file, and *target_filename* is the resultant **sortcl** data definition file (typically with a **.ddf** extension). If no target file is specified, the data definition statements will be written to the console.

Optionally, you can specify:

-s separator where *separator* is a field SEPARATOR string that you specify (see *SEPARATOR* on page 95). The default separator string is a pipe (`|`). The SEPARATOR and POSITION attributes are required for LDIF input fields defined in **sortcl**, but they have only internal significance (as described in *LDIF* on page 61).

ldif2ddf translates all of the LDIF fields that are found in the source LDIF file, so you may want to manually delete unwanted elements in the target DDF.

3 EXAMPLE

The following is an excerpt from the file **plant_catalog.ldif**:

```
common: Phlox, Woodland
botanical: Phlox divaricata
zone: 3
light: Sun or Shade
price: $2.80
availability: Jan/22/2008

common: Cardinal Flower
botanical: Lobelia cardinalis
zone: 2
light: Shade
price: $3.02
availability: Feb/22/2008

common: California Poppy
botanical: Eschscholzia californica
zone: Annual
light: Sun
price: $7.89
availability: Mar/27/2008

common: Mayapple
botanical: Podophyllum peltatum
zone: 3
light: Mostly Shady
price: $2.98
availability: Jun/05/2008
```

The following is the **sortcl** DDF translation produced by running **ldif2ddf**:

```
/FIELD=(common, POSITION=1, SEPARATOR='|')
/FIELD=(botanical, POSITION=2, SEPARATOR='|')
/FIELD=(zone, POSITION=3, SEPARATOR='|')
/FIELD=(light, POSITION=4, SEPARATOR='|')
/FIELD=(price, POSITION=5, SEPARATOR='|')
/FIELD=(availability, POSITION=6, SEPARATOR='|')
```

odbc2ddf

1 PURPOSE

The program **odbc2ddf** allows **CoSort** users to convert database table layouts specified into a **sortcl** data definition file (**.ddf**) containing `/FIELD` layout descriptions, provided the database is compatible with ODBC (Open Database Connectivity). This **.ddf** can then be used in **sortcl** job scripts (see *Data Definition Files* on page 46). ODBC is an interface that accesses certain databases.

SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. **sortcl** is the **CoSort** user interface that uses the SORTCL syntax and the `cosort()` library routine. The SORTCL language supports multiple, differently-formatted input and output files, structured reports, and has features that encompass and surpass COBOL and legacy sort capabilities.

odbc2ddf also allows you to query the database for such information as the list of ODBC drivers.

The **odbc2ddf** program is located in the directory **\$COSORT_HOME/bin** on Unix (**install_dir\bin** on Windows).

2 **USAGE**

The syntax for basic usage of **odbc2ddf**, that is, to generate `/FIELD` statements from a table description is:

```
odbc2ddf [Table] [outfile]
```

where `Table` is the name of an RDBMS table, and `outfile` is the resultant **sortcl** data definition file (typically with a **.ddf** extension). If no target file is specified, data definition statements are written to the console.

Complete syntax for the **odbc2ddf** command is:

```
odbc2ddf [-d] [-u username] [-p password] [DSN] [Table] [outfile]  
        [-f frame] [-s separator] [-l recordlength]
```

where:

- d** Lists the installed ODBC drivers.
- u** User name for the specified DSN. If no DSN is given, the possible DSNs are listed. If no `Table` is given, the tables in the specified DSNs are listed.
- p** Password for the specified DSN.

DSN The name of the DSN.

Table A specific table name from which to generate `/FIELD` statements.

outfile The resultant **sortcl** data definition file (typically with a **.ddf** extension). If no `outfile` is specified, the data definition statements will be written to the console.

if no DSN is given, the possible DSNs are listed

if no table is given, the tables in the specified DSNs are listed

- f** Specifies the frame character used in the `/FIELD` statement.
- s** Specifies the separator string used in the `/FIELD` statement.
- l** Output fixed length fields.

Additional options include:

- h** Displays help.
- v** Verbose mode.

odbc2ddf translates all of the table columns that are found in the source table, so you may want to manually delete unwanted columns (/FIELDS) in the target file.

3 EXAMPLE

The following is a database description of a table, **tablea**, residing on the database with DSN **my_oracle**:

Field	Type	Null	Default
name	varchar(40)	No	
age	smallint(5)	No	0
address	mediumint(6)	No	0
st	varchar(2)	No	
zip	text	No	
rate	float(4,2)	No	0.00
year	year(4)	No	0000
date	date	No	0000-00-00
timestamp	timestamp	Yes	CURRENT_TIMESTAMP
time	time	No	00:00:00
active	tinyint(1)	No	0
offset	int(11)	No	0
key	var(64)	No	
spanish	varchar(8)	No	
adjust	double(8,3)	No	0.00
filled	int(8)	No	00000099

The following is the **sortcl** DDF translation produced by running the following:

odbc2ddf my_oracle tablea table.ddf:

```
/FILE="tablea;DSN=my_oracle"  
/PROCESS=ODBC  
/FIELD=(name, POSITION=0001, SEPARATOR='|')  
/FIELD=(age, POSITION=0002, SEPARATOR='|', NUMERIC)  
/FIELD=(address, POSITION=0003, SEPARATOR='|', NUMERIC)  
/FIELD=(st, POSITION=0004, SEPARATOR='|')  
/FIELD=(zip, POSITION=0005, SEPARATOR='|')  
/FIELD=(rate, POSITION=0006, SEPARATOR='|', NUMERIC)  
/FIELD=(year, POSITION=0007, SEPARATOR='|', NUMERIC)  
/FIELD=(date, POSITION=0008, SEPARATOR='|')  
/FIELD=(timestamp, POSITION=0009, SEPARATOR='|')  
/FIELD=(time, POSITION=0010, SEPARATOR='|')  
/FIELD=(active, POSITION=0011, SEPARATOR='|')  
/FIELD=(offset, POSITION=0012, SEPARATOR='|', NUMERIC)  
/FIELD=(key, POSITION=0013, SEPARATOR='|')  
/FIELD=(spanish, POSITION=0014, SEPARATOR='|')  
/FIELD=(adjust, POSITION=0015, SEPARATOR='|', NUMERIC)  
/FIELD=(filled, POSITION=0016, SEPARATOR='|', NUMERIC)
```

Note that the comments in the above example refer to the ODBC data type value, which can be any of the following that have **sortcl** data type equivalents:

Table 10: Supported ODBC Data Types

ODBC Data Type	sortcl Equivalent
CHAR	ASCII
NUMERIC	NUMERIC
DECIMAL	NUMERIC
INTEGER	NUMERIC, PRECISION=0
SMALLINT	NUMERIC, PRECISION=0
FLOAT	NUMERIC
REAL	NUMERIC
DOUBLE	NUMERIC
DATETIME	ASCII
VARCHAR	ASCII
DATE	ASCII
TIME	ASCII
TIMESTAMP	ASCII



Date, Time, and Timestamp columns will map to ASCII using `odbc2ddf` because ASCII is most likely to load successfully into all `odbc`-supported database types. If sorting or manipulations are required, such as date intervals (see the *Date Intervals* chapter on page 216) then you must declare the appropriate date or time stamp data type, rather than ASCII, and exact size. Contact IRI Support with any special requirements.

xml2ddf

1 PURPOSE

xml2ddf (eXtensible Markup Language format-to-DDF) is a translation program for scanning XML files to produce descriptive file name and field-layout text that can be referenced by, or pasted directly into, a **sortcl** job specification file. The **xml2ddf** program is located in the directory **\$COSORT_HOME/bin** on Unix (*install_dir\bin* on Windows).

SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. **sortcl** is the **CoSort** user interface that uses the SORTCL syntax and the `cosort()` library routine. The SORTCL language supports multiple, differently-formatted input and output files, structured reports, and has features that encompass and surpass COBOL and legacy sort capabilities.

After translation, you can edit the format of the field statements within the **sortcl** data definition file, or within the **sortcl** job specification file if the DDF contents have been pasted or appended.

For details on how to reference a **.ddf** file from within a **sortcl** job script, see *Data Definition Files* on page 46.

For details on using the `/PROCESS=XML` command in the `/INFILE` and `/OUTFILE` sections of a **sortcl** specification file, see *XML* on page 68 in the *sortcl PROGRAM* chapter on page 37.

2 USAGE

The syntax of **xml2ddf** is:

```
xml2ddf [options] source_filename [target_filename]
```

where *source_filename* is the name of the XML file (typically with a **.xml** extension), and *target_filename* is the resultant **sortcl** data definition file (typically with a **.ddf** extension). If no target file is specified, the data definition statements will be written to the console.

Additional options include:

- f *frame*** where *frame* is a **FRAME** character that you specify (see *FRAME* on page 101). The default frame character is a double quote (").
- s *separator*** where *separator* is a field **SEPARATOR** string that you specify (see *SEPARATOR* on page 95). The default separator string is a pipe (|). The **SEPARATOR** and **POSITION** attributes are required for XML input fields defined in **sortcl**, but they have only internal significance (as described in *XML* on page 68).

xml2ddf translates all of the XML fields that are found in the source XML file, so you may want to manually delete unwanted elements in the target DDF.

3 EXAMPLE

The following is an excerpt from the file **plant_catalog.xml**:

```
<CATALOG>
  <PLANT>
    <COMMON>Bloodroot</COMMON>
    <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
    <ZONE>4</ZONE>
    <LIGHT>Mostly Shady</LIGHT>
    <PRICE>$2.44</PRICE>
    <AVAILABILITY>2011-03-15</AVAILABILITY>
  </PLANT>
  ...
  <PLANT>
    <COMMON>Don's Flower</COMMON>
    <BOTANICAL>Thornicus purnhagenus</BOTANICAL>
    <ZONE>Perennial</ZONE>
    <LIGHT>Sun & Shade</LIGHT>
    <PRICE>$99.95</PRICE>
    <AVAILABILITY>2011-08-15</AVAILABILITY>
  </PLANT>
</CATALOG>
```

The following is the **sortcl** DDF translation produced by running **xml2ddf**:

```
/FIELD=(COMMON0001, POS=0001, SEP='|', FRAME='', XDEF="/CATALOG/PLANT/COMMON")
/FIELD=(BOTANICAL0002, POS=0002, SEP='|', FRAME='', XDEF="/CATALOG/PLANT/BOTANICAL")
/FIELD=(ZONE0003, POS=0003, SEP='|', FRAME='', XDEF="/CATALOG/PLANT/ZONE")
/FIELD=(LIGHT0004, POS=0004, SEP='|', FRAME='', XDEF="/CATALOG/PLANT/LIGHT")
/FIELD=(PRICE0005, POS=0005, SEP='|', FRAME='', XDEF="/CATALOG/PLANT/PRICE")
/FIELD=(AVAILABILITY0006, POS=0006, SEP='|', FRAME='', XDEF="/CATALOG/PLANT/AVAILABILITY")
```


CLF TEMPLATES

1 PURPOSE

NCSA Separate log (or *three-log*) format is a convention for storing NCSA common log data in three separate log files, rather than as a single file. The three log formats are:

Common (Access) log	Contains basic information from the NCSA log.
Referral log	Contains corresponding referral information.
Agent log	Contains corresponding agent information.

CoSort furnishes metadata support for these formats by providing three data definition file (**.ddf**) templates, each containing /FIELD layout descriptions usable in **sortcl** job scripts (see *Data Definition Files* on page 46).

2 USAGE

In the **\$COSORT_HOME/example/sortcl** directory, three **.ddf** templates with NCSA log field layouts are provided:

CLF_Access.ddf **sortcl** field layouts for common log or access log files.

CLF_Referral.ddf **sortcl** field layouts for referral log files.

CLF_Agent.ddf **sortcl** field layouts for agent log files.

To create job scripts which reference these layouts, you can open the **.ddf** you require, modify it to include job statements and any output file specifications, and save it as an **.scl** job script that can run with **sortcl** (see *EXAMPLES* on page 423).

Alternatively, you can create a new job script, and use a **/SPEC** statement to invoke the field layouts from the **.ddf** you require (see *Specification Files* on page 45).

3 EXAMPLES

3.1 Common (Access) Log

Given the NCSA common log file input data, **common.dat**:

```
129.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
127.125.125.125 - rjones [10/Oct/1999:21:15:04 +0500] "GET /index.html HTTP/1.0" 200 1043
127.125.125.125 - dsmith [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
125.125.125.125 - dsmith [10/Oct/1999:21:15:02 +0500] "GET /index.html HTTP/1.0" 200 1043
128.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
129.125.125.125 - dsmith [10/Oct/1999:21:15:08 +0500] "GET /index.html HTTP/1.0" 200 1043
125.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
126.125.125.125 - rjones [10/Oct/1999:21:15:04 +0500] "GET /index.html HTTP/1.0" 200 1043
126.125.125.125 - dsmith [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
128.125.125.125 - dsmith [10/Oct/1999:21:15:02 +0500] "GET /index.html HTTP/1.0" 200 1043
```

You can open **\$COSORT_HOME/example/sortcl/CLF_Access.ddf**, and modify it to create **common.scl**, which performs a two-key sort:

```
/INFILE=common.dat
  /FIELD=(all,POSITION=1)      # provided for remapping purposes only
  /FIELD=(host,POSITION=1,SEPARATOR=' ',IP_ADDRESS)
  /FIELD=(rfc931,POSITION=2,SEPARATOR=' ')
  /FIELD=(username,POSITION=3,SEPARATOR=' ')
  /FIELD=(time,POSITION=4,SEPARATOR=' ',EUROPEAN_TIMESTAMP)
  /FIELD=(timezone,POSITION=5,SIZE=5,SEPARATOR=' ',NUMERIC)
  /FIELD=(request,POSITION=6,SEPARATOR=' ',FRAME='')
  /FIELD=(statuscode,POSITION=7,SEPARATOR=' ',NUMERIC)
  /FIELD=(bytes,POSITION=8,SEPARATOR=' ',NUMERIC)
/SORT
  /KEY=username
  /KEY=host
/OUTFILE=stdout
  /FIELD=(all)
```

When executed, this produces the following sorted output:

```
125.125.125.125 - dsmith [10/Oct/1999:21:15:02 +0500] "GET /index.html HTTP/1.0" 200 1043
126.125.125.125 - dsmith [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
127.125.125.125 - dsmith [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
128.125.125.125 - dsmith [10/Oct/1999:21:15:02 +0500] "GET /index.html HTTP/1.0" 200 1043
129.125.125.125 - dsmith [10/Oct/1999:21:15:08 +0500] "GET /index.html HTTP/1.0" 200 1043
125.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
126.125.125.125 - rjones [10/Oct/1999:21:15:04 +0500] "GET /index.html HTTP/1.0" 200 1043
127.125.125.125 - rjones [10/Oct/1999:21:15:04 +0500] "GET /index.html HTTP/1.0" 200 1043
128.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
129.125.125.125 - rjones [10/Oct/1999:21:15:07 +0500] "GET /index.html HTTP/1.0" 200 1043
```

3.2 Referral Log

Given the NCSA referral log file input data, **ref.dat**:

```
[04/Sep/2004:21:15:05 +0500] "http://www.ibm.com"
[12/Oct/2004:21:18:05 +0500] "http://www.aol.com/index.html"
[22/Nov/2004:21:15:05 +0500] "http://www.microsoft.com"
[29/Dec/2004:21:16:05 +0500] "http://www.ibm.com/index.html"
[19/Oct/2004:21:15:05 +0500] "http://www.apple.com"
[04/Nov/2004:21:15:05 +0500] "http://www.aol.com/index.html"
[10/Oct/2004:21:15:05 +0500] "http://www.microsoft.com/index.html"
[10/Oct/2004:21:15:01 +0500] "http://www.ibm.com/index.html"
[10/Oct/2004:21:15:05 +0500] "http://www.apple.com/index.html"
[10/Oct/2004:21:15:05 +0500] "http://www.aol.com"
```

You can open **\$COSORT_HOME/example/sortcl/CLF_Referral.ddf**, and modify it to create **referral.scl**, which performs a two-key sort:

```
/INFILE=ref.dat
/FIELD=(time,POSITION=2,SIZE=20,EUROPEAN_TIMESTAMP) # fixed field
/FIELD=(referrer,POSITION=3,SEPARATOR=' ',FRAME='')# delimited field
/SORT
/KEY=referrer
/KEY=time
/OUTFILE=ref.out
/HEADREC="URL Referrer Timestamp\n-----\n"
/FIELD=(referrer)
/FIELD=(date_time,POSITION=40,EUROPEAN_TIMESTAMP)
```

When executed, this produces the following ordered report, sorted by referrer, then by timestamp:

URL Referrer	Timestamp

http://www.aol.com	10/Oct/2004:21:15:05
http://www.aol.com/index.html	12/Oct/2004:21:18:05
http://www.aol.com/index.html	04/Nov/2004:21:15:05
http://www.apple.com	19/Oct/2004:21:15:05
http://www.apple.com/index.html	10/Oct/2004:21:15:05
http://www.ibm.com	04/Sep/2004:21:15:05
http://www.ibm.com/index.html	10/Oct/2004:21:15:01
http://www.ibm.com/index.html	29/Dec/2004:21:16:05
http://www.microsoft.com	22/Nov/2004:21:15:05
http://www.microsoft.com/index.html	10/Oct/2004:21:15:05

3.3 Agent Log

Given the NCSA agent log file input data, **agent.dat**:

```
[10/Nov/2004:21:10:05 +0500] "Microsoft Internet Explorer - 5.0"
[15/Oct/2004:20:11:05 +0500] "Microsoft Internet Explorer - 6.0"
[16/Oct/2004:22:15:07 +0500] "Microsoft Internet Explorer - 6.0"
[09/Nov/2004:09:15:25 +0500] "Microsoft Internet Explorer - 5.0"
[11/Nov/2004:21:17:35 +0500] "Microsoft Internet Explorer - 5.0"
[11/Oct/2004:01:35:25 +0500] "Microsoft Internet Explorer - 6.0"
[10/Oct/2004:11:11:15 +0500] "Microsoft Internet Explorer - 6.0"
[05/Nov/2004:23:15:45 +0500] "Microsoft Internet Explorer - 5.0"
[14/Oct/2004:03:15:05 +0500] "Microsoft Internet Explorer - 6.0"
[03/Nov/2004:19:25:45 +0500] "Microsoft Internet Explorer - 5.0"
```

You can open **\$COSORT_HOME/example/sortcl/CLF_Agent.ddf**, and modify it to create **agent.scl**, which performs a two-key sort:

```
/INFILE=agent.dat
/FIELD=(all,POSITION=1) # provided for remapping purposes only
/FIELD=(time,POSITION=2,SIZE=20,EUROPEAN_TIMESTAMP) # fixed field
/FIELD=(agent,POSITION=3,SEPARATOR=' ',FRAME='')# delimited field
/SORT
/KEY=agent
/KEY=time
/OUTFILE=stdout
/FIELD=(all)
```

When executed, this produces the following results, sorted by agent then by timestamp:

```
[10/Oct/2004:11:11:15 +0500] "Microsoft Internet Explorer - 6.0"
[11/Oct/2004:01:35:25 +0500] "Microsoft Internet Explorer - 6.0"
[14/Oct/2004:03:15:05 +0500] "Microsoft Internet Explorer - 6.0"
[15/Oct/2004:20:11:05 +0500] "Microsoft Internet Explorer - 6.0"
[16/Oct/2004:22:15:07 +0500] "Microsoft Internet Explorer - 6.0"
[03/Nov/2004:19:25:45 +0500] "Microsoft Internet Explorer - 5.0"
[05/Nov/2004:23:15:45 +0500] "Microsoft Internet Explorer - 5.0"
[09/Nov/2004:09:15:25 +0500] "Microsoft Internet Explorer - 5.0"
[10/Nov/2004:21:10:05 +0500] "Microsoft Internet Explorer - 5.0"
[11/Nov/2004:21:17:35 +0500] "Microsoft Internet Explorer - 5.0"
```


sorti2scl

1 PURPOSE

For sort/merge users migrating from **CoSort**'s **sorti** sort interactive program to **sortcl**, IRI has produced a **sorti** parameter translation program. The **sorti2scl** utility scans batch **sorti** specifications to produce text that can be executed immediately by the **sortcl** program (see the *sortcl PROGRAM* chapter on page 37).

sortcl is the command line program that uses the SORTCL syntax and the `cosort()` library routine. SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. SORTCL supports multiple, differently-formatted input and output files, structured reports, and has features that encompass and surpass the capabilities of **sorti** (see *ADDITIONAL FUNCTIONS OF SORTCL* on page 432).

2 USAGE

2.1 Execution

You must have permission to access both the **sorti2scl** and **sortcl** programs. These are standalone programs, i.e., they are not interdependent and do not require any other modules. They can be found in the **\$COSORT_HOME/bin** directory on Unix, or in **\install_dir\bin** on Windows.

To execute **sorti2scl**, enter:

```
sorti2scl sorti_script
```

where *sorti_script* represents your **sorti** script file. The output is sent to **stdout** (standard output). If you wish the output to go to a file, then use the following:

```
sorti2scl sorti_script sortcl_script
```

where *sortcl_script* is a generic name for your new SORTCL script file. This file can then be examined, modified, and executed. **sortcl** execution takes the form:

```
sortcl /SPECIFICATIONS=sortcl_script [sortcl_options]
```

where *sortcl_options* are optional controls to **sortcl** (see the *sortcl PROGRAM* chapter on page 37 for details).

3 EXAMPLES

3.1 Example #1

Given the following **sorti** sort script, **test1.spc**

```
# SORTI specifications by user@NT (user)
# Fri Sep 30 17:44:23 2011
sort          # Action
0             # Record length (Variable)
1             # Number of input files
chiefs        # Input file 1
1 Stable      # Number of keys
descending    # Key 1 direction
fixed         #      location
24            #      starting column
5             #      length
ALPHABETIC    #      format
ASCII         #      data type
none          #      space/tab trimming
no            #      case-folded
both          # Output
test.out      # Output file name
```

The command to convert it for use by **sortcl** is:

```
sorti2scl test1.spc test1.scl
```

test1.scl is as follows:

```
# sortcl specs created by sorti2scl on Sun May 01 16:41:59 2011
#SORTI specifications by user@NT (user)
#Fri Sep 30 17:44:23 2011
/SORT
/STABLE
/INFILES=chiefs
  /KEY=(position=24,size=5,NONE,CASE_OFF,DESCENDING,ASCII)
/OUTFILE=stdout
/OUTFILE=test.out
```

The resultant **sortcl** specification file can be run with the command:

```
sortcl /spec=test1.scl
```

3.2 Example #2

As a second example, we will use the **sorti** parameters saved in **test2.spc**:

```
# SORTI specifications by C:M (Supra 7.5.1)
sort                                #action
0                                  # <- variable record size
1                                  # Number of input files
INDAT$$                            # Input file name
6                                  #number of sort keys
ascending                          #key number 1
fixed                              #key location
22                                 #start column
14                                 #length
alpha                              #format
none                               #space/tab trimming
no                                 #case sensitive
ascending                          #key number 2
fixed                              #key location
36                                 #start column
25                                 #length
alpha                              #format
none                               #space/tab trimming
no                                 #case sensitive
ascending                          #key number 3
fixed                              #key location
8                                  #start column
14                                 #length
alpha                              #format
none                               #space/tab trimming
no                                 #case sensitive
ascending                          #key number 4
fixed                              #key location
74                                 #start column
15                                 #length
alpha                              #format
none                               #space/tab trimming
no                                 #case sensitive
ascending                          #key number 5
fixed                              #key location
61                                 #start column
9                                  #length
alpha                              #format
none                               #space/tab trimming
no                                 #case sensitive
ascending                          #key number 6
fixed                              #key location
70                                 #start column
4                                  #length
alpha                              #format
none                               #space/tab trimming
no                                 #case sensitive
terminal                           #output file name
```

To create the **sortcl** equivalent, enter the following on the command line:

```
sorti2scl test2.spc test2.scl
```

The resultant **sortcl** specification file, **test2.scl**, is:

```
# sortcl specs created by sorti2scl on Sun Apr 01 16:38:51
2001
/SORT
/INFILES=INDAT$$
  /KEY=(position=22,size=14,NONE,CASE_OFF,ASCENDING,ASCII)
  /KEY=(position=36,size=25,NONE,CASE_OFF,ASCENDING,ASCII)
  /KEY=(position=8,size=14,NONE,CASE_OFF,ASCENDING,ASCII)
  /KEY=(position=74,size=15,NONE,CASE_OFF,ASCENDING,ASCII)
  /KEY=(position=61,size=9,NONE,CASE_OFF,ASCENDING,ASCII)
  /KEY=(position=70,size=4,NONE,CASE_OFF,ASCENDING,ASCII)
/OUTFILE=stdout
```

To run the sort, enter:

```
sortcl /spec=test2.scl
```

4 ADDITIONAL FUNCTIONS OF SORTCL

The SORTCL language and its methods for creating reports are described in detail in the *sortcl PROGRAM* chapter on page 37. The following are some additional capabilities of SORTCL which go beyond **sorti**:

- output files with multiple formats which can also have data types translated in the same pass
- summary files which have sums, counts, averages, minimum and maximum values
- running summary fields in detail records
- detail and multiple summary records can be written to the same output file to give a structured report
- mathematical and trigonometric functions across the records
- break conditions for intra-record and inter-record events
- vertical record selection via INCLUDE-OMIT statements
- Micro Focus Variable Length, ELF, CSV, and other file processing
- supports central file/record data definition sharing
- cross-table join (matching) functionality
- interface to cross-platform GUI
- ranking
- substrings
- field-length sensing and manipulation / alignment
- ASCII string find and replace
- custom field-level transformations
- field-level de-identification and encryption functions
- table look ups.

mvs2scl

1 PURPOSE

For MVS users migrating from an IBM mainframe z/OS (MVS) environment to open systems, IRI has produced an MVS sort parameter translation program. The **mvs2scl** program scans the MVS Job Control Language statements to produce text that can be executed immediately by the **sortcl** program.

SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. **sortcl** is the command line program that uses the SORTCL syntax and the `cosort()` library routine. The SORTCL language supports multiple, differently-formatted input and output files, and structured reports.

2 USAGE

2.1 Execution

You must have permission to access both the **mvs2scl** and **sortcl** programs. These are standalone programs, i.e., they are not interdependent and do not require any other modules. They can be found in the **\$COSORT_HOME/bin** directory in Unix, or in **\install_dir\bin** on Windows.

To execute **mvs2scl**, enter:

```
mvs2scl mvs_script
```

where *mvs_script* represents your MVS Job Control script file. The output is sent to standard out. If you wish the output to go to a file, then use the following:

```
mvs2scl mvs_script sortcl_script
```

where *sortcl_script* is a name for your new SORTCL script file.

If your input source is standard input, you can execute **mvs2scl** as follows:

```
cat mvs_script | mvs2scl
```

where the output is written to standard out. To write output to a file, such as *out.scl*:

```
cat mvs_script | mvs2scl stdin out.scl
```

where *cat mvs_script*, in these cases, represents the command to provide the input that will function as **stdin**.

Command-Line Options

The following options can follow the word **mvs2scl** when used on the command line:

- p Intersperses original non-sort JCL statements with the converted sort parameters, and comments out the sort-only JCL statements.
- a Uses ASCII as the character type when the incoming data type is CH.
- e Uses EBCDIC as the character type (the default) when the incoming data type is CH.

Environment Variable Options

There are also environment variables you can set to alter the behavior of **mvs2scl** whenever which will more closely imitate the JCL behavior on the mainframe:

COSORT_ZD_FILL=0

When you set **COSORT_ZD_FILL** to 0, all **ZONED_DECIMAL** fields will contain the attribute **FILL='0'** in the output section of the translated scripts. When **sortcl** is executed, this will ensure that the leading zeros are produced for the **ZONED_DECIMAL** values (see *FILL* on page 106).

MVS2SCL_MAPALL=yes

When you set **MVS2SCL_MAPALL** to **yes**, any fields that are not specifically referenced in the original job script are derived for the purposes of translation to the **sortcl** script. **mvs2scl** derives these fields (including positions and sizes) to complete the full record layout based on the specified record length in the JCL script. This way, the entire record is mapped to the output when **sortcl** is run.

MVS2SCL_NODOLLAR=1

When you set **MVS2SCL_NODOLLAR** to 1, the entries **/INFILE=SORTIN** and **/OUTFILE=SORTOUT** are created in the resultant **sortcl** script when there is no **SORTIN** or **SORTOUT** reference, respectively, in the source script. Without this setting, **/INFILE=\$SORTIN** and **/OUTFILE=\$SORTOUT** would be created in the resultant **sortcl** script (the default).

Resultant **.scl** files can be examined, modified, and executed. Execution takes the form:

```
sortcl /SPECIFICATIONS=sortcl_script [sortcl_options]
```

where *sortcl_options* are optional controls to **sortcl** (see the *sortcl PROGRAM* chapter on page 37 for details).



In some cases, the execution of **mvs2scl** can create multiple **sortcl** scripts from a single source. For example, from a batch script produced with Clarity's Mainframe Batch Manager. Contact your IRI agent for more details.

3 CONVERSION RULES

MVS scripts can contain statements that are not translated, such as references to physical locations (i.e., disk storage and cylinders). Also, most MVS operating system references are not applicable in the Unix or Windows environment.

Fields are defined with symbolic names (field_0, field_1, etc.) that are associated with absolute field positions and lengths. Subsequent references to these same fields refer to those given names. After the translation, you can change the field names to any name you want.

Data Types Not Recognized

- Leading Overpunch Sign.

Data Types Recognized

- Whole numbers, reals, floating (Alphanumeric)
- Character, Natural (ASCII)
- Character, Signed
- Character, Unsigned (EBCDIC)
- Integer, Short Signed
- Integer, Short Unsigned
- Integer, Natural Signed
- Integer, Natural Unsigned
- Integer, Long Signed
- Integer, Long Unsigned
- Float, Single Precision
- Float, Double Precision
- Zoned Decimal, Trailing Overpunch Sign.

Micro Focus COBOL

- COMP, Signed
- COMP, Unsigned
- COMP-3, Signed (Packed Decimal)
- COMP-3, Unsigned
- COMP-5, Signed
- COMP-5, Unsigned
- COMP-X
- DISP, Unsigned
- DISP, Sign leading
- DISP, Sign leading separate
- DISP, Sign trailing
- DISP, Sign trailing separate.

Ryan McFarland COBOL

- COMP, Signed
- COMP, Unsigned
- COMP-1
- COMP-3, Signed (Packed Decimal)
- COMP-3, Unsigned
- COMP-6
- DISP, Unsigned
- DISP, Sign leading
- DISP, Sign leading separate
- DISP, Sign trailing
- DISP, Sign trailing separate.

4 EXAMPLES

4.1 Example #1—OMIT Statement

Using the following MVS sort script, **mvs1**:

```
//OMIT      JOB
//SORT1     EXEC  PGM=SORT
//STEPLIB   DD    DSN=sort.DIRECTORY,DISP=SHR
//SYSOUT    DD    SYSOUT=A
//SORTIN    DD    DSN=pres18.dat,UNIT=2400-3,
//              VOL=SER=112233,DISP=(OLD,KEEP),
//              DCB=(LRECL=47,RECFM=FB,
//              BLKSIZE=900),LABEL=(1,SL)
//              DD    DSN=pres19.dat,UNIT=2400-3,
//              VOL=SER=223344,DISP=(OLD,KEEP),
//              DCB=(LRECL=47,RECFM=FB,
//              BLKSIZE=900),LABEL=(1,SL)
//SORTOUT    DD    DSN=dupl1840,VOL=SER=765456,
//              UNIT=2400-3,DISP=(NEW,KEEP),
//              DCB=(LRECL=47,RECFM=FB,
//              BLKSIZE=900),LABEL=(1,SL)
//SORTWK01   DD    SPACE=(CYL,(20)),UNIT=SYSDA
//SORTWK02   DD    SPACE=(CYL,(20)),UNIT=SYSDA
//SORTWK03   DD    SPACE=(CYL,(20)),UNIT=SYSDA
//SORTWK04   DD    SPACE=(CYL,(20)),UNIT=SYSDA
//SORTWK05   DD    SPACE=(CYL,(20)),UNIT=SYSDA
//SYSIN      DD    *
              SORT  FIELDS=(23,2,A,29,3,D),
                    FORMAT=CH,EQUALS
OMIT  COND=(29,1,CH,EQ,34,1,CH,&,
            1,5,CH,NE,C'Tyler')
              END
/*
```

here is the command to convert it for use with SORTCL and place the conversion in the file **mvs1.scl**:

```
mvs2scl mvs1 mvs1.scl
```

mvs1.scl is as follows:

```

/INFILES=(pres18.dat, pres19.dat)
/STABLE
/LENGTH=47
/FIELD=(field_0, POSITION=23, SIZE=2, EBCDIC)
/FIELD=(field_1, POSITION=29, SIZE=3, EBCDIC)
/FIELD=(field_2, POSITION=29, SIZE=1, EBCDIC)
/FIELD=(field_3, POSITION=34, SIZE=1, EBCDIC)
/FIELD=(field_4, POSITION=1, SIZE=5, EBCDIC)
/FIELD=(field_5, POSITION=29, SIZE=2, INT)
/CONDITION=(cond_0, TEST=(field_2 == field_3 AND
field_4 != "Tyler"))
/OMIT=(CONDITION=cond_0)
/KEY=(field_0, ASCENDING)
/KEY=(field_1, DESCENDING)
/OUTFILE=(dupl1840)
/LENGTH=47

```

The names of the input files have been grouped because they have a common format. The logical record length has become the value of the /LENGTH for the /INFILES. The fields have been given names that the user can substitute with more meaningful ones. The EQUALS command in the JCL has been changed to /STABLE.

Two /CONDITION statements are created to be used with /OMIT. Notice that the second /CONDITION statement is defined using the first /CONDITION statement.

The resultant **sortcl** specification file is used for sorting if you enter:

```
sortcl /spec=mvs1.scl
```

Here are is an input file with the US presidents, term starting in the 1800's, **pres18.dat**:

Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Adams, John Quincy	1825-1829	D-R	MA
Jackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA
Polk, James K.	1845-1849	DEM	NC
Taylor, Zachary	1849-1850	WHG	VA
Fillmore, Millard	1850-1853	WHG	NY
Pierce, Franklin	1853-1857	DEM	NH
Buchanan, James	1857-1861	DEM	PA
Lincoln, Abraham	1861-1865	REP	KY
Johnson, Andrew	1865-1869	REP	NC
Grant, Ulysses S.	1869-1877	REP	OH
Hayes, Rutherford B.	1877-1881	REP	OH
Garfield, James A.	1881-1881	REP	OH
Arthur, Chester A.	1881-1885	REP	VT
Cleveland1, Grover	1885-1889	DEM	NJ
Harrison, Benjamin	1889-1893	REP	OH
Cleveland2, Grover	1893-1897	DEM	NJ
McKinley, William	1897-1901	REP	OH

and the US presidents, term starting in the 1900's, **pres19.dat**:

Roosevelt, Theodore	1901-1909	REP	NY
Taft, William H.	1909-1913	REP	OH
Wilson, Woodrow	1913-1921	DEM	VA
Harding, Warren G.	1921-1923	REP	OH
Coolidge, Calvin	1923-1929	REP	VT
Hoover, Herbert C.	1929-1933	REP	IA
Roosevelt, Franklin D.	1933-1945	DEM	NY
Truman, Harry S.	1945-1953	DEM	MI
Eisenhower, Dwight D.	1953-1961	REP	TX
Kennedy, John F.	1961-1963	DEM	MA
Johnson, Lyndon B.	1963-1969	DEM	TX
Nixon, Richard M.	1969-1973	REP	CA
Ford, Gerald R.	1973-1977	REP	NE
Carter, James E.	1977-1981	DEM	GA
Reagan, Ronald W.	1981-1989	REP	IL
Bush, George H.W.	1989-1993	REP	TX
Clinton, William J.	1993-2001	DEM	AR

The output is written to the file **dupl1840**:

Clinton, William J.	1993-2001	DEM	AR
McKinley, William	1897-1901	REP	OH
Tyler, John	1841-1845	WHG	VA

4.2 Example #2—Summary Output

As a second example, we will use a similar input file, **chiefs.votes**.

Washington, George	201	1789-1797	FED	VA
Adams, John	202	1797-1801	FED	MA
Jefferson, Thomas	203	1801-1809	D-R	VA
Madison, James	204	1809-1817	D-R	VA
Monroe, James	205	1817-1825	D-R	VA
Adams, John Quincy	206	1825-1829	D-R	MA
Jackson, Andrew	207	1829-1837	DEM	SC
Van Buren, Martin	208	1837-1841	DEM	NY
Harrison, William H.	209	1841-1841	WHG	VA
Tyler, John	210	1841-1845	WHG	VA
Polk, James K.	211	1845-1849	DEM	NC
Taylor, Zachary	212	1849-1850	WHG	VA
Fillmore, Millard	213	1850-1853	WHG	NY
Pierce, Franklin	214	1853-1857	DEM	NH
Buchanan, James	215	1857-1861	DEM	PA
Lincoln, Abraham	216	1861-1865	REP	KY
Johnson, Andrew	217	1865-1869	REP	NC
Grant, Ulysses S.	218	1869-1877	REP	OH
Hayes, Rutherford B.	219	1877-1881	REP	OH
Garfield, James A.	220	1881-1881	REP	OH
Arthur, Chester A.	221	1881-1885	REP	VT
Cleveland1, Grover	222	1885-1889	DEM	NJ
Harrison, Benjamin	223	1889-1893	REP	OH
Cleveland2, Grover	224	1893-1897	DEM	NJ
McKinley, William	225	1897-1901	REP	OH
Roosevelt, Theodore	226	1901-1909	REP	NY
Taft, William H.	227	1909-1913	REP	OH
Wilson, Woodrow	228	1913-1921	DEM	VA
Harding, Warren G.	229	1921-1923	REP	OH
Coolidge, Calvin	230	1923-1929	REP	VT

In this case, we are interested in giving a sum of the numbers of the terms of presidency. The keys will be party and state. Suppose that the JCL is called **mvs2** and looks like this:

//SUMS	JOB		1
//	EXEC	PGM=SORT	2
//STEPLIB	DD	DSN=SORT.RESI.DENCE,DISP=SH	3
//SYSOUT	DD	SYSOUT=A	4
//SORTIN	DD	DSN=chiefs30.votes,	5
//		UNIT=2400-3,VOL=SER=887766,	
//		DISP=(OLD,KEEP)	
//SORTOUT	DD	DSN=termsums,	6
//		UNIT=2400-3,VOL=SER=554433.	
//	DD	DISP=(NEW,KEEP)	
//SORTWK01	DD	UNIT=SYSDA,SPACE=(CYL,20)	7
//SYSIN	DD	*	8
	SORT	FIELDS=(40,3,CH,A,45,2,CH,A)	9
	SUM	FIELDS=(24,3,CH)	10
/	*		11

To create the **sortcl** equivalent, enter the following on the command line:

```
mvs2scl mvs2 mvs2.scl
```

The resultant SORTCL specification file, **mvs2.scl**, is:

```
/INFILE=chiefs30.votes
  /FIELD=(Field_0,POSITION=40,SIZE=3, EBCDIC)
  /FIELD=(Field_1,POSITION=45,SIZE=2, EBCDIC)
  /FIELD=(Field_2,POSITION=24,SIZE=3, EBCDIC)
/CONDITION=(cond_0,TEST=(Field_0 OR Field_1))
  /KEY=(field_0,ASCENDING)
  /KEY=(field_1,ASCENDING)
/OUTFILE=termsums
  /FIELD=(field_2,POSITION=24,SIZE=3, EBCDIC)
  /FIELD=(field_0,POSITION=40,SIZE=3, EBCDIC)
  /FIELD=(field_1,POSITION=45,SIZE=2, EBCDIC)
  /SUM field_2 from field_2 BREAK cond_0
```

To run the sort, enter:

```
sortcl /spec=mvs2.scl
```

The key fields and summary field are output to **termsums**:

206	D-R	MA
612	D-R	VA
211	DEM	NC
214	DEM	NH
446	DEM	NJ
208	DEM	NY
215	DEM	PA
207	DEM	SC
228	DEM	VA
202	FED	MA
201	FED	VA
216	REP	KY
217	REP	NC
226	REP	NY
***	REP	OH
451	REP	VT
213	WHG	NY
631	WHG	VA

After the sort, each party-state pair is accompanied by its term number total. For Republicans from Ohio, the sum exceeds the number of bytes allotted for the result; the sum is shown as asterisks (***) to indicate that the size of field `Field_2` needs to be larger so that the data will fit in the field.



It should be noted that the `char` data type translated to `ebcdic`. If your data has already been translated to `ascii` or if you want to convert to `ascii` in the output then you should make adjustments to your translated script.

vse2scl

1 PURPOSE

For sort/merge users migrating from an IBM mainframe (VSE) environment to open systems, IRI has produced a VSE sort parameter translation program. The **vse2scl** program scans the VSE Job Control Language sort statements to produce text that can be executed immediately by the **sortcl** program.

SORTCL, or Sort Control Language, is a high-level language for defining and manipulating data. **sortcl** is the command line program that uses the SORTCL syntax and the `cosort()` library routine. The SORTCL language supports multiple, differently-formatted input and output files, and structured reports.

2 USAGE

2.1 Execution

You must have permission to access both the **vse2scl** and **sortcl** programs. These are standalone programs, i.e., they are not interdependent and do not require any other modules. They can be found in the **\$COSORT_HOME/bin** directory on Unix or in **\install_dir\bin** on Windows.

To execute **vse2scl**, enter:

```
vse2scl vse_script
```

where *vse_script* represents your VSE Job Control script file. The output is sent to standard out. If you wish the output to go to a file, then use the following:

```
vse2scl vse_script sortcl_script
```

where *sortcl_script* is a generic name for your new SORTCL script file.

If your input source is standard input, you can execute **vse2scl** as follows:

```
cat vse_script | vse2scl
```

where the output is written to standard out. To write output to a file, such as *out.scl*:

```
cat vse_script | vse2scl stdin out.scl
```

where *cat vse_script*, in these cases, represents the command to provide the input that will function as **stdin**.

Command-Line Options

The following options can follow the word **vse2scl** when used on the command line:

- a Uses ASCII as the character type when the incoming data type is CH.
- e Uses EBCDIC as the character type (the default) when the incoming data type is CH.

Environment Variable Options

There are also environment variables you can set to alter the behavior of **vse2scl** whenever which will more closely imitate the JCL behavior on the mainframe:

COSORT_ZD_FILL=0

When you set **COSORT_ZD_FILL** to 0, all **ZONED_DECIMAL** fields will contain the attribute **FILL='0'** in the output section of the translated scripts. When **sortcl** is executed, this will ensure that the leading zeros are produced for the **ZONED_DECIMAL** values (see *FILL* on page 106).

VSE2SCL_MAPALL=yes

When you set **VSE2SCL_MAPALL** to **yes**, any fields that are not specifically referenced in the original job script are derived for the purposes of translation to the **sortcl** script. **vse2scl** derives these fields (including positions and sizes) to complete the full record layout based on the specified record length in the JCL script. This way, the entire record is mapped to the output when **sortcl** is run.

Resultant **.scl** files can be examined, modified, and executed. Execution takes the form:

```
sortcl /SPECIFICATIONS=sortcl_script [sortcl_options]
```

where *sortcl_options* are optional controls to **sortcl** (see the *sortcl PROGRAM* chapter on page 37 for details).

3 CONVERSION RULES

Any statements in the VSE script that cannot be translated will be ignored, such as references to physical locations (i.e., disk storage and cylinders). Also, most VSE operating system references are not applicable in the Unix or Windows environment.

Fields are defined with symbolic names (field_0, field_1, etc.) that are associated with absolute field positions and lengths. Subsequent references to these same fields refer to those given names. It is suggested that you then change the field names to names that suggest the meaning of the data.

Data Types Not Recognized

- Leading Overpunch Sign.

Data Types Recognized

- Integers, reals, floating (Alphanumeric)
- Character, Natural (ASCII)
- Character, Signed
- Character, Unsigned (EBCDIC)
- Integer, Short Signed
- Integer, Short Unsigned
- Integer, Natural Signed
- Integer, Natural Unsigned
- Integer, Long Signed
- Integer, Long Unsigned
- Float, Single Precision
- Float, Double Precision
- Zoned Decimal, Trailing Overpunch Sign.

Micro Focus COBOL

- COMP, Signed
- COMP, Unsigned
- COMP-3, Unsigned (Packed Decimal)
- COMP-5, Signed
- COMP-5, Unsigned
- COMP-X
- DISP, Unsigned
- DISP, Sign leading
- DISP, Sign leading separate
- DISP, Sign trailing
- DISP, Sign trailing separate.

Ryan McFarland COBOL

- COMP, Signed
- COMP, Unsigned
- COMP-1
- COMP-3, Signed
- COMP-3, Unsigned (Packed Decimal)
- COMP-6
- DISP, Unsigned
- DISP, Sign leading
- DISP, Sign leading separate
- DISP, Sign trailing
- DISP, Sign trailing separate.

4 EXAMPLES

4.1 Example #1 - Two-Key Sort

Using the following VSE sort script, **vse1.app**:

```
// EXEC      SORT, SIZE=256K
              SORT  FIELDS=(3,17,A,20,11,D), FORMAT=CH
              RECORD TYPE=F, LENGTH=(98)
              INPFIL VSAM
              OUTFIL ESDS, REUSE
              OPTION ROUTE=LST
              END
/*
```

here is the command to convert it for use with SORTCL and place the conversion in the file **vse1.scl**:

```
vse2scl vse1.app vse1.scl
```

vse1.scl is as follows:

```
/INFILE=($DD_SORTIN)
/LENGTH=98
/FIELD=(field_0, POSITION=3, SIZE=17, EBCDIC)
/FIELD=(field_1, POSITION=20, SIZE=11, EBCDIC)
/KEY=(field_0, ASCENDING)
/KEY=(field_1, DESCENDING)
/OUTFILE=($DD_SORTOUT)
```

The resultant SORTCL specification file is used for sorting if you enter:

```
sortcl /spec=vse1.scl
```

4.2 Example #2 - Omit Condition

The following is a VSE sort script, **vse2.app**:

```
SORT FIELDS=(1,4,A,18,4,A,5,13,A,22,12,A,140,7,A,63,10,A),FORMAT=CH
OMIT COND=(1,2,EQ,C'15',AND,400,1,EQ,C'C'),FORMAT=CH
RECORD TYPE=F,LENGTH=(650)
INPFIL VSAM
OUTFIL ESDS,REUSE
OPTION ROUTE=LST
END
/*
```

To create the **sortcl** equivalent, enter the following on the command line:

```
vse2scl vse2.app vse2.scl
```

The resultant SORTCL specification file, **vse2.scl**, is:

```
/INFILE=($DD_SORTIN)
/LENGTH=650
/FIELD=(field_0, POSITION=1, SIZE=4, EBCDIC)
/FIELD=(field_1, POSITION=18, SIZE=4, EBCDIC)
/FIELD=(field_2, POSITION=5, SIZE=13, EBCDIC)
/FIELD=(field_3, POSITION=22, SIZE=12, EBCDIC)
/FIELD=(field_4, POSITION=140, SIZE=7, EBCDIC)
/FIELD=(field_5, POSITION=63, SIZE=10, EBCDIC)
/FIELD=(field_6, POSITION=1, SIZE=2, EBCDIC)
/FIELD=(field_7, POSITION=400, SIZE=1, EBCDIC)
/CONDITION=(cond_0, TEST=(field_6 == "15" AND
                          field_7 == "C"))
/OMIT=(CONDITION=cond_0)
/KEY=(field_0, ASCENDING)
/KEY=(field_1, ASCENDING)
/KEY=(field_2, ASCENDING)
/KEY=(field_3, ASCENDING)
/KEY=(field_4, ASCENDING)
/KEY=(field_5, ASCENDING)
/OUTFILE=($DD_SORTOUT)
```

To run the sort, enter:

```
sortcl /spec=vse2.scl
```



It should be noted that the `char` data type translated to `ebcdic`. If your data has already been translated to `ascii` or if you wish to convert to `ascii` in the output then you should make adjustments to your translated script.

COBOL TOOLS

1 OVERVIEW



The names of **CoSORT** library files have changed for version 9, but **CoSORT** is backward-compatible if you have upgraded from an older version and are using the older names. It is recommended that you modify the names to those introduced in version 9 for organizational purposes and for consistency with future releases. The version 9 file names (listed before their previous names) are as follows:

```
libcosort.a cosort.a
libsortcl.a sortcl.a
libsortcl_vsam.a sortcl_vsam.a
libsortcl_idx.a sortcl_idx.a
libsortcl_vis.a sortcl_vis.a
libmfcosortwb.a mfcosortwb.a
libmfcosortse.a mfcosortse.a
libacucosort.a acucosort.a
libnat2cs.a nat2cs.a
```

CoSort offers several techniques for accelerating sort/merge operations within COBOL programs. There are four options available:

- replace MF COBOL Workbench, Server Express, and Net Express sort routines without reprogramming
- execute consecutively with a standalone **CoSort** program
- execute concurrently with a standalone **CoSort** program
- send/receive records directly to/from **CoSort** using the COBOL API.

The first option accelerates sort/merge operations already running in Micro Focus COBOL source programs. By linking the **mfcosort** library on the command line, the SMP `cosort()` coroutine sort engine is called (instead of COBOL's internal sort routine) to produce new executables or build a new RTS. Performance improves significantly, and the need for temporary work space is often halved.

Linking instructions are provided in the following sections:

- *WORKBENCH 4.1 FOR UNIX* on page 455
- *WORKBENCH 4.0 AND NET EXPRESS 2.X FOR WINDOWS* on page 457
- *MF COBOL SERVER EXPRESS SORT REPLACEMENT FOR UNIX* on page 459
- *MF COBOL NET EXPRESS SORT REPLACEMENT FOR WINDOWS* on page 461
- *ACUCOBOL SORT REPLACEMENT FOR UNIX* on page 462.

The second option is to make a system call to a stand-alone **CoSort** utility like **sortcl** from the COBOL application. Sort control can be invoked to perform simple to complex sort and report operations. Your program waits for the job to complete before continuing. See the *sort PROGRAM* chapter on page 481, the *sorti PROGRAM* chapter on page 507, and the *sortcl PROGRAM* chapter on page 37.

The third option has a **CoSort** utility program running in parallel with the application. The COBOL program runs (in the foreground) concurrently with a sort/merge/join/aggregate/report job (in the background). Pipes (|) can be used to coordinate data transfer between the program and the sort. *Calling sortcl from within a COBOL Program* on page 464 shows how to invoke the **sortcl** report generator to read input file(s) and produce output file(s) or formatted reports. See also the *sortcl PROGRAM* chapter on page 37.

The fourth option is to use the Application Program Interface (API). The program calls `cosort()` directly. The ability to call the parallel coroutine sort eliminates the I/O of intermediate files. As in traditional COBOL procedure-to-procedure logic, records stream one-by-one or in blocks between the application and the sorter. Sorted records are available for processing as they are generated. See the *API (Application Program Interface)* chapter on page 541 for this information.

2 WORKBENCH 4.1 FOR UNIX

The following files are required for linking Micro Focus (MF) COBOL Workbench 4.1¹ compilers and **CoSort**:

- **libmfcosortwb.a**
- **libcosort.a**

To include **CoSort** modules into the standalone executable program **myprog**, use:

```
cob -x -C CALLSORT myprog.cbl \  
-L$COSORT_HOME/lib -lmfcosortwb -lcosort
```

To include **CoSort** modules into the new runtime system **csrts**, use:

```
cob -xe "" -o csrts \  
-L$COSORT_HOME/lib -lmfcosortwb -lcosort
```

Note that intermediate files (**.int** and **.gnt**) must be compiled with the CALLSORT compiler directive prior to execution with **csrts**:

```
cob -C CALLSORT myprog.cbl
```

1. **CoSort**'s MF COBOL sort replacement is not currently available for Server Express.



Certain MF COBOL file types such as RELATIVE, INDEXED, and LINE SEQUENTIAL are not native to **CoSort**. To sort these types of files, you should instruct mfcosort to use the MF COBOL File Handler for I/O operations. This is achieved by means of the environment variable COSORT_USEEXTFH. To enable MF File Handler operation, use

```
COSORT_USEEXTFH=1;export COSORT_USEEXTFH
```

To disable it, use:

```
unset COSORT_USEEXTFH
```

Some performance degradation may be incurred due to the MF File Handler's overhead. LINE SEQUENTIAL files that do not contain characters in the range 00-1F do not require that **CoSort** use the MF File Handler.

3 WORKBENCH 4.0 AND NET EXPRESS 2.X FOR WINDOWS

The following files are required for linking Micro Focus (MF) COBOL Workbench 4.0 (or Net Express 2.x) and **CoSort**:

libmfcosortwb.obj	MF COBOL sort --> CoSort interface routines
libc.lib	Auxiliary module required by CoSort .
oldnames.lib	Auxiliary module required by CoSort .
user32.lib	Auxiliary module required by CoSort .
gdi32.lib	Auxiliary module required by CoSort .
advapi32.lib	Auxiliary module required by CoSort .
libcosort_static.lib	CoSort sort engine (can be found in the \$COSORT_HOME/lib directory).

To prepare COBOL source files for use with **CoSort**, compile with the **CALLSORT** directive and the value "CS_EXTSM". For example, to produce the object **myprog.obj** from **myprog.cbl**, use the command:

```
cobol myprog.cbl CALLSORT"CS_EXTSM"
```

To include **CoSort** modules into the executable program **myprog**, use:

```
cbllink myprog.obj libmfcosortwb.obj cosortlib_rec_count.lib  
libc.lib oldnames.lib user32.lib gdi32.lib advapi32.lib
```

To link **CoSort** modules into the standalone executable program **myprog** (not RTS-dependent), use:

```
cbllink -t myprog.obj libmfcosortwb.obj  
cosortlib_rec_count.lib libc.lib oldnames.lib  
user32.lib gdi32.lib advapi32.lib
```



If you are using **mfcosort.obj** for Workbench 4.0, you must set the **cosortrc** parameter `USE_RECORDCOUNT_API` before linking COBOL programs with **mfcosort** (see *Resource Control Settings* on page 647).

Certain MF COBOL file types such as `RELATIVE`, `INDEXED`, and `LINE SEQUENTIAL` are not native to **CoSort**. To sort these types of files, you should instruct **mfcosort** to use the MF COBOL File Handler for I/O operations. This is achieved by means of the environment variable `COSORT_USEEXTFH`.

To enable MF File Handler operation, set `COSORT_USEEXTFH=1`

To disable it, set `COSORT_USEEXTFH=`

Some performance degradation may be incurred due to the MF File Handler's overhead. `LINE SEQUENTIAL` files that do not contain characters in the range 00-1F do not require that **CoSort** use the MF File Handler.

4 MF COBOL SERVER EXPRESS SORT REPLACEMENT FOR UNIX

The following files are required for linking Micro Focus COBOL Server Express 2.2, 4.0, or 5.0 and **CoSort**:

- **libmfcosortse.a**
- **libcosort.a**

To include **CoSort** modules into the standalone executable program **myprog**, use either of the following methods:

- `cob32 -x -C CALLSORT myprog.cbl -L. -lmfcosortse -lcosort -lpthread -lrt`
- `cob32 -x myprog.cbl -L. -lmfcosortse -lcosort -lpthread -lrt`

To include the **COSORT** modules during runtime, use:

```
cob32 -xe "" -o csrts -L. -lmfcosortse -lcosort -lpthread -lrt
```

The program needs to be compiled with the **CALLSORT** compiler directive prior to execution:

```
cob32 -C CALLSORT myprog.cbl
```



The above linking method is for 32-bit COBOL compilers. If you have a 64-bit compiler, replace all `cob32` instances above to `cob64`.

Certain MF COBOL file types such as **RELATIVE**, **INDEXED**, and **LINE SEQUENTIAL** are not native to **CoSort**. To sort these types of files, you should instruct **mfcosort** to use the MF COBOL File Handler for I/O operations. This is achieved by means of the environment variable `COSORT_USEEXTFH`. To enable MF File Handler operation, use

```
COSORT_USEEXTFH=1;export COSORT_USEEXTFH
```

To disable it, use:

```
unset COSORT_USEEXTFH
```

Some performance degradation may be incurred due to the MF File Handler's overhead. **LINE SEQUENTIAL** files that do not contain characters in the range 00-1F do not require that **CoSort** use the MF File Handler.

4.1 JCL Sort Replacement

Improve sort performance by calling the CoSORT sort engine to replace sort calls by the Micro Focus Enterprise Server, Sever Express, or Net Express. The Micro Focus server environment must be set to specify the location of the dynamic library containing the CoSort JCL sort replacement.

The CoSort JCL sort replacement library is **csjextsm.so** on Unix and Linux, and **csjextsm.dll** on Windows. This library also requires **libcosort.so** or **libcosort.dll**, provided by IRI. Your system administrator must place these libraries in a directory listed in the library search path of the system.

The environment variable **MFJEXTSM** must be set to **csjextsm(.so or .dll)** in the Micro Focus server environment. The CoSort external sort replacement can be called for all sort calls or for individually selected sort calls. Use the CoSort resource control settings for performance tuning.

Set the sort call **ALIAS** in the server **ALIAS** table to **EXTJSORT** to use the CoSort sort replacement for all sort calls. For example, set the **ALIAS** for **SORT** to **EXTJSORT**.

To use CoSort for individually selected calls, do not set the sort **ALIAS**. Instead, modify the JCL script to replace individual calls with **EXTJSORT**.



For sort operations not supported by CoSort, the request will be passed back to and performed by the default sort call.

The log file contains information showing when the external function was used.

5 MF COBOL NET EXPRESS SORT REPLACEMENT FOR WINDOWS

The following files are required for linking Micro Focus COBOL Net Express 3.0, 4.0 and 5.0 for **CoSort**:

libmfcosortne.obj	MF Cobol - CoSort interface routines.
libcosort_static.lib	CoSort sort engine (in the \$COSORT_HOME/lib directory).
libcmt.lib	Auxiliary module required by CoSort .
oldnames.lib	Auxiliary module required by CoSort .
Kernel32.lib	Auxiliary module required by CoSort .
advapi32.lib	Auxiliary module required by CoSort .
ws2_32.lib	Auxiliary module required by CoSort .

To sort COBOL data files with **CoSort**, compile the COBOL program with the **CALLSORT** compiler directive. For example, to compile **myprog.cbl**:

```
cobol myprog.cbl CALLSORT "CS_EXTSM"
```

To include the **CoSort** modules in the executable program, use:

```
cbllink myprog.obj libmfcosortne.obj libcosort_static.lib libcmt.lib  
oldnames.lib advapi32.lib ws2_32.lib kernel32.lib
```



Certain MF COBOL file types such as **RELATIVE**, **INDEXED**, and **LINE SEQUENTIAL** are not native to **CoSort**. To sort these types of files, you should instruct **mfcosort** to use the MF COBOL File Handler for I/O operations. This is achieved by means of the environment variable **COSORT_USEEXTFH**.

To enable MF File Handler operation, set **COSORT_USEEXTFH=1**

To disable it, set **COSORT_USEEXTFH=**

Some performance degradation may be incurred due to the MF File Handler's overhead. **LINE SEQUENTIAL** files that do not contain characters in the range 00-1F do not require that **CoSort** use the MF File Handler.

6 ACUCOBOL SORT REPLACEMENT FOR UNIX

You can use **CoSort** to replace the sort in ACUCOBOL-GT, versions 6.2 and higher. To perform the sort replacement, you must:

- 1) Go to the **/lib** directory under the ACUCOBOL installation directory.
- 2) Edit the Makefile as follows:
 - a) Find the **EXTSM** configuration and remove the comment characters from top set of three lines.
 - b) Insert comment characters for the second set of three lines in the **EXTSM** configuration.
 - c) Set the **EXTSM_LIB** variable to point to the desired **libacucosort.a** and **libcosort.a** files.
 - d) Insert the following linker options:

```
-lpthread -lrt -lposix4 -lm -ldl
```

- e) Save the Makefile.
- 3) Type **make**. This re-links the libraries to generate the new **runcbl**. You can then check the version/license details by typing

```
runcbl -vv
```

- 4) Copy **runcbl** into the **/bin** directory.
- 5) In order to start the runtime, you must start the acushare daemon by typing

```
acushare -start
```

- 6) You can now test the SORT routines: Compile a COBOL program that uses sorting by typing

```
ccbl program.cbl
```

- 7) Before you run the program, you will need to set the ACUCOBOL environment variable **USE_EXTSM** to 1. You can now run it using **runcbl program.acu**.



If you are using any external input or output files you must also set the environment variable **COSORT_USEEXTFH** to 1.

7 BATCH-MODE SORTS

The file **miami.dat** is used in the following examples. **miami.dat** has four ASCII fields: TITLE, PUBLISHER, QUANTITY, and PRICE. The COBOL program that uses **miami.dat** contains the following lines:

```

07  ENVIRONMENT DIVISION.
08  CONFIGURATION SECTION.
09  SOURCE-COMPUTER.  HP9000.
10  OBJECT-COMPUTER.  HP9000.
11  INPUT-OUTPUT SECTION.
12  FILE-CONTROL.
13      SELECT MIAMI-FILE
14          ASSIGN TO "MIAMI.DAT"
15          ORGANIZATION IS SEQUENTIAL
16          ACCESS IS SEQUENTIAL
17      .
18  DATA DIVISION.
19  FILE SECTION.
20  FD MIAMI-FILE
21      RECORD CONTAINS 35 CHARACTERS
22      LABEL RECORDS ARE OMITTED
23      .
24  01 MIAMI-RECORD.
25      05 TITLE          PIC X(15).
26      05 PUBLISHER      PIC X(13).
27      05 QUANTITY       PIC 9(3).
28      05 PRICE          PIC 99V99

```

The **miami.dat** data is as follows:

Table 11: Data for miami.dat

Title	Publisher	Quantity	Price
Reasoning For	Prentice-Hall	150	10.25
Murder Plots	Harper-Row	160	5.90
Still There	Dell	80	13.05
Pressure Cook	Harper-Row	228	9.95
Sending Your	Valley Kill	130	15.75
People Please	Valley Kill	75	11.50
Map Reader	Prentice-Hall	200	14.90

This file contains 35-byte, fixed-length records.

7.1 Using sortcl from the Prompt

From a command-line prompt, to sort **miami.dat** by title, you might use:

```
sortcl /infile=miami.dat /length=35
      /outfile=miami-by-title.dat
```

Notice that the record length parameter is required with COBOL data files because COBOL does not automatically insert a carriage return at the end of every record.

To sort **miami.dat** by the publisher's name, use:

```
sortcl /infile=miami.dat /length=35
      /key=' (position=16,size=13) ' /outfile=by-pub.dat
```

The single quotes are used here to protect the **sortcl** syntax from being interpreted by the shell in Unix. Windows users do not need the quotes.

In this case, *publisher* is a 13-character field which starts at the 16th column of every record. The results of the sort are put in a file named **by-pub.dat**. Refer to the *sortcl PROGRAM* chapter on page 37 to explore other **sortcl** features.

7.2 Calling sortcl from within a COBOL Program

7.2.1 Literal-String Passing

From within a COBOL program, the **CALL** statement is used to execute the **sortcl** program. We are showing a simple **sortcl** command line for brevity, but pre-existing script files are more desirable. Using the previous examples, equivalent COBOL procedure statements are:

```
CALL "system" USING "sortcl /infile=miami.dat
                    "/length=35 /outfile=by-title.dat"
```

and:

```
CALL "system" USING "sortcl /infile=miami.dat /length=35
                    "/key=' ' (position=16,size=13) ' '
                    "/outfile=miami-by-title.dat"
```

Note that the **USING** string is continuous in the last example, and contains two consecutive apostrophes to become a single apostrophe in the COBOL compiler.

U This COBOL statement calls the standard Unix system function. The parameter for this call is a string which is passed to the command line of a shell (**/bin/sh**) in a new process. ♦

W This COBOL statement calls the standard command function, **system**. The parameter for this call is a string which is passed to the command line in a new process. ♦

The COBOL program (or process) is suspended until the execution of the command is completed.

7.2.2 Concurrent Processing Extensions in Unix

You can place the sort/merge/report in the background while continuing to execute the COBOL program in the foreground. By adding the **&** symbol to the command, the concurrent process operates in parallel with the COBOL program.

For example, to sort **miami** by title in the background, the **CALL** statement is as follows:

```
CALL "system" USING "sortcl /infile=miami.dat /length=35
                    "/outfile=miami-by-title.dat &"
```

This calling convention is useful when sorting large files, and when the calling program does not depend on an immediate result of the sort.

7.2.3 Supported COBOL Data Types

Keys that are defined in COBOL data-types are supported in **sortcl**. The following example shows **miami.dat** with a different record layout, and the **sortcl** command sorting it on the now packed-decimal quantity field:

```
24 01 MIAMI-RECORD.
25    05 TITLE      PIC X(15) .
26    05 PUBLISHER  PIC X(13) .
27    05 QUANTITY   PIC 9(3)    USAGE COMP-3 .
28    05 PRICE      PIC 99V99   USAGE COMP-3 .
```

```
CALL "system" USING "sortcl /infile=miami.dat /length=33
                    "/key=' '(position=29,size=2,ASC,MF_CMP3)''
                    "/outfile=miami-by-qty.dat"
```



Record length and key size have changed since packed decimal fields take up less space than in the record layout of the previous example. See your COBOL compiler manual for formulas to calculate type sizes.

The third parameter to the key command is the direction of the sort, in this case, ASCENDING. The fourth parameter indicates the data type of the key. *Table 12* and *Table 13* on page 466 list other COBOL-related data types supported by **sortcl**:

Table 12: Micro Focus Data Types

sortcl Data Type	PIC	USAGE clause
MF_COMP or COMPUTATIONAL	signed	COMP
UMF_COMP	unsigned	COMP
MF_CMP3, PACKED, PACKED_DECIMAL	signed	COMP-3
UMF_CMP3	unsigned	COMP-3
MF_CMP5	signed	COMP-5
UMF_CMP5	unsigned	COMP-5
MF_CMPX	unsigned	COMP-X
MF_DISP	unsigned	DISPLAY
MF_DISPSTL	signed	DISPLAY SIGN LEADING
MF_DISPSTLS	signed	DISPLAY SIGN LEADING SEPARATE
MF_DISPST	signed	DISPLAY SIGN TRAILING
MF_DISPSTS	signed	DISPLAY SIGN TRAILING SEPARATE

Table 13: Ryan-McFarland Data Types

sortcl Data Type	PIC	USAGE clause
RM_COMP	signed	COMP
URM_COMP	unsigned	COMP
RM_CMP1	signed	COMP-1
RM_CMP3	signed	COMP-3
URM_CMP3	unsigned	COMP-3
RM_CMP6	unsigned	COMP-6
URM_DISP	unsigned	DISPLAY

Table 13: Ryan-McFarland Data Types

sortcl Data Type	PIC	USAGE clause
RM_DISPST	signed	DISPLAY SIGN LEADING
RM_DISPSTLS	signed	DISPLAY SIGN LEADING SEPARATE
RM_DISPST	signed	DISPLAY SIGN TRAILING
RM_DISPSTLS	signed	DISPLAY SIGN TRAILING SEPARATE

7.2.4 Micro Focus Line-Sequential Records

COBOL programs that remove trailing spaces from a record are identified by the ORGANIZATION IS LINE SEQUENTIAL clause in the SELECT statement:

```

7  ENVIRONMENT DIVISION.
8  CONFIGURATION SECTION.
9  SOURCE-COMPUTER.  HP9000.
10 OBJECT-COMPUTER.  HP9000.
11 INPUT-OUTPUT SECTION.
12 FILE-CONTROL.
13     SELECT MANY-SPACES-FILE
14         ASSIGN TO "many-spaces.dat"
15         ORGANIZATION IS LINE SEQUENTIAL
16         ACCESS IS SEQUENTIAL
17     .

```

The trailing spaces removed from the end of a record are automatically replaced by a line-feed character in COBOL. According to **sortcl**, this file type is handled by specifying a record length of 0:

```

CALL "system" USING  "sortcl /infile=line-seq.dat
                      "/length=0
                      "/outfile=line-seq-sorted.dat"

```

Zero is the default value for the record length in **sortcl**, and therefore need not be specified explicitly (as above). See the *sortcl PROGRAM* chapter on page 37 for more information on **sortcl** and supported data and file types.

7.2.5 Micro Focus Variable-Length Records

Data files written by Micro Focus COBOL programs that use the V or VARIABLE option in the RECORDING MODE are also supported by **sortcl**.

Provide **sortcl** with a `/PROCESS=MFVL` command to identify this type of file:

```
CALL "system" USING  "sortcl /infile=many-spaces.dat  
                     "/process=MFVL  
                     "/outfile=many-spaces-sorted.dat"
```

See the *sortcl PROGRAM* chapter on page 37 for more information on supported data and file types.

7.2.6 Using Copy-In Source Files

Copying-in the command string structure is more efficient when performing many similar sorts. By `COPYing-in` **sortclA.cpy** and **sortclB.cpy** into your COBOL program's working storage area, you can make multiple calls with similar parameters to **sortcl** with relative ease and efficiency (see *SORTCL INCLUDE FILES* on page 471). The following two examples illustrate these calls.

The following COBOL program sorts the same input file **miami.dat** into four files, each corresponding to a sort of the file by each field:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHANGING-KEY-SORT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP9000.
OBJECT-COMPUTER. HP9000.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY "sortclA.cpy" REPLACING SORTCL-NO-INFILES BY 1.
COPY "sortclB.cpy" REPLACING SORTCL-NO-KEYS BY 1.
PROCEDURE DIVISION.
* set up sort parameters in SORTCL-CMD items (copied-in)
* common parameters:
MOVE "miami.dat" TO SORTCL-INFILE(1).
MOVE 36 TO SORTCL-REC-LENGTH.
* sort 1 - sort by title
MOVE 1 TO SORTCL-KEY-START(1).
MOVE 15 TO SORTCL-KEY-SIZE(1).
MOVE "miami-by-title.dat" TO SORTCL-OUTFILE.
* execute the sortcl command
CALL "system" USING SORTCL-CMD.
* sort 2 - sort by publisher
MOVE 16 TO SORTCL-KEY-START(1).
MOVE 13 TO SORTCL-KEY-SIZE(1).
MOVE "miami-by-pub.dat" TO SORTCL-OUTFILE.
* execute
CALL "system" USING SORTCL-CMD.
* sort 3 - sort by quantity
MOVE 29 TO SORTCL-KEY-START(1).
MOVE 3 TO SORTCL-KEY-SIZE(1).
MOVE "miami-by-qty.dat" TO SORTCL-OUTFILE.
* execute
CALL "system" USING SORTCL-CMD.
* sort 4 - sort by price
MOVE 32 TO SORTCL-KEY-START(1).
MOVE 5 TO SORTCL-KEY-SIZE(1).
MOVE "miami-by-price.dat" TO SORTCL-OUTFILE
* execute
CALL "system" USING SORTCL-CMD.
STOP-RUN.

```

The following COBOL program merges two pre-sorted data files into one:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. MERGE-MIAMIS.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
SOURCE-COMPUTER. HP9000.  
OBJECT-COMPUTER. HP9000.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
COPY "sortclA.cpy" REPLACING SORTCL-NO-INFILES BY 2.  
COPY "sortclB.cpy" REPLACING SORTCL-NO-KEYS BY 1.  
PROCEDURE DIVISION.  
*  set up parameters  
  MOVE "merge" TO SORTCL-FUNCTION.  
  MOVE "miami-A.dat" TO SORTCL-INFILE(1).  
  MOVE "miami-B.dat" TO SORTCL-INFILE(2).  
  MOVE "miami-AB.dat" TO SORTCL-OUTFILE.  
  MOVE 36 TO SORTCL-REC-LENGTH.  
*  merge by publisher requires definition of a key  
  MOVE 16 TO SORTCL-KEY-START(1).  
  MOVE 13 TO SORTCL-KEY-SIZE(1).  
  DISPLAY "Executing " SORTCL-CMD.  
*  execute the sortcl command  
  CALL "system" USING SORTCL-CMD.  
  STOP-RUN.
```

8 SORTCL INCLUDE FILES

The following COBOL source files are inserted into COBOL applications via two COPY statements (one for each file).

Both COPY statements must provide REPLACING clauses, specifically:

- **sortclA.cpy** requires a value for the number of input files, SORTCL-NO-INFILES
- **sortclB.cpy** requires an integer replacement for SORTCL-NO-KEYS.

The names of the items that are declared within these files are now able to be modified by the application program. The application program can also dynamically adjust the parameters in order to perform multiple calls to the **sortcl** program.

These files are also included in the software distribution in the **examples/API** directory:

sortclA.cpy:

```
* SORTCL COBOL Header File.
01 SORTCL-CMD.
05 FILLER                PIC X(9)  VALUE "../sortcl".
05 FILLER                PIC X(2)  VALUE " /".
05 SORTCL-FUNCTION       PIC X(5)  VALUE "sort".
*      * either "sort" or "merge"
05 SORTCL-INFILE-AREA OCCURS SORTCL-NO-INFILES TIMES.
*                                * number of input files
10 FILLER                PIC X(9)  VALUE " /infile=".
10 SORTCL-INFILE         PIC X(40) VALUE SPACES.
*      * input file name
```

sortclB.cpy:

```

05 FILLER                      PIC X(9)  VALUE " /length=".
05 SORTCL-REC-LENGTH          PIC 9(5)  DISPLAY VALUE 0.
*      * input record length (0=variable,n=fixed)
05 SORTCL-KEY OCCURS SORTCL-NO-KEYS TIMES.
*      * number of keys
10 FILLER                      PIC X(12) VALUE " /key=' '(position=".
* 10 FILLER                      PIC X(1) VALUE X"27".
* 10 FILLER                      PIC X(5) VALUE "(position=".
10 SORTCL-KEY-START PIC 9(5)  DISPLAY VALUE 1.
*      * key start position in record
10 FILLER                      PIC X(6)  VALUE ",size=".
10 SORTCL-KEY-SIZE PIC 9(5)  DISPLAY VALUE 1.
*      * key size (length in characters)
10 FILLER                      PIC X(1)  VALUE ", ".
10 SORTCL-KEY-DIR PIC X(3)  VALUE "ASC".
*      * key direction (ASCending or DESCending)
10 FILLER                      PIC X(1)  VALUE ", ".
10 SORTCL-KEY-TYPE PIC X(10) VALUE "ASCII".
*      * key type, eg. ASCII, NUMERIC, MF-COMP
*      * see sortcl chapter for more
10 FILLER                      PIC X(2)  VALUE ")' '".
* 10 FILLER                      PIC X(1) VALUE X"27".
05 SORTCL-OUTFILE-AREA.
10 FILLER                      PIC X(10) VALUE " /outfile=".
10 SORTCL-OUTFILE PIC X(40) VALUE SPACES.
*      * output file name
05 SORTCL-BACKGROUND PIC X  VALUE SPACE.
*      * process control (space=CALLer waits,
*      * "&"=runs in background)

```


9 ACUCOBOL-GT VISION RECORDS

9.1 Enabling CoSort Support for Vision Files

CoSort's **sortcl** supports data files in ACUCOBOL-GT Vision format.

In order to process Vision files, you will need a valid Vision **.vlc** license file generated by Acucobol. You must rename this file to **sortcl.vlc**, and place it in the directory containing the **sortcl** executable.

The instructions below describe how to build a Vision-enabled **sortcl** executable.



To obtain the requisite Vision-enabled **CoSort** libraries for linking to Vision libraries, contact IRI or your IRI agent.



For compiling and linking on Windows systems, the following files are required:

- *install_dir\lib\libsortcl_vis.lib*
- *install_dir\lib\libcosort_static.lib*
- *install_dir\lib\sortcl_main.obj*.

You also need the Acucobol libraries **acme.lib** and **avisionx.lib**, where **x** is the Vision version supported by your Acucobol software. The Microsoft Visual C++ linker **link.exe** is needed to build the Vision-enabled **sortcl** executable.

For example, to build a **sortcl** with **avision5.lib**, use the following command:

```
link /OUT:sortcl.exe libsortcl_vis.lib sortcl_main.obj  
libcosort_static.lib acme.lib avision5.lib kernel32.lib ws2_32.lib
```

:

U For compiling and linking on UNIX and Linux systems, the following files are required:

- **\$COSORT_HOME/etc/Makefile.vis**
- **\$COSORT_HOME/lib/libsortcl_vis.a**
- **\$COSORT_HOME/lib/libcosort.a**

Make sure that the libraries you requested from IRI are the same bit architecture as your Acucobol. You will also need access to a C compiler and the `make` command.

The following are the steps you must follow to edit **Makefile.vis**, depending on your operating system, Acucobol bit architecture, and your C compiler options:

- 1) Make sure that **\$COSORT_HOME** is set in your environment, and that it points to the **CoSort** install directory.
- 2) Set **ACULIB_DIR** to the Acucobol **/lib** directory.
- 3) Set **CSLIB_DIR** to the **CoSort /lib** directory. By default, this should be **\$COSORT_HOME/lib**.
- 4) Make sure you have the C compiler `gcc` or `cc` in your path. Otherwise, you must set the **PATH** to point to the compiler.
- 5) Set **LD_LIBS** appropriately for your operating system. The options given should work for most operating systems.
- 6) Save the changes to **Makefile.vis** and exit.

You must then perform the following steps to build a Vision-enabled **sortcl**:

- 1) Rename **Makefile.vis** to **Makefile**.
- 2) Run the command `make`.

If you encounter any errors while building the Vision-enabled **sortcl** executable, email support@iri.com.

9.2 Usage

The syntax for specifying a file as Vision (either as an input source or output target) is as follows:

```
/PROCESS=VISION[, version]
```

where *version* can be any of the following:

- VERSION2** Vision version 2 files.
- VERSION3** Vision version 3 files.
- VERSION4** Vision version 4 files.
- VERSION5** Vision version 5 file (the default).

For example, you can indicate that your file is a Vision version 3 file using the following syntax in a **sortcl** job script:

```
/PROCESS=VISION, VERSION3
```

For an example of using **sortcl**'s ACUCOBOL-GT Vision file support, consider the following Vision data file:

```
Bush|George H.W. |1989-1993|REP|TX
Adams|John|1797-1801|FED|MA
Buchanan|James|1857-1861|DEM|PA
Arthur|Chester A. |1881-1885|REP|VT
Adams|John Quincy|1825-1829|D-R|MA
Carter|James E. |1977-1981|DEM|GA
```



The readable format shown above was produced using the extract option of Acucorp's **vutil32.exe** utility (**vutil** on Unix). Viewing data in this form helps you to determine the `/FIELD` layouts shown below.

You must use a `/PROCESS=VISION` command in **sortcl** to specify Vision-format files:

```
/INFILE=chiefs_sep
/PROCESS=VISION
/INCOLLECT=6
/FIELD= (fnme, POSITION=1, SEPARATOR='|')
/FIELD= (name, POSITION=2, SEPARATOR='|')
/FIELD= (year, POSITION=3, SEPARATOR='|')
/FIELD= (part, POSITION=4, SEPARATOR='|')
/FIELD= (stat, POSITION=5, SEPARATOR='|')
/SORT
/KEY=fnme
/OUTFILE=chiefs_sep.out
/PROCESS=RECORD
/FIELD= (fnme, POSITION=1, SEPARATOR='|')
/FIELD= (name, POSITION=2, SEPARATOR='|')
/FIELD= (year, POSITION=3, SEPARATOR='|')
/FIELD= (part, POSITION=4, SEPARATOR='|')
/FIELD= (stat, POSITION=5, SEPARATOR='|')
```

chiefs_sep.out contains:

```
Adams|John Quincy|1825-1829|D-R|MA
Adams|John|1797-1801|FED|MA
Arthur|Chester A.|1881-1885|REP|VT
Buchanan|James|1857-1861|DEM|PA
Bush|George H.W.|1989-1993|REP|TX
Carter|James E.|1977-1981|DEM|GA
```

Conversely, **sortcl** can convert from a non-Vision file format to a Vision file. To do this, declare the process type of your input file beneath the `/INFILE` statement (`/PROCESS=RECORD` is the default), and use `/PROCESS=VISION` beneath the `/OUTFILE` statement. After data has been converted to Vision format, you can use the utility **vutil32.exe** (or **vutil**) to view, or display information about, the converted file.

For example, consider the following script which converts an input file with the format MFVL_SMALL (Micro Focus variable-length with two-byte binary short header records) into Vision file format:

```
/INFILE=chiefsmfvl.dat
/PROCESS=MFVL_SMALL
  /FIELD=(f1, POSITION=1, SIZE=27)
  /FIELD=(f2, POSITION=28, SIZE=12)
  /FIELD=(f3, POSITION=40, SIZE=5)
  /FIELD=(f4, POSITION=45, SIZE=2)
/SORT
  /KEY=(f1, desc)
/OUTFILE=chiefsvision.dat
/PROCESS=VISION
  /FIELD=(f1, POSITION=1, SIZE=27)
  /FIELD=(f2, POSITION=28, SIZE=12)
  /FIELD=(f3, POSITION=40, SIZE=5)
  /FIELD=(f4, POSITION=45, SIZE=2)
```

See *DATA SOURCE AND TARGET FORMATS (/PROCESS)* on page 53 for more information on supported file formats. See *DATA TYPES* on page 611 for a list of supported data types.

10 MICRO FOCUS COBOL ISAM RECORDS

10.1 Enabling CoSort Support for MF ISAM Records

CoSort's sortcl supports the collation and conversion of the proprietary index file formats of Micro Focus COBOL (MF ISAM).

Process MFISAM uses third party libraries requiring licensing that is provided by IRI when the COBOL MFISAM and VISION file support option is purchased. The Micro Focus License Manager is installed with CoSort and must be running to validate the use of the third party libraries.(indows) installed on your system.

10.2 Usage

The syntax for specifying a file as MF ISAM (either as an input source or output target) is as follows:

```
/PROCESS=MFISAM[, type]
```

where *type* can be any of the following Micro Focus indexed file types:

IDX8	IDXFORMAT"8" - The default.
IDX4	IDXFORMAT"4"
IDX3	IDXFORMAT"3"
C-ISAM	C-Indexed Sequential Access Method
ESDS	Entry Sequenced Data Set
XESDS	Extended ESDS
VISION	ACUCOBOL-GT Vision

If you indicate the following in the input section of a job script:

```
/PROCESS=MFISAM, VISION
```

CoSort will process an MF ISAM indexed file in the Vision file format. If you do not specify a type, the indexed file is assumed to be of the default type (IDX8).

When index file types are formatted in separate data and index files, both files must exist in the same directory, and the job script must specify the data file name.

For example, on input, *filename.dat* and *filename.idx*, the job script specifies the *filename.dat*. On output, *filename.idx* will be created along with *filename.dat*.

Runtime error messages will display Micro Focus error codes.

sort PROGRAM

1 PURPOSE

The Unix system `sort` command was designed to sort and merge variable-length ASCII character records (text files and piped data). It usually requires that all input records are in memory. While the native `sort` command can accomplish tasks quickly for a moderate number of records, it causes thrashing, or fails, at a large number.

CoSort provides a drop-in replacement for the Unix `/bin/sort` that has the same user interface, but does not produce the adverse effects of high data volumes. The **cosort/bin/sort** program, one of several standalone interfaces within the **CoSort** package, is AT&T SVID Issue 2 and X/Open Issue 3-conformant. For instructive purposes, we will refer to the Unix `sort` and the **CoSort** `sort` interchangeably (because the syntax is identical) as **sort**.



Windows users are provided with the executable **unixsort.exe** which uses the same syntax. ♦



CoSort's sort program is useful if you want to continue using the same interface and commands featured in a Unix `/bin/sort`, but to make use of all **CoSort** data transformation and reporting features, it is recommended that you use the **sortcl** interface (see the *sortcl PROGRAM* chapter on page 37) for all jobs, from simple to complex.

The **sort** program is limited relative to **CoSort's** other standalone utilities. Elaborate report generation and file-format manipulation are not possible, as would be with **sortcl**, for example. This chapter is intended to improve your understanding of the Unix **sort** facility, and make **CoSort's sort** a more viable, convenient tool for limited file-to-file operations, or to accelerate existing jobs.

Note that in addition to replacing the Unix `sort`, **CoSort** also replaces the Micro Focus COBOL (see the *COBOL TOOLS* chapter on page 453) `sort`, the SAS System `sort`, and the Software AG Natural `sorts` on Unix. The latter two `sort` replacements, also known as **PROCSORT** and **SORT** respectively, are described later in this chapter. For a list of third-party `sort` tools for which **CoSort** provides direct replacements and/or free conversion tools or services, see *PRODUCT OVERVIEW* on page 27.

2 PERFORMANCE CONSIDERATIONS

In high-volume jobs, **CoSort's sort** program can run up to several times faster than the Unix system's sort program. It does not fail after the input size exceeds available memory. In addition, **CoSort's sort** has no limits on line length, and record lengths can be as high as 65,535 bytes each. Furthermore, if the final byte of an input file is not a line-feed, **CoSort's sort** will silently supply it. The following table compares the two sort programs running a 3-key sort on a Sun E450 with 4GB of RAM, running under Solaris release 2.9:

Table 14: Benchmarks

File Size / # of Records	CoSort Sort	Unix sort
~402MB (variable-length) 2,273,818 records	2m:15s	1.5 hrs
~3.53GB (variable-length) 19,945,961 records	31 minutes	12 hours

Files have 100 bytes/record

If you have an investment in sort scripts, you should consider replacing the UNIX sort with **CoSort's bin/sort**. It is recommended that you set system or individual users' profiles to include the `PATH` to **CoSort's bin/sort** using the appropriate search order. For system administrators who want to give it precedence, use:

```
PATH="$ {COSORT_HOME}": "$ {PATH}"
```

In this manner, the native **bin/sort** program can still be used by specifying its full path.

If you want only **sortcl** and other utilities to be found in the path, and not to supercede existing commands, use:

```
PATH="$ {PATH}": "$ {COSORT_HOME}"
```

The description that follows is adapted, and paraphrased for easier reference, from the standard UNIX *man* page named `sort (CMD)`, `sort (C)`, `sort (1)`, etc. For additional information, UNIX users can access the system man page on **/bin/sort** by executing:

```
man sort
```

or

```
more $COSORT_HOME/cat/sort.1
```

Windows users can use:

```
type C:\Progra~1\IRI\CoSort91\docs\unixsort.txt | more
```

or open the same file with your preferred text editor.



If you are running batch scripts, make sure that your existing batch files do not reference the full path to the native **bin/sort** executable if you want them to utilize **CoSort's bin/sort**.

3 EXECUTION

The **sort** and **unixsort** programs are run from the command line. The syntax respectively on Unix and Windows systems is:

```
sort [input source] [input source] [-key flags] \
    [-output flags] [-performance flags]

unixsort [input source] [input source] [-key flags] \
    [-output flags] [-performance flags]
```



You can adjust the `MONITOR_LEVEL` resource control setting which determines the degree of verbosity for reporting the progress of a job (see *Resource Control Settings* on page 647). For details on all **CoSort** error messages, see *ERROR and RUNTIME MESSAGES* on page 660.

The various command-line flags, or options, are described below.

Note that in examples (where applicable), there will first be a line for Unix users, and then a line for Windows users, as above.

3.1 Input Sources

Any set of characters not preceded by - is assumed to be an input file. As many input files as desired are allowed. If no input files are provided, input is taken from **stdin**. We will use the file **chiefs10h** below, which has the name, term, party, and state from the first 10 U.S. Presidents, for our sample input data.

Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Adams, John Quincy	1825-1829	D-R	MA
Jackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA

3.1.1 Single Input File

From the command line, the easiest way to sort one input file is to name it following the word `sort`, for example:

```
sort chiefs10h
unixsort chiefs10h
```

The results of the sort go to standard out (that is, to the screen if no pipe is specified). Since no sort key was specified, the entire input line (record) serves as the key, starting with the first character in each line. The ordering (collating) sequence is determined by the machine's locale. See `locale(M)` for information on your (Unix) machine's setting.

The output sent to the to the screen is sorted from left to right:

Adams, John	1797-1801	FED	MA
Adams, John Quincy	1825-1829	D-R	MA
Harrison, William Henry	1841-1841	WHG	VA
Jackson, Andrew	1829-1837	DEM	SC
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Tyler, John	1841-1845	WHG	VA
Van Buren, Martin	1837-1841	DEM	NY
Washington, George	1789-1797	FED	VA



If your input file is empty, you can use a resource control setting to determine the disposition of the output file (see *ON_EMPTY_INPUT* option on page 652). By default, when an input file is empty, an output file is created assuming zero-value input data.

3.1.2 Multiple Input Files

More than one input file can be sorted. Without any command flags (see *FEATURES* on page 487), a simple example of a two-file input is:

```
sort chiefs10h chiefs10h
unixsort chiefs10h chiefs10h
```

In this case, the files would be sorted and appear merged together, with each President appearing twice.

3.1.3 Standard Input

Pipes can also be used to provide input for the **sort** program. For example, a file can be sorted from **stdin**:

```
cat chiefs10h | sort
type chiefs10h | unixsort
```

Piping records in from **stdin** to the **sort** program is particularly useful for manipulating system information, for example:

```
who | sort +4n
dir | unixsort +4n
```

sorts the contents of the current logins in time order, oldest first. You can modify the output by following the `sort` command with key options like that above. *FEATURES* on page 487 describes the options available with **CoSort**'s **sort** program.

4 **FEATURES**

For the remainder of this chapter, **CoSort**'s `sort` command for Unix will be used in examples. Windows users should replace `sort` with `unixsort`.

Three options are available in the **sort** program that alter its default behavior of sorting: check only, merge only, and unique.

Check Only

- c Checks the input file, before sorting begins, to see if it is already in the order that other **sort** program flags (if any) indicate. If the file is already in order, the **sort** program exits with a 0 (no message). If it is out of order, the program displays a message indicating the first line of disorder, i.e., an exit status of 1. For example:

```
sort -c chiefs10h
unixsort -c chiefs10h
```

would check to see whether **chiefs10h** was already sorted from left to right according to the locale-defined collating sequence of the machine. Since the file is in date order, rather than in alphabetic order by last name, we get the message:

```
sort: disorder: Adams, John                1797-1801    FED    MA
```

President Adams, who is on the second line of the input file, is the first record out of order because it does not precede Washington in the file.

Merge Only

- m Specifies a merge, not a sort, when the input file(s) is already in sorted order. Merging two or more sorted input files would work without specifying a merge, but when the files are already sorted, the merge option is faster.

If the files are not in sorted order, and merge is specified, the files are merged into sorted order, which is a much slower process than sorting them together instead.

Unique

- u Specifies unique keys, which eliminates all but one output record with equal keys. That is, there will be not records with duplicate keys. For demonstration, if you input the **chiefs10h** file twice, the output will be the same as sorting only one file:

```
sort -u chiefs10h chiefs10h
```

4.1 Ordering Options

The following options override the default sort rules. When options appear independent of key field specifications (as discussed in *Defining Key Fields* on page 490), then they will be applied globally to all keys. When the options are attached to a specific key, they override global options for that key.

4.1.1 Dictionary Order

- d Dictionary, or phone directory, order, as defined by the machine's locale setting, is the new collating sequence. This means that only letters, digits, spaces, and tabs (blanks) are compared. No other ASCII characters are significant.

4.1.2 Fold Lowercase into Uppercase

- f Case folds lowercase letters into uppercase for output so that lowercase letters do not take precedence over uppercase letters, or vice versa. The machine's locale setting determines how the conversion is governed.

4.1.3 Ignore

- i Ignores non-printable characters, as defined by the locale setting, in non-numeric sorts. Normally, these are characters outside the ASCII range 040-0176 octal (inclusive).

4.1.4 Numeric Sort

- n Sorts a numeric string by arithmetic value, even when optional blanks, minus signs, 0 or more digits, or decimal points are used. The -b option is also implied here (see *Blanks* on page 489).

4.1.5 Reverse Order

- r Reverses the order of the comparisons, so that the collating sequence is descending and greater key values appear earlier in the output, rather than the default (which is ascending order). In a descending alphabetic sort, for example, B comes before A, 2 comes before 1, and so on. For example:

```
sort -r chiefs10h
```

produces the following output:

Washington, George	1789-1797	FED	VA
Van Buren, Martin	1837-1841	DEM	NY
Tyler, John	1841-1845	WHG	VA
Monroe, James	1817-1825	D-R	VA
Madison, James	1809-1817	D-R	VA
Jefferson, Thomas	1801-1809	D-R	VA
Jackson, Andrew	1829-1837	DEM	SC
Harrison, William Henry	1841-1841	WHG	VA
Adams, John Quincy	1825-1829	D-R	MA
Adams, John	1797-1801	FED	MA

4.2 Field Separator Options

4.2.1 Blanks

- b Ignores leading blanks when determining the starting and ending positions of a restricted sort key. A restricted sort key is any key that is less than the entire line. If the -b option is applied before the first key field designation, then it is applied to all key fields.



By default, leading blanks are viewed by the **sort** and **unxsort** program as field separators. They include a single blank, a sequence of blanks, a tab, or a line-feed. All blanks in a sequence of blanks are considered part of the next field; e.g., all blanks at the beginning of a line are considered part of the first field. Field separators can be changed from blanks to other characters using the -t option (see *Field Separator Character* on page 490).

4.2.2 Field Separator Character

`-t char`

char is the separator character to delimit fields within a sort command.



Do not use a separator character that occurs within the fields.

4.3 Defining Key Fields

4.3.1 Field Specifiers

`+m[.n]` Marks the exact start position of a sort key. Counting starts at zero and includes the separator that precedes the field. To apply options to a specific key, the numerical position can be followed by any of the options (*bdfinr*).

A starting position of `+m.n` indicates the $n+1$ st character of the $m+1$ st field. For example:

```
sort +1.2 chiefs10h
```

indicates that the first key starts on the third character of the second field in the record, which for the first record is the *a* in *James*:

Monroe, James	1817-1825	D-R	VA
Madison, James	1809-1817	D-R	VA
Van Buren, Martin	1837-1841	DEM	NY
Washington, George	1789-1797	FED	VA
Jefferson, Thomas	1801-1809	D-R	VA
Harrison, William Henry	1841-1841	WHG	VA
Jackson, Andrew	1829-1837	DEM	SC
Adams, John	1797-1801	FED	MA
Tyler, John	1841-1845	WHG	VA
Adams, John Quincy	1825-1829	D-R	MA

If want the key to start with the first character of the second field, use the following:

```
sort +1 chiefs10h
```

The result will be:

Jackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Washington, George	1789-1797	FED	VA
Monroe, James	1817-1825	D-R	VA
Madison, James	1809-1817	D-R	VA
Adams, John	1797-1801	FED	MA
Tyler, John	1841-1845	WHG	VA
Adams, John Quincy	1825-1829	D-R	MA
Jefferson, Thomas	1801-1809	D-R	VA
Harrison, William Henry	1841-1841	WHG	VA

As you can see, the space is the delimiter character so the second field is sorted by the presidents' first names, with the exception of Van Buren, where Buren is the value for the second field.

`-m [.n]` Marks the exact ending position of a sort key. If no `-m.n` is given, the sort key beginning with `+m.n` above continues until the end of the line.

An ending position of `-m.n` indicates that the sort key finishes (the comparison stops) before the $n+1$ st character of the $m+1$ st field. For example:

```
sort +1.0 -1.2 chiefs10h
```

indicates that the sort key is only the first two characters of the second field, i.e., it starts on the first character and ends before the third character. Remember that the first character is the blank:

Jackson, Andrew	1829-1837	DEM	SC
Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Adams, John Quincy	1825-1829	D-R	MA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Tyler, John	1841-1845	WHG	VA
Van Buren, Martin	1837-1841	DEM	NY
Jefferson, Thomas	1801-1809	D-R	VA
Harrison, William Henry	1841-1841	WHG	VA

A missing `.n` indicates `.0`, the first character of the m th field. For example:

```
sort +3 -4 chiefs10h
```

indicates that the fourth field is the sort key, since it starts and ends on the fourth field (Party):

Adams, John Quincy	1825-1829	D-R	MA
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Van Buren, Martin	1837-1841	DEM	NY
Jackson, Andrew	1829-1837	DEM	SC
Adams, John	1797-1801	FED	MA
Washington, George	1789-1797	FED	VA
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA

When a numeric sort is specified, blanks are ignored and the file is sorted in number order. The following sorts the parties first and then the date in reverse numeric order:

```
sort +3 -4 +2nr chiefs10h
```

Following is the result:

Adams, John Quincy	1825-1829	D-R	MA
Monroe, James	1817-1825	D-R	VA
Madison, James	1809-1817	D-R	VA
Jefferson, Thomas	1801-1809	D-R	VA
Van Buren, Martin	1837-1841	DEM	NY
Jackson, Andrew	1829-1837	DEM	SC
Adams, John	1797-1801	FED	MA
Washington, George	1789-1797	FED	VA
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA

4.3.2 Alternate Field Specifiers

Solaris and other modern Unix sort commands use the `-k` flag to specify keys. **CoSort's** **sort** (and **unixsort**) programs support both syntax conventions. White space denotes the end of a field unless a field separator character is defined.

```
-k field_start [type] [,field_end [type]]
```

The key field begins at *field_start* and ends at *field_end* inclusive. *field_start* has the form *field_number*[*.first character*] and *field_end* has the form *field_number*[*.last character*]. Numbering begins at 1. To designate *type*, use any of the modifiers (*bdfinr*).

To sort **chiefs10h** starting with the second field use:

```
sort -k 2 chiefs10h
```

This will give the following output:

Jackson, Andrew	1829-1837	DEM	SC
Washington, George	1789-1797	FED	VA
Monroe, James	1817-1825	D-R	VA
Madison, James	1809-1817	D-R	VA
Adams, John	1797-1801	FED	MA
Tyler, John	1841-1845	WHG	VA
Adams, John Quincy	1825-1829	D-R	MA
Van Buren, Martin	1837-1841	DEM	NY
Jefferson, Thomas	1801-1809	D-R	VA
Harrison, William Henry	1841-1841	WHG	VA

To sort starting at the second character of the first field, use the following command:

```
sort -k 1.2 chiefs10h
```

This will result in the following output:

Jackson, Andrew	1829-1837	DEM	SC
Madison, James	1809-1817	D-R	VA
Van Buren, Martin	1837-1841	DEM	NY
Harrison, William Henry	1841-1841	WHG	VA
Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Adams, John Quincy	1825-1829	D-R	MA
Jefferson, Thomas	1801-1809	D-R	VA
Monroe, James	1817-1825	D-R	VA
Tyler, John	1841-1845	WHG	VA

To sort by the fourth and fifth fields, use the following:

```
sort -k 4,5 chiefs10h
```

The output is as follows:

Adams, John Quincy	1825-1829	D-R	MA
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Van Buren, Martin	1837-1841	DEM	NY
Jackson, Andrew	1829-1837	DEM	SC
Adams, John	1797-1801	FED	MA
Washington, George	1789-1797	FED	VA
Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA

To sort by the fourth and fifth fields in reverse order, use the following command:

```
sort -k 4,5r chiefs10h
```

to get the following output:

Harrison, William Henry	1841-1841	WHG	VA
Tyler, John	1841-1845	WHG	VA
Washington, George	1789-1797	FED	VA
Adams, John	1797-1801	FED	MA
Jackson, Andrew	1829-1837	DEM	SC
Van Buren, Martin	1837-1841	DEM	NY
Jefferson, Thomas	1801-1809	D-R	VA
Madison, James	1809-1817	D-R	VA
Monroe, James	1817-1825	D-R	VA
Adams, John Quincy	1825-1829	D-R	MA



All of the key options/flags apply globally to all of the sort keys described in the command statement, unless the flags are attached to a specific sort key. When attached to a restricted sort key, the option(s) overrides all global ordering options that may have been previously specified.

4.4 Output Flags

4.4.1 Output File

`-o file`

Instead of standard output, sorted results are saved to a file that is named immediately following the `-o` flag. For example:

```
sort 15chiefs -o 15chiefs.out
```



You should only give the output file the same name as the input file when you want to replace (overwrite) the input file.

4.4.2 Standard Output

Output from the **sort** program is automatically sent to **stdout**, the console. It can also be redirected to other devices by the use of pipes on the command line. This saves the step of writing to a file and then sending the file to the device. For example:

```
ls -l | sort | lp  
dir | unixsort | prn
```

sends a sorted directory list to the printer.

4.5 Performance Flags

4.5.1 Temporary Directory Assignment

`-T tmpdir`

To allocate additional workspace for overflow, name a usable directory to hold temporary files. There must be a space between `-T` and `tmpdir`.

By default, the **sort** and **unixsort** program attempts to store temporary files in **/tmp** and **/usr/tmp**.

If `-T tmpdir` (the name of the alternative directory) is specified, then **tmpdir** and **/tmp** are tried.

If `-T tmpdir` (the name of the alternative directory) is specified, temporary files will be written to `tmpdir` and **/tmp** (if **/tmp** was previously created).

The `-T tmpdir` option overrides the **cosortrc** environment variable setting (or Windows Registry setting), **WORK_AREA**, which may have already been set by your system administrator to direct overflow workspace.



You must have space in, and permission to write into, your selected `tmpdir`. Contact your system administrator for more information when sort volume exceeds the limits of your designated `tmpdir`.



WARNING!

Solaris users should note that `/tmp` is generally reserved for system swap space. Either reassign the overflow directory with this command or check the **WORK_AREA** value(s) in the applicable **cosortrc** file.

For more information on overflow directories, see *WORK_AREA path* on page 648.

4.5.2 Memory Allocation Technique

- ykmem IRI's **sort** and **unixsort** programs do not recognize this option, which limits the amount of main memory used for sorting. The **cosortrc** file (or Windows Registry) contains a `MEMORY_MAX` parameter to do this (see *PERFORMANCE TUNING* on page 639 for complete information).

4.5.3 I/O Buffer Setting

- zrecsz CoSort's **sort** and **unixsort** programs do not recognize this option which sets the size of a buffer used in the merge phase. Again, the `COSORT_TUNER` variable allows the system administrator to control the size of the I/O blocks (read/write buffer), and the number of records in memory, the RAM limit, and overflow directories.

4.5.4 Version Display

- v Displays the current **CoSort** release level, internal source code tag, serial number, and any additional license information, for example:

```
sort with CoSort V 9.5.1 S060705A-1508 #11027.9516 6 CPUs
© 1978-2011 Innovative Routines International Non-expiring
```


USING COSORT WITH THE SAS SYSTEM

1 PURPOSE

CoSort can replace the SAS System 7-8 sort function on Unix. This section of the documentation provides instructions for linking **CoSort** to improve native sort performance.

2 EXECUTION

To use **CoSort** with the SAS System, complete the following steps:

1. Be sure the shared **CoSort** library, **libcosort.so**, or other similarly named dynamically linked library routine is installed and licensed in the **\$COSORT_HOME/lib** directory on your system.
2. Make **libcosort.so** available to the SAS System by allowing the instructions in the section *MAKING COSORT AVAILABLE*.
3. Submit an options statement in a SAS session to specify **libcosort.so** in the section “Using CoSort in a SAS session”.

3 MAKING COSORT AVAILABLE

This section describes the system-specific instructions for making the **CoSort** routine available to the SAS System.



If you are running SAS programs through a batch facility, make sure the environment variables described below are set correctly.

3.1 For AIX

Use either of the following two methods.

- Create symbolic links to **libcosort** from one of the directories searched by default, such as **/usr/lib**, as shown in the following example:

```
ln -s /usr/local/cosort/lib/libcosort.so /usr/lib
```

- Set the environment variable **\$LIBPATH** to the directory containing **CoSort**. For example,

*** Using Bourne Shell**

```
LIBPATH=/usr/local/cosort/lib:$LIBPATH export LIBPATH
```

*** Using C Shell**

```
setenv LIBPATH /usr/local/cosort/lib:$LIBPATH
```

3.2 For Compaq Tru64 Unix, Intel ABI, IRIX, and Solaris

Use either of the following methods:

- Create symbolic links to the host sort libraries from one of the directories searched by default, such as **/usr/lib** as shown in the following example:

```
ln -s /usr/local/cosort/lib/libcosort.so /usr/lib
```

- Set the environment variable **\$LD_LIBRARY_PATH** to the directory containing the host sort library. For example,

*** Using Bourne Shell**

```
LD_LIBRARY_PATH=/usr/local/cosort/lib:$LD_LIBRARY_PATH  
export LD_LIBRARY_PATH
```

*** Using C Shell**

```
setenv LD_LIBRARY_PATH /usr/local/cosort/  
lib:$LD_LIBRARY_PATH
```

3.3 For HP-UX

Use either of the following two methods:

- Create symbolic links to the host sort libraries from one of the directories searched by default, such as **/usr/lib**, as shown in the following example:

```
ln -s /usr/local/cosort/lib/libcosort.sl /usr/lib
```

- Set the environment variable **\$SHLIB_PATH** to the directory containing the host sort library. For example,

*** Using Bourne Shell**

```
SHLIB_PATH=/usr/local/cosort/lib:$SHLIB_PATH  
export SHLIB_PATH
```

*** Using C Shell**

```
setenv SHLIB_PATH /usr/local/cosort/lib:$SHLIB_PATH
```

4 OPTION STATEMENTS



The options statements throughout this section specify the syntax to submit to the SAS System. You can also specify these options as command line options and options in the **sasv8.cfg** file. Refer to SAS Companion for Unix Environments for more information on setting options.

Use the **SORTNAME** option to tell the SAS System which host sort routine should be used. Submit this option statement in a SAS session:

```
OPTIONS SORTNAME=COSORT;
```

Once the host sort routine is available, use the **SORTPGM=HOST** or **SORTPGM=BEST** options statements to tell the SAS System when to use the host sort routine. Submit one of the following options statements in a SAS session:

- `OPTIONS SORTPGM=HOST;`
tells the SAS System to always use the host sort routine made available.
- `OPTIONS SORTPGM=BEST;`
tells the SAS System to choose the best sorting method in a given situation, the SAS System sort or **CoSort**.

There are two options that define how the SAS System chooses the *best* sort algorithm. The following examples use the syntax of an options statement that needs to be submitted to the SAS System:

```
-sortcut <n>,
```

where *n* specifies a number of observations.

- `OPTIONS SORTPGM=BEST SORTCUT=500;`
`-sortcut` tells the SAS System to choose the host sort routine if the number of observations is greater than the number you specify, and to use the SAS System sort if the number of observations is equal to or less than the number specified.

```
-sortcutp size[kKmM],
```

where *size* specifies a file size in either kilobytes or megabytes.

- `OPTIONS SORTPGM=BEST SORTCUTP=40M;`
`-sortcutp` tells the SAS System to choose the host sort routine if the size of the data being sorted exceeds the size you specify, and to use the SAS System sort if the size of the data is equal to or smaller than the size you specify.

If these options are not defined or these options are set to zero, the SAS System chooses the SAS System sort routine. If you specify both options and either condition is met, the SAS System chooses **CoSort**.

You can change the work directory used for temporary sort files by using the option `sortdev dir`, where *dir* is the directory in which you want the temporary files to be created. For example, submit the following statement if you want the temporary files to be created in `/tmp`:

```
OPTIONS SORTPGM=BEST SORTCUT=500 sortdev="/tmp";
```

You can specify the host sort option `sortanom t` to print timing and resource information to the SAS log after each phase of a sort. The following is an example of this option:

```
OPTIONS SORTPGM=HOST SORTANOM=t;
```

You can specify the host sort option `sortanom v` to print to the SAS log the arguments parsed to the sort, which can be useful for tuning or debugging:

```
OPTIONS SORTPGM=HOST SORTANOM=v;
```

You can attempt to increase your sort performance by increasing the values of the `sortsize` and `memsize` SAS options. However, make sure that `sortsize` is at least 4M less than `memsize`. You can see other SAS performance statistics in the SAS log using the `FULLSTIMER` option:

```
OPTIONS FULLSTIMER;
```

USING COSORT WITH SOFTWARE AG NATURAL

1 PURPOSE

There are two ways that you can improve high-volume data processing performance in the Natural environment using **CoSort**:

- Call **sortcl** directly to make use of **CoSort**'s full range of data manipulation features with your Natural work files. See *DIRECT CALLS TO SORTCL*.
- Use the **CoSort** drop-in sort replacement to invoke **CoSort**'s high-performance sorting engine rather than the default sort in Natural (see *USING THE COSORT SORT REPLACEMENT* on page 504 for full implementation instructions).

2 DIRECT CALLS TO SORTCL

CoSort's general-purpose data transformation and reporting program, **sortcl**, can be called from Natural directly with:

```
CALL shcmd  
$COSORT_HOME/bin/sortcl /spec=filename.scl
```

where the `/INFILE(s)` into, and `/OUTFILE(s)` from **sortcl** can be Natural work files. This allows you greater flexibility for off-loading high-volume file manipulation and reformatting jobs to **sortcl**. See the *sortcl PROGRAM* chapter on page 37 for complete details on using **sortcl** and creating **sortcl** job scripts (**.scl** files).



If you are using the call method above for sorting only, you do not need to install the **CoSort** drop-in replacement as described in the following section.

The following section provides instructions for linking **CoSort** to improve Natural native sort performance.

3 USING THE COSORT SORT REPLACEMENT

CoSort can replace the Software AG Natural 4GL sort function, Version 2.2 through 6.2 on Unix. To use the drop-in sort replacement, you must first install it as follows:

- 1) Confirm that the file **libnat2cs.a** has been installed with **CoSort** (in the **\$COSORT_HOME/lib** directory). **sortcl** must be licensed on your system.
- 2) Copy the makefile **\$COSORT_HOME/src/Makefile.nat2cs** into your (equivalent) **~sag/nat/vxxx/bin/build** directory, where **xxx** is the Natural version number such as 4.1.2. If you have an incompatible version of Natural, advise your IRI agent.
- 3) Install the replacement with these commands:

```
cd ~sag/nat/v412/bin/build
mv Makefile Makefile.orig
cp Makefile.nat2cs Makefile
```

- 4) Edit **Makefile** to uncomment the **LIB_COSORT** entry to be applicable to your operating system.
- 5) Link with this command:

```
make natural cosort=yes
```

- 6) Run with these commands:

```
setenv PATH $PATH:$COSORT_HOME/bin
natural [...]
```



You can enable debugging messages for sorts by setting the environment variable **NAT2SCL_DEBUG=1**.

COSORT LOAD ACCELERATOR FOR DB2 (CLA4DB2)

1 PURPOSE

CoSort provides a load accelerator for IBM DB2 UDB 6.x, 7.x, and 8.x.

The **CoSort** Load Accelerator for DB2 (**CLA4DB2**) is a drop-in replacement for the sort within the DB2 sort utility. In one sort-load step, you can double load performance when a pre-defined index is required. Once you enable **CLA4DB2** to work on your system, you can continue to use the same DB2 commands, and the **CLA4DB2** utility will be invoked automatically.

2 ENABLING CLA4DB2 ON YOUR SYSTEM

CLA4DB2 is found in **\$COSORT_HOME/lib** upon installation of **COSORT**.

However, you must first enable your system to use it successfully. These steps are operating-system-dependent:



- 1) Set the environment variable **\$COSORT_HOME** to **/usr/local/cosort**.
- 2) Make sure that the files **cosort.lic** and **cosortrc** are located in **\$COSORT_HOME/etc**, that is, the default location at installation.
- 3) If you have a 32-bit version of **libcla4db2.so**, copy the 32-bit **libcosort.so** to **/usr/lib**. If you have a 64-bit **libcla4db2.so**, copy the respective 64-bit **libcosort.so** to **/usr/lib/64**.

- 4) To activate **CLA4DB2**, type:

```
db2set DB2SORT = shared_object_path/libcla4db2.so
```

where *shared_object_path* is the path where **libcla4db2.so** is located.

- 5) You must stop and restart the database instance for which you just ran **db2set**. You can now use DB2 as normal, and **CLA4DB2** is automatically utilized for all loads. ◆

W

- 1) To activate **CLA4DB2**, type:

```
db2set DB2SORT = shared_object_path/libcla4db2.dll
```

where *shared_object_path* is the path where **libcla4db2.dll** is located.

- 2) You must stop and restart the database instance for which you just ran `db2set`. You can now use DB2 as normal, and **CLA4DB2** is automatically utilized for all loads.

sorti PROGRAM

1 PURPOSE

CoSort's sort interactive (**sorti**) program provides easy command-line and batch access to the `cosort()` engine for basic sort and merge operations.

In an on-screen session, you will be responding to a series of prompts. At each prompt, a *help* facility explains the query and/or provides a sample response. Each response is analyzed for obvious errors and prompts you to make an appropriate response, or to overrule the objection. Sort/merge output can be directed to both a file and the terminal, so you can verify that the specifications are working as intended.



CoSort's **sorti** program is useful only for simple sorting and merging jobs, and is provided for users with little or no scripting experience who would prefer to specify their job requirements by responding to prompts. However, to make use of all **CoSort** data transformation and reporting features, it is recommended that you use the **sortcl** interface (see the *sortcl PROGRAM* chapter on page 37) for all jobs, from simple to complex.

sorti specifications can be saved for later use in a batch script. In standalone batch operations, sorts and merges are executed with **sorti** using existing sort/merge specifications. Batch users do not have to know the data formats, keys, or any of the details of sorting and merging.

The **sorti** program is standalone because no other programs are required; however, you can customize **sorti** to meet specific user requirements. In the areas of input record selection, key comparisons, and record output handling, you can supply a customizing procedure in any programming language to replace the standard operations (see *CUSTOM INPUT, OUTPUT AND COMPARE PROCEDURES* on page 556 for more information).

For sort/merge operations requiring selection, matching, aggregation, calculation, data conversion, or reformatting, see the *sortcl PROGRAM* chapter on page 37. To move existing **sorti** batch specifications to **sortcl** job scripts, see the *sorti2scl* chapter on page 427.

2 AN INTERACTIVE SESSION

An interactive session with **sorti** begins on the command line by entering:

```
sorti
```

2.1 Start-up

When **sorti** begins, the terminal displays, for example:

```
CoSort Sort Interactive Version 9.5.1 D91080301-1500
Copyright 2011 Innovative Routines International, Inc.
S/N 11027.9518 1 CPUs Non-expiring
Eastern Daylight Time 11:54:08 AM Friday, Aug 26 2011
```

```
Directory: C:\IRI\CoSORT95\bin
BlockSize: 2097152 bytes
MaxMemory: 268435456 bytes
WorkAreas: .\
```

```
Responses: <cr> takes offered default
           'stop' restarts the Action
           'help' explains each query
           $variable defined in shell
           # starts comment for batch
           !command for shell command
```

```
Action:  SORT  MERGE  DISPLAY  GUIDE (default)  END:
```

This banner shows the version of **CoSort** and its internal source code tracking number, plus your hardware configuration, company name, serial number, licensed number of CPUs, expiration date (if applicable), and runtime (performance) settings.

Performance parameters are set in the resource control file (**cosortrc** on Unix) or the Windows registry. This can be a file named by you with the environment variable **COSORT_TUNER**, or it can be the default file **cosortrc** or registry settings which **sorti** will use at execution time (see *PERFORMANCE TUNING* on page 639).

The tuning parameters shown at the start-up display are:

Directory	the directory in which sorti is being executed.
# Threads	the number of CPUs (or sort processes/threads) to be used for the sort.
BlockSize	the size, in bytes, of the I/O read/write buffers.
MaxMemory	the limit, in bytes, of the RAM to be used by each sort process.
WorkAreas	the list of directories for overflow in large sort/merge operations, and the maximum number of bytes assigned in each (overflow) directory.

Responses

Responses to prompts are user entries, an optional comment, and the [Enter] key. Except for file names, uppercase and lowercase letters are interchangeable and shortened forms are understood. For example, a sort can be specified as: *Sort*, *SOR*, *sO*, or *s*. Some additional response options are:

`<cr>` Hitting the [Enter] or [Return] key elects the default option offered with most prompts. For the default, enter either [Enter] or:

`white space # comments [Enter]`

If a comment with the default is desired, some white space is necessary before the #. Otherwise, **sorti** will accept the comments, but will repeat the prompt.

`stop` On each query, if the user responds `stop`, **sorti** will cancel the current sequence of responses and return to the `Action` query.

`help` The help response produces both an explanation of the current query and examples of reasonable responses.

`$variable` You can respond to prompts with a shell variable. As in the Unix command, enter a \$ immediately followed by an alphanumeric string. If the variable has been defined, it is used.

If the variable is not defined, **sorti** cannot execute. However, **sorti** will allow you to reference an undefined variable provided it does not affect your subsequent entries. For example, you can use an undefined variable for input and output file names. However, undefined variables can not be specified for the action, number of input files, keys, etc.

`#comments` Comments in the specifications provide information for batch execution in the future. Because the specifications and comments are text, the saved file can be edited with a text editor to amend, enhance,

or remove all comments.

User identification is placed at the head of the file to identify the licensee and time of day. Every prompt and response is given a succinct explanation by **sorti**. You can add your own full lines of comments and you can overwrite **sorti**'s short comments.

If you begin your response with the #, you can enter a full line of text. The current prompt will be repeated and you can enter as many lines as necessary.

If you respond to the query and follow it with white space and #, that comment will overwrite the one made by **sorti**.

! command This invokes the standard shell capability so you can investigate and control your environment while **sorti** runs.

This manual also discusses the use of standalone **sorti** for batch operations. That program is initiated with:

```
sorti specifications file
```

In an interactive **sorti** session, the **!** capability allows you to perform a sort or merge from existing specifications by entering:

```
!sorti specifications file
```

which performs the specifications without having to stop and restart **sorti**. This and the ability to:

```
DISPLAY specifications file
```

give the interactive user firm control of specification development.

**WARNING!**

Because `help` and `stop` have special meanings, files with these names should include their directory name. For example, to see a local file called **help**, enter:

```
DISPLAY ./help
```

2.2 A Sample Sort

To get an idea of how you can interact with **sorti**, a simple example of a sort is provided. The file **newchiefs** will be used as our input file. **newchiefs** is a shortened form of **chiefs**, which is delivered with your **CoSort** package and should be found in the **\$COSORT_HOME/examples/sorti** directory.



To create **newchiefs** on a Unix machine, enter:

```
tail -17 $COSORT_HOME/examples/sorti/chiefs>newchiefs
```

Now, type `sorti` to begin the **sorti** program. Examine the file by entering:

```
display newchiefs
```

For information on the `display` command, see *DISPLAY* on page 531. ♦

The following should appear on your screen:

```

=====1=====2=====3=====4=====5
Taft, William H.      1909-1913  REP  OH
Wilson, Woodrow      1913-1921  DEM  VA
Harding, Warren G.   1921-1923  REP  OH
Coolidge, Calvin     1923-1929  REP  VT
Hoover, Herbert C.   1929-1933  REP  IA
Roosevelt, Franklin D. 1933-1945  DEM  NY
Truman, Harry S.     1945-1953  DEM  MI
Eisenhower, Dwight D. 1953-1961  REP  TX
Kennedy, John F.     1961-1963  DEM  MA
Johnson, Lyndon B.  1963-1969  DEM  TX
Nixon, Richard M.    1969-1973  REP  CA
Ford, Gerald R.      1973-1977  REP  NE
Carter, James E.     1977-1981  DEM  GA
Reagan, Ronald W.    1981-1989  REP  IL
Bush, George H.W.    1989-1993  REP  TX
Clinton, William J.  1993-2001  DEM  AR
Bush, George W.      2001-        REP  TX
=====1=====2=====3=====4=====5

```

Name
Term
Party
State
line-feed

An invisible line-feed exists after each record, so the record length is 47 (see *Record Length: 0 (variable (default)) / fixed:* on page 517). The fields are arranged as shown in Table 15.

Table 15: Fields—newchiefs

Field	Start Pos.	Length	Type
Name	1	27	ASCII
Term	28	9	ASCII
Party	40	3	ASCII
State	45	2	ASCII

As you perform this example, you can refer to the corresponding section in this chapter for more information.

Table 16: Query/Response—newchiefs

Query	Response	Comment	Section
Action:	sort	initializes a sort	<i>section 3</i>
Record Length:	0 or <cr>	sets the record length as fixed or variable	<i>section 3.1</i>
Number of input files:	1 or <cr>	sorts one file only	<i>section 3.2</i>
Input File #x:	newchiefs	names the file to be sorted	<i>section 3.3</i>
Keys Unique:	2	two keys will be specified, Party and Name	<i>section 3.4</i>
Key 1 Direction:	ascending or <cr>	indicates low→high (A→Z) sorting of Party	<i>section 3.5</i>
Location:	fixed or <cr>	Party is found at a fixed location	<i>section 3.6</i>
Start Position:	40	indicates byte position where Party begins	<i>section 3.7</i>
Field size:	3	length of Party field	<i>section 3.8</i>
Format:	alpha or <cr>	Party consists of text characters	<i>section 3.9</i>
Type:	ASCII or <cr>	party is ASCII characters	<i>section 3.10</i>
Alignment:	none or <cr>	Party is already aligned	<i>section 3.11</i>
Case fold:	no or <cr>	Party is already capitalized	<i>section 3.12</i>
Key 2 Direction:	asc or <cr>	indicates low→high (A→Z) sorting of Name	<i>section 3.5</i>
Location:	fixed or <cr>	Name is found at a fixed location	<i>section 3.6</i>
Start position:	1 or <cr>	indicates byte position where Name starts	<i>section 3.7</i>

Table 16: Query/Response—newchiefs (cont.)

Query	Response	Comment	Section
Field size:	27	the length of Name key	<i>section 3.8</i>
Format:	alpha or <cr>	Name consists of text characters	<i>section 3.9</i>
Type:	ASCII or <cr>	Name is ASCII characters	<i>section 3.10</i>
Alignment:	none or <cr>	Name is already aligned	<i>section 3.11</i>
Case fold letters:	no or <cr>	Name is already capitalized properly (first letter only)	<i>section 3.12</i>
Output:	both	sends the output to your terminal and creates a file	<i>section 3.13</i>
Output file name:	demrep	names the output file	<i>section 3.14</i>
EXECUTE...:	yes or <cr>	watch your screen for output	<i>section 3.15</i>
Save specifications for reruns:	yes	allows you to create a file with the above specifications	<i>section 3.17</i>
Specifications filename:	new.spc	you have created a specification file with this name	<i>section 3.18</i>

The following is the output from the example shown in *Table 16* on page 513:

```

=====1=====2=====3=====4=====5=====6===== 15:56:44

CoSORT Version 8.2.1 D8040728-0921 (c) 1978-2004 IRI, Inc. www.cosort.com
EDT 03:55:56 PM Thursday, July 29 2004. #04117.8286 Monitor Level 1
<00:00:00.00> event (66): cosort() process begins
Carter, James E. 1977-1981 DEM GA
Clinton, William J. 1993-2001 DEM AR
Johnson, Lyndon B. 1963-1969 DEM TX
Kennedy, John F. 1961-1963 DEM MA
Roosevelt, Franklin D. 1933-1945 DEM NY
Truman, Harry S. 1945-1953 DEM MI
Wilson, Woodrow 1913-1921 DEM VA
Bush, George H.W. 1989-1993 REP TX
Bush, George W. 2001- REP TX
Coolidge, Calvin 1923-1929 REP VT
Eisenhower, Dwight D. 1953-1961 REP TX
Ford, Gerald R. 1973-1977 REP NE
Harding, Warren G. 1921-1923 REP OH
Hoover, Herbert C. 1929-1933 REP IA
Nixon, Richard M. 1969-1973 REP CA
Reagan, Ronald W. 1981-1989 REP IL
Taft, William H. 1909-1913 REP OH
<00:00:47.78> event (67): cosort() process ends
=====1=====2=====3=====4=====5 17 Records 15:56:44

Save specifications for reruns? YES or NO (default):

```


The political Party field took first priority in the sort (DEM's are displayed before REP's), followed by the last name of each president (DEM names are in alphabetical order, followed by REP names in order).



You can adjust the MONITOR_LEVEL resource control setting which determines the degree of verbosity for reporting the progress of a job (see *Resource Control Settings* on page 647). For details on all CoSort error messages, see *ERROR and RUNTIME MESSAGES* on page 660.

3 SORTING

After entering `sorti` from the command line, the banner appears and the cursor appears at the end of the `Action` query:



```
-----  
Action:  SORT  MERGE  DISPLAY  GUIDE <default>  END SORT█
```

Responding `SORT` to the `Action` query initiates the sequence of prompts for sorting.

With the exception of the name of the input file, you can elect the default option to all the queries. The result will be the same as if you had entered the following at the console:

```
sort filename
```

At any point, if you want information about the query, enter:

```
help
```

And if you need to restart, enter:

```
stop
```

3.1 Record Length: 0 (variable (default)) / fixed:

At the prompt `INPUT Record Length`, enter the length, in bytes, of the input records. 0 indicates variable-length records. When **sorti** reads variable-length records it scans across the record for the line-feed. An entry greater than 0 indicates that every record in the input has that fixed length.

The minimal length of a fixed-length record is 1 byte, and the maximum is 65,535. Do not use a comma to represent a value greater than 999. Each input record is assumed to be that fixed size. Control characters and every other bit pattern are ignored.

The following is a sample from the **newchiefs** file. Note the byte positions at the bottom:

Carter, James E.	1977-1981	DEM	GA
Reagan, Ronald W.	1981-1989	REP	IL
Bush, George H.W.	1989-1993	REP	TX
Clinton, William J.	1993-2001	DEM	AR
Bush, George W.	2001-	REP	TX
====,====1====,====2====,====3====,====4====,====5			

Although the final field (state) ends at position 46, there are actually 47 bytes in each record. There is an invisible carriage return after each record, which brings the fixed-record length of the input file **newchiefs** to 47. The record lengths are both fixed and variable at the same time.

The internal representation of variable-length records is different than the external representation. This is important to you if you will be handling variable-length records directly in any of your procedures. In that case, you should follow the internal format of variable-length records discussed in *RECORD TYPES* starting on page 633.



WARNING!

Do not specify records to be of variable length if the record contains binary data, unless you are certain that the binary characters will not cause a line feed.

3.2 Number of files: 0 (proc) / +- numb (1 default):

This value (without the sign) equals the number of files to be read. The next set of prompts will ask the name of each file indicated. Subsequent prompts will apply to all input files collectively.

Negative values and 0 are used when you want **sorti** to use your own input procedure to produce records. You will create `cs_input()` which will be called repeatedly to deliver one record at a time until the procedure indicates end-of-file.

If you have created this procedure, you can enter:

- 0 to use your routine exclusively
- n* to read *n* number of files and then use your routine.

3.3 Input file #x:

The #*x* above means that this query will be repeated as many times as was indicated in *section 3.2*. Here, you must enter the name of one file, as follows:

```
[path]filename[|skip bytes[|accept bytes]] [>e]
```

where *path* is specified with forward slashes, and *filename* is a single file name. The name `stdin` indicates that input will be from standard input. Otherwise, the file will be opened by **sorti** to ensure it will be available for input. If it is not available, the following message is given:

```
filename: Permission denied. Use Anyway?  
filename: No such file or directory. Use Anyway?
```

Answer yes (y) or no (n).

The reason for overriding a warning is to build specifications for future execution when the data will be available.

The following options are also available:

- | *skip bytes* for example, enter `chiefs|47` to skip 47 bytes from the front of the file before sorting.
- | *accept bytes* for example, enter `chiefs||470` to limit the total number of bytes for processing to 470.
- > *e* for example, enter `chiefs>x` is used when the character *x* is used to terminate each variable-length record instead of the standard line-feed character.



If your input file is empty, you can use a resource control setting to determine the disposition of the output file (see *ON_EMPTY_INPUT* option on page 652). By default, when an input file is empty, an output file is created assuming zero-value input data.

3.4 Keys [number] [stable] [unique / dups only] (default) 1:

Enter the total number of keys on which the file will be sorted. Pairs of records will be examined for every key specified in the order in which each key is specified. Key comparisons determine which of two records precedes the other.

Regardless of the number of keys specified, key comparisons will stop as soon as an inequality is discovered. However, if all the keys are examined and they are all equal, the relative arrangement of these records cannot be determined.

If (s) *stable* is specified, the order of any records with equal keys will be the order in which they appear in the input. Because stability requires processing overhead, only invoke this option when necessary.

If (u) *unique* is specified, only records with unique keys are output. This is the functional equivalent of removing all duplicate records.

If (d) *duplicates* is specified, only records having duplicate keys will be output.

An entry greater than 0 indicates that *cosort()* will use its built-in routines to analyze keys. You will next be prompted for details on each key. An entry of 0 indicates that you have provided your own procedure for key comparisons. The *cosort()* routine will call your procedure, *cs_compare()*, every time two records are to be compared.

3.5 Key x Direction: ASCENDING (default) or DESCENDING:

Key x indicates that the next series of responses will be repeated as many times as was indicated in *section 3.4*.

When the key is *ASCENDING*, key values are arranged from low to high. If the key is a character string, for example, the characters ABCD will appear earlier than BCDE. If they are external numeric keys, then -10 appears before 0 which appears before +5.

When key values are *DESCENDING*, the key values decrease from beginning to end; e.g., ZYXW comes before YXWV, and 1000E+2 comes before 99999.

3.6 Location: *FIXED* (default) / <separator char>:

Location describes how the key field is to be found in the record. All fields begin in byte position 1.

The *FIXED* response indicates that the key will be found at a fixed position in every record.

A single character entered here is a field delimiter. The key field is located in a variable position in the record just after the field delimiter character, e.g., a comma, colon, or semicolon.

To enter a special character, or to avoid ambiguity between a separator character and an abbreviation or control character, use a leading backslash (\).

General character entry rules are as follows:

Table 17: Separator Characters

Separator Character	Enter
any ASCII character but F f B b # ! \ or 'space'	<character>
F B \ # ! or 'space'	\<character>
f or b	\\f or \\b
formfeed backspace linefeed tab or null	\f \b \n \t or \0
any character	\octal value

See *Field Size: 65535 (maximum) 50 (default)*: on page 522 for an example of fixed and variable key locations.

3.7 Starting byte position: 1 (default):

If the response to the preceding `Location` was `FIXED`, you must enter the starting byte position of the key. Field 1 is always at byte position 1. The minimum start position is 1; the maximum is 65,535. Do not use a comma to represent values greater than 999.

If the response to `Location` was `BLANK`, or a field delimiter character, you must enter a field counter. An entry of 1 indicates that the field starts at byte position 1; an entry of 2 locates the field on or after the first appearance of the delimiter; an entry of 3 locates the field on or after the second appearance, and so on.

`BLANK`-separated fields include the spaces or tabs while *character*-separated fields do not include the *character*.

This record has two `BLANK`-delimited fields:

1	2	3	4	5	6	7	8	9
	<tab>		A	B			C	D

Field 1 2

This record has four `$`-delimited fields:

1	2	3	4	5	6	7	8	9
A	\$	B	C	\$	\$	D	E	F

Field 1 2 3 4



WARNING!

The starting position is different between `BLANK` and *character*-separated fields where the *character* is a space or tab.

3.8 Field Size: 65535 (maximum) 50 (default):

You must specify the length of the key field. The minimal length of a field is 1; the maximum is 65,535. The length you specify, plus the start position, can exceed the actual length of the record. If you do not specify a length, the maximum length that will be compared for the key field is 50 bytes.

Comparison stops at the logical field end.

In the following example, the key field is italicized:

Location	Start	Length	Record
Fixed	2	6	<i>Washington</i> , George
Variable	2	6	Washington, <i>George</i>

Note that the variable-position field includes the space.

3.9 Form: ALPHA (def), NUM, DATE, TIMESTAMP:

The key's data type is specified as a form-type pair. Your response to form describes the format group (see *OUTPUT: TERMINAL (default)*, *FILE*, *BOTH* or *PROGRAM*: on page 524 for the type choices). See *DATA TYPES* on page 611 for more detail on form-type pairs.

The ALPHA form includes signed characters, EBCDIC characters, DBCS/MBCS character sets, and three special types:

MONTH_DAY	where months and days of the week are compared logically rather than alphabetically; in ascending order: Jan < Feb < ... < Dec Sun < Mon < ... < Sat Upper-case characters are also permitted.
ASC_IN_EBC	ASCII characters are compared in EBCDIC sequence. An obvious difference is that in EBCDIC, digits > alpha characters.
ASC_IN_NATURAL	ASCII characters are sorted according to the collating sequence order of the current locale environment.

See *OUTPUT: TERMINAL (default)*, *FILE*, *BOTH* or *PROGRAM*: on page 524 for the various possibilities. Other types of the NUMERIC form are COBOL types and zoned decimals.

The DATE, TIME, and TIMESTAMP forms have types which are dependent on the American, European, Japanese, or ISO standards.

3.10 Type ASCII (default) or LIST to show options:

sorti recognizes a large number of data types, which are listed and described in *DATA TYPES* on page 611. It is important to declare the data type of a field so that **sorti** can interpret the data correctly for comparisons, conversions, and/or display purposes. If the data type is not given for an input or output field, the field is assumed to be ASCII. A key field is assumed to have the same data type that was declared in the input.

They will all, by default, display right-justified with a precision of two. MONEY and CURRENCY can be used interchangeably. They have MILL set to *on* and display the monetary symbol for the active *locale* at the beginning of the field.

The following are examples of how the ASCII-numeric data types display in the USA:

ASCII	Numeric	Currency	Currency w/Fill
3.2	3.20	\$3.20	\$*****3.20
150	150.00	\$150.00	\$*****150.00
3.25	3.25	\$3.25	\$*****3.25
9.4562	9.46	\$9.46	\$*****9.46
1023.45	1023.56	\$1,023.45	\$***1,023.45
29384.56	29384.56	\$29,384.56	\$**29,384.56

3.11 Alignment: NONE (default), LEFT, or RIGHT:

If an ALPHA format was specified in *section 3.9*, the character string in the key field can be shifted left or right before the comparison. A response of LEFT will trim white space on the left and add it to the right. A response of RIGHT will remove white space from the right and add it to the left. The following examples indicate the apparent shifting of the characters in a key field:

NONE	_ _ _ G e o r g e _ _
LEFT	G e o r g e _ _ _ _
RIGHT	_ _ _ _ _ G e o r g e

3.12 Case fold letters: NO (default) or YES:

If an ALPHA format was specified in *section 3.9*, the characters within the key field can be shifted to uppercase before they are compared. In this case, uppercase and lowercase letters will compare equally, for example, A will be equal to a, Z will come after y, etc. The response YES elects this option. The following example indicates a NO and YES response, respectively:

NONE	_ _ _ G e o r g e _ _
CASE	_ _ _ G E O R G E _ _

No other responses are required to complete the specification of an alphabetic key.



You will not be prompted for alignment or case information unless your key fields contain alphabetic ASCII characters.

3.13 OUTPUT: TERMINAL (default), FILE, BOTH or PROGRAM:

Output is sent to the `TERMINAL` (standard out), by default. You can also send the output to a file, or a file and the terminal simultaneously (`BOTH`). `FILE` and `BOTH` will invoke another prompt asking for an output file name (see *Output file name:* on page 525).

The `PROGRAM` response means that **sorti** will use your output routine to handle output records.



Additional runtime information will appear in the output if the `MONITOR_LEVEL` value in the Unix **cosortrc** file or Windows registry is `>0`. **CoSort** runtime information is sent to **stderr**, however, while **sorti** terminal output is sent to **stdout**.

3.14 Output file name:

This prompt is displayed if you responded `FILE` or `BOTH` at the previous prompt (see *OUTPUT: TERMINAL (default), FILE, BOTH or PROGRAM:* on page 524). The name of the output file is given in the form:

```
[>>] filename [>e]
```

Enter a single file name to receive the output. System devices can be used.



For example, on many Unix systems, if the output file selected is `/dev/lp`, output will be directed to the system printer. ◆

Use the file name **stdout** to represent the system's standard output file.

When sorting, output records are not produced until the last input file has been completely read and closed. Therefore, the output file name can be the same as an input file.

If the selected file already exists, you will receive the message:

```
filename: Exists. Use anyway (Yes/No)?
```

If you respond `No`, you will be asked for another file name. If you respond `Yes`, the existing file will be overwritten.



WARNING!

Do not use the same name for the output file as any input file when merging.

The optional `>e` extension can be used with variable-length records to replace the line-feed character with an `e`.

The optional `>>` added to the beginning of a pre-existing file name indicates that the output will be appended to the end of the existing file. For example, if you specify `>>report`, the output of the sort or merge will be appended to the file **report**. If the file does not already exist, it will be created.

3.15 EXECUTE with these specifications? YES (default) or NO:

At this point, all the specifications have been entered, and you can choose whether to execute them. If you do not, your next prompt will be at *section 3.17*.

3.16 Execution Time

```

=====|=====| =.===== |           Start Time
      .
      .
      .
=====|=====.= == xxx Records      Stop Time

```

If you respond **Yes** at the **EXECUTE** prompt, the **Start Time** line is drawn and the sort or merge begins using the given specifications. When control returns to the **sorti** program, a record or byte count is shown with the **Stop Time**. The lines are marked at 5-character and 10-character intervals for reference. If **MONITOR_LEVEL** is set above 0 in the Unix **cosortrc** file or Windows registry, additional runtime messaging can appear with the output, although technically it is sent to **stderr**.

3.17 Save specifications for reruns? YES or NO (default):

The specifications entered in the previous steps can be saved for batch execution. A **YES** or **SAVE** response will invoke a prompt asking for a file name. The **NO** default response clears the current specifications in preparation for the next operation.

3.18 Specifications file name:

This prompt is displayed if you responded **YES** or **SAVE** at the previous prompt (see *section 3.17*). This file will hold the specifications you have just entered. Naming conventions are not required by **sorti**, but you might want to adopt your own. Since the specifications will carry the developers name, date, and input data name, it is suggested that your specifications file name states the purpose of the sort or merge, for example:

```

Job1.cs_specs           J1T2V3.spc
inventory.merge.cs      Outstanding.sorter

```

If you choose the name of an existing file, **sorti** responds:

```
filename: Exists.    Use anyway (Yes/No)?
```

If you respond **No**, you must choose a different name.

BATCH OPERATIONS on page 534 discusses the format of the specifications file. If you want to examine it, you can make use of the display option discussed in **DISPLAY** on page 531.

4 MERGING

Action: SORT MERGE DISPLAY GUIDE (default) END MERGE

Responding MERGE initiates the queries for merging.

In merging, one or more sorted input files are combined into a single sorted output file. The keys used for the merge must be the same as those used from the previous sort(s). The results are the same as sorting the input files, but, of course, merging is much faster.

Since the input files are in order, the output starts immediately. The output file should have a different name than any of the input files so that the input file will not be overwritten.

When there are more files to merge than the system permits (approximately 25 or more), **sorti** will group the files and create temporary files in the declared overflow directory(ies). This work is transparent to the user, but when it occurs, output does not start immediately.

The queries and responses for merging are the same as those for sorting. Follow the directions beginning with *Record Length: 0 (variable (default)) / fixed:* on page 517.



When two or more records have the same keys, their output order is determined by the order in which the input files were specified for the merge, i.e., merges are inherently stable.

4.1 A Sample Merge

An example of a merge is provided to show the user entries required. Two input files will be used: **dem**s will contain only Democrats; **rep**s, only Republicans.

U To create **dem**s and **rep**s, perform the sample sort in *A Sample Sort* starting on page 511. To produce the output shown below from the **demrep** file, enter:

```
head -7 demrep > dems
tail -10 demrep > reps ◆
```

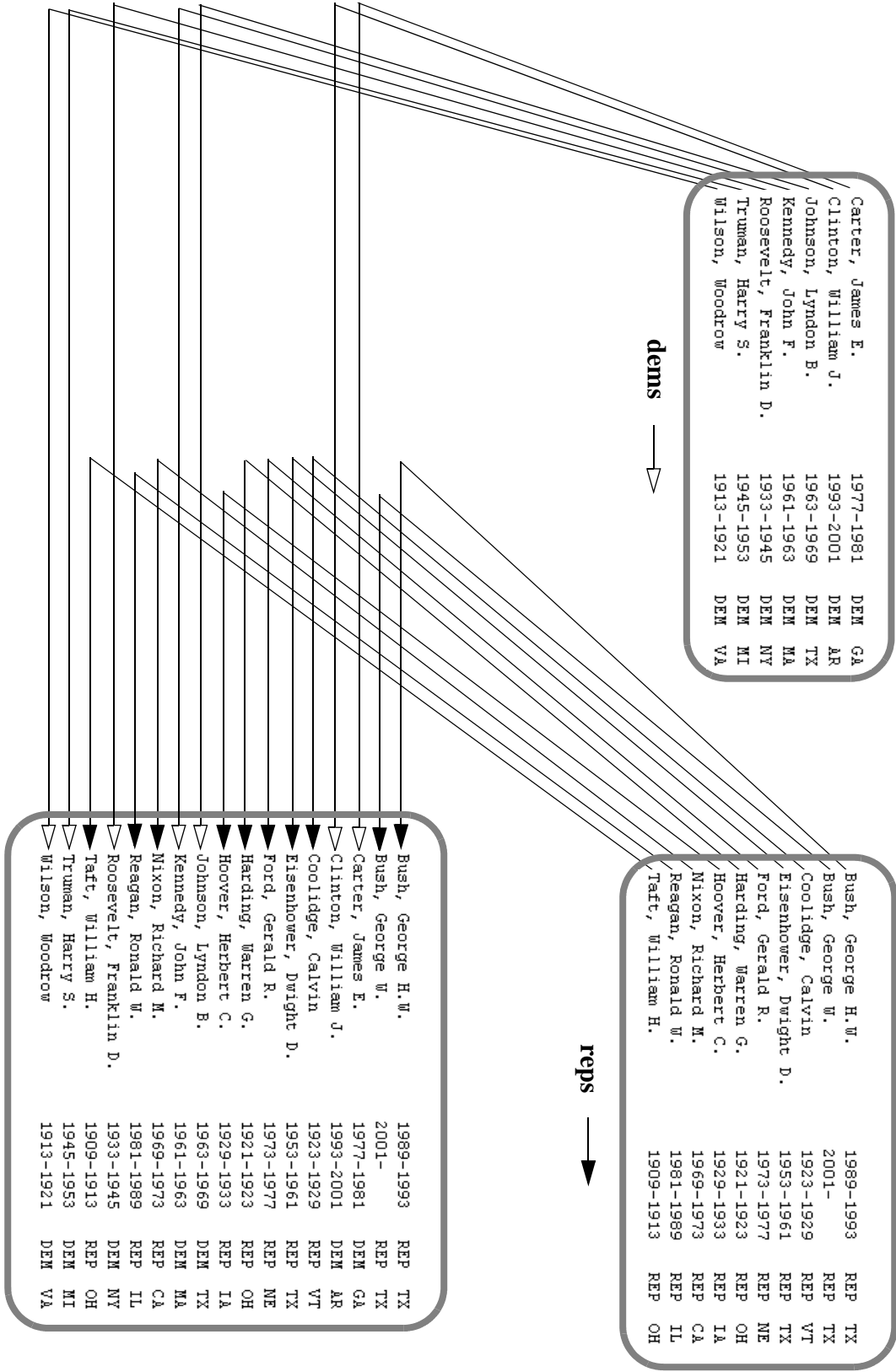
```
====1====2====3====4====5
Carter, James E.      1977-1981   DEM   GA
Clinton, William J.  1993-2001   DEM   AR
Johnson, Lyndon B.   1963-1969   DEM   TX
Kennedy, John F.     1961-1963   DEM   MA
Roosevelt, Franklin D. 1933-1945   DEM   NY
Truman, Harry S.     1945-1953   DEM   MI
Wilson, Woodrow       1913-1921   DEM   VA
====1====2====3====4====5
```

dems

```
====1====2====3====4====5
Bush, George H.W.     1989-1993   REP   TX
Bush, George W.       2001-       REP   TX
Coolidge, Calvin      1923-1929   REP   VT
Eisenhower, Dwight D. 1953-1961   REP   TX
Ford, Gerald R.       1973-1977   REP   NE
Harding, Warren G.    1921-1923   REP   OH
Hoover, Herbert C.    1929-1933   REP   IA
Nixon, Richard M.    1969-1973   REP   CA
Reagan, Ronald W.     1981-1989   REP   IL
Taft, William H.      1909-1913   REP   OH
====1====2====3====4====5
```

reps

Refer to *Table 18* on page 530 for help in performing this example:



Merging dems and reps into mrgrps using the Name key

Table 18: Query/Response—mrgprs

Query	Response	Comment	Section
Action:	merge	initializes a merge	4
Record Length:	0 or <cr>	sets the record length as fixed or variable	3.1
Number of input files:	2	merges 2 files	3.2
Input File #1:	dems	first file to be used in merge	3.3
Input file 2:	reps	second file to merge	3.3
Keys Unique:	1 or <cr>	indicates only one key has been pre-sorted	3.4
Key 1 Direction:	ascending or <cr>	indicates low→high (A→Z) sorting of Name	3.5
Location:	fixed or <cr>	Name is found at a fixed location	3.6
Start Position:	1 or <cr>	indicates byte position where Name begins	3.7
Field size:	27	length of Name field	3.8
Format:	alpha or <cr>	Name consists of text characters	3.9
Type	ASCII or <cr>	Name is ASCII characters	3.10
Alignment:	none or <cr>	Name is already aligned	3.11
Case fold letters:	no or <cr>	Name is already capitalized properly (first letter only)	3.12
Output:	both	sends the output to your terminal and creates a file	3.13
Output file name:	mrgprs	names the output file	3.14
EXECUTE...:	yes or <cr>	watch your screen for output	3.15
Save specifications for reruns:	yes	allows you to create a file with the above specifications	3.17
Specifications file name:	mrgprs.spc	you have created a specification file with this name	3.18

5 DISPLAY

```
-----  
Action:  SORT  MERGE  DISPLAY  GUIDE (default)  END DISPLAY█
```

DISPLAY brings a text file to the foreground so that the format and key locations of a file can be seen. A simple prompt for the file name follows. You can also use the form:

`DISPLAY filename`

If you use the simpler form, you will be prompted for the file name. If the file is not present, the following message appears:

`filename: No such file or directory:`

The prompt will be repeated until you indicate an available file name or stop.

6 SPECIFICATION ENTRY GUIDE



```
Action:  SORT  MERGE  DISPLAY  GUIDE (default)  END GUIDE█
```

The **sorti** start-up sequence is repeated and an explanation of the features is displayed (see *Start-up* on page 508).

7 ENDING INTERACTIVE

```
-----  
Action:  SORT  MERGE  DISPLAY  GUIDE (default)  END END█
```

This ends the interactive session with **sorti**, and returns you to your command prompt. Pressing the <Control>-C or <Delete> key will also take you out of **sorti** on many systems.

8 **BATCH OPERATIONS**

In batch operations, **sorti** takes its specifications from a text file rather than the user. The easiest way to make such a file is to generate and save it from an interactive **sorti** session. Provided the formatting is correct, the file can also be produced by a text processor, an application program, or a properly configured shell script.

Normally, there are no messages during execution. The start-up banner is not shown. The only messages that occur are error messages sent to the standard error device. On an error, **sorti** exits with a non-zero status.



Following **sorti** execution, write a script similar to:

```
if test $? ne 0; then echo sorti problem $?
```

to warn of a problem. The value of the status is the error condition explained in *ERROR and RUNTIME MESSAGES* on page 660. ◆

9 BATCH EXECUTION

A **sorti** batch program is initiated from the command line or shell script with:

```
sorti specifications file
```

The `cs_input()` routine supplied in the package allows input records to come from standard-in. If it is used, and the specifications direct that `cs_input()` is the source of records, then the following form can also be used:

```
program | sorti specifications file
```

If the specification indicates that output is to go to the terminal (**stdout**), the following form is used:

```
sorti specifications file | program
```

where *specifications file* is the source of the specifications.

To verify that you have the right specifications file, you may want to list it first. Usually, there are comments and references to data files which include warnings or instructions for the run, such as which executing program to use.

If you issue the command and the **sorti** program is not known, you will get the message:

```
sorti: not found or Permission denied
```

You will have to find the program or change its access level in order to use it. The original **sorti** program is typically held in the **\$COSORT_HOME/bin** directory. Contact your system administrator to make **sorti** accessible to you.

If the program starts and issues the message:

```
sorti:      specifications file: no such file \
           or directory
```

then the specifications file is not available. Some possible reasons are:

- the file name is incorrectly spelled
- the file is not on your path
- the security access code is incorrect.

Again, the solution is to make the file accessible. A specifications file is likely to be with application program files. The application files should be in a user directory rather than a system directory.

10 SPECIFICATION SOURCES

In a development environment, the most convenient sources of specifications are the files produced by the interactive use of **sorti**. With this program, you can experiment before creating a specifications file. At the end of every sort or merge operation, interactive **sorti** prompts:

Save specifications for reruns? YES or NO

If you respond YES, you will be prompted with:

Specifications File Name

That file has the developer's responses and comments, and is in the proper format for input to batch **sorti**. A file created with interactive **sorti** can act as a template for other specifications.

You can use a text editor to quickly develop or modify specifications. Be sure that your text editor does not insert special control characters. To modify specifications, load an existing specifications file into your text editor, make the necessary changes, and then save the new file.

You can also generate specification text from an application program using the format discussed in *SPECIFICATIONS FORMAT* on page 537.

One way to integrate sorting and merging into an application program is as follows:

- 1) The user program generates specifications.
- 2) The program stops or initiates an independent process.
- 3) Batch **sorti** executes on the given specifications file.
- 4) The user program restarts and continues by reading in the sorted records.

11 SPECIFICATIONS FORMAT

The following conventions apply to entries in a specifications file:

- each record has a variable length of not more than 65,535 bytes
- a comments field starts with the # character
- each record consists of one of the following:
 - a comment of not more than 128 bytes
 - a command of 1 to 128 bytes
 - a command followed by a comment.
- a null entry elects a default command; if a comment is given, white space is needed before the comment
- commands can be uppercase or lowercase and abbreviated to one character
- comments of 128 bytes or less can be repeated as required and are not interpreted as commands
- the number of records depends on the job that is being specified
- groups of specifications, each controlling a job, can be repeated.

12 SPECIFICATION SUMMARY

Table 19 shows the available specification options.

Table 19: Specification Options

Number	Command Options	Meaning
1	<code>SORT/MERGE/CHECK</code>	Action
2	<code>#</code>	Record Length 0 (Variable)
3	<code>#</code>	Number of Files 0 for Procedure
4	<i>input file name</i>	(repeat for each file specified in 3)
5	<code>#/# Stable Unique</code>	Number of Keys 0 for Procedure
6	<code>ASC / DES</code>	Key Direction (repeat 6-12 for each key specified in 5)
7	<code>FIXED char</code>	Locator: Fixed Delimited
8	<code>#</code>	Start Position Field Number
9	<code>#</code>	Field Length
10	<code>ALPHA / NUMERIC/ DATE / TIME / TIMESTAMP</code>	Format
11 Alphanumeric	<code>NO / LEFT / RIGHT</code>	Alignment Readable or Binary
12 ALPHA NUMERIC TIME DATE TIMESTAMP	<code>YES / NO</code> Select type for the given group	Case Shift see <i>DATA TYPES</i> on page 611

Table 19: Specification Options (cont.)

Number	Command Options	Meaning
13	PROGRAM TERMINAL FILE BOTH	Output to procedure Standard output device File output Terminal and file
14	<i>output file name</i>	13:FILE/BOTH

The following is an example of a specifications file. The numbers on the left do not appear. The comments are made by interactive **sorti**.

```

1      # sorti spec's login @mach name
2      # create date time year
3      Sort      # Action
4      0          # Record Length
5      1          # Number of Input Files
6      chiefs     # Input File #1
7      1          # Number of Keys
8      ascend    # Key #1 direction
9      ,          # field separator character
10     2          # starting field
11     10         # length
12     alpha      # format
13     none       # space/tab trimming
14     no         # case folded
15     file       # Output
16     ch.out     # Output file name

```

Notes:

1-2 Comments are application-independent; you may want to describe what and why, for example:

```

# Inventory Report 1 invtrpt.spc
# set Environment Directories to /extra1 and /extra2
# With 100,000 records, job runs in 2 mins.

```

6 This line is repeated as many times as indicated in line 5.

8-14 The command at Line 8 can be abbreviated to A. The series between lines 8-14 is repeated as many times as was indicated in line 7.

16 This line does not appear if the output, line 15, does not call for a file.

13 RUN-TIME ERROR CONDITIONS

Unlike the errors that occur in interactive **sorti**, problems in batch **sorti** are fatal. An error message is sent to the standard error output device and a non-zero exit status is returned, describing the problem.

The error number and message are shown in *ERROR and RUNTIME MESSAGES* on page 660. Some errors can be repaired by the user. Three categories of errors are identified:

1. Definition Error

An obvious semantic problem such as a typographical error:

- misspelling, e.g., SRT for SORT
- missing field, e.g., no output name at the end.

These problems might be corrected by reviewing the specifications file for obvious errors.

2. Execution Error

There is a less obvious error in the definitions:

- missing (non-accessible) input file name
- impossible format, e.g., missing field separator.

These problems might be corrected by reviewing your files and data.

3. Operation Error

These problems occur later in a sort or merge when there is insufficient overflow space, or when you are not permitted to write to an output file. The problems involve workspace and file permissions. They will probably require the attention of the specification developer and the system administrator.

API (Application Program Interface)

1 OVERVIEW



The names of **CoSort** library files have changed for version 9, but **CoSort** is backward-compatible if you have upgraded from an older version and are using the older names. It is recommended that you modify the names to those introduced in version 9 for organizational purposes and for consistency with future releases. The version 9 file names (listed before their previous names) are as follows:

<code>libcosort.a</code>	<code>cosort.a</code>
<code>libsortcl.a</code>	<code>sortcl.a</code>
<code>libsortcl_vsam.a</code>	<code>sortcl_vsam.a</code>
<code>libsortcl_idx.a</code>	<code>sortcl_idx.a</code>
<code>libsortcl_vis.a</code>	<code>sortcl_vis.a</code>
<code>libmfcosortwb.a</code>	<code>mfcosortwb.a</code>
<code>libmfcosortse.a</code>	<code>mfcosortse.a</code>
<code>libacucosort.a</code>	<code>acucosort.a</code>
<code>libnat2cs.a</code>	<code>nat2cs.a</code>

There are two primary thread-safe interfaces available to **CoSort** users:

- sortcl_routine()** Provides users with access to all the sorting, joining, data conversion, selection, aggregation and other data manipulation functionality available in **CoSort**'s standalone tool, **sortcl**.
- cosort_r()** A direct interface into the **CoSort** sort engine, allowing access to sorting and merging. This is the successor to the `cosort()` and `mcs()` APIs from prior releases.

The `sortcl_routine()` API is used when your data processing specifications can be provided in a **sortcl** script. `sortcl_routine()` has a very simple interface and can be easily accessed. It is therefore recommended that you use `sortcl_routine()` whenever possible.

`cosort_r()` is used for access to the **CoSort** sorting engine from within a user application program. `cosort_r()` has a more complex interface and requires some advanced programming. It is also functionally limited to sort/merge operations, but its calling conventions are familiar to those upgrading from using prior `cosort()` API calls.



The non-thread-safe `cosort()` API continues to be supported for pre-version 9 **CoSort** users, but it is no longer documented. The use of the thread-safe `cosort_r()` API is recommended.

The `mcs()` API also continues to be supported for legacy users, but the `cosort_r()` API is recommended.

1.1 Calling the CoSort APIs

All **CoSort** APIs follow a C calling convention. Therefore, in order to call any of the **CoSort** APIs using a different programming language, the language must support this convention. The parameters must be set up and provided as expected by the APIs.

Because languages like FORTRAN and Pascal pass parameters by reference rather than by value, this is a relatively easy task. Among others, this includes C++, COBOL, Visual Basic and Java. Java users can call `sortcl_routine()` from an intermediate C program, and use the Java Native Interface (JNI) to call the intermediate program. For details, see the example *Calling sortcl_routine() Using Java on page 551*.

2 sortcl_routine()

This section introduces the programming conventions for calling the `sortcl_routine()` procedure from within your application program.

`sortcl_routine()` provides a direct interface between an application program and all of the commands offered by **CoSort**'s standalone tool, **sortcl**. It provides you with access to sorting, joining, merging, data manipulation, aggregation and all of **sortcl**'s extensive feature set. `sortcl_routine()` allows you to pass in a **sortcl** script that details all the data definition and data manipulation specifications.

Within the **sortcl** script, you can also customize your own input, output, and compare routines to best suit the requirements of your calling program.

2.1 Usage

The `sortcl_routine()` interface consists of three primary calls, which are made in the following order:

- 1) `sortcl_alloc()`.
- 2) `sortcl_routine()` *on page 544*.
- 3) `sortcl_free()` *on page 545*.

`cs_sortcl_t` is defined in **sortcl_routine.h**, and its elements are defined as follows:

```
typedef struct cs_sortcl_s  cs_sortcl_t;

struct cs_sortcl_s {
    void* priv;
};
```

2.1.1 sortcl_alloc()

`sortcl_alloc()` takes no parameters and returns a variable of type `cs_sortcl_t`. This call allocates some of the memory resources, and initializes variables needed by `sortcl_routine()` for execution:

```
cs_sortcl_t* sortcl_alloc();
cs_sortcl_t* sortcl = sortcl_alloc();
```

2.1.2 **sortcl_routine()**

This routine is where the **sortcl** commands are passed by the calling program and the job is executed.

`sortcl_routine()` is a function that accepts two arguments:

- a pointer to the structure `cs_sortcl_t`, which was allocated in `sortcl_alloc()`
- a pointer to a char.

The return value is an integer that is set to 0 upon success, and an error number upon failure (see **ERROR** and **RUNTIME MESSAGES** *on page 660*). All syntax and execution messages are sent to **stderr**:

```
int sortcl_routine(cs_sortcl_t* sortcl, char* command);
sortcl_routine(sortcl, command);
```

where *command* is a **sortcl** command expressed as either a set of command line arguments, or a reference to script file. Examples of acceptable syntax are as follows:

- `sortcl_routine("/specification=job1");`
- `sortcl_routine("/inf=sam /field=(first,pos=1,size=2) /outf=john")`
- `strdup(job2, "/infile=sam /spec=key_script /outfile=john")`
`sortcl_routine(job2)`

Environment variables are expanded by `sortcl_routine()`. The use of backslashes(\) to protect shell reserved symbols (such as parenthesis, periods, pound signs, and ampersands) is not required.

For complete details on the **sortcl** language see the **sortcl PROGRAM** *chapter on page 37*. For details on using specification files, see **Specification Files** *on page 45*.

While using the `sortcl_routine()` API, the calling program can also set up the **sortcl** job script to perform custom input, output and compares, as described in **CUSTOM INPUT, OUTPUT AND COMPARE PROCEDURES** *on page 556*.

2.1.3 *sortcl_free()*

This routine is called to free all the memory resources allocated by **CoSort** for the execution of the job. It accepts one parameter, which is a variable of the type `cs_sortcl_t`, and returns a void:

```
void sortcl_free(cs_sortcl_t* sortcl);
sortcl_free(cosort);
```

2.2 Linking with *sortcl_routine()*

2.2.1 Linking on Unix/Linux

To resolve the function call to `sortcl_routine()` in the calling program, it needs to be linked with the static libraries **libsortcl.a** and **libcosort.a**, or with the dynamic libraries **libsortcl.so** and **libcosort.so**. These exist by default in the **\$COSORT_HOME/lib** directory. If **libsortcl.so** is present in any of the directories **/lib**, **/usr/lib**, or **/usr/local/lib** while performing dynamic linking, it can be referenced with `-lsortcl`.

The calling program will also require the header files **cosort.h** and **sortcl_routine.h**, which reside in the **\$COSORT_HOME/include** directory.

The static linking can be executed using the following command:

```
cc -o user test's main and other objects \
  $COSORT_HOME/libs/libsortcl.a \
  $COSORT_HOME/libs/libcosort.a -ldl -lm -lpthread -lposix
```

where `-lposix` must be replaced by `-lrt` if the POSIX library is not available on your system, or if you encounter undefined references to AIO symbols.



For 64-bit architectures, the options will vary depending on your operating system. Some of these options might be as follows:

Solaris: `-xarch=v9`

AIX: `-q64`

HP-UX: `+DD64, +DA2.0w`

2.2.2 Linking on Windows

To resolve the function call to `sortcl_routine()` in the calling program, it needs to be linked with the static libraries **libsortcl_static.lib** and **libcosort_static.lib**, or with the dynamic link libraries **libsortcl_dynamic.lib** and **libcosort_dynamic.lib**. These exist by default in the *install_dir\lib* directory.

The compiler also requires the header files **sortcl_routine.h** and **cosort.h**. Some additional libraries, including **advapi32.lib**, **kernel32.lib**, and **ws2_32.lib**, might be required to resolve external objects.

If compiling using Microsoft Visual Studio, be sure to select the multi-threaded option, and the structure alignment as one-byte.

If compiling dynamically, you must define `__DLL_IMPORT__`. Also, note that the dynamic runtime libraries **libsortcl.dll** and **libcosort.dll** must reside in the directory in which the executable will be run. Otherwise, they should reside in the system directory, such as **C:\WINNT\System32**.

You can perform static linking as follows:

```
cl apples_program_objects libsortcl_static.lib libcosort_static.lib  
[additional_libraries]
```



This example represents a generic Windows linking command using the Microsoft C/C++ compiler. Depending on your programming language, you might use a different method to link to the **CoSort** library.



WARNING!

You must re-link with the latest **CoSort** libraries whenever you install a new patch or a new version of **CoSort**.

2.3 sortcl_routine() Examples

This section contains basic `sortcl_routine()` examples. More extensive examples that illustrate the use of custom input, output and compare procedures are provided at the end of the next section (see *Examples using Custom Input, Output, and Compare Procedures on page 565*).

2.3.2 Using `sortcl_routine()` with a Summary Function

This example prompts a user to enter a number representing a particular store within a chain. Based on this entry, **sortcl** calculates the sales total from every department within that store (see Using WHERE with Sums, Cross-Calcs on Sums *on page 301* for a standalone version similar to this example).

Given the input file **sales**, a portion of which looks like:

```
01 25 2103.54 2002-01-13
12 03 868.51 2002-01-03
05 25 103.24 2002-01-10
01 03 7969.68 2002-02-17
05 03 756.24 2002-02-10
12 25 5640.34 2002-02-09
12 45 7879.67 2002-02-05
```

and the data definition file **api_sum.ddf**:

```
/FIELD=(store, POSITION=1, SIZE=2)
/FIELD=(dept, POSITION=4, SIZE=2)
/FIELD=(amount, POSITION=7, SIZE=7, NUMERIC)
/FIELD=(date, POSITION=16, SIZE=10, JAPANESE_DATE)
/CONDITION=(newdept, TEST=(dept))
/CONDITION=(st_01, TEST=(store == "01"))
/CONDITION=(st_05, TEST=(store == "05"))
/CONDITION=(st_12, TEST=(store == "12"))
```

You can write the program **store_totals.c**, containing `sortcl_routine()`, that produces one of three output files, depending on the user entry:

CONTINUED ON NEXT PAGE...

```
case 12:
    strcat(szSpec,
        "/OUTFILE=store12.out\n"
        "/HEADREC=\"Dept Store12\\n\\n\"\n"
        "/FIELD=(dept,POSITION=2,SIZE=2)\n"
        "/FIELD=(t12,POSITION=4,SIZE=9)\n"
        "/SUM t12 FROM amount WHERE st_12 BREAK newdept\n");
    break;
default:
    printf("Error: invalid store number\n");
    iRetVal = 1;
}
if (iRetVal == 0) {
    /* allocate the main sortcl context variable */
    sortcl = sortcl_alloc();
    if (sortcl) {
        /* call sortcl_routine api */
        iRetVal = sortcl_routine(sortcl, szSpec);
        /* free the sortcl variable */
        sortcl_free(sortcl);
    }
}
/* if return value != 0 it is an error */
return (iRetVal);
}
```

For example, if the user selects 1 when prompted, the output file would be **store1.out**:

```
Dept Store01
03      7970
25      2104
45       0
```

2.3.3 Calling `sortcl_routine()` Using Java

You can call `sortcl_routine()` from Java using Java Native Interface (JNI) and an intermediate C program as described below.

The procedure for calling `sortcl_routine()` using Java is:

- 1) Create a native interface method in your Java application. Generate the header file for C program using `javah`.
- 2) Create an intermediate C program that invokes `sortcl_routine()`.
- 3) Compile the C program so that it can be dynamically linked to from a Java program, such as `.dll` on Windows or `.so` on Unix, via the JNI.
- 4) Complete the rest of your Java application and call the method defined by the native interface.

Generate the header file for the C program using `javah`. Be sure to provide the directory for the `.class` files. In this example, the files are in the directory `com/iri/jni/sort` so the `javah` call should match the package name followed by the class name. The output of this command will be a C header file, in this case **`com_iri_jni_sort_JavaToSortcl.h`**.

The following header file, **`com_iri_jni_sort_JavaToSortcl.h`**, is used by the previous C program:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_iri_jni_sort_JavaToSortcl */

#ifndef _Included_com_iri_jni_sort_JavaToSortcl
#define _Included_com_iri_jni_sort_JavaToSortcl
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      com_iri_jni_sort_JavaToSortcl
 * Method:     callsortcl
 * Signature:  (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_com_iri_jni_sort_JavaToSortcl_callsortcl
    (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

The following intermediate C program, **callcosort.c**, calls sortcl_routine():

```
/* Copyright © 2013 - CoSort / Innovative Routines Int'l, Inc.
Purpose: This C program calls sortcl_routine() and acts as
an interface for calls from a JavaToSortcl.java Java program. */

#include "com_iri_jni_sort_JavaToSortcl.h"
#include "cosort.h"
#include "sortcl_routine.h"

JNIEXPORT jint JNICALL Java_com_iri_jni_sort_JavaToSortcl_callsortcl (JNIEnv *env, jobject obj, jstring jspec) {
    /* Convert Java String object to C characters */
    const char *spec= (*env)->GetStringUTFChars(env, jspec,0);

    cs_sortcl_t* sortcl; /* main sortcl context variable */

    /* allocate the main sortcl context variable */
    sortcl = sortcl_alloc();
    if (sortcl) {
        return sortcl_routine(sortcl,spec);
    }
    return -1;
}
```

W For Windows users, the following batch file creates the .dll (used by the JNI) from the intermediate C program callcosort.c:

The following example is on Windows and uses the Visual Studio Command Prompt that contains the path of Visual Studio libraries. Note that the COSORT_HOME location must also be set.

```
cl -I"C:\Program Files (x86)\Java\jdk1.7.0\include" -I"C:\Program Files
(x86)\Java\jdk1.7.0\include\win32" /I%COSORT_HOME%include callsortcl.c
%COSORT_HOME%lib\libsortcl_static.lib %COSORT_HOME%lib\libcosort_static.lib
advapi32.lib -LD -Fecallsortcl.dll
```



The following Java program demonstrates the ability to call `sortcl_routine()` by linking with the C program `callcosort.c`:

```
// Copyright © 2013 - CoSort / Innovative Routines Int'l, Inc.
// Purpose: This Java program demonstrates the ability to call
// sortcl_routine() API via another C program "callsortcl.c"
// with a Sort Control Language (SortCL) specification
package com.iri.jni.sort;

public class JavaToSortcl {
    public native int callsortcl(String spec);

    static {
        try {
            System.loadLibrary("callsortcl");
        } catch (UnsatisfiedLinkError e) {
            System.err.println("Native code library failed to load.\n" + e);
            System.exit(1);
        }
    }

    public static void main(String[] args) {
        String spec = "/spec=example1a.scl";
        System.out.println("Calling callsort (sortcl_routine())... ");
        try{
            JavaToSortcl jToS = new JavaToSortcl();
            int ret = jToS.callsortcl(spec);
            System.out.println("callsort (sortcl_routine()) returned with error code " + ret);
        }catch (UnsatisfiedLinkError e) {
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```

The sortcl script being called in this example is **example1a.scl**, which can be any sortcl job specification file (see Specification Files *on page 45*). Alternatively, you can specify a set of sortcl commands directly.

2.3.4 Calling sortcl from within a Java Program

The **sortcl** executable can be called directly from within a Java program, as shown in the following example:

```
// Copyright © 2011 - CoSort / Innovative Routines Int'l, Inc.
// Purpose: This simple Java program demonstrates the ability to call
// standalone sortcl executable from a Java program

import java.io.*;
class RunSortclExe {
public static void main(String args[]) {
    Process sortcl= null;
    String output;
    BufferedReader stderr;
    int ret = -1;

    try {
        sortcl = Runtime.getRuntime().exec("sortcl.exe /infile=chiefs
            /outfile=chiefs.out /monitor=1");
    }
    catch (IOException e) {
        System.out.println("unable to launch sortcl.exe " + e.getMessage());
        System.exit(-1);
    }
    stderr = new BufferedReader(new InputStreamReader(sortcl.getError
        Stream()));

    /* Read the output (stderr) from sortcl */
    try {
        ret = sortcl.exitValue();
    }
    catch (IllegalThreadStateException e) {
        ReadProcess(stderr);
    }
}

public static void ReadProcess(BufferedReader stderr) {
    StringBuffer sb = new StringBuffer();
    try {
        char[] buf = new char[8];

        int numRead = stderr.read(buf);

        while (numRead != -1) {
            for (int i = 0; i < numRead; i++) {
```

CONTINUED ON NEXT PAGE...

```
        if (buf[i] == '\n' || buf[i] == '\r' || buf[i] == '\b'
            || (int)buf[i] == -1) {
            /* ignore these characters in the output and print the buffer */
            /* constructed so far*/
            if (!sb.toString().equals(""))
                System.out.println(sb.toString());
            sb = new StringBuffer();
        }
        else {
            /* construct the buffer */
            sb.append(buf[i]);
        }
    }

    /* read again */
    numRead = stderr.read(buf);
}
}
catch (java.io.IOException e) {
    System.err.println("ERROR reading from process " + e.getMessage());
}
}
```

3 CUSTOM INPUT, OUTPUT AND COMPARE PROCEDURES

While using **CoSort**'s standalone tool, **sortcl**, or the `sortcl_routine()` API to execute **sortcl** commands, the calling program can also set up the **sortcl** script to perform custom input, output, and compares.

A simple way to dynamically link your procedure with **sortcl**:

- 1) place your procedures in the program `cs_user.c` (the default reference)
- 2) compile and link with this command:

```
cc -shared cs_user.c -o cs_user.so
```

At **sortcl** execution, the names used in your script will be matched to the procedure in `cs_user.c`.

3.1 Custom Input

sortcl allows the calling program to set up a custom input procedure by way of the command `/INPROCEDURE`.

You can use a custom input procedure to process flat records from a proprietary DBMS or ETL tool, for example. In these cases, you can program a *hook* between the API of the tool (such as ODBC or OCI) and your **CoSort** input procedure.

Another benefit of writing a custom input procedure is horizontal and/or vertical selection. You can reduce the data that the sort or merge will process by removing entire records (horizontal selection) or removing unnecessary parts of records (vertical selection).

One of the uses for `/INPROCEDURE` is to pre-process records, such as Control Detail Records (CDRs). For more information, contact your IRI representative.

The following section describes the syntax for invoking a custom input procedure.

3.1.1 Syntax: /INPROCEDURE

To ensure your input procedure is called by **sortcl**, use the following syntax within a job script:

```
/INPROCEDURE [=procedure_name]
[/PROCESS=process_type]
[/LENGTH=fixed_record_length] or [/LENGTH=0]
[other_attributes]
```

`/LENGTH=0` (the default) is used for variable-length records.

The default *procedure_name* is assumed to be `cs_input()`. Otherwise, the name specified by the calling program is used to resolve the procedure. The default process type is `RECORD`, which can be either fixed- or variable-length (records terminated by a linefeed or carriage return). The calling program can also specify an alternate process type, as described in *DATA SOURCE AND TARGET FORMATS (/PROCESS)* on page 53.

The following sections describe the implementation process, in the order in which it is performed:

- Custom Input Procedure Declaration on page 557.
- Registering the Custom Input Procedure on page 557.
- Building the Custom Input Procedure for Runtime Linking on page 558

3.1.2 Custom Input Procedure Declaration

A custom input procedure accepts two parameters:

- a pointer to the buffer that contains the input records
- a pointer to the size of the buffer in bytes.

The integer pointed to by the buffer size should be set to the actual number of bytes placed into the buffer by the input procedure.

The procedure(s) returns an integer value that should be set to 1, until there are no more records to pass. After the last record, the return value should be set to 0:

```
int procedure_name(char* buffer, int* buffersize)
```

The format of the data in the *buffer* depends on the `/PROCESS` specification in your **sortcl** script (see *DATA SOURCE AND TARGET FORMATS (/PROCESS)* on page 53).

3.1.3 Registering the Custom Input Procedure

When using a custom input procedure with the `sortcl_routine()` API, the calling program can choose to register the input procedure with **CoSort**. This needs to be done after the call to `sortcl_alloc()`, but before the call to `sortcl_routine()`. This adds the calling program's custom procedure(s) to an internal list within **CoSort**, which is resolved at runtime. If the calling program chooses not to do this, **CoSort** will dynamically resolve and load the procedure name from the `/INPROCEDURE` definition in the **sortcl** script.

The calling program can register the procedure using a call to `sortcl_reg_userproc()`. This function accepts three parameters:

- a pointer to the structure `cs_sortcl_t` (see *Usage on page 543*)
- a pointer to a character -- this stores the name of the procedure you are trying to register
- a pointer of the type `cs_function_ptr()` -- a function pointer that points to the address of the procedure.

A return value of 0 indicates a successful registration. If there are invalid input parameters, -1 is returned:

```
typedef int (*cs_function_ptr)();  
int sortcl_reg_userproc(cs_sortcl_t* sortcl, char *funcname,  
cs_function_ptr funcptr)
```

If you have a custom input procedure called **sortcl_routine_input**, for example, it would be set up as follows.

```
cs_sortcl_t* sortcl = sortcl_alloc();  
sortcl_reg_userproc(sortcl, "sortcl_routine_input", sortcl_routine_input)  
sortcl_routine(char* command);
```

Every new custom procedure must be set up using an additional call to `sortcl_reg_userproc()`.



At runtime, the registration list will be searched by **sortcl** to locate the procedure addresses. If the procedures cannot be resolved using this list, **cs_user.so** (Unix) or **cs_user.dll** (Windows) will be opened for address resolution. Failure to resolve these procedures results in an error.

3.1.4 Building the Custom Input Procedure for Runtime Linking

If the user is executing a standalone **sortcl** script with an `/INPROCEDURE` statement, or the custom input procedure is not registered, **sortcl** will resolve any references at runtime. In order to do this, the custom procedures need to be built into a shared object **cs_user.so** (Unix) or a dynamic linked library **cs_user.dll** (Windows).

sortcl looks for them in the current directory, and in the list of directories pointed to by the shared library path (Unix) or the System32 folder (Windows). The environment variable `CS_USER_DLL` can also be used to point to the exact location of the library.

If a custom input procedure is registered using `sortcl_reg_userproc()`, then the application can be compiled and linked statically. For instructions, see *Linking on Unix/Linux on page 545* or *Linking on Windows on page 546*.

For an example of a custom input procedure, See *Custom Input and Output using sortcl_routine on page 565*.

3.2 Custom Output

sortcl allows the calling program to set up a custom output procedure by way of the command `/OUTPROCEDURE`.

You can use a custom output procedure to redirect sorted records to a proprietary DBMS or ETL tool, for example. In these cases, you can program a *hook* between the API of the tool (such as ODBC or OCI) and your **CoSort** output procedure.

The following section describes the syntax for invoking a custom output procedure:

3.2.1 Syntax: /OUTPROCEDURE

To ensure your input procedure is called by **sortcl**, use the following syntax:

```
/OUTPROCEDURE [=procedure_name]
[/PROCESS=process_type]
[/LENGTH=fixed_record_length] or [/LENGTH=0]
[other_attributes]
```

`/LENGTH=0` (the default) is used for variable-length records.

The default procedure name is assumed to be `cs_output()`. Otherwise, the name specified by the calling program is used to resolve the procedure. The default process type is `RECORD`, which can be either fixed- or variable-length (records terminated by a linefeed or carriage return). The calling program can also specify an alternate process type, as described in *DATA SOURCE AND TARGET FORMATS (/PROCESS)* on page 53.

The following sections describe the implementation process, in the order in which it is performed:

- Custom Output Procedure Declaration on page 560.
- Registering the Custom Output Procedure on page 560.
- Building the Custom Output Procedure for Runtime Linking on page 561.

3.2.2 Custom Output Procedure Declaration

A custom output procedure accepts two parameters:

- a pointer to the buffer that contains one output record
- the length of that record.

After **CoSort** has returned all the records, the value of the second parameter (the length) is set to NULL:

```
int procedure_name(char* buffer, int bufferlen)
```

The format of the data in the *buffer* depends on the /PROCESS specification in your **sortcl** script (see DATA SOURCE AND TARGET FORMATS (/PROCESS) on page 53).

3.2.3 Registering the Custom Output Procedure

When using a custom output procedure with the `sortcl_routine()` API, the calling program can choose to register the output procedure with **CoSort**. This needs to be done after the call to `sortcl_alloc()`, but before the call to `sortcl_routine()`. This will add the calling program's custom procedure to an internal list within **CoSort**, which is resolved at runtime. If the calling program chooses not to do this, **CoSort** will dynamically resolve and load the procedure name from the /OUPROCEDURE definition in the **sortcl** script.

The calling program can register the procedure using a call to `sortcl_reg_userproc()`. This function accepts three parameters:

- a pointer to the structure `cs_sortcl_t` (see Usage on page 543)
- a pointer to a character -- this stores the name of the procedure you are trying to register
- a pointer of the type `cs_function_ptr()` -- a function pointer that points to the address of the procedure.

A return value of 0 indicates a successful registration. If there are invalid input parameters, -1 is returned:

```
typedef int (*cs_function_ptr)();  
int sortcl_reg_userproc(cs_sortcl_t* sortcl, char *funcname,  
cs_function_ptr funcptr)
```


If you have a custom output procedure called **sortcl_routine_output**, for example, it would be set up as follows:

```
cs_sortcl_t* sortcl = sortcl_alloc();  
sortcl_reg_userproc(sortcl,"sortcl_routine_output",sortcl_routine_output)  
sortcl_routine(char* script);
```

Every new custom procedure must be set up using an additional call to `sortcl_reg_userproc()`.



At runtime, the registration list will be searched by **sortcl** to locate the procedure addresses. If the procedures cannot be resolved using this list, then **cs_user.so** (Unix) or **cs_user.dll** (Windows) will be opened for address resolution. Failure to resolve these procedures results in an error.

3.2.4 Building the Custom Output Procedure for Runtime Linking

If the user is executing a standalone **sortcl** script with an `/OUTPROCEDURE` statement, or the custom output procedure is not registered, **sortcl** will resolve any references at runtime. In order to do this, the custom procedures need to be built into a shared object **cs_user.so** (Unix) or a dynamic linked library **cs_user.dll** (Windows).

sortcl looks for them in the current directory, and in the list of directories pointed to by the shared library path (Unix) or the System32 folder (Windows).

The environment variable `CS_USER_DLL` can also be used to point to the exact location of the library.

If a custom output procedure is registered using `sortcl_reg_userproc()`, then the application can be compiled and linked statically. For instructions, see *Linking on Unix/Linux on page 545* or *Linking on Windows on page 546*.

3.2.5 Examples

For an example of a custom output procedure, see *Custom Input and Output using sortcl_routine on page 565*.

3.3 Custom Compares

sortcl allows you to invoke custom compares at the:

- record level using `/KEYPROCEDURE`
- field level using an extended `/KEY` syntax.

Both the record and the field compares use the same function prototype and implementation technique, except the **sortcl** syntax differs on how they are defined within a script. The following two sections address the syntax, and then a common section addresses the function declaration, parameters and implementation (Implementation of Custom Compares *on page 563*).

3.4 Custom Record Compare Using `/KEYPROCEDURE`

sortcl allows the calling program to set up a custom compare using the command `/KEYPROCEDURE`.

To ensure that your compare procedure is called by **sortcl**, use the following syntax:

```
/KEYPROCEDURE [=procedure_name]
```

The default procedure name is assumed to be `cs_compare()`. Otherwise, the name specified by the calling program is used to resolve the procedure. The default process type is `RECORD`, which can be either fixed- or variable-length (records terminated by a linefeed or carriage return). The calling program can also specify an alternate process type (see `DATA SOURCE AND TARGET FORMATS (/PROCESS)` *on page 53*).

3.5 Custom Key Compares Using `/KEY`

With key- or field-level compares, you can continue to use the built-in command, `/KEY` (see `KEYS` *on page 160*), while expanding the collating capabilities of **sortcl** by writing a custom compare procedure.

The custom procedure is called via the `/KEY` statement. Instead of inserting a supported data type name in the `/KEY` statement, use can reference your custom procedure:

```
/KEY=(field_name,compare=procedure_name)
```

There are no assumed defaults, that is, you must specify a compare *procedure_name*.

3.6 Implementation of Custom Compares

3.6.1 Custom Compare Procedure Declaration

The custom compare procedure accepts three parameters. The first two are pointers to the records that need to be compared, and the third is a pointer to the **CoSort** internal structure `cs_cmpdata_t` that is defined in `cosort.h`. The return value is an integer indicating which record precedes the other:

```
int procedure_name(char* record1, char* record2, cs_cmpdata_t* cmp);
```

The structure `cs_cmpdata_t` is defined in **cosort.h** as follows:

```
struct cs_cmpdata_s {
    void      *pPrivate; /* Priv pointer for use by external function */
    cs_cmp_mode iMode;
    int        iErrorNum; /* Return error value. For future use */
    int        iRecordLen1; /* Length of first field or record */
    int        iRecordLen2; /* Length of second field or record */
    int        iFieldOffset;
    int        iFieldLen;
};
typedef struct cs_cmpdata_s cs_cmpdata_t;
```

The custom compare function is called in the following stages by **CoSort**:

Initialization

CoSort first calls the custom compare using an `iMode` value of `CS_CMP_BEGIN`. At this time, the first two parameters are set to `NULL`. The external function can allocate and initialize the `pPrivate` pointer and any other resources during this call. A return value of 0 indicates a successful initialization by the external routine:

```
procedure_name(NULL, NULL, cs_cmpdata_t* cmp);
```

Compare Processing

CoSort then calls the custom compare using an `iMode` value of `CS_CMP_VALUE`, which indicates that two records or fields are being passed into the function to be compared. During this call, `iRecordLen1` and `iRecordLen2` are set to the lengths of the records or fields that are indicated by the first and the second parameter respectively:

```
procedure_name(char* ptr1, char* ptr2, cs_cmpdata_t* cmp);
```

The return value is an integer which indicates the precedence of one record or field over the other:

- If the first record precedes the second, return a positive value.
- If the second record precedes the first, return a negative value.
- If both records are equal, return 0.

Termination

After all the records have been compared, **CoSort** calls the custom compare routine one final time with an `iMode` value of `CS_CMP_END`. At this point, the routine can free any allocated resources. A return value of 0 indicates successful completion:

```
procedure_name(NULL, NULL, cs_cmpdata_t* cmp);
```

3.6.2 Building the Compare Procedure for Runtime Linking

If you are executing a **sortcl** script which contains a custom compare, **sortcl** will resolve any references at runtime. In order to do this, the custom compare procedures need to be built into a shared object on Unix or a dynamic linked library on Windows. The environment variable `CS_USER_DLL` should be used to point to the exact location of the library, which has all the custom compare functions.

If `CS_USER_DLL` is not set:

- On Windows, **sortcl** will look for the library **cs_user.dll**, in the current directory or in the **System32** folder.
- On Unix, **sortcl** will look for the library **cs_user.so** or **cs_user.sl** (HPUX) in the shared library path (which can be `LD_LIBRARY_PATH`, `SHLIB_PATH`, or `LIBPATH`).

For an example of a custom compare, *see Custom Input and Output using sortcl_routine on page 565*.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* Copyright 1978-2011 CoSort / Innovative Routines International (IRI), Inc.
* All Rights Reserved.
*
* SortCL API example to use custom input and output routines
*   ScriptFile:      custom.scl
*   Input File:      chiefs
*   Output File:     stdout (via custom output routine)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <string.h>

/* cosort header file */
#ifdef (_WIN32)
#define __DLL_IMPORT__ /* IMPORTANT: needed to work with Windows DLL */
#pragma pack(1) /* or compile with /Zpl option */
#endif /* (_WIN32) */
#include "cosort.h"
#include "sortcl_routine.h"

int input_routine(char* pBuffer, int* pLength);
int output_routine(char* pBuffer, int iLength);
FILE* ifile;

int main()
{
    int iRetVal; /* return value */
    cs_sortcl_t* sortcl; /* main sortcl context variable */

    /* allocate the main sortcl context variable */
    sortcl = sortcl_alloc();
    if (sortcl) {
        /* Register the Custom Input and Output procedures */
        sortcl_reg_userproc(sortcl, "input_routine", input_routine);
        sortcl_reg_userproc(sortcl, "output_routine", output_routine);
        if ((ifile = fopen("chiefs","rb")) == (FILE*)0) {
            printf("error reading input file\n");
            iRetVal = 2;
        }
    }
    else {
        /* Call the sortcl_routine API */
        iRetVal = sortcl_routine(sortcl, "/spec=custom.scl");
    }
    /* Free the sortcl object */
    sortcl_free(sortcl);
}

```

CONTINUED ON NEXT PAGE...

```
    else {
        iRetVal = 1;
    }
    printf("Press Enter to exit\n");
    getchar();
    return (iRetVal);
}

/* Custom Input Procedure */
int input_routine(char* pBuffer, int* pLength)
{
    int ret = 0;

    if (fgets(pBuffer, 500, ifile)) {
        *pLength = (int)strlen(pBuffer);
        ret = 1;
    }
    return (ret);
}

/* Custom Output Procedure */
int output_routine(char* pBuffer, int iLength)
{
    /* print the output record to stdout */
    if (pBuffer) {
        pBuffer[iLength] = '\0';
        printf("Buffer size = %d\n\n%s\n", iLength, pBuffer);
    }
    return 0;
}
```

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* Copyright 1978-2011 CoSort / Innovative Routines International (IRI), Inc.
* All Rights Reserved.
*
* SortCL API example to run a script file
*   ScriptFile:      keyproc.scl
*   Input File:      chiefs
*   Output File:     chiefs.out
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* cosort header file */
#ifdef ( _WIN32 )
#define __DLL_IMPORT__ /* IMPORTANT: needed to work with Windows DLL */
#pragma pack(1) /* or compile with /Zp1 option */
#endif /* ( _WIN32 ) */
#include "cosort.h"
#include "sortcl_routine.h"

int main()
{
    int iRetVal; /* return value */
    cs_sortcl_t* sortcl; /* main sortcl context variable */

    /* allocate the main sortcl context variable */
    sortcl = sortcl_alloc();
    if (sortcl) {
        /* call sortcl_routine api */
        iRetVal = sortcl_routine(sortcl, "/spec=keyproc.scl");
        /* free the sortcl variable */
        sortcl_free(sortcl);
    }
    /* if return value != 0 it is an error */
    return (iRetVal);
}

```

The following script, **record.scl**, invokes a custom record compare:

```
/INFILE=chiefs
  /FIELD= (name, POSITION=1, SIZE=27)
  /FIELD= (year, POSITION=28, SIZE=12)
  /FIELD= (party, POSITION=40, SIZE=5)
  /FIELD= (state, POSITION=45, SIZE=2)
/SORT
  /KEYPROCEDURE=cs_user_keycmp_ascii
/OUTFILE=chiefs.out
```

The following script, **keyproc.scl**, invokes a custom key compare:

```
/INFILE=chiefs
/LENGTH=0
  /FIELD= (name, POSITION=1, SIZE=27)
  /FIELD= (year, POSITION=28, SIZE=12)
  /FIELD= (party, POSITION=40, SIZE=5)
  /FIELD= (state, POSITION=45, SIZE=2)
/SORT
  /KEY= (state, compare=cs_user_keycmp_ascii, DESC)
/OUTFILE=chiefs.out
```


The function `cs_user_keycmp_ascii` can be written and compiled into a dynamic linked library on Windows or a shared object on Unix, which is then resolved at runtime. The Windows example is as follows:

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* Copyright 1978-2011 CoSort / Innovative Routines International (IRI), Inc.
* All Rights Reserved.
*
* SortCL API example to implement a custom key or record compare
*
* Compile and link as a DLL or shared object library
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdlib.h>

#if defined (_WIN32)
#pragma pack(1) /* or compile with /Zpl option */
#define _LIBSPEC __declspec( dllexport )
#else
#define _LIBSPEC
#endif /* (_WIN32) */
#include "cosort.h"

_LIBSPEC int cs_user_keycmp_ascii(char *s1, char *s2, cs_cmpdata_t *cmp)
{
    int iRetVal;

    switch (cmp->iMode) {
        case CS_CMP_BEGIN:
            cmp->pPrivate = malloc(100);
            iRetVal = 0;
            break;
        case CS_CMP_VALUE:
            iRetVal = strncmp(s1, s2, __max(cmp->iRecordLen1, cmp->iRecordLen2));
            break;
        case CS_CMP_END:
            free(cmp->pPrivate);
            iRetVal = 0;
            break;
    }
    return (iRetVal);
}

```

4 **cosort_r()**

This section describes the programming conventions for calling the `cosort_r()` routine from within your application program.

There are five stages in which `cosort_r()` parameters are defined and executed. The first four stages each consist of one definition call, and the final stage has one or more execution calls. Multiple execution calls occur when the calling program sends and receives records via a buffer:

- 1) **Initialization** The sort, merge, or check is defined.
- 2) **Input** The input source(s) are defined.
- 3) **Compare** The sort keys are defined.
- 4) **Output** The output source is defined.
- 5) **Execution** The sort or merge is executed.

4.1 Usage

The `cosort_r()` API consists of three primary calls, which are made in the following order:

- 1) `cosort_alloc()` on page 571.
- 2) `cosort_r()` on page 571.
- 3) `cosort_free()` on page 572.

The primary **CoSort** structure, `cs_cosort_t` is defined in **cosort.h**, and its elements are defined below. The first two elements are integers, and the second two are pointers to characters:

```
struct cs_cosort_s {  
    void* priv;  
    int sw;  
    int mesg;  
    char* porta;  
    char* portb;  
    int total_reccount;  
}cs_cosort_t;
```

4.1.1 cosort_alloc()

`cosort_alloc()` takes no parameters and returns a variable of type `cs_cosort_t`. This call allocates some of the memory resources and initializes variables needed by `cosort_r()` for execution:

```
cs_cosort_t* cosort_alloc();
cs_cosort_t* cosort = cosort_alloc();
```

4.1.2 cosort_r()

`cosort_r()` is the main API call to **CoSort** which is then divided into the five stages where the parameters are set up and the job is executed. All the stages of `cosort_r()` are explained in this chapter, beginning with `cosort_r()` Initialization Stage *on page 572*.

`cosort_r()` is a function that accepts one argument. It is a pointer to the structure `cs_cosort_t`, and it returns a void:

```
void cosort_r(cs_cosort_t* cosort);
cosort_r(cosort);
```

cosort_r() Parameters:

SWITCH

SWITCH is set once by the calling program to initialize a job. A job is defined as a sort, merge or check. SWITCH is then set only by `cosort_r()` as a control parameter. It can signal an error to the calling program, indicate a request for an input record(s), indicate a return of output record(s), or indicate the completion of the job.

After each call to `cosort_r()`, the value of SWITCH will change. SWITCH reports the status of `cosort_r()`, including and errors with specifications. You can check for SWITCH equal to `CS_ERROR` after each call to `cosort_r()`. If there is an error, MESSAGE will contain an error number describing the problem. You can display the error message by using MESSAGE as an index into the `CS_Return[]` array.

MESSAGE

MESSAGE is used by the calling program in the initialization stage to indicate a sort, merge, or check. It is used in the input stage to specify the number of input files, in the compare stage to specify the number of keys, and in the output stage to describe how the output can be directed.

The `cosort_r()` routine also passes back the error number using MESSAGE if an error has occurred.

PORTA

In the initialization stage, the calling program uses `PORTA` to pass **CoSort** performance tuning values. In the input and output stages, `PORTA` is used to supply input and output file names. In the compare stage, it is used to supply the keys. And in the execution stage, the calling program can use `PORTA` to pass input records.

PORTB

In the input stage, `PORTB` is used to set the input record size. On execution, `PORTB` can point to a buffer allocated by the calling program, and is then used by `cosort_r()` to return the sorted records.

4.1.3 cosort_free()

`cosort_free()` is called to free all the memory resources allocated by **CoSort** for the execution of the job. It accepts one parameter, which is a variable of the type `cs_cosort_t`:

```
void cosort_free(cs_cosort_t* cosort);
cosort_free(cosort);
```

4.2 cosort_r() Initialization Stage

To initialize a new job, set `SWITCH` to `CS_INIT` as shown in *Table 20*. A job can be a sort, join, or check.

Table 20: Initialization Call

Parameter	Previous Value		Next Value		Description
	Macro	Value	Macro	Value	
<code>SWITCH</code>			<code>CS_INIT</code>	0	Initiate
<code>MESSAGE</code>			<code>CS_SORT</code>	0	Sort
			<code>CS_MERGE</code>	1	Merge
			<code>CS_CHECK</code>	2	Check
<code>PORTA</code>				ptr.	Work Environment
<code>PORTB</code>				ptr.	Options

`PORTA`, a character pointer, can point to a file name that sets the **CoSort** tuning options. If the pointer is null, the values are taken from values defined in the file specified by the environment variable `$COSORT_TUNER`. If the environment variable is not defined, **CoSort** will search through other assigned locations. The location, format, and meaning of the variables are described in *Using Customized Resource Control Files on page 643*.

PORTB is a character pointer. If not null, it points to an integer that controls sorting options. PORTB can have the following values:

PORTB Parameter		Option
Macro	Bit Value	
CS_NODUPS	0x01	Unique (no duplicates)
CS_STABLE	0x02	Stable (maintain order)
CS_DUPSONLY	0x08	Duplicates Only
CS_MONITOR	(m << 4)	Monitor Level m

For example, to enable unique stable and monitor level 5:

```
options = CS_NODUPS | CS_STABLE | (5 << 4);
```

See MONITOR_LEVEL *on page 650* for a description of monitor levels.

4.3 cosort_r() Input Definition Stage

After a successful initialization call, `cosort_r()` will set a SWITCH equal to CS_INPUT. This value should not be altered by the calling program.

The calling program should use the table below to provide information about the input.

Parameter	Previous Value		Next Value		Comments
	Macro	Value	Macro	Value	
SWITCH	CS_INPUT	1			Define Input
MESSAGE				> 0 = 0 < 0	Number of files Return to calling program Both of the above
PORTA				ptr.	File names
PORTB				ptr.	Record Size

A positive MESSAGE value indicates that `cosort_r()` is to read records from that number of files. PORTA will point to a variable-length record containing a string of file names. (The format of that record is described in Variable-length Records *on page 633*.) A file name is defined as follows:

```
filename[|skip_bytes[|accept_bytes|]
```

where `skip_bytes` is the number of bytes to skip in the file `filename`, and `accept_bytes` is the number of bytes to process. The file name declaration is repeated for each file that is specified by `MESSAGE`.

A zero `MESSAGE` value indicates that `cosort_r()` should return to the calling program in the execution stage to request input records. This is described in `cosort_r()` Execution Stage *on page 582*.

A negative `MESSAGE` combines the above two methods. Its value is interpreted as a positive number for setting the number of input files. After all the files are read, `cosort_r()` will return to the calling program for more input records.

To specify the record length, the fourth parameter points to a short integer. This is 0 for variable-length records, or a value between 1 and 65,535 to indicate the length of fixed-length records.

If the input source is a file, `cosort_r()` assumes the end of a variable-length record to be a linefeed character. `cosort_r()` can also receive input records via a buffer from the calling program in the execution stage. In that case, the input data can consist of variable-length records which need to be set up in the correct format. See Variable-length Records *on page 633* for details on setting up variable-length records.

When fixed-length records are passed to `cosort_r()` by the calling program, no format conversions are necessary. See Fixed-length Records *on page 636* for the format of fixed-length records.



If your input file is empty, you can use a resource control setting to determine the disposition of the output file (see `ON EMPTY INPUT` *on page 652*). By default, when an input file is empty, an output file is created assuming zero-value input data.

4.4 `cosort_r()` Compare Keys Setup Stage

After a successful input definition call, `cosort_r()` will set a `SWITCH` equal to `CS_COMPARE`, indicating that it expects the comparison keys to be defined next. This value should not be altered by the calling program.

The calling program should use the table below to provide information about the comparison keys.

Table 21: Define Comparisons

Parameter	Previous Value		Next Value		Comments
	Macro	Value	Macro	Value	
SWITCH	CS_COMPARE	2			Define Compares
MESSAGE				> 0 0	Number of Keys Use custom record compare
PORTA				ptr.	Key Descriptors Custom record compare function name when MESSAGE = 0
PORTB				ptr.	Custom record compare function pointer when MESSAGE = 0

If the calling program sets a MESSAGE value of 0, it indicates that `cosort_r()` will compare records via a custom user-defined function, `cs_compare()`. For more details on how to write a custom compare function, see `cosort_r()` Custom Compares on page 578.

A MESSAGE greater than 0 indicates that PORTA points to an array of that many key structures. Each element of the array describes one key. `cosort_r()` analyzes these keys in the order specified. Evaluation order during the actual record comparison begins with the first element of the array representing the most significant key. Subsequent keys are invoked in order to resolve equal comparisons.

The PORTB argument is not used during the keys setup stage.

4.4.1 CoSort Key Structure Format

The **CoSort** key structure is declared in **cosort.h**. Each key is specified using the format shown in Table 22 on page 576. Multiple keys simply repeat the format. Table 22 describes the **CoSort** key structure and the values that each element of the structure can contain. The structure is declared in **cosort.h**.

Declaration of the CoSort Key Structure:

```

struct ckey {
  char  k_dir;
  char  k_ftype;
  unsigned short k_pos;
  unsigned short k_length;
  char  k_form;
  char  k_align;
  char  k_type;
  char  k_fsep;
  cs_function_ptr k_custom_proc_funcptr;
  char* k_custom_proc_name;
} cs_key_t;

```

Table 22: Key Format Table

Size	CoSort Key Structure Name	C Macro Value	Value	Meaning
char	k_dir	CS_ASCEND CS_DESCEND	0 1	Direction: Ascending Descending
char	k_ftype	CS_FCHAR CS_FANY CS_FBLANK	0 1 2	Location: Fixed Character Delimited White Space Delimited
short	k_pos		#	Start for Position/Field#
short	k_length		#	Length in bytes
char	k_form	CS_ALPHA CS_NUMERIC CS_DATE CS_TIME CS_TIMESTAMP CS_CUSTOMKEY	0 1 2 3 4	Form: Alphabetic Numeric Date Time Timestamp User-defined key compare. See Custom Key Compares Using /KEY on page 562.
char	k_align (Set to zero(0) for any k_form other than CS_ALPHA)	CS_NOJUST CS_JUSTRIGHT CS_JUSTLEFT CS_NOCASEFOLD CS_CASEFOLD	0x00 0x01 0x02 ~(0x04) 0x04	No Justification Right Justification Left Justification No Casefolding Casefolding
char	k_type	Datatype Name		see DATA TYPES on page 611
char	k_fsep		char	Delimiter Character (if the value for location was CS_FANY)

Table 22: Key Format Table

Size	CoSort Key Structure Name	C Macro Value	Value	Meaning
cs_function_ptr	k_custom_proc_funcptr	Custom Compare Function Pointer		Internal Use Only
char*	k_custom_proc_name	Custom Compare Function Name		Function name for the custom key compare

4.4.2 Non-C Specification

The calling program needs to provide the data described, in contiguous order, as shown in the above table. Depending on your operating system and hardware specifications, the key definitions can be set up and passed to `cosort_r()`.

To replicate the key structure in versions of FORTRAN that do not allow structured data types, declare a `CHARACTER*n` array, where *n* is ten times the number of keys. The two short integers representing key position and length can be associated with bytes 3,4 and 5,6, respectively, of each key segment by using `EQUIVALENCE`. Similar techniques with strings can be used in other languages that do not support data structures.

COBOL permits a data structure and definition that are similar to C. The `COMP-X` and `COMP-6` data types can be employed in Micro Focus COBOL and RM-COBOL compilers, respectively, to impose short integers. For example, Micro Focus COBOL programmers can use the following structure (be sure to specify the `IBMCOMP` compiler directive when compiling):

```

01 CS-KEY [OCCURS n TIMES] .
   05 KEY-DIRECTION  PIC 9      COMP-X.
   05 KEY-LOCATION     PIC 9      COMP-X.
   05 KEY-START-POS  PIC 9(4)   COMP.
   05 KEY-LENGTH     PIC 9(4)   COMP.
   05 KEY-FORMAT     PIC 9      COMP-X.
   05 KEY-ALIGN      PIC 9      COMP-X.
   05 KEY-TYPE       PIC 9      COMP-X.
   05 KEY-FSEP       PIC X.

```



Visual Basic programmers should refer to the example provided in the `\install_dir\examples\API\VB` directory. ◆

4.4.3 *cosort_r()* Custom Compares

The API allows you to perform custom compares at both the record and field level. Although the method for setting up the compares using the API differ for each, the actual prototype for writing the compares and the implementation are the same. The next two sections describe how to set up the API program to perform the compares, and Custom Compare Procedure Declaration *on page 579* details the implementation process.

Custom Record Compares

In the compare keys setup stage, if the calling program has set `MESSAGE` to 0, this indicates a custom record compare. In this case, the calling program can:

- set `PORTB` to the function pointer of the custom record compare function and cast it to the data type `cs_function_ptr` (defined in `cosort.h`). This reference should be resolved during compile time.
- set `PORTA` the function name for the custom record compare, which is resolved by **CoSort** at runtime.

The function pointer takes precedence over the function name. If neither is present, **CoSort** tries to resolve the default function name `cs_compare()` at runtime.

Custom Key Compares

In the compare keys setup stage, the calling program needs to set `k_form` (*See the Key Format Table chapter on page 576*) for a particular key to `CS_CUSTOMKEY` in order to indicate that it is a custom key compare. The calling program can then:

- choose to set `k_custom_proc_funcptr` (*See the Key Format Table chapter on page 576*) directly to the function pointer for the custom key compare function, if using a function which will be resolved at compile time.
- set `k_custom_proc_name` (*See the Key Format Table chapter on page 576*) to the name of the custom key compare function which is resolved at runtime by **CoSort**.

The function pointer takes precedence over the function name. There are no assumed defaults in this case. The variable `k_dir` (*See the Key Format Table chapter on page 576*) is ignored in this case.



The calling program must set `k_custom_proc_funcptr` to `NULL`, if it is not being used, in order to allow correct resolution of the function address at runtime.

4.4.4 Custom Compare Procedure Declaration

This is a common section for the custom record and key compares, and explains the actual implementation.

The custom compare procedure accepts three parameters. The first two are pointers to the records that need to be compared, and the third is a pointer to a **CoSort** internal structure, `cs_cmpdata_t`, defined in **cosort.h**. The return value is an integer indicating which record precedes the other:

```
int procedure_name(char* record1, char* record2, cs_cmpdata_t* cmp)
```

The structure `cs_cmpdata_t` is defined in **cosort.h** as follows:

```
struct cs_cmpdata_s {
    void      *pPrivate; /* Priv pointer for use by external function */
    cs_cmp_mode iMode;
    int        iErrorNum; /* Return error value. For future use */
    int        iRecordLen1; /* Length of first field or record */
    int        iRecordLen2; /* Length of second field or record */
    int        iFieldOffset;
    int        iFieldLen;
};
typedef struct cs_cmpdata_s cs_cmpdata_t;
```

The custom compare function is called in the following stages by **CoSort**

Initialization

CoSort first calls the custom compare using an `iMode` value of `CS_CMP_BEGIN`. At this time, the first two parameters are set to `NULL`. The external function can allocate and initialize the `pPrivate` pointer and any other resources during this call. A return value of 0 indicates a successful initialization by the external routine:

```
procedure_name(NULL, NULL, cs_cmpdata_t* cmp)
```

Compare Processing

CoSort then calls the custom compare using an `iMode` value of `CS_CMP_VALUE`, which indicates that two records or fields are being passed into the function to be compared. During this call, `iRecordLen1` and `iRecordLen2` are set to the lengths of the records or fields that are indicated by the first and the second parameter, respectively:

```
procedure_name(char* ptr1, char* ptr2, cs_cmpdata_t* cmp)
```

The return value is an integer that indicates the precedence of one record or field over the other.

- If the first record precedes the second, return a positive value.
- If the second record precedes the first, return a negative value.
- If both records are equal, return 0.

Termination

After all the records have been compared, **CoSort** calls the custom compare routine one final time with an `iMode` value of `CS_CMP_END`. At this point, the routine can free any allocated resources. A return value of 0 indicates successful completion:

```
procedure_name(NULL, NULL, cs_cmpdata_t* cmp)
```

4.4.5 Building the Compare Procedure for Runtime Linking

If you are executing an API program that has a custom compare, **CoSort** will resolve any references at runtime. In order to do this, the custom compare procedures need to be built into a shared object on Unix or a dynamic linked library on Windows. The environment variable `CS_USER_DLL` should be used to point to the exact location of the library, which has all the custom compare functions.

If `CS_USER_DLL` is not set:

- On Windows, **sortcl** will look for the library **cs_user.dll**, in the current directory or in the **System32** folder.
- On Unix, **sortcl** will look for the library **cs_user.so** or **cs_user.sl** (HPUX) in the shared library path (which can be `LD_LIBRARY_PATH`, `SHLIB_PATH`, or `LIBPATH`).

4.4.6 Example

For an example of a custom record compare, see *Custom Record Compare using `cosort_r()` on page 601*.

For an example of a custom key compare, see *Custom Key Compare using `cosort_r()` on page 605*.

4.5 `cosort_r()` Output Definition Stage

After a successful compare keys setup call, `cosort_r()` will set `SWITCH` equal to `CS_OUTPUT`. This value should not be altered by the calling program.

The calling program should use the following table to provide information for the output destination of the job.

Table 23: Output Options

Parameter	Previous Value		Next Value		Comments
	Macro	Value	Macro	Value	
SWITCH	CS_OUTPUT	3			Define Output
MESSAGE			CS_PROGRAM CS_FONLY CS_UONLY CS_BOTH CS_APPEND	0 1 2 3 4	Return to calling program File Standard Output Both 1 and 2 above Append to file
PORTA				ptr.	Output file name
PORTB					

A MESSAGE value of CS_PROGRAM indicates that `cosort_r()` will return sorted records each time, as they become available. The calling program can then process the records as needed (see `cosort_r()` API Examples *on page 586* for execution examples), without waiting for the sort job to complete.

If MESSAGE is CS_FONLY, PORTA will point to a variable-length record containing the output file name. For a description of the format, see Variable-length Records *on page 633*.

If MESSAGE is CS_UONLY, records are written to the standard output stream.

A MESSAGE value of CS_BOTH combines the actions of CS_FONLY and CS_UONLY.

4.6 cosort_r() Execution Stage

After a successful call to `cosort_r()` to define output options, `SWITCH` remains at the value of `CS_OUTPUT`. `cosort_r()` is now ready to execute the job. The next call will start the execution, as shown in *Table 24*.

Table 24: Execution Call

Parameter	Previous Value		Next Value		Comment
	Macro	Value	Macro	Value	
SWITCH	CS_OUTPUT CS_INPUT CS_EOJ	3 1 5			Begin Execution/ Receive output Input record request Completion
MESSAGE				n 0	# of bytes being passed End of User Input
PORTA				ptr.	Pointer to User Buffer
PORTB				ptr.	Pointer to User Buffer

4.6.1 Simple Execution

If, in the input and output setup stages, the calling program did not set up the options for sending and receiving data via the calling program, `cosort_r()` is called only once for the job execution, and at the end of this, the sort or merge will be completed, and `cosort_r()` should return with a value of `CS_EOJ()` if successful. No buffer references are required here. `MESSAGE`, `PORTA`, and `PORTB` are not needed in this case.

The first call to `cosort_r()` is mandatory, then **CoSort** determines if further calls are necessary, based on how the input and output were set up in the definition stages.

4.6.2 Passing Records to cosort_r() via a Buffer

If the calling program had specified 0, or a negative number for the number of files during the input definition stage, then `cosort_r()` will return at least once with `SWITCH` set to `CS_INPUT (1)`, requesting one or more records.

Your program should either move one or more input records to the buffer, or indicate that there are no more records.

If there are more records to pass, set `MESSAGE` equal to the byte count of the buffer. The records are to be moved into the buffer pointed to by `PORTA`. This could be accomplished by moving either the data to the buffer, or the buffer pointer to the data.

Variable-length Records *on page 633* and Fixed-length Records *on page 636* describe the proper format of variable-length and fixed-length records, respectively.

If there are no more records, set `MESSAGE` to 0 (zero) and call `cosort_r()` to process the prior records.

At this point, if the calling program had not set up the output to be re-directed back through buffers, the sort or merge should return with a `SWITCH` value of `CS_EOJ`, indicating success. Otherwise, it will continue as explained further.

4.6.3 Receiving Records from `cosort_r()` via a Buffer

In the output definition stage, if the calling program indicated that `cosort_r()` should return to the calling program with ordered output records, `cosort_r()` returns with a `SWITCH` value of `CS_OUTPUT` with ordered output records.

`cosort_r()` returns with records in the `PORTB` buffer. The very first return will have a single record in the `PORTB` buffer. On the next call, set the value of `MESSAGE` to the number of bytes the calling program requires to be returned from `cosort_r()`. At the same time, also set `PORTB` to point to a buffer that can contain that same number of bytes.

The records in `PORTB` can be used for any purpose, including writing the block of records to an output file (see `RECORD TYPES` *on page 633* for the format of records being returned).

4.6.4 End of Job

When there are no records left, the sort or merge is successful, and `cosort_r()` will return a `SWITCH` of `CS_EOJ`.

See *Passing Records to `cosort_r()` Using a Buffer on page 590* for an example that incorporates all stages of the `cosort_r()` API.



WARNING!

Following a return from `cosort_r()` that is not an error or a signal that the job is completed, `cosort_r()` expects to be invoked again. It has allocated various resources -- in particular, memory, disk space, shared memory, and semaphores -- that will remain allocated until the job or calling program finishes. In the current release of **CoSort**, the file space is to remain allocated unless an error or the end of job occurs, regardless of the continued execution of the calling program.

4.7 `cosort_r()` Error Handling

If `cosort_r()` returns a SWITCH of `CS_ERROR`, an error situation has been detected.

The value in `MESSAGE` stores the specific error number. All errors are described in `ERROR` and `RUNTIME MESSAGES` on page 660. In order to display the error message, you can pass `MESSAGE` as an index to the array `CS_Return[]` defined in **`cosort.h`**. In the event of an error, the calling program should exit. Attempt to fix the error and then re-execute.

In this event, the sort or merge has been terminated and all the allocated space has been returned. Subsequent calls to `cosort_r()` that do not initiate a new sort or merge will receive an error return with `MESSAGE = CE_CLRE`.

4.8 Final Record Count

A count of input records is maintained by `cosort_r()`. Because it is possible to specify `CS_NODUPS` in the initialization call, this might not be the same as the output record count.

The output record count can be accessed by the calling program via the variable `total_reccount`, which is an integer defined the `cs_cosort_t` structure.

4.9 Linking with `cosort_r()` on Unix or Linux

To resolve the function call to `cosort_r()` in the calling program, it needs to be linked with the static library **`libcosort.a`**, or with or the dynamic library **`libcosort.so`**. These are present in the `$COSORT_HOME/lib` directory by default. If **`libcosort.so`** is present in the directories `/lib`, `/usr/lib`, or `/usr/local/lib` while dynamic linking, the file can be referenced with `-lcosort`.

The program will also require the header file **`cosort.h`**, and the C file **`cs_ret.c`**, if the calling program needs to resolve references to the error messages.

4.9.1 Linking with `libcosort.a`

You can link to the **CoSort** library statically, but if the **CoSort** library changes, it is necessary to link dynamically so that the most current **CoSort** library is linked to at runtime.

For example, the following commands would compile and link the project **`test_link`** with the **CoSort** library (on Solaris)

Static Linking

```
cc -o test_link test_link.c libcosort.a -lm -lpthread -lposix -ldl
```


Dynamic Linking

```
cc -o test_link test_link.c libcosort.so -lm -lpthread -lposix -ldl
```

where `-lposix` must be replaced by `-lrt` if the POSIX library is not available on your system, or you encounter undefined references to AIO symbols.



For 64-bit architectures, depending on your operating system, the options will vary. Some of these options may be as follows:

Solaris: `-xarch=v9`

AIX: `-q64`

HP-UX: `+DD64, +DA2.0w`



WARNING!

You must re-link with the **CoSort** library whenever you install a new patch or new version of the **CoSort** product.

4.10 Linking with `cosort_r()` On Windows

To resolve the function call to `cosort_r()` in the calling program, it needs to be linked with the static library **libcosort_static.lib**, or to the dynamic library **libcosort.dll**. These are present in the *install_dir*\lib directory by default.

The compiler will also require the header file **cosort.h**, and the C file **cs_ret.c**, if the calling program needs to resolve references to the error messages. Some additional libraries including **advapi32.lib**, **kernel32.lib**, and **ws2_32.lib** might be required to resolve external objects.

4.10.1 Linking with the CoSort Library

If compiling using Microsoft Visual Studio, be sure to select the multi-threaded option, and the structure alignment as one-byte.

If compiling with **libcosort_dynamic.lib**, you must define `__DLL_IMPORT__`. Also, note that **libcosort.dll** must reside in the directory from which the executable will be run. If it does not reside here, it must be in the System directory, such as **C:\WINNT\System32**.

For example, to link a project **apples** with **libcosort_static.lib**, you might use:

```
cl apples_program_objects libcosort_static.lib [additional_libraries]
```

**WARNING!**

You must re-link with the **CoSort** library whenever you install a new patch or new version of the **CoSort** product.



This example represents a generic Windows linking command using the Microsoft C/C++ compiler. Depending on your programming language, you might use a different method to link to the **CoSort** library.

4.11 cosort_r() API Examples

This section provides some sample C and COBOL programs that demonstrate how the `cosort_r()` API works. You can find all the example programs in your `examples/API` directory under your install directory.

4.11.1 Simple Sort

This is a basic sort which does not perform any special input or output processing by the calling program.

```

/*****

* Copyright 1978 - 2011 CoSort/Innovative Routines International(IRI), Inc.
* All Rights Reserved.
*
* CoSort API example to pass records using a file
* Input File: chiefs
* Output File: chiefs.out
* Record Length: Fixed
* Key: ASCII characters. positions 1-27, ascending */
*****/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* cosort header file */

#include "cosort.h"

/* File structure to setup the input and output files. The file names need to
   be prepended by the length of the name before they are passed in */
struct fname{
    short len;
    char name[30];
};

struct fname ifile[] = {6,"chiefs"};
struct fname ofile[] = {10,"chiefs.out"};

int main(){

    cs_cosort_t* cs;
    CKEY pkey[1];
    short rlen = 0;

    /* Allocate the main CoSort object */
    cs = cosort_alloc();

    cs->mesg = CS_SORT;
    cs->porta = NULL; /* not needed in this case */
    cs->portb = NULL; /* not needed in this case */

    /* CoSort intialization stage */
    cosort_r(cs);
    if (cs->sw == CS_ERROR){
        printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
        cosort_free(cs);
        exit(1);
    }
}
*****/

```

CONTINUED ON NEXT PAGE...

```

/* Fixed length records */
#ifdef WIN32
    rlen = 48;
#else
    rlen = 47;
#endif

/* CoSort input stage */
cs->mesg = 1; /* the input is a file(s) */
cs->porta = (char*)ifile;
cs->portb = (char*)&rlen;

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
/*****

/* CoSort compare stage */
pkey[0].k_dir = CS_ASCEND;
pkey[0].k_ftype = CS_FCHAR;
pkey[0].k_pos = 1;
pkey[0].k_length = 27;
pkey[0].k_form = CS_ALPHA;
pkey[0].k_type = CS_ASCII;
pkey[0].k_align = '\0';
pkey[0].k_fsep = 0;

cs->mesg = 1; /* have one sort key */
cs->porta = (char*)pkey; /* point to the pkey structure */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
*****/

/* CoSort output stage */
cs->mesg = CS_FONLY; /* send final output to a file */
cs->porta = (char*)ofile; /* point to the output file structure */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
*****/

```

CONTINUED ON NEXT PAGE...

```

/* Basic CoSort execution stage */

cs->porta = NULL; /* not needed */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
/*****/

printf(" \ncosort exits normally. Records count %d\n",
       cs->total_reccount);

/* Free the main CoSort object */
cosort_free(cs);

printf("Press any key to exit\n");
getchar();

return 0;
} /* end of main */

```

CoSort

This example demonstrates how to pass records to `cosort_r()` using a buffer. In this example, the following sequence of events is defined:

- 1) The first four `cosort_r()` calls set up the initialization, input, compare, and output stages.
- 2) The fifth call to `cosort_r()` sets up the execution stage, which then returns to the calling program with a switch value of `CS_INPUT`. This requests the calling program to send the input data via `PORTA` to `cosort_r()`.

`cosort_r()` is called each time the buffer is filled up, in a loop, until the calling program runs out of data.

- 3) `cosort_r()` is called once with `SWITCH=CS_INPUT` and `MESSAGE=0` to indicate the end of input/data.
- 4) `cosort_r()` returns with the first sorted record in `PORTB`.
- 5) `cosort_r()` is called repeatedly until all the remaining output records are obtained.
- 6) `cosort_r()` returns with `SWITCH=CS_EOJ`.

```

/* ** ** ** **
 * Copyright 1978-2011 CoSort / Innovative Routines International (IRI), Inc.
 * All Rights Reserved.
 *
 * CoSort API example pass records in using a buffer
 *   Input Source:      chiefs
 *   Output Destination: stdout
 *   Record Length:     Variable
 *   Key:               ASCII characters. positions 1-27, ascending
 * ** ** ** **
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

/* CoSort header file */
#ifdef (_WIN32)
#define __DLL_IMPORT__ /* IMPORTANT: needed to work with Windows DLL */
#pragma pack(1) /* or compile with /Zp1 option */
#endif /* (_WIN32) */
#include "cosort.h"

/* The following are the getushort() and putushort() macros defined for
different architectures, please uncomment the one you need for your system.
These are be used to add the short to the front of the record for variable
length records */

/* #define DEC */
/* #define RISC */
#define x86

```

CONTINUED ON NEXT PAGE...

```

#ifdef DEC
#define putushort(v,s)  (*((s) +1) = ((v) >> 8), *(s) = (v) )
#define getushort(v,s) (v) = ((long)(s) & 1) ? *(unsigned char *)((s) + 1) * 256 +
*(unsigned char *) (s) : *(unsigned short *) (s)
#else
#ifdef RISC
#define putushort(v,s)  (*(s) = ((v) >> 8), *((s) + 1) = ((v) & 255))
#define getushort(v,s)  (v) =  (unsigned short)((*(s) << 8) | (*(s) + 1) & 255))
#else
#ifdef x86
#define putushort(v,s)  *(unsigned short *) (s) = (v)
#define getushort(v,s)  (v) = *(unsigned short *) (s)
#endif
#endif
#endif

int main()
{
    cs_cosort_t* cs;    /* main CoSort object */
    FILE* inputdata;   /* a record source */
    short  rlen;
    char* ptr;
    short  SS = sizeof(short);
    struct ckey pkey[1]; /* key structure */

    /* allocate CoSort object */
    cs = cosort_alloc();

    /* * * * CoSort intialization stage * * * */
    cs->sw      = CS_INIT;    /* initiate cosort */
    cs->mesg     = 0;         /* signals to do a sort */
    cs->porta    = NULL;      /* not needed */
    cs->portb    = NULL;      /* not needed */

    cosort_r(cs);
    if (cs->sw == CS_ERROR) {
        printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
        cosort_free(cs);
        exit(1);
    }

    /* * * * CoSort input stage * * * */
    rlen = 0; /* variable length records */
    cs->mesg = 0; /* signals input via procedure */
    cs->porta = NULL;
    cs->portb = (char*)&rlen;

    cosort_r(cs);
    if (cs->sw == CS_ERROR) {
        printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
        cosort_free(cs);
        exit(1);
    }
}

```

CONTINUED ON NEXT PAGE...

```
/* * * * CoSort compare stage * * * */
pkey[0].k_dir    = CS_ASCEND;
pkey[0].k_ftype  = CS_FCHAR;
pkey[0].k_pos    = 1;
pkey[0].k_length = 27;
pkey[0].k_form   = CS_ALPHA;
pkey[0].k_type   = CS_ASCII;
pkey[0].k_align  = '\0';
pkey[0].k_fsep   = 0;

cs->mesg = 1; /* signals one key */
cs->porta = (char*)pkey;
cs->portb = NULL;

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}

/* * * * CoSort output stage * * * */
cs->mesg = CS_PROGRAM; /* signals output via procedure */
cs->porta = NULL;
cs->portb = NULL;

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}

/* * * * CoSort execution stage * * * */
/* open the data source */
if ((inputdata = fopen("chiefs", "rb")) == (FILE*)NULL) {
    printf("error reading input file\n");
    exit(2);
}

/* Allocate buffers for input and output.
   Alternatively, you can also use the same buffer for both,
   and point the correct variable to it. */
cs->porta = (char*)malloc(550 * sizeof(char));
cs->portb = (char*)malloc(550 * sizeof(char));

for (;;) { /* endless loop */
    /* Call CoSort for Execution */
    cosort_r(cs);

    switch (cs->sw) {
        case CS_INPUT:
            /* Send record(s) to CoSort */
            cs->mesg = 0; /* total nb of bytes in inbuffer */
            ptr = cs->porta;
```

CONTINUED ON NEXT PAGE...


```

/* Loop until 500 bytes are added to the input buffer */
while (cs->mesg < 500) {
    if (fgets(ptr + SS, 500, inputdata) == NULL) {
        /* end of file */
        break;
    }

    rlen = strlen(ptr + SS); /* rec length */
    /* print the record to stdout */
    printf(" in[%4d]    <%2d> |%.s", cs->mesg, rlen, rlen, ptr + SS);
    putushort(rlen, ptr); /* insert length before actual record */
    cs->mesg += (rlen + SS); /* add length + size of short to
                           total number of bytes*/

    ptr += (rlen + SS); /* advance inbuffer position */
} /* end of while */
break;

case CS_OUTPUT:
    /* Receive a sorted record(s) from CoSort */
    ptr = cs->portb;

    /* Loop until all the bytes received from CoSort
       have been read */
    while (cs->mesg) {
        getushort(rlen, ptr); /* get record length */
        /* print sorted record to stdout */
        printf("out[%4d]    <%2d> |%.s", cs->mesg, rlen, rlen, ptr + SS);
        cs->mesg -= (rlen + SS); /* decrease bytes read from
                               the total */

        ptr += (rlen + SS); /* advance outbuffer position */
    } /* end of while */
    cs->mesg = 500; /* set message value to the maximum number
                   of bytes CoSort should return */

    break;

case CS_ERROR:
    /* Received an error from CoSort */
    printf("\nCsort returns Error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);

case CS_EOJ:
    /* CoSort Job has successfully completed */
    printf("\nCsort exits normally.  Records count   %d\n",
          cs->total_reccount);

    /* Free the main CoSort object */
    free(cs->porta);
    free(cs->portb);
    cosort_free(cs);

    printf("\nPress Enter to exit\n");
    getchar();

    return 0;
}
}
}

```

CoSort

CONTINUED ON NEXT PAGE...

```

/* Copy second file length */
f2len = 7;
memcpy(infiles+8, &f2len, 2);
/* copy second file name */
memcpy(infiles+10, "chiefs2", 7);

/* the input records are variable length */
reclen = 0;

cs->mesg = 2;
cs->porta = infiles;
cs->portb = (char*)&reclen;

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}

/* * * * CoSort compare stage * * * */
pkey[0].k_dir = CS_DESCEND;
pkey[0].k_ftype = CS_FCHAR;
pkey[0].k_pos = 40;
pkey[0].k_length = 3;
pkey[0].k_form = CS_ALPHA;
pkey[0].k_type = CS_ASCII;
pkey[0].k_align = '\0';
pkey[0].k_fsep = 0;

pkey[1].k_dir = CS_ASCEND;
pkey[1].k_ftype = CS_FCHAR;
pkey[1].k_pos = 1;
pkey[1].k_length = 27;
pkey[1].k_form = CS_ALPHA;
pkey[1].k_type = CS_ASCII;
pkey[1].k_align = '\0';
pkey[1].k_fsep = 0;

cs->mesg = 2; /* we have 2 sort keys */
cs->porta = (char*)pkey;
cs->portb = NULL;

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}

/* * * * CoSort output stage * * * */
cs->mesg = CS_UONLY; /* send output to stdout */
cs->porta = NULL;
cs->portb = NULL;

```

CONTINUED ON NEXT PAGE...

```
cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}

/* * * * CoSort execution stage * * * */

cs->mesg = 0;
cs->porta = NULL;
cs->portb = NULL;

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}

if (cs->sw == CS_EOJ){
    printf("\ncosort exits normally.  Records count  %d\n",
          cs->total_reccount);
}

/* Free the main CoSort object */
cosort_free(cs);

printf("Press Enter to exit\n");
getchar();

return 0;
}
```

4.11.4 Custom Input Selection

You can perform any action to produce a record. Horizontal selection can be applied to select a record from the database you are examining. Vertical selection can be applied where you select only those items from each record that are needed for the sort. The record can also be expanded to include additional data.

By trimming the length and the width of the data on input, less data is moved through the sort, enabling it to run faster.

The following example demonstrates both vertical and horizontal selection of records by processing the list of U.S. presidents in the **chiefs** file. Horizontal trimming is accomplished by discarding any president who was not a Democrat. Vertical trimming is done by removing all data except the name and home state fields.

```
/* Input File -- "chiefs"
 * Output File -- stdout
 * Record Length -- fixed
 * Key1: Name, Ascending */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* CoSORT Header File */
#include "cosort.h"

int main()
{
    cs_csort_t* cs;
    FILE *fp; /* source file */
    char orig[50]; /* sort input */

#define NAME orig[0]
#define STATE orig[44]

    CKEY pkey[1];
    short rlen; /* length of the sort record */
    char* ptr;

    /* Open Input Source */
    if ((fp = fopen("chiefs", "r")) == (FILE *)NULL) {
        exit(-1);
    }
}
```

CONTINUED ON NEXT PAGE...

```

/* Allocate main CoSORT object */
cs = cosort_alloc();

/* CoSORT Initialization Stage */

cs->mesg = CS_SORT; /* specify a sort */
cs->porta = NULL;
cs->portb = NULL;

cosort_r(cs);

if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
/*****

/* CoSORT Input Stage */

rlen      = 30; /* record length */
cs->mesg = 0; /* input from program */
cs->porta = NULL;
cs->portb = (char*)&rlen;

cosort_r(cs);

if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
/*****

/* CoSORT compare stage */
pkey[0].k_dir      = CS_ASCEND;
pkey[0].k_ftype    = CS_CHAR;
pkey[0].k_pos      = 1;
pkey[0].k_length   = 27;
pkey[0].k_form     = CS_ALPHA;
pkey[0].k_type     = CS_ASCII;
pkey[0].k_align    = '\0';
pkey[0].k_fsep     = 0;

cs->mesg = 1;          /* have one sort key */
cs->porta = (char*)pkey; /* point to the pkey structure */
cs->portb = NULL;      /* not needed */
                        /* 1 key */

cosort_r(cs);

if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(1);
}
/*****

```

CONTINUED ON NEXT PAGE...

```

    /* CoSORT output stage */
    cs->mesg = CS_UONLY; /* send output to stdout */
    cs->porta = NULL;
    cs->portb = NULL;

    cosort_r(cs);

    if (cs->sw == CS_ERROR){
        printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
        cosort_free(cs);
        exit(1);
    }
    /*****

/* Allocate porta before execution */
cs->porta = (char*)malloc(100 * sizeof(char));

while (1){
    /* Call CoSort for Execution */
    cosort_r(cs);

    switch(cs->sw) {
        /* Send variable length record(s) to CoSort */
        case CS_INPUT:
            cs->mesg = 0; /* total nb of bytes in inbuffer */
            ptr      = cs->porta;

            while (fgets(orig, 48, fp) != NULL)/* e-o-f */{

                /* This is a basic example of selection where
                 the record is modified before being sent
                 to CoSORT */
                /* Look for all Democrats */
                if ((orig[39] == 'D') && (orig[40] == 'E')){
                    strncpy(ptr,&NAME,27);
                    strncpy(&ptr[27],&STATE,3);
                    strncpy(&ptr[30],"\\0",1);

                    cs->mesg = 30; /* set total number of bytes being
                                   passed to CoSORT */
                    break;
                }
            }
            break;

        /* Received an error from CoSort */
        case CS_ERROR:
            printf(" \ncosort returns Error: %s\n", CS_Return[cs->mesg]);
            cosort_free(cs);
            exit(1);
    }
}

```

CONTINUED ON NEXT PAGE...

```
    /* CoSORT Job has successfully completed */
    case CS_EOJ:
        printf(" \ncosort exits normally. Records count  %d\n", cs-
>total_reccount);

        /* Free the main CoSort object */
        free(cs->porta);
        cosort_free(cs);

        printf(" \nPress any key to exit out of the program\n");
        getchar();
        return 0;
    } /* cs->sw ends */

}/* while ends */

} /* end of main program */
```


4.11.5 Custom Record Compare using cosort_r()

```

/*****
* Copyright 1978 - 2011 CoSORT/Innovative Routines International (IRI), Inc.
* All Rights Reserved.
*
* CoSORT API example to use a Custom Record Compare
* File: customcomparerec.c
* Input File: chiefs_sep
* Output File: chiefs.out
* Record Length: Variable delimited
*****/

#include <stdio.h>
#include <stdlib.h>

/* cosort header file */
#include "cosort.h"

int cs_record_compare(char *s1, char *s2, cs_cmpdata_t* cmp);
int main(){

    cs_cosort_t* cs;

    /* File structure to setup the input and output files. The file names need to
    be prepended by the length of the name before they are passed in */
    struct filename{
        short len;
        char name[30];
    };

    struct filename ifile[] = {10,"chiefs_sep"};
    struct filename ofile[] = {10,"chiefs.out"};

    /* Allocate the main CoSORT object */
    cs = cosort_alloc();

    cs->mesg = CS_SORT;
    cs->porta = NULL; /* not needed in this case */
    cs->portb = NULL; /* not needed in this case */

    /* CoSORT intialization stage */
    cosort_r(cs);
    if (cs->sw == CS_ERROR){
        printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]); cosort_free(cs);
        exit(cs->mesg);
    }
    *****/

    /* CoSort input stage */
    cs->mesg = 1; /* the input is a file(s) */
    cs->porta = (char*)ifile;
    cs->portb = 0;

```

CONTINUED ON NEXT PAGE...

```

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("CoSort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}
/*****

/* CoSort compare stage */

cs->mesg = 0; /* Message is 0 indicating a custom record compare call */
cs->porta = strdup("cs_record_compare"); /* custom record compare function name
*/
cs->portb = NULL; /* can be used to pass function pointer if needed */

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}
/*****

/* CoSort output stage */
cs->mesg = CS_FONLY; /* send final output to a file */
cs->porta = (char*)ofile; /* point to the output file structure */
cs->portb = NULL; /* not needed */
cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}
/*****

/* Basic CoSort execution stage */

cs->porta = NULL; /* not needed */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR){
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}

/*****

printf(" \ncosort exits normally. Records count %d\n", cs->total_reccount);

/* Free the main CoSort object */
cosort_free(cs);

printf("Press any key to exit\n");
getchar();

return 0;
} /* end of main */

```

CONTINUED ON NEXT PAGE...

```

int cs_record_compare(char *s1, char *s2, cs_cmpdata_t* cmp)
{
    int LenA, LenB, retval;

    switch(cmp->iMode){
        case CS_CMP_BEGIN:
            cmp->pPrivate = malloc(100);
            retval = 0;
            break;
        case CS_CMP_VALUE:
            LenA = cmp->iRecordLen1;
            LenB = cmp->iRecordLen2;
            retval = strncmp(s1, s2, (LenB>LenA? LenA:LenB));
            break;
        case CS_CMP_END:
            free(cmp->pPrivate);
            retval = 0;
            break;
    }

    return retval;
}

```

The custom compare routine is written a file **customcompare.c**, which is then compiled into **customcompare.so** or **customcompare.sl** (Unix) or **customcompare.dll** (Windows). The environment variable `CS_USER_DLL` is exported to point to the library, which can be resolved at runtime.

A sample custom compare function for Windows may look like this.

```
__declspec( dllexport ) int cs_record_compare(char *s1,
                                             char *s2, cs_cmpdata_t* cmp)
{
    int LenA, LenB, retval;

    switch(cmp->iMode){
    case CS_CMP_BEGIN:
        cmp->pPrivate = malloc(100);
        retval = 0;
        break;
    case CS_CMP_VALUE:
        LenA = cmp->iRecordLen1;
        LenB = cmp->iRecordLen2;
        retval = strncmp(s1, s2, (LenB>LenA? LenA:LenB));
        break;
    case CS_CMP_END:
        free(cmp->pPrivate);
        retval = 0;
        break;
    }
    return retval;
}
```

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* Copyright 1978-2011 CoSort / Innovative Routines International (IRI), Inc.
* All Rights Reserved.
*
* CoSort API example to demonstrate custom record comparison
*   Input File:          chiefs_sep
*   Output Destination: chiefs.out
*   Record Length:       Variable delimited
*   Keys:                 3 keys (custom, internal, custom)
*
* In this example, the custom compare function is passed in as a function
* pointer and resolved during compile time. It can also be compiled
* separately into a dll and passed in as the function name,
* thereby being resolved at runtime
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <stdio.h>
#include <stdlib.h>

/* CoSort header file */
#ifdef (_WIN32)
#define __DLL_IMPORT__ /* IMPORTANT: needed to work with Windows DLL */
#pragma pack(1) /* or compile with /Zp1 option */
#endif /* (_WIN32) */
#include "cosort.h"

/* File structure to setup the input and output files. The file names need to
be prepended by the length of the name before they are passed in */
struct filename {
    short len;
    char name[30];
};

int cs_user_keycmp_ascii(char* a, char* b, cs_cmpdata_t* cmp);

int main()
{
    cs_cosort_t* cs;
    cs_key_t pkey[3];
    short rlen = 0;

    struct filename ifile[] = {10,"chiefs_sep"};
    struct filename ofile[] = {10,"chiefs.out"};

    /* Allocate the main CoSort object */
    cs = cosort_alloc();

    /* * * * * CoSort initialization stage * * * */
    cs->mesg = CS_SORT;
    cs->porta = NULL; /* not needed in this case */
    cs->portb = NULL; /* not needed in this case */

```

CONTINUED ON NEXT PAGE...

```

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}

/* * * * CoSort input stage * * * */
cs->mesg = 1; /* the input is a file(s) */
cs->porta = (char*)ifile;
cs->portb = 0;

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}

/* * * * CoSort compare stage * * * */
/* Key 1 -- Ascending by state */
pkey[0].k_dir    = CS_ASCEND;
pkey[0].k_ftype  = CS_FANY;
pkey[0].k_pos    = 5;
pkey[0].k_form   = CS_CUSTOMKEY; /* indicates a custom key compare */
pkey[0].k_type   = CS_ASCII;
pkey[0].k_align  = '\0';
pkey[0].k_fsep   = '|';
pkey[0].k_custom_proc_name = NULL;
pkey[0].k_custom_proc_funcptr = cs_user_keycmp_ascii; /* function pointer */

/* Key 2 -- Descending by last name */
pkey[1].k_dir    = CS_DESCEND;
pkey[1].k_ftype  = CS_FANY;
pkey[1].k_pos    = 1;
pkey[1].k_form   = CS_ALPHA; /* indicates a regular key compare */
pkey[1].k_type   = CS_ASCII;
pkey[1].k_align  = '\0';
pkey[1].k_fsep   = '|';
pkey[1].k_custom_proc_name = NULL;
pkey[1].k_custom_proc_funcptr = NULL;

/* Key 3 -- Ascending by party */
pkey[2].k_dir    = CS_ASCEND;
pkey[2].k_ftype  = CS_FANY;
pkey[2].k_pos    = 4;
pkey[2].k_form   = CS_CUSTOMKEY; /* indicates a custom key compare */
pkey[2].k_type   = CS_ASCII;
pkey[2].k_align  = '\0';
pkey[2].k_fsep   = '|';
pkey[2].k_custom_proc_name = NULL;
pkey[2].k_custom_proc_funcptr = cs_user_keycmp_ascii; /* function pointer */

```

CONTINUED ON NEXT PAGE...

```

cs->mesg = 3; /* three sort keys */
cs->porta = (char*)pkey; /* point to the pkey structure */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}

/* * * * CoSort output stage * * * */
cs->mesg = CS_FONLY; /* send final output to a file */
cs->porta = (char*)ofile; /* point to the output file structure */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}

/* * * * Basic CoSort execution stage * * * */
cs->porta = NULL; /* not needed */
cs->portb = NULL; /* not needed */

cosort_r(cs);
if (cs->sw == CS_ERROR) {
    printf("Cosort exits with error: %s\n", CS_Return[cs->mesg]);
    cosort_free(cs);
    exit(cs->mesg);
}

printf("\nCosort exits normally.  Records count  %d\n",
        cs->total_reccount);

/* Free the main CoSort object */
cosort_free(cs);

printf("Press Enter to exit\n");
getchar();

return 0;
}

/* Custom key compare routine. This just does a regular ASCII
   compare for example purposes */
int cs_user_keycmp_ascii(char *s1, char *s2, cs_cmpdata_t* cmp)

```

CONTINUED ON NEXT PAGE...

```

{
    int LenA, LenB, retval;

    switch (cmp->iMode) {
        case CS_CMP_BEGIN:
            cmp->pPrivate = malloc(100);
            retval = 0;
            break;

        case CS_CMP_VALUE:
            LenA = cmp->iRecordLen1;
            LenB = cmp->iRecordLen2;
            retval = strncmp(s1, s2, ((LenB > LenA) ? (LenA) : (LenB)));
            break;

        case CS_CMP_END:
            free(cmp->pPrivate);
            retval = 0;
            break;
    }
    return retval;
}

```

4.11.7 COBOL Program with a C Routine that Calls `cosort_r()`

The following example shows how you can call a C routine, from within your COBOL application, that calls `cosort_r()` and the **CoSort** library.

For example, the following COBOL program is named **test.cbl**:

```

*****
*****Micro Focus COBOL  Select here and return here for display*
*****

      IDENTIFICATION DIVISION.
      PROGRAM-ID. SELECTION.
      *REMARKS. MICRO FOCUS COMPILER.
      ENVIRONMENT DIVISION.
      INPUT-OUTPUT SECTION.
      *PROGRAM DIVISION.
      CALL "do_cosort_r".
      STOP RUN.

```

which contains a reference to the C module `do_cosort_r()`. This can be declared in the file **inter.c**, for example:


```

int do_cosort_r()
{
    structfile_inflow_infile;
    structfile_inflow_outfile;
    structckeyw_keys[100];
    int sw, mess;

    strcpy(w_infile.name, "chiefs");
    w_infile.len = strlen(w_infile.name);
    strcpy(w_outfile.name, "chiefs.out");
    w_outfile.len = strlen(w_outfile.name);

    w_keys[0].k_dir = 0;
    w_keys[0].k_ftype = 0;
    w_keys[0].k_pos = 21;
    w_keys[0].k_length = 4;
    w_keys[0].k_type = 8;
    w_keys[0].k_form = 1;

    sw = CS_INIT; mess = 0;
    cosort(&sw, &mess, NULL, NULL);
    if (sw == CS_ERROR) return -1;

    mess = 1; /* infile number */
    cosort(&sw, &mess, (char *)&w_infile, NULL);
    if (sw == CS_ERROR) return -1;

    mess = 1; /* key number */
    cosort(&sw, &mess, (char *)&w_keys, NULL);
    if (sw == CS_ERROR) return -1;

    mess = 1; /* out file number */
    cosort(&sw, &mess, (char *)&w_outfile, NULL);
    if (sw == CS_ERROR) return -1;

    cosort(&sw, &mess, NULL, NULL);
    if (sw == CS_ERROR) return -1;

    return 0;
}

```

You could then compile the COBOL program **test.cbl**, together with **inter.c**, and link with `cosort_r()` as follows:

```
cob -x -o tcob -Q -aarchive test.cbl inter.c libcosort.a
```


APPENDIX

A DATA TYPES

This section lists the data types supported natively by **CoSort**. The method for referencing each data type depends on which program in the **CoSort** package you are using: **sorti**, **sortcl**, or the `cosort_r()` API.

Generally, both a form and type are required. There are five basic forms of data that **CoSort** recognizes, shown in *Table 25*:

Table 25: Forms

Form	Name
0	Alphabetic
1	Numeric
2	Date
3	Time
4	Timestamp

Both the form number and reference type within each form must be declared in the **sorti** program and in API calls to `cosort_r()` (see *Type ASCII (default) or LIST to show options*: on page 523).

In **sortcl**, however, only the data type's name is declared in **sortcl** field statements (see *Numeric Attributes* on page 109). For example:

```
/FIELD= (Price, POSITION=35, SIZE=4, INTEGER)
```

The more than 100 internally supported data types meet the needs of most **CoSort** users. **CoSort** also has the ability to support special data types, such as encrypted data, multi-byte character sets, and ad hoc forms. This is done through user-written procedures (see *CUSTOM INPUT, OUTPUT AND COMPARE PROCEDURES* on page 556 for details). The **sort** program supports only ASCII data, so no type of specification is necessary (see the *sort PROGRAM* chapter on page 481).

A.1 Form 0 -- Alphabetic Data Types

Form 0 types include ASCII, EBCDIC, and *multi-byte character* strings and a special date form.

API users should precede the *Reference Type* with CS_. For example, EBCDIC should be CS_EBCDIC.

Table 26: Single-Byte Types of Form 0

sorti and sortcl Reference Type^a	Reference Standard
ASCII [LATIN1]	ISO 8859-1
EBCDIC	IBM Standard
LATIN2	ISO 8859-2
LATIN3	ISO 8859-3
BALTIC	ISO 8859-4
CYRILLIC	ISO 8859-5
ARABIC	ISO 8859-6
GREEK	ISO 8859-7
HEBREW	ISO 8859-8
TURKISH [LATIN5]	ISO 8859-9
LATIN6	ISO 8859-10
sorti and sortcl Reference Type^a	Examples
ASC_IN_EBC	"123" > "ABC"
ASC_IN_NATURAL	"ABC" > "A-D" (- is ignored)

The ASCII data type has two additional options -- alignment and case folding. The alignment of ASCII characters in the field (none, left, or right) and their case sensitivity determine their exact collating sequence (see *ASCII COLLATING SEQUENCE* on page 680).

Data types MONTH_DAY and ASC_in_EBC (ASCII in EBCDIC sequence) do not have the alignment and case folding options.

Table 27: ASCII Supplement

ASCII options	sorti and sortcl Reference	cosort_r() Reference	Data Example
Not Aligned	ALIGNMENT NONE	CS_NOJUST	" Chars "
Left Aligned	ALIGNMENT LEFT	CS_JUSTLEFT	"Chars "
Right Aligned	ALIGNMENT RIGHT	CS_JUSTRIGHT	" Chars"
Case Sensitive	CASEFOLD YES	CS_NOCASEFOLD	" Chars "
Not Sensitive	CASEFOLD NO	CS_CASEFOLD	" CHARS "

For all Form 0 data types, API users should precede the *Reference Type* with **CS_**.
For example, KEF should be **CS_KEF**.

Multi-byte characters in Form 0 are sorted by the encoding order.

Table 28: Multi-Byte Types of Form 0 (ASCII form)

sorti and sortcl Reference Type	Reference / Encoding Standard	Encoding Standard	Description / Comments
JOHAB	KS X 1001:1992	JOHAB	Korean single-byte separator only
KEF	Korean EBCDIC Format	EBCDIC	Korean single-byte separator only
EHANGUL	IBM DBCS-HOST	EBCDIC	Korean single-byte separator only
HANGUL	Hitachi Hangul	EUC-KR	Korean single-byte separator only
UTF8 / UNICODE (Unicode)	ISO 10646:1993-1	UTF8	Unicode
UTF16 / UCS2 (Unicode)	ISO10646	UTF16	Unicode
UTF32 / UCS4 (Unicode)	ISO10646	UTF32	Unicode
GBK / EUC_CN	ISO10646	GB13000	Chinese national standard
BIG5	BIG5	GBT-12345	Hong Kong Chinese
EUC_TW	CNS 11643-1992 (Planes 1 - 3)	EUC	Taiwan
EUC_KR	KS X 1001:1992	EUC	Korean (WANSUNG)

Table 28: Multi-Byte Types of Form 0 (ASCII form)

sorti and sortcl Reference Type	Reference / Encoding Standard	Encoding Standard	Description / Comments
EUC_JP	Plane 0: JIS X 0201-1976 Plane 1: JIS X 0208-1990 Plane 2: H/W Katakana Plane 3: JIS X 0212-1990	EUC	Japanese
SJIS	JIS X 0208-1990	Shift JIS	Japanese
IBM_DBCS_HOST	IBM Japanese IBM Korean IBM Simplified Chinese IBM Traditional Chinese	IBM DBCS HOST	IBM Simplified Chinese (mainframe encoding)
IBM_DBCS_PC	IBM Japanese IBM Korean IBM Simplified Chinese IBM Traditional Chinese	IBM DBCS PC	IBM Simplified Chinese (PC encoding)

Table 29 through Table 32 show the form 6 multi-byte data types. All Unicode data types shown in these tables use UTF-16, which is a double-byte encoding. Non-Unicode types support a mixture of 8-bit ASCII and double-byte native encodings. Sort orderings are noted in the *Collation Order* columns.



Dynamic libraries (.dll for Windows and .so for UNIX and Linux) are required for all form 6 data types in Table 29 through Table 32. These libraries are provided at installation in the **\$COSORT_HOME/lib/modules** directory. The default location for all **CoSort** library files for Windows is **install_dir\lib\modules**.

Form 6 multi-byte character data types include Chinese Big5, Chinese GBK, Japanese, Korean and UTF16_UNICODE. Data type names specify a subset of the encoding that applies to RowGen and PROCESS=RANDOM. The full range of valid encodings and conversions are available for any data type name used.

THAI_TIS_620 is a form 6 data type and implements Thai Industrial Standard 620-2533 (TIS-620). TIS-620 is a single byte Thai character encoding. The collation order is specified by Thai Royal Institute Dictionary 2525 B.E. Edition.

Conversion between Unicode and native types is possible for characters existing in both data sets.

Collation for native types is in traditional order. For example, Chinese BIG5 is ordered by stroke, while Chinese GBK is ordered by Pinyin (pronunciation). All Unicode types are ordered in dictionary order.

Table 29: Chinese Big5 Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
CHINESE_UNICODE_STROKE	Unicode 5.2.0: http://www.unicode.org/versions/Unicode5.2.0/	Chinese Unicode used in Taiwan, Hong Kong and Macao.	Includes Unicode characters. Includes CJK characters.
CHINESE_BIG5, HK, MO, ROC, TW	Bihua (Stroke) Windows Codepage 950: http://msdn.microsoft.com/en-us/global/cc305155.aspx	General, common usage BIG5.	Includes commonly used characters. Excludes punctuation, symbols, and rarely used characters.
		Commonly used Big5 in Hong Kong.	
		Commonly used Big5 in Macao.	
		Commonly used Big5 in the Republic of China.	
		Commonly used Big5 in Taiwan.	
CHINESE_BIG5_RARE, HK_RARE, MO_RARE, ROC_RARE, TW_RARE	Bihua (Stroke) Windows Codepage 950: http://msdn.microsoft.com/en-us/global/cc305155.aspx	General, rare usage BIG5.	Includes rarely used characters. Excludes punctuation, symbols, and commonly used characters.
		Rarely used Big5 in Hong Kong.	
		Rarely used Big5 in Macao.	
		Rarely used Big5 in the Republic of China.	
		Rarely used Big5 in Taiwan.	
CHINESE_BIG5_DIGITS	Encoding order	Digits.	Includes 0 through 9 non-numeric digit characters.

Table 30: Chinese GBK Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
CHINESE_UNICODE_PINYIN	Unicode 5.2.0: http://www.unicode.org/versions/Unicode5.2.0/	Chinese Unicode used in mainland China and Singapore.	Includes Unicode characters. Includes CJK characters.
CHINESE_GBK_SIMPLIFIED, PRC_SIMPLIFIED, SG_SIMPLIFIED	Pinyin Windows Codepage 936: http://msdn.microsoft.com/en-us/global/cc305153.aspx	General, simplified GBK.	Includes simplified GBK. Excludes punctuation, symbols, commonly used traditional characters, and rarely used traditional characters.
		Simplified GBK used in the People's Republic of China.	
		Simplified GBK used in Singapore.	
CHINESE_GBK_TRADITIONAL, PRC_TRADITIONAL, SG_TRADITIONAL	Pinyin Windows Codepage 936: http://msdn.microsoft.com/en-us/global/cc305153.aspx	General, common usage traditional GBK type.	Includes commonly used traditional characters. Excludes punctuation, symbols, simplified characters and rarely used traditional characters.
		Commonly used traditional GBK used in the Republic of China.	
		Commonly used traditional GBK used in Singapore.	
CHINESE_GBK_TRADITIONAL_RARE, PRC_TRADITIONAL_RARE, SG_TRADITIONAL_RARE	Pinyin Windows Codepage 936: http://msdn.microsoft.com/en-us/global/cc305153.aspx	General, rare usage traditional GBK.	Includes rarely used traditional characters. Excludes punctuation, symbols, simplified characters and commonly used traditional characters.
		Rarely used traditional GBK used in the Republic of China.	
		Rarely used traditional GBK used in Singapore.	
CHINESE_GBK_DIGITS	Encoding order	Digits.	Includes 0 through 9 non-numeric digit characters.

Table 31: Japanese Shift_JIS Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
JAPANESE_ALPHABET, JP_ALPHABET	Dictionary order Windows Codepage 932: http://msdn.microsoft.com/en-us/goglobal/cc305152.aspx	Includes 56 Japanese alphabetic letters.	Includes Shift_JIS and half (ANSII) characters.
JAPANESE_HIRAGANA_BIG, JP_HIRAGANA_BIG		Hiragana upper-case. Includes 12 Hiragana upper-case characters.	
JAPANESE_HIRAGANA_SMALL, JP_HIRAGANA_SMALL		Hiragana lower-case. Includes 71 Hiragana lower-case characters.	
JAPANESE_HIRAGANA, JP_HIRAGANA		Hiragana upper-and lower-case. Includes 83 Hiragana characters.	
JAPANESE_KATAKANA_FULL_BIG, JP_KATAKANA_FULL_BIG		Katakana upper-case. Includes 12 Katakana upper-case characters.	
JAPANESE_KATAKANA_FULL_SMALL, JP_KATAKANA_FULL_SMALL		Katakana lower-case. Includes 71 Katakana lower-case characters.	
JAPANESE_KATAKANA_FULL, JP_KATAKANA_FULL		Katakana upper- and lower-case. Includes 83 Katakana characters.	
JAPANESE_KATAKANA_HALF, JP_KATAKANA_HALF		Katakana (single-byte). Includes 55 Katakana single-byte characters.	
JAPANESE_DIGITS, JP_DIGITS		Digits.	Includes 0 through 9 non-numeric digit characters.

Table 31: Japanese Shift_JIS Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
JAPANESE_UNICODE_ALPHABETIC, JP_UNICODE_ALPHABETIC	Dictionary order Unicode 5.2.0: http://www.unicode.org/versions/Unicode5.2.0/	Includes 52 Japanese Unicode alphabetic letters.	Includes Unicode characters, with Japanese characters sorted in dictionary order.
JAPANESE_UNICODE_HIRAGANA_BIG, JP_UNICODE_HIRAGANA_BIG		Unicode Hiragana upper-case. Includes 71 Hiragana upper-case characters.	
JAPANESE_UNICODE_HIRAGANA_SMALL, JP_UNICODE_HIRAGANA_SMALL		Unicode Hiragana lower-case. Includes 12 Hiragana lower-case characters.	
JAPANESE_UNICODE_HIRAGANA, JP_UNICODE_HIRAGANA		Unicode Hiragana upper-and lower-case. Includes 83 Hiragana characters	
JAPANESE_UNICODE_KATAKANA_FULL_BIG, JP_UNICODE_KATAKANA_FULL_BIG		Unicode Katakana upper-case. Includes 12 Katakana upper-case characters.	
JAPANESE_UNICODE_KATAKANA_FULL_SMALL, JP_UNICODE_KATAKANA_FULL_SMALL		Unicode Katakana lower-case. Includes 71 Katakana lower-case characters.	
JAPANESE_UNICODE_KATAKANA_FULL, JP_UNICODE_KATAKANA_FULL		Unicode Katakana upper-and lower-case. Includes 83 Katakana characters.	
JAPANESE_UNICODE_KATAKANA_HALF, JP_UNICODE_KATAKANA_HALF		Unicode Katakana (single-byte). Includes 55 Katakana single-byte characters.	

Table 32: Korean KSC5601 Multi-Byte Types of Form 6

Data Type	Collation Order / Encoding Standard	Description	Notes
KOREAN_HANGUL, KR_HANGUL	Dictionary order	Include 2,350 commonly used Hangul characters.	Includes KSC506 and ASCII characters.
KOREAN_HANGUL_RARE, KR_HANGUL_RARE	Windows Codepage 949: http://msdn.microsoft.com/en-us/global/cc305154.aspx	Includes 8,822 rarely used Hangul characters.	
KOREAN_DIGITS, KR_DIGITS		Digits.	Includes 0 through 9 non-numeric digit characters.
KOREAN_UNICODE_HANGUL, KR_UNICODE_HANGUL	Dictionary order	Korean Unicode Hangul. Includes 11,172 Hangul characters.	Includes Unicode chars, with Korean characters sorted by Japanese dictionary order

Table 33: Standard Unicode Data Types

Data Type	Collation Order / Encoding Standard	Description	Notes
UTF16_UNICODE	Dictionary order Windows Codepage 949: http://msdn.microsoft.com/en-us/global/cc305154.aspx	Full standard Unicode set	Includes all valid UTF-16 encodings. Should be used with / CHARSET=UTF16
UTF16_DIGITS	Encoding order	Includes 0 through 9 numeric digit characters.	Can be summed and converted to numeric data types.

Table 34: THAI_620 Single-Byte Type

Data Type	Collation Order / Encoding Standard	Description	Notes
THAI_TIS_620	Specified by Thai Royal Institute Dictionary 2525 B.E. Edition		Implements Thai Industrial Standard 620-2533 (TIS-620)

A.2 Form 1 -- The Numeric Types

If you select `NUMERIC` for form and `ASCNUM` for type, the data can be any combination of external numeric, human-readable characters. Numeric data can be used in calculations and can be classified as either external (human-readable) or internal.

External values can be a combination of signed or unsigned integers and reals. Within the specified field, leading spaces are ignored and a trailing space ends the quantity.

Examples of external numeric values are:

```
Integers    +123 -5 12345
Reals       -122.456 0.03+12345.
```

Because these data types can be intermixed, no further specifications are necessary. API users writing in C/C++ should precede the *Reference Type* with `CS_`. For example, `INT` should be `CS_INT`. With the COBOL data types the embedded underscores ("_") should be removed so that `MF_CMP3` should be `CS_MFCMP3`.

Table 35: Numeric Data Types

sorti and sortcl Reference Type		Meaning
Alphanumeric		
0	ASCNUM, NUMERIC	Integers, reals, floating points
C Types		
1	CHAR	Character, Natural (ASCII)
2	SCHAR	Character, Signed
3	UCHAR	Character, Unsigned (EBCDIC)
4	SHORT	Integer, Short Signed
5	USHORT	Integer, Short Unsigned
6	INT, INT32, INT64	Integer, Natural Signed (32- or 64-bit)
7	UINT, UINT32, UINT64	Integer, Natural Unsigned (32- or 64-bit)
8	LONG	Integer, Long Signed
9	ULONG	Integer, Long Unsigned
10	FLOAT	Float, Single Precision
11	DOUBLE	Float, Double Precision



For types 1-9 in *Table 35* on page 620, all **sortcl** /FIELD declarations referencing these data types must contain **SIZE** attributes, even when the fields are delimited (see *SIZE* on page 99).

A.2.1 C Types

The first group of C types are supported by the C library in your computer's operating system. The fields are used by C, FORTRAN, Pascal and similar languages. The exact interpretation of the key field depends on the compiler and the hardware. Values are produced by setting the field according to the caller's specifications.

The specified key has a known length. If the length of the given field is longer than (and an even multiple of) the key type, successive compares will be performed on adjacent keys until the values are unequal or the field is exhausted.

Data types larger than a single byte are affected by endianness. The /ENDIAN statement can be used to write endian-independent scripts. See /ENDIAN on page 79.

Characters

Characters are one-machine-byte long, with no assumptions made about byte width for most comparison cases -- assuming a standard 8-bit width with ranges $-128 \leq \text{signed character} \leq 127$ and $0 \leq \text{unsigned character} \leq 255$. *Natural* characters are either signed or unsigned, depending on the interpretation of `char` used by your machine.

- | | |
|------------------------|------------------------|
| 1. Character, Natural | $1 \leq \text{Length}$ |
| 2. Character, Signed | $1 \leq \text{Length}$ |
| 3. Character, Unsigned | $1 \leq \text{Length}$ |

Natural means that **CoSort** will use the native C library `memcmp()` function to evaluate the keys. *Natural* comparison is faster and should be used whenever collation of meta-characters (those with the most significant bit on) is not a problem. If meta-characters occur, `memcmp()` does not necessarily give the same results as the default `char` type on a machine, on signed or unsigned characters.

On machines without a signed character declaration, performance is degraded by forcing an additional interpretation. On most machines, characters are naturally signed, but `memcmp(a, b, 1)` will return the correct result when:

- $0 \leq a \leq 126$ and $129 + a \leq b \leq 255$
- $28 \leq a \leq 255$ and $0 \leq b \leq a - 128$

Because the results of Character comparisons can differ for meta-characters across machines and across libraries, it is advisable to build test cases for every combination of meta-characters so that you will know what to expect.

Integers and Floats

This category includes:

- multi-byte natural (single-byte ordering only)
- signed and unsigned short and long integers
- signed floats.

For many computers, a multi-byte field must be properly aligned on a memory address. This means that the field starting address must be an even multiple of *n* where *n* is the length of the key. Some operating systems sense misalignment and quickly and correctly shift the field. Most machines with RISC processors do not, however. **CoSort** checks every compare for alignment, and applies correction when needed.

4. Integer, Short Signed $2 \leq \text{Length.}$
5. Integer, Short Unsigned $2 \leq \text{Length.}$

Short integers are stored as 2's complement numbers (in binary form) usually in two 8-bit bytes with ranges $-32,767 \leq \text{signed short} \leq 32,767$ and $0 \leq \text{unsigned short} \leq 65535$. No width assumptions are made. The order of the byte pair (whether higher valued bits are in the first or second byte) is the natural ordering of the hardware.

6. Integer, Natural Signed $4 \leq \text{Length.}$
7. Integer, Natural Unsigned $4 \leq \text{Length.}$

These two data types represent 2's complement integers stored in the width allocated by an `int` declaration with no `short` or `long` adjective. Two and four bytes are common widths. When `INT` or `UNINT` are used, byte ordering is in the natural order of the machine, unless specified with an `/ENDIAN` statement. Specific 32- and 64-bit data types - `INT32`, `INT64`, `UINT32`, and `UINT64` - are also available.

8. Integer, Long Signed $4 \leq \text{Length.}$
9. Integer, Long Unsigned $4 \leq \text{Length.}$

Four bytes is the typical width for this 2's complement data type, with ranges $-2147483648 \leq \text{signed} \leq 2147483647$ and $0 \leq \text{unsigned} \leq 4294967295$.

10. Float, Single Precision $4 \leq \text{Length.}$
11. Float, Double Precision $8 \leq \text{Length.}$

Single Precision and Double Precision Floats are usually placed in four and eight bytes.

ASCII Numeric Data Types

The following are ASCII-numeric data types that are specific to **sortcl**:

- Currency
- IP Address
- Whole Number
- Bit

They will all, by default, display right-justified with a precision of two. **MONEY** and **CURRENCY** can be used interchangeably. They have **MILL** set to *on* and display the monetary symbol for the active *locale* at the beginning of the field.

The following are examples of how the ASCII-numeric data types display in the USA:

ASCII	Numeric	Currency	Currency w/Fill
3.2	3.20	\$3.20	\$*****3.20
150	150.00	\$150.00	\$*****150.00
3.25	3.25	\$3.25	\$*****3.25
9.4562	9.46	\$9.46	\$*****9.46
1023.45	1023.56	\$1,023.45	\$***1,023.45
29384.56	29384.56	\$29,384.56	\$**29,384.56

IP_ADDRESS consists of positive whole number sub-fields separated by a period("."). You can have any number of sub-fields and any number of digits in these sub-fields. When sorting, each sub-field is compared numerically in sequence from left to right. Succeeding sub-fields are only compared if all previous sub-fields have been determined to be equal. Following are examples of valid **IP_ADDRESS** fields:

```
101.5.5.245
1.2.5
301.231.456789.27193.2.34
5
```

WHOLE_NUMBER is an ASCII-numeric data type that displays right-justified and contains only integers.

BIT is used on input fields for which an ASCII or EBCDIC display of bit representation, e.g. 0110010, is required on output. The size of each field declared as **BIT** on input must be multiplied by eight for ASCII or EBCDIC output in order to display the complete bit representation. For more details, see the description of **BIT** on page 125.

Table 36: Types of Form 1

sorti and sortel Reference Type		Data Type
RM COBOL Types		
12	RM_COMP	COMP, Signed
13	URM_COMP	COMP, Unsigned
14	RM_CMP1	COMP-1
15	RM_CMP3	COMP-3, Signed
16	URM_CMP3	COMP-3, Unsigned
17	RM_CMP6	COMP-6
18	RM_DISP	DISP, Signed
19	URM_DISP	DISP, Unsigned
20	RM_DISPSSL	DISP, Sign Leading
21	RM_DISPSSLs	DISP, Sign Leading Separate
22	RM_DISPST	DISP, Sign Trailing
23	RM_DISPSTS	DISP, Sign Trailing Separate
Micro Focus COBOL Types		
24	MF_COMP	COMP, Signed
25	UMF_COMP	COMP, Unsigned
26	MF_CMP3, PACKED, PACKED_DECIMAL	COMP-3, Packed Decimal
27	UMF_CMP3	COMP-3, Unsigned
28	MF_CMP4	COMP-4, Signed
29	UMF_CMP4	COMP-4, Unsigned
30	MF_CMP5	COMP-5, Signed
31	UMF_CMP5	COMP-5, Unsigned
32	MF_CMPX	COMP-X
33	MF_DISP	DISP, Signed
34	UMF_DISP	DISP, Unsigned
35	MF_DISPSSL	DISP, Sign Leading
36	MF_DISPSSLs	DISP, Sign Leading Separate
37	MF_DISPST	DISP, Sign Trailing
38	MF_DISPSTS	DISP, Sign Trailing Separate
Other COBOL Types		
39a	PSIGNF	Packed Decimal

Table 36: Types of Form 1 (cont.)

39b	PD	Packed Decimal
-----	----	----------------

sorti and sortcl Reference Type		Data Type
Miscellaneous		
40	ZONED_DECIMAL	Zoned Decimals
41	ZONED_EBCDIC	Zoned Decimals in EBCDIC
EBCDIC Native RM COBOL Types		
42	ERM_COMP	COMP, Signed_
43	ERM_UCOMP	COMP, Unsigned
44	ERM_CMP1	COMP-1
45	ERM_CMP3	COMP-3, Signed
46	ERM_UCMP3	COMP-3, Unsigned
47	ERM_CMP6	COMP-6
48	ERM_DISP	DISP, Signed
49	ERM_UDISP	DISP, Unsigned
50	ERM_DISPSSL	DISP, Sign Leading
51	ERM_DISPSSLs	DISP, Sign Leading Separate
52	ERM_DISPST	DISP, Sign Trailing
53	ERM_DISPSTS	DISP, Sign Trailing Separate
EBCDIC Native Micro Focus COBOL Types		
54	EMF_COMP	COMP, Signed
55	EMF_UCOMP	COMP, Unsigned
56	EMF_CMP3	COMP-3, Packed Decimal
57	EMF_UCMP3	COMP-3, Unsigned
58	EMF_CMP4	COMP-4, Signed
59	EMF_UCMP4	COMP-4, Unsigned
60	EMF_CMP5	COMP-5, Signed
61	EMF_UCMP5	COMP-5, Unsigned
62	EMF_COMPMX	COMP-X
63	EMF_DISP	DISP, Signed
64	EMF_UDISP	DISP, Unsigned
65	EMF_DISPSSL	DISP, Sign Leading

Table 36: Types of Form 1 (cont.)

66	EMF_DISP_SLS	DISP, Sign Leading Separate
67	EMF_DISP_ST	DISP, Sign Trailing
68	EMF_DISP_STS	DISP, Sign Trailing Separate

A.2.2 RM COBOL Data Types

The following data types can be identified for comparisons.

- 12. RM COMP, Signed
- 13. RM COMP, Unsigned

COMPUTATIONAL variables are stored one digit per byte with a trailing byte for signed data. Each digit is stored as its binary value (that is, 0x01 for 1). The sign byte is 0x0D for negative values and 0x0B for positive values.

14. RM COMP-1

COMPUTATIONAL-1 variables are stored as signed, two byte big-endian (most significant byte first) 2's complement numbers between -32,768 and 32,767. If your machine is big-endian and C shorts are two bytes, it will be faster to use one of the C short integer comparison types.

- 15. RM COMP-3, Signed
- 16. RM COMP-3, Unsigned

A COMPUTATIONAL-3 item, also known as *packed decimal*, is composed of a string of hex digits and a sign. Decimal digits (0 through 9) are held left to right. Each decimal digit is represented as a hex digit with two hex digits per byte.

The last hex digit holds the sign, as shown in *Table 36*:

Table 37: Hexadecimal/Signs

Decimal	Hex	Sign
11	b	Positive
13	d	Negative
15	f	Unsigned (positive)

The sign is the last hex digit so that an odd number or decimal digits needs to be

retained. If there are an even number of digits, a 0 hex digit is prepended to the value to make full bytes.

Table 38 following are sample data representations:

Table 38: Sample Data Representations

Numeric Value	Hex Patterns	Bit Patterns
-123	12 3d	0001 0010 0011 1101
-1234	01 23 4d	0000 0001 0010 0011 0100 1101
+123	12 3b	0001 0010 0011 1100
+1234	01 23 4b	0000 0001 0010 0011 0100 1100

17. RM COMP-6, Unsigned

COMPUTATIONAL-6 is stored just like COMPUTATIONAL-3, but as unsigned values without the need for a sign half-byte.

18. RM DISP, Signed

19. RM DISP, Unsigned

20. RM DISP, Sign leading

21. RM DISP, Sign leading separate

22. RM DISP, Sign trailing

23. RM DISP, Sign trailing separate

USAGE IS DISPLAY values are stored byte-by-byte as the ASCII values for each digit for up to 18 digits. Each digit is in the range 0x30 through 0x39.

SIGN IS SEPARATE causes a leading or trailing byte to be added to the width, with a value of 0x2B (ASCII '+') or 0x2D (ASCII '-').

Included signs cause the leading or trailing byte to be incremented by 16 for positive values if the digit is 1 through 9, to A (0x41) through I (0x49), or to (0x7B) for 0. Negative values increment 1 through 9 to J (0x4A) through R (0x52), and 0 to } (0x7D).

A.2.3 MF COBOL Data Types

These Micro Focus data types can be compared by **CoSort**. The width of data fields is generally based on a maximum of 18 characters in a PICTURE clause.

- 24. MF COMP, Signed
- 25. MF COMP, Unsigned

COMPUTATIONAL, a.k.a. COMPUTATIONAL-4 a.k.a. BINARY, negative values are stored as 2's complement numbers with the most significant byte first. The number of bytes of storage depends on the magnitude of the value (9s in PICTURE) and on the storage mode of the COBOL program which generates the data, as shown in *Table 39*:

Table 39: Picture 9s/Storage

Picture 9s		Storage	
Signed	Unsigned	Byte	Word
1-2	1-2	1	2
3-4	3-4	2	2
5-6	5-7	3	4
7-9	8-9	4	4
10-11	10-12	5	8
12-14	13-14	6	8
15-16	15-16	7	8
17-18	17-18	8	8

If the machine is big-endian, consider using a C, FORTRAN, or Pascal internal type of the same width; it should be faster.

- 26. MF COMP-3, Signed
- 27. MF COMP-3, Unsigned
- 28. MF COMP-4, Signed
- 29. MF COMP-4, Unsigned

Micro Focus COMPUTATIONAL-3, packed decimal, is like Ryan-McFarland (RM/COBOL) COMPUTATIONAL-3 but with 0xC used for the positive values sign in the half-byte instead of 0xB (see *Table 37* on page 626).

- 30. MF COMP-5, Signed
- 31. MF COMP-5, Unsigned

COMPUTATIONAL-5 is like COMPUTATIONAL-4 but the byte order depends on the hardware.



For **CoSort**'s purposes, if you are using big-endian data you should use COMP-4. The COMP-5 algorithm explicitly does little-endian comparisons.

For better performance, consider using a C Library-supported integer type if the data width is suitable.

32. MF COMP-X

COMPUTATIONAL-X is like COMP-4, but its width is the number of 9s in the PICTURE clause and its allowable values are any unsigned integer which fits the width, i.e., 41 significant decimal digits at most.

- 33. MF DISP, Signed
- 34. MF DISP, Unsigned
- 35. MF DISP, Sign leading
- 36. MF DISP, Sign leading separate
- 37. MF DISP, Sign trailing
- 38. MF DISP, Sign trailing separate

Micro Focus DISPLAY values are stored in the same manner as Ryan-McFarland, except with sign included. MF does not alter any bytes when a positive sign is included, and increments by 64, from 0 through 9 (0x30-0x39) to p through y (0x70-0x79).

39a. PSIGNF, Packed Decimal

A COBOL packed data type where the positive signed numbers use H'F' (0xf).

39b. PD, Packed Decimal

A mainframe packed decimal data type for different sign handling. On input, COMP-3 accepts only d as negative, and accepts b, c, f as positive signs. PD accepts both d and b as negative. Note that b is positive with CM-3 and negative with PD.

40. ZONED, Zoned Decimal

41. ZONED_EBCDIC, Zoned Decimals in EBCDIC

Zoned Decimals are alphanumeric digits. If the decimal quantity is negative, the last character is zoned, i.e., written as a lower-case character. Only integers are represented and only integer values are recognized. **CoSort** reads upper- and lower- case characters and writes lower case.

If the quantity is negative and:

the last character is	the string ends with
0	p
1	q
...	...
9	y

Table 40 contains examples of 3 character strings:

Table 40: Decimal/Zoned Decimal

Decimal	Zoned Decimal
1	001
-1	00q
10	010
-10	01p
999	999
-999	99y



In order to pad a zoned decimal output field with zeroes to the `SIZE` given, you must include the attribute `FILL= ' 0 '` (see *FILL* on page 106).

A.2.4 EBCDIC Native RM COBOL Data Types

42 - 53 see *RM COBOL Data Types* on page 626 for descriptions.

A.2.5 EBCDIC Native MF COBOL Data Types

54 - 68 see *MF COBOL Data Types* on page 627 for descriptions.

A.3 Forms 2-4 -- Date/Time/Timestamp Data Types

CoSort recognizes the four distinct ways of describing time, date, and timestamp: American, European, Japanese, and International Standards Organization (ISO).

In *Table 41*, the example column illustrates the different forms possible for input entries. Output values, however, are always in the form of the last example in the table cells.

Table 41: Date/Time/Timestamp Data Types

cosort () Form Type		sortcl and sorti Name	Syntax
2	42	AMERICAN_DATE	<i>month</i> (name or integer)/ <i>day</i> / <i>year</i>
3	42	AMERICAN_TIME	<i>hour</i> [: <i>minute</i>][: <i>second</i>] <i>xM</i>
4	42	AMERICAN_TIMESTAMP	<i>month</i> / <i>day</i> / <i>year</i> <i>hour</i> [: <i>minute</i>][: <i>second</i>] <i>xM</i>
2	43	EUROPEAN_DATE	<i>day</i> . <i>month</i> (name or integer). <i>year</i>
3	43	EUROPEAN_TIME	<i>hour</i> [: <i>minute</i>][: <i>second</i>]
4	43	EUROPEAN_TIMESTAMP	<i>day</i> . <i>month</i> . <i>year</i> <i>hour</i> [: <i>minute</i>][: <i>second</i>]
2	44	JAPANESE_DATE	<i>year</i> - <i>month</i> (name or integer)- <i>day</i>
3	44	JAPANESE_TIME	<i>hour</i> [: <i>minute</i>][: <i>second</i>]
4	44	JAPANESE_TIMESTAMP	<i>year</i> - <i>month</i> - <i>day</i> <i>hour</i> [: <i>minute</i>][: <i>second</i>]
2	45	ISO_DATE	<i>year</i> - <i>month</i> (name or integer)- <i>day</i>
3	45	ISO_TIME	<i>hour</i> [: <i>minute</i>][: <i>second</i>]
4	45	ISO_TIMESTAMP	<i>year</i> - <i>month</i> - <i>day</i> <i>hour</i> [: <i>minute</i>][: <i>second</i>]
2	40	MONTH_DAY	"Jan" < "Feb" and "Wed" < "Thu"



When using AMERICAN_DATE, the attribute SIZE=10 specifies the format MM/DD/YYYY, for example 12/31/2009. SIZE=11 specifies a 3-character representation of the month, for example Dec. You can specify a longer representation of the month by increasing the size, for example SIZE=12 can be used for Dece, and so on.

CoSort also provides two date types for use across the year 2000 mark. These are to be used in conjunction with the MINIMUM_YEAR setting in your Unix **cosortrc** file or

Windows registry. Any date less than the `MINIMUM_YEAR` setting will be considered to occur after the year 1999 and will sort accordingly. If `MINIMUM_YEAR` is not set, 1970 will be the `MINIMUM_YEAR`.

cosort () Form Type		sorti & sortel Name	Syntax
2	75	Y2K_ASCII_YR	<i>2-digit year</i>
2	76	Y2K_ASCII_JULIAN	<i>5-digit julian date</i>

B RECORD TYPES

This section contains descriptions of the formats of variable-length and fixed-length records supported by **CoSort**. Programs passing records to and from the `cosort_r()` routines should conform to these standards.

B.1 Variable-length Records

Variable-length records are used internally by **CoSort** to hold data records and file names. The format is different from the external representation of variable-length records. You will need to know the correct representation when manipulating these records in your program.

B.1.1 Single Records

Variable-length records are held by **CoSort** as character arrays whose leading bytes contain a short integer indicating the length of the record. The minimal length is 0, the maximum is 65,535.

When `cosort_r()` reads a record from a file whose length is declared `variable`, it scans for the line-feed character and constructs a record with the length of the character string only. When `cosort_r()` writes a variable-length record to an output file or to the standard output device, the length characters are removed and the line-feed character is added back, thereby restoring the record to its original form.

An eleven character record `Adams, John` is held as:

1	2	3	4	5	6	7	8	9	10	11	12	13
0b	00	A	d	a	m	s	,		J	o	h	n

`0b 00` is a representation of 11 as a short integer.

Internal data movement is dependent only on the indicated length so that variable-length binary records can be managed. The array that carries the records should be large enough to accommodate the largest record plus the size of the short integer containing that value. On input, the length must be counted and inserted into the record.

B.1.2 Multiple Logical Records (File Names)

Input and output file names are passed as variable-length records. When multiple input file names are required, the length-name pairs are repeated.

As an example, the two files **ABC** and **DEFG** would be passed in a character array as follows:

1	2	3	4	5	6	7	8	9	10	11
03	00	A	B	C	04	00	D	E	F	G

Input file string as a structure

In this example, taken from **breaker.c** in the **examples/API** directory, a structure is used to specify the name of the input file.

```
struct fname {                                /* file name struct */
    short len;
    char name[10];
} ifile[] = { 6, "chiefs" }; /* input file passed in */
```

Note that the `ifile` variable is declared as an array of `fname` structures. These could be repeated for multiple input files to name two copies of **chiefs** like so:

```
struct fname ifile[] = { {10, "chiefs"},
                        {10, "chiefs"}
                      };
```

The `name` field is longer than it seemingly needs to be to align each structure in the array on a four-byte boundary, which some machines, like those based on RISC hardware, need for referencing. Alignment constraints on your machine can vary.

The `len` count was incremented to indicate the full width of `name`; using the smaller value of six would cause unpredictable results. The four bytes which follow each null character that terminates a name are mostly wasted space. In the multiple input file case, however, the nulls are needed to delimit the real end of the file names.

Passing file names in a character array

The following program, **name_builder.c**, demonstrates the building of a character array consisting of multiple logic records:

```
#include <stdio.h>
#include <stdlib.h>

main(int ac, char **av) {

    char   In_Names[1000];
    char   *In_Name_ptr;
    int     Nb_of_Names = 0,
            i,j,t;
    short   sh_length;

    if (ac < 2) {
        fprintf(stderr,"usage: name_builder  name1 name2 ... \n");
        exit(1);
    }
    In_Name_ptr = In_Names;
    Nb_of_Names = ac - 1;

    for (i = 1; i < ac; i++) {
        sh_length = strlen(av[i]);
        memcpy(In_Name_ptr,&sh_length,sizeof(short));
        memcpy(In_Name_ptr + sizeof(short), av[i],sh_length);
        In_Name_ptr += (sizeof(short) + sh_length);
    }
    /* * * * * * IN_Names has Nb_of_Names  names * * * * */
    j = In_Name_ptr - In_Names;
    printf("   Names %3d  length %3d\n    [",Nb_of_Names,j);
    for (i = 0; i < j; i++) {
        t = In_Names[i];
        if (t <= ' ') printf(" x%x",t); else printf(" %c",t);
    }
    printf("]\n");
}
```

Compile and execute:

```
name_builder "chiefs" "abc.def" "A/BB/CCC"
```

which displays:

```
Names   3  length  27
[ x6 x0 c h i e f s x7 x0 a b c . d e f x8 x0 A / B B / C C C ]
```

B.2 Fixed-length Records

When the caller specifies fixed-length records, every record is assumed to have that same given length. **CoSort** reads and writes the records without regard to content. Line-feed and non-ASCII characters have no effect and the reading ends when records of the specified length cannot be read from the specified input file(s).

A fixed-length record is held by **CoSort** as it was read from the caller or from the input stream. In preparing, comparing, or receiving a fixed length, the same format is followed. As an example, an eleven character record Adams, John is maintained as:

1	2	3	4	5	6	7	8	9	10	11
A	d	a	m	s	,		J	o	h	n

C SYSTEM ROUTINES

The following information is useful for those who will be specifying library routines or providing special handling for system interrupts.

C.1 Library Routines

The following library routines are used by **CoSort**.

abort	fstat64	nl_strcmp	strchr
abs	ftok	nl_strncmp	strcmp
access	getenv	open	strcoll
atof	gethostname	printf	strcpy
atoi	getpid	read	strdup
atol	getppid	realloc	strerror
calloc	getrlimit	rewind	strftime
clock	kill	semctl	strlen
close	localtime	semget	strncmp
currlangid	longjmp	semop	strncoll
cuserid	madvise	setbuf	strncpy
dup	malloc	setjmp	strnxfrm
errno	memccpy	setlocale	strstr
exit	memchr	shmat	strtok
fclose	memcmp	shmctl	strtoken
fflush	memcpy	shmdt	syscon
fgets	memset	shmget	time
fopen	mmap	signal	tolower
fopen64	mmap64	sprintf	toupper
fork	msgctl	sscanf	unlink
fprintf	msgget	stat	valloc
fread	msgrcv	stat64	wait
free	msgsnd	statvfs	waitpid
fseek	munmap	strcasecmp	write
fseeko64	nl_langinfo	strcat	



WARNING!

System programmers should be aware that if they replace these routines, the result can influence the performance of programs dynamically linked to the `cosort()` library.

C.2 Signal Processing

It is necessary to catch interrupt signals in order to remove allocated work files when the program executing `cosort()` is about to end prematurely. *Table 42* shows the signals that are processed:

Table 42: Signals and their Meanings

Signal	Meaning
SIGHUP	Hang-up
SIGINT	Interrupt
SIGQUIT	Quit
SIGILL	Illegal Instruction
SIGTRAP	Trace trap
SIGABRT	Abort
SIGFPE	Floating point
SIGBUS	Bus error
SIGSEGV	Segment error
SIGSYS	System call error
SIGPIPE	No pipe reader
SIGALRM	Alarm clock
SIGTERM	Software Terminate
SIGUSRI	User Signal 1

The following function is called:

```
int cs_signal(signal)  int signal;
```

If `cs_signal()` returns 0:

- allocated memory and files are deleted; and,
- the signal is regenerated for subsequent user action.

If `cs_signal()` returns a value other than 0, `cosort_r()` processing continues and no further signal action occurs.

The default `cs_signal()` function returns 0. You can provide your own routine to replace the default. It is recommended that you bind the routine to your application program rather than to **libcosort.a** on Unix, **libcosort_static.lib** on Windows.

D PERFORMANCE TUNING

CoSort is designed to allow the user to control machine resources, and to be a good neighbor on a system with multiple users and applications. There are no mechanisms to assume special privilege, disable interrupts, bypass the kernel, or restrict other programs performing I/O. This *good neighbor* approach assures that multiple sorts and non-sort jobs can run well concurrently. Accordingly, **CoSort** will perform with minimal impact on the system with the default resource control settings provided.

CoSort is capable of symmetric multiprocessing through advanced, POSIX-compliant multi-threading technology. Therefore, the number of available CPUs, threads and disks, and especially the actual amount of installed memory available to the **CoSort** user, will affect throughput in high-volume sorting operations. Therefore, job performance can be improved, or conversely, deprioritized, by changing the:

- balance of memory and I/O
- AIO resources (Unix)
- size of **CoSort**'s I/O buffers
- size and location of overflow files.

Your system administrator can use the advice and instructions in this section to re-tune **CoSort** for faster or slower performance. Manual performance tuning will affect the execution of the standalone **CoSort** programs, and any program that integrates the `cosort_r()` procedure.

Because **CoSort** is designed for high-volume, high-performance sorting in a typical multi-job environment, your expectations for sort performance must be weighed against the needs of other users on the machine. When **CoSort** runs (alone or concurrently with other jobs), you should monitor system and user times, and include the effect of **CoSort** on the total environment when evaluating performance.

Memory Usage

The amount of sorting performed in memory has the greatest impact on the length of time a job takes to run. For a large job, keeping few records in memory and writing many to disk generally causes slower sort performance due to the physical reading and writing of files. It can also cause job failure if there is insufficient disk space for the records.

Conversely, using too much memory can cause thrashing, which also degrades performance. On most computer systems, the best timings are achieved when all the records of the sort can easily be retained in RAM.

For small sorts on lightly loaded machines, sorting in memory works well, especially with the defaults provided by IRI. For larger jobs, or those running on very heavily loaded machines, it is desirable to find new tuning values that will optimize the resources spent. For huge jobs, or large jobs that you expect to run more than once, it might be worthwhile to experiment with various memory-disk schemes to find a good combination. A script is provided in *Benchmarking* on page 659 to suggest how this is done.

The **cs_setup** program works at installation time to find the best amount of total memory and shared memory for sorting. The values generated by setup assume that approximately 35-40% of the machines resources can be given to a sort job.

On a heavily-used system, it is likely that you will not get any memory without swapping. The main concern is that requesting too much memory will thrash the system or lead to complete memory usage. Advanced users will note that **sar** and **vmstat** on various Unix flavors, and **mem** on Windows machines can provide more detailed memory usage information. See `MEMORY_MAX` in *Resource Control Settings* on page 647.

Disk Usage

Overflow occurs in sorting when the input data volume exceeds available or specified memory limits. When the memory is filled, the records in memory are sorted and written to a temporary work file. When all the overflow data are distributed to temporary files, these files and the internal data are merged to produce output. Typically, the required overflow space is less than the size of the input. If the system does allow all the work files to be open at once, it is possible to temporarily need more than the input size.

You are able to control where temporary files will be placed. Where multiple drives are available, you can achieve increased speed, as the files are written to read back in parallel. If your system provides *striping*, the I/O system will automatically accommodate this. Otherwise you should assign different devices as overflow directories. Furthermore, using different physical devices for work space and output will distribute the space requirement in a multi-threaded fashion, avoiding I/O conflicts in the merge phase.

In addition to specifying the location of sort overflow, the capacity of each work area can be set. By restricting the amount of overflow allowed, system administrators can more precisely control the impact and performance of large sort jobs on a busy machine; see *WORK_AREA path* on page 648.

D.1 Tuning CoSort for Windows

During First Time Setup, a set of Windows Registry values are created that contains the default resource control values that **CoSort** uses when you run a job.

Additionally, a template file containing resource control values, **cosort.rc**, resides in `\install_dir`. You can edit this file, or create a separate one, to override the registry settings at the user or job level.

See *Search Order for Resource Controls* on page 645 for details on how **sortcl** prioritizes the recognition of resource control settings.

To access and edit the Windows Registry, you must run **regedit.exe** and select:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Innovative Routines Int'l  
Inc.\CoSort_Version\Global Configuration.
```

Generally, the values you will be most concerned with are `WORK_AREAS`. These are the directories where **CoSort** will put temporary files. For the best performance, it is recommended that you name as many directories as possible which are on *physically separate* devices



You can include multiple entries for any given `WORK_AREA` setting, for example `c:\sortwork1,d:\sortwork2`.

The number of threads used for sorting can be controlled by the values of `THREAD_MAX` and `THREAD_MIN`. Other registry values should not be changed unless recommended by IRI's technical support staff.

D.2 Tuning CoSort for Unix

During First Time Setup, a resource control file is created in the **\$COSORT_HOME/etc** directory, the initial values for which are determined by installer user prompts. This file, **cosortrc**, is used to set the resources that **CoSort** will use by default¹.

It is recommended that you set **COSORT_TUNER** to the name of a resource control file. This allows you to create and modify multiple resource control files, which can be selected for different users, jobs, and system conditions. For details, see *Using Customized Resource Control Files* on page 643.

See *Search Order for Resource Controls* on page 645 for details on how **sortcl** prioritizes the different ways of setting resource control settings.



For optimal tuning, it is recommended that you read the note in *MEMORY_MAX fixed_value or %value or auto or minimize* on page 647.

-
1. Previous versions of **CoSort** used the environment variable **COSORT_TUNER** for this job, and its old syntax will still continue to work.

D.3 Using Customized Resource Control Files

The files **cosortrc** (Unix) and **cosort.rc** (Windows) contain a template of tuner values. At the user and job level, you can customize your own resource control file in either of the following ways:

- edit the existing tuner-values template, and give it a name that is meaningful to a specific job
- create your own tuner-values file and give it a name that is meaningful to a specific job.

When creating your own tuner file, it is necessary to include only those settings that you wish to be used other than the defaults. For example, the following is a valid tuner file entry in Windows:

```
THREAD_MAX      3
```

In this case, if you specify the use of this tuner file, the maximum number of sort processes to be designated for the job will be 3 (for details on all the possible settings, see *Resource Control Settings* on page 647). All other values for that job will be determined in the order described in *Search Order for Resource Controls* on page 645.

U If you are in the C shell, the syntax for specifying a new resource control file to be used is:

```
setenv COSORT_TUNER [path] filename
```

For example:

```
setenv COSORT_TUNER /usr/mis/cosortrc.job1
```

In the Bourne or Korn shell the command would be:

```
COSORT_TUNER=/usr/mis/cosortrc.job1
export COSORT_TUNER
```

You can also use the `/MEMORY-WORK` statement to designate a resource control file. The syntax is:

```
/MEMORY-WORK=" [path] filename"
```

For example:

```
/MEMORY-WORK="/usr/mis/cosortrc.job1"
```

The settings in the file **cosortrc.job1** would be read before the settings in any other resource control file, and therefore take precedence.

API users integrating the sort engine directly can pass a resource file name to `cosort_r()`.

Here is an example in C:

```
char Tuner []="/usr/mis/cosortrc/job1;  
...  
cosort (&sw, &msg, Tuner, null); ◆
```



To specify the use of a new resource control file from a Windows command line, type:

```
set COSORT_TUNER=c:\IRI\CoSort95\bin\filename
```

For example:

```
set COSORT_TUNER=c:\IRI\CoSort95\bin\cosort_job1.rc
```



The complete path is required on Windows when setting `COSORT_TUNER`.

You can also use the `/MEMORY-WORK` statement to designate a resource control file. The syntax is:

```
/MEMORY-WORK=" [path] filename "
```

For example:

```
/MEMORY-WORK="\Progra~1\IRI\CoSort91\etc\cosort.rc2"
```

The settings in the file **cosort.rc2** would be read before the settings in any other resource control file, and therefore take precedence.

API users integrating the sort engine directly can pass a resource file name to `cosort_r()`.

Here is an example in C:

```
char Tuner []="C:\\user\\jobs\\job1";  
...  
cosort (&sw, &msg, Tuner, null); ◆
```

D.4 Search Order for Resource Controls

This section describes the level of priority given to other methods of setting resource controls, for both Unix and Windows. **CoSort** will first evaluate a `/MEMORY-WORK` command or API call for the old tuning values or a file referenced to by `COSORT_TUNER` (see *Using Customized Resource Control Files* on page 643).



Any controls not set by these methods will be searched in order in the files named below. Note that for some directories, **.cosortrc** is the file, and in others, it is **cosortrc**.

Table 43: Search Order—cosortrc

Priority	File
1	./cosortrc (current directory)
2	\$HOME/.cosortrc (user's home directory)
3	/etc/default/cosortrc
4	/etc/cosortrc
5	\$COSORT_HOME/etc/cosortrc
6	/opt/cosort/etc/cosortrc
7	/usr/local/cosort/etc/cosortrc

CoSort will search for each of the above files in order until all the values have been set. Once a value is set, it will not be changed by settings in any subsequent files. It is therefore possible that different variables can be set in different files. Factory defaults will be used for any values not already set.

If you wish to have all the variables set in one place, you should set `COSORT_TUNER` to a file name and make sure that all the variables are set in that file. Remember that you can use any legal Unix name for this file and that it will only be searched in the path designated in the `COSORT_TUNER` declaration. If all the values are not set in this file, a search will be done as shown in *Table 43*.

If you find performance unsatisfactory, run **cs_setup** and choose **Tune CoSort** from the menu. This is the same utility that runs during **First Time Setup**. You can set different values for the tuner variables based on anticipated conditions at the time of the sort. Any values that you do not set yourself will be given factory default values.

You are also given the opportunity to set the path and name for the resource control file being created. In this way, you can generate multiple resource control files, store them in the same location, and then activate the desired one by setting `COSORT_TUNER` or `/MEMORY-WORK` to the appropriate file name. ◆

W *Table 44* shows the order in which **sortcl** prioritizes resource control settings. Factory (registry) defaults will be used for any values not manually set.

Table 44: Resource Control Priority on Windows

Priority	File
1	files pointed to by a <code>/MEMORY-WORK</code> statement
2	resource control files pointed to by <code>COSORT_TUNER</code>
3	settings taken from cosort.rc if it is located in the same directory from which the job is run (that is, the command-line directory)
4	Windows Registry

◆

D.5 Resource Control Settings

This section provides a complete listing, with definitions and syntax, of the variables that can be set in the default **cosortrc** file (or **cosort.rc**), the Windows registry, or some other **CoSort** resource control file.

The names are not case-sensitive and they can be specified in any order. A sample resource control file will be presented after the definitions.

All *size* and *count* values must be positive numbers without commas, though decimal values are supported. *Units* can be designated as K or KB for kilobytes, M or MB for megabytes, G or GB for gigabytes.

If no units are designated, the default units are bytes. Units are not case-sensitive. All values and path/file names can contain references to environment variables.



All resource control settings prior to **CoSort** version 8 are still supported, but their values will not be optimal. It is therefore recommended that you update any older settings (and values) with the settings described here.

THREAD_MAX *count*

The maximum number of threads designated for all processes that make use of multiple threads (when licensed), including sorts, merges, and aggregations. The default **THREAD_MAX** value is the number of cores licensed for CoSort use. Set this number lower to minimize overhead in smaller jobs, or to accommodate the needs of other programs running concurrently. If **THREAD_MAX** is set higher than the number of licensed cores, an error message displays.

THREAD_MIN *count*

The minimum number of threads to allot for a sort or merge before aborting due to a lack of memory. The default is 1. 0 is understood as 1.

MEMORY_MAX *fixed_value or %value or auto or minimize*

The upper memory limit used for internal (in-memory) sorts before they will overflow to disk-based temporary files. After overflow, the same value represents the size of temporary files for sort operations. Note that sorts occur not only when **/SORT** is specified, but also when the **NOT_SORTED** option is applied in either **SET** file invocations (see *Set files* on page 192) or **JOIN** statements (see */JOIN* on page 218). These constitute functional tasks in the same manner as a **/SORT** operation.

At installation, you can set **MEMORY_MAX** to be a literal value, such as 50GB. You can also express this value as a percentage of physically-detected RAM. For example, if

physical RAM is 20,480MB, setting MEMORY_MAX 50% tells CoSort to allocate no more than 10,240MB. You can modify this value in the cosortrc file at any time.

W The default MEMORY_MAX value on Windows at installation is based on 50% of physically-accessible RAM. On 32-bit systems, there is a hardware limitation of 2GB of accessible memory. ◆

U The default MEMORY_MAX value on Unix and Linux systems at installation is based on a percentage of the memory detected when CSMEMTEST is run at installation:

- 10% over 32GB
- 15% over 16GB
- 25% over 2GB
- 35% under 2GB
- 50% under 512MB



WORK_AREA path

Specifies the overflow directories for sort, and join, temporary file read/write activity. You may specify as many WORK_AREA entries as you wish, but the optimal quantity should match your THREAD_MAX value, and their optimal locations would be on different physical drives, each with sufficient capacity and space to hold at least 1x the largest sort data source (e.g., input file). Assigning multiple overflow directories to different physical devices should improve performance because the I/O can occur in parallel (and not conflict). Try also to ensure that WORK_AREA paths are not on the same physical drive as with input or output files.

path can be an environment variable or contain an environment variable within it. The default overflow directory is ./ on Unix or .\ on Windows (current). Be sure to follow your operating system convention for specifying a path, for example D:\tmp\work (Windows) or /tmp/work (Unix).

Work files take the form *CSprocess_number*, and are removed automatically at the completion of a successful job.

WORK_AREA directories/files are written to in round-robin fashion in the order specified until the job is finished. Use the ON_WORKAREAS_FULL option to determine which action to take when one or more areas becomes full during a job.

When you type `sortcl /rc` from the command line, a 0 next to a specified WORK_AREA entry indicates that there is space available, and -1 indicates that there is no space available.

U If your sort input, temporary, and/or output files will be read from and/or written to remote, NFS-mounted drives, then you should add the following entry to the **cosortrc** file:

```
AIO OFF
```

which disables **CoSort**'s use of AIO (Asynchronous Input/Output). By default, AIO ON (enabled) is set. ◆

ON_WORKAREAS_FULL *option*

Determines the pause/resume behavior when one or more specified **WORK_AREAS** (the paths where temporary work files are written to in large sort jobs) has run out of disk space during a **CoSort** job. The options are:

ABORT The default behavior. When a work area(s) is full, the job is aborted, temporary work files are purged, and the job must be re-run. Before restarting the job after an abort, you must either free up space in your specified work area(s), or assign different **WORK_AREAS** entries. See *WORK_AREA path* on page 648.

RETRY_PROMPT The following prompt is displayed when a work area(s) is filled up during a job:

```
out of space in work area: [a]bort or [r]esume?
```

Enter a to abort the job (see **ABORT** above).

If you want to resume, you must first free up space in the specified **WORK_AREAS** (see *WORK_AREA path* on page 648), and then enter r to resume the job.

RETRY_ADD_WORKAREA_PROMPT

The following prompt is displayed when a work area(s) is filled up during a job:

```
out of space in work area: [a]bort, [r]esume, [n]ew
work area?
```

Enter a to abort the job (see **ABORT** above).

To resume the job, you must first free up space in the specified **WORK_AREAS** (see *WORK_AREA path* on page 648), and then enter r to resume the job.

If you enter `n`, you are then prompted as follows:

```
enter path of new work area (empty line to stop):
```

Enter one or more paths, pressing <Enter> after each entry, to add to the list of `WORK_AREAS` (see *WORK_AREA path* on page 648). After you have added one or more path names, press <Enter> again to create an empty line. Each path entered is checked for read/write access, and, if acceptable, added to the list of `WORK_AREAS`. All previous work files are then considered to be non-full, and **CoSort** attempts to resume the job. And if all other work areas are still full, **CoSort** will utilize only the new work areas entered here.

BLOCKSIZE *size [unit]*

Defines the size of the I/O buffer. Optimal blocksize varies depending on the amount of physical memory available in your environment. The default is set to 1200KB on Unix and 2048KB on Windows.

MONITOR_LEVEL *level number*

Set the monitor level to determine the degree of on-screen reporting detail on a **CoSort** job. Monitor events are reported through **stderr**. Setting the monitor at higher levels can adversely impact the efficiency of the sort. The default `MONITOR_LEVEL` is 1. Setting the level with a **sortctl** /`MONITOR` statement will override the global default for that particular job.

This chart describes the output display at each monitor level:

Level	Description
0	no monitoring
1	Show job initiation and job completion of events only. This includes the program itself, the sort processes, and the merge (output write) processes.
2	Includes Level 1 plus the opening and closing of input and output files
3-9	Includes Level 2 plus the opening and closing of temporary files. Each progressive level will show the number of records processed with an increasing degree of frequency.
3	every 1,000,000 records
4	every 100,000 records
5	every 10,000 records

Level	Description
6	every 1,000 records
7	every 100 records
8	every 10 records
9	every 1 record



High monitor levels degrade large file processing performance due to screen I/O overhead.

LOG [*path/*] *filename*

Runtime configuration and sort timing information can be sent to a self-appending log file. By showing the actual system values used during execution, the log file can be used in benchmarking and performance analysis. Specifying different filenames in other resource control files will create additional log files.

When an error prevents a job from completing, the log file is not created. Instead, you can refer to the file **.cserrlog** which is created and overwritten with each job. The location of **.cserrlog** is also determined by the *path* specified here. If you do not specify a path, any log and **.cserrlog** files are written to the current working (user) directory.



The permissions assigned to the *path* that can be specified here are the same permissions given to the **.cserrlog** file.

By the time a **sortel** job script is processed, **.cserrlog** has already been opened and cannot be changed. Therefore, any LOG setting existing in a file pointed to by a /MEMORY-WORK statement (inside a **sortel** job script) cannot affect the path to where **.cserrlog** is written (see *Resource Control File* on page 44). You can specify the path of **.cserrlog** only within the default **cosortrc** file, or by using \$COSORT_TUNER (see *Using Customized Resource Control Files* on page 643).

MINIMUM_YEAR *2-digit year*

CoSort supports a *sliding century window* with two data types for specifying a minimum century year, which recognizes or facilitates remediations of pre-Year 2000 data. The relevant CoSort data types are CS_Y2K_ASCII_JULIAN (5-digit Julian date) and CS_Y2K_ASCII_YR (2-digit year). Any dates less than the minimum date specified are considered to be part of the current century, and sort accordingly. The default MINIMUM_YEAR is 70.

ON_EMPTY_INPUT *option*

Determines the disposition of output files when all input files in a **sortcl** job are empty (that is, when they exist but contain 0 bytes). The options do not apply to piped input or input procedures, or when any single `/INFILE` in a multi-source list contains records. The following options are supported:

PROCESS_WITH_ZEROS

The default. The output file is created assuming zero-value input data. The output file will not be empty. It can contain zero values for summary columns and other requested **sortcl** formatting data.

EMPTY_OUTPUT

An empty output file (with zero bytes) is created.

UNCHANGE_OUTPUT

If an output file of the same name already exists prior to **CoSort** processing of empty input, the original output file will be unchanged.

DELETE_OUTPUT

If an output file of the same name already exists prior to **CoSort** processing of empty input, the original output file will be deleted. No new output file will be created.



For all actions except the default, a warning message will be issued if any input file is empty, and that file will be ignored.

OUTPUT_TERMINATOR *option*

Determines the record terminator used when your output is specified as variable-length, and when `/PROCESS=RECORD` (see *DATA SOURCE AND TARGET FORMATS (/PROCESS)* on page 53). This option is ignored when your output is specified as fixed-length. You can include this setting when you want to do any of the following on output:

- change variable-length records from the Windows format to the Unix format, or vice versa
- add special characters to terminate records
- prevent double-linefeed terminators.

The following options are supported:

- INFILE** The default behavior. When input is variable-length, this option uses the same terminator as the input file, that is, either a linefeed (LF) or a carriage return-linefeed (CRLF). For fixed-length input, the default output terminator is LF. Therefore, use the **DOS** option (see below) if you prefer CRLF.
- DOS** A special case of `character(s)` equal to `\r\n`, that is, a carriage return-linefeed.
- Unix** A special case of `character(s)` equal to `\n`, that is, a linefeed.
- character(s)**
The character, or character string, that you specify -- possibly the null string "" -- is appended to each output record.



Specifying "" (null string) as the output terminator character can be useful when input data is declared as fixed-length (which is done to save processing time when all records are of equal length). In these cases, using the null string for the output terminator character will prevent double-linefeeds in the output when converting from fixed- to variable-length.

AUDIT [*path*]filename

sortcl can produce a self-appending log file, in XML format, that contains comprehensive job information for the purposes of auditing. Auditing is enabled only when the **AUDIT** entry is included in the **cosortrc** file (or Windows Registry).

An audit record is appended to the log after each **sortcl** invocation, and includes statistical information regarding the job and the complete **sortcl** job script. Additional entries/lines that do not appear in the original job script (that is, entries referenced via one or more **/SPEC** commands) are expanded and included in the audit log. An environment display shows the literal equivalents of all environment variables used in the job script. The setting is:

```
AUDIT [path] filename
```

where *path* is the directory that will contain the self-appending audit file, and *filename* is the name of the audit file. It is recommended that the file name is given the extension **.xml** to conform with its file type, for example:

```
AUDIT /home/compliance/audit/audit_trail.xml
```

To disable auditing, you must remove the `AUDIT` entry from the **cosortrc** file (Windows Registry).



You can assign an environment variable in place of the path, for example:

```
AUDIT    $syspath/audit_log.xml
```

For details on the components of the audit file, and how to write **sortel** job scripts that query XML audit logs, see *AUDITING* on page 82.

D.5.1 Optional Settings

The following rc settings are not displayed upon installation of **CoSort**. Users who require non-default behavior must manually add these settings and the appropriate values to their **cosortrc** file (or Windows registry or Windows **cosort.rc** file).

ON_MISSING_OUTLENGTH *option*

(This option applies only to users migrating from a version of **CoSort** earlier than 8.1.1.). You might want to preserve the default behavior of earlier versions regarding the disposition of the output file(s) when an output length is not specified. The following options are supported:

<code>VARLENGTH</code>	The default behavior. Indicates that the output will be variable-length. In this case, the output records will be terminated by characters specified by the <code>OUTPUT_TERMINATOR</code> rc setting (see <i>OUTPUT_TERMINATOR option</i> on page 652).
<code>INLENGTH</code>	Indicates that the output will have the same length specified for the input file (or in the last input file if more than one are defined). If the input file was declared as variable-length, then variable-length is preserved on output (use the <code>OUTPUT_TERMINATOR</code> rc option to set the default terminator character). If the input file was specified as fixed length, then this fixed length is preserved on output.



This option is relevant only to users who are migrating from a pre-version 8 **CoSort**. For all other users, failure to specify a record length on output defaults to variable-length output (the default behavior). When fixed-length input records terminate with invisible characters, it is common to miscount the length of the records. A simple way to verify record length is to divide the file size by the record length you intend to specify. The quotient (the number of records) must be an integer.

OUTRECLENGTH

Option outputs records using the length of the output record and with no line termination characters added to the end of the record.

```
PREVENT_DOUBLE_TERM ON
```

(This option applies only to users migrating from a version of **CoSort** earlier than 8.1.1.) Include this setting when a **sortcl** script contains a fixed `/LENGTH` input to define equal-length records that are LF- or CRLF-terminated, and when output is `/LENGTH=0` (or no length specified).

The default behavior in **CoSort** version 7 and earlier resulted in records with single LF or CRLF terminators in these cases. The current default behavior is to add an additional LF or CRLF. Therefore, include this parameter and set it to `ON` (it is `OFF` by default) to avoid the double LF or CRLF when running such scripts.

USE_RECORDCOUNT_API

(This option applies only to users migrating from a version of **CoSort** earlier than 8.1.1.) Include this parameter and setting if you have written API programs that call the `cosort()` or `mcs()` API prior to **CoSort** version 8. Because the initialization calls in the current version now use byte counts by default, you can use this **cosortrc** parameter to instruct **CoSort** to use the old record count method instead, assuring backward-compatibility with applications using these libraries.

You should add this parameter setting if you use **CoSort** to replace the SAS sort on Unix (see the *USING COSORT WITH THE SAS SYSTEM* chapter on page 499), or if you use any other third-party software that was written with calls to the `cosort()` or `mcs()` API prior to **CoSort** version 8 (that is, prior to 2003).

U AIO ON

If your sort input, temporary, and/or output files will be read from and/or written to remote, NFS-mounted drives, then you should add the following entry to the **cosortrc** file:

```
AIO OFF
```

which disables **CoSort**'s use of AIO (Asynchronous Input/Output). By default, AIO `ON` (enabled) is set. ◆

ON_EMPTY_OUTPUT *option*

This parameter determines the disposition of output files when the target of a **sortcl** job is created, but contains 0 bytes. The following options are supported:

PROCESS_WITH_ZEROS

The default. The output file is created assuming zero-value output data. It can contain zero values for summary columns and other requested **sortcl** formatting data.

EMPTY_OUTPUT

An empty output file (of zero bytes) is created.

U **AIO_RETRY_TIMEOUT** *time_value unit*

Sets the amount of time before an error is generated. It provides a time-out -- that is, a duration of time to wait to retry the AIO request in the case of a resource shortage. If the request does not succeed by the amount of time you specify, then **sortcl** will return a CE_AIORES error. The default is 60 seconds. You can assign a *time_value* such as 20, and then a unit, which can be any of the following:

s or sec Seconds.

m or min Minutes.

h or hr Hours.

If the *time_value* is set to 0, **sortcl** will error out immediately. ◆

SUMMARY_OVERFLOW_BREAK 1

Include this option to force overflow values of numeric fields to be written to a new, subsequent record when the specified size of an output field is too small to accommodate its full value. This is particularly useful for aggregation when large sums are computed, and the output field is not sized appropriately.

The following data types are supported when applying this option:

- NUMERIC
- MF_COMP
- MF_CMP3
- MF_DISP
- ZONED_DECIMAL

Note that only the overflow characters (from one or more fields of the supported data types) are written to any new, subsequent record, and non-affected fields are left empty in such records.

ENDIANNESS option

This parameter specifies the endianness of the input source(s) and output target(s). It is used when the input and/or output PROCESS type is `VARIABLE_SEQUENTIAL` (or `VS`). See *VARIABLE_SEQUENTIAL (or VS)* on page 57. The options are:

- `MACHINE` The default. Uses the endianness of the machine on which **CoSort** is running.
- `BIG` Big-endian.
- `LITTLE` Little-endian.

ON_COLLATION_FAILURE option

When sorting fields with a multi-byte data type, this parameter determines **sortcl**'s behavior when key field values are encountered that are not included in that data type's collation sequence. The options are:

- `FAIL` The default. The job terminates.
- `WARN` One or more warnings is displayed for each record that cannot be sorted; `WARNINGSON` must be specified (see *29.4.Runtime Warnings (/WARNINGSON and /WARNINGSOFF)*). Non-sortable records appear at the beginning of the output file (with an ascending sort), or at the end of the output file (with a descending sort).
- `IGNORE` Warnings are not given and the job continues. Non-sortable records appear at the beginning of the output file (with an ascending sort), or at the end of the output file (with a descending sort).

ON_CONVERSION_FAILURE option

When performing data type conversion with multi-byte fields, some characters might not have character equivalents between the source and target data types. This parameter determines **sortcl**'s behavior when this occurs:

- `FAIL` The default. The job terminates.
- `WARN` One or more warnings is displayed for each record containing the field(s) that has no equivalent conversion character; `WARNINGSON` must be specified (see *29.4.Runtime Warnings (/WARNINGSON and /WARNINGSOFF)*). Non-convertible characters keep their original encoding values.
- `IGNORE` Warnings are not given and the job continues. Non-convertible characters keep their original encoding values.

Example Resource Control File

The following is an example of **cosortrc** (on Unix) with default settings:

```
# CoSort 9.5.1 resource control file
# This file can be modified to tune sort performance.
# See the CoSort User Manual, Appendix D for help.
#
# Created by user root
# MET 02:58:23 PM Friday, Feb 26 2011
# If you copy or modify this file, change the name and/or date above
THREAD_MAX 4                                # Maximum number of sort threads
THREAD_MIN 1                                # Minimum number of sort threads
# CSMEMTEST RESULT      = 1837105152
# SYSTEM DETECTED RAM    = 4227330048
# MANUALLY REPORTED RAM = 4294967296
MEMORY_MAX 918552576      # Memory Setting
WORK_AREA /t1/tmp
ON_WORKAREAS_FULL ABORT    # Pause/resume behavior
# WORK_AREA x               # Additional overflow drive
BLOCKSIZE 1228800          # IO buffer size
MONITOR_LEVEL 1            # Runtime monitor level
LOG /opt/cosort91/etc/cosort.log # Self-appending runtime log
MINIMUM_YEAR 70            # Century window
ON_EMPTY_INPUT PROCESS_WITH_ZEROS # Output file option
OUTPUT_TERMINATOR INFILE   # Output terminator
# EOF
```

W The above example would appear the same in a **cosort.rc** file (on Windows) with the exception of the way **WORK_AREA** directories are specified and the actual values for certain settings such as **THREAD_MAX**, and **BLOCKSIZE**. ◆

D.6 Benchmarking

Given a recurring job in a constant mix of running programs, it is possible to find new tuning variables that can further minimize throughput time.

Since there are several tuning values which affect performance, it makes sense to isolate one or two that seem to make the most difference in a given job or user mix (usually `THREAD_MAX` and `MEMORY_MAX` in your particular runtime environment. These and other values can be modified by hand or through benchmarking scripts, an example of which is provided below.

For small jobs or those only needing a few configuration change tests, ad hoc modifications of resource control file values will suffice, and the running log file will record actual values in use, and their corresponding performance results.

For large jobs with multiple configuration tests, the use of automatically changing values is indicated. This can be accomplished by declaring the resource control values as environment variables, which are then changed through a separate shell script run as a batch command.

For example, we first decide to assign the `THREAD_MAX` value to the environment variable `$CPU`, and the `MEMORY_MAX` value to the environment variable `$MEM`, and build them into a shell script that will invoke changing resource control values. The following is a sample batch script for the Unix Bourne shell that executes a **sorti** program using the resource control file **cosortrc.no1** in the current directory:

```
#!/bin/sh
echo "CoSort Tuner Values Changer"

COSORT_TUNER=cosortrc.no1
export COSORT_TUNER

for CPU in 2 4 6 8 10 12; do
    for MEM in 50MB 150MB 300MB 500MB; do

        export CPU
        export MEM
        echo THREAD_MAX=$CPU
        echo MEMORY_MAX=$MEM

        sorti cs_spec.no1

    done
done
```

The **cosortrc.no1** file contains:

THREAD_MAX	\$CPU
THREAD_MIN	1
MEMORY_MAX (\$CPU)	\$MEM
WORK_AREA	/tmp
BLOCKSIZE	122880
MONITOR_LEVEL	4
LOG	rclog.no1
MINIMUM_YEAR	1970
ON_EMPTY_INPUT	PROCESS_WITH_ZEROS
OUTPUT_TERMINATOR	INFILE

Note that if the sort script were to be set up in **sortcl**, this command would be added to the specification file:

```
/MEMORY-WORK="cosortrc.no1"
```

and the batch testing (shell) script on the previous page.

At the end of the batch cycle, review the actual values and performance results in the **rclog.no1** file to determine the best range for the tuner values. Once this *saddle* area is found, recast the range around that area with smaller increments and re-run the batch script. Eventually, this method will identify the best tuner values for a repeated job of this kind on your system. Remember to also look at system and user times, and consider the impact your sorts have on concurrent jobs.

E ERROR and RUNTIME MESSAGES

This section contains a table of **CoSort** error and runtime values and messages (see *Detailed Error and Runtime Messages* on page 669 for elaboration on each message).

E.1 Table of Error Values

The errors and messages in *Table 45*, presented in value order, can occur throughout the **CoSort** suite (see *E.2.Detailed Error and Runtime Messages* for an alphabetical listing of messages). When an error occurs, batch programs stop with an error message, the system status error is set, and work files are purged. Depending on how your operating system was generated, fatal job or operator intervention errors might not be caught by **CoSort** and may go on to cause undesirable results. In these cases, contact your IRI agent for assistance.

Table 45: Error Values

Value	Message	Meaning
0	Normal Return	N/A
1	Reserved for future use	N/A
2	Insufficient Memory	Required memory space could not be dynamically allocated.
3	Unknown Exception	An internal error detected due to invalid data and/or specifications. Check whether the input data and specifications are valid. See <i>Unknown Exception Error -- Logging</i> on page 680. Contact IRI Support if you cannot resolve the problem.
4	Invalid Format Definition	Not currently returned.
5	File Creation Error	A given file cannot be created. Make sure you have access to the directory and that the directory has enough room to create the file.
6	Reserved for future use	N/A
7	Error in WORK_AREA	WORK_AREA specification in your resource control file is invalid or cannot be used.
8	Unknown Action Requested	A job was specified which was not CS_SORT, CS_MERGE, or CS_CHECK.
9	Uncorrected Error Condition	cosort_r() was called without a CS_INIT call after it reported an error.
10	Parameter Error	Indicates that a routine was called with an illegal parameter. Check your sortcl specification.
11	Record Length Improper	A record length less than 0 was specified.
12	Invalid Number of Keys	A number of keys less than 0 was specified.
13	Invalid Key Specification	Key location was not specified as CS_FCHAR, CS_FANY, or CS_FBLANK.
14	Merge Invalid Number of Files	CoSort could not open all of the workfiles that it created.
15	Unspecified Error	Not currently returned.

Table 45: Error Values (cont.)

Value	Message	Meaning
16	Invalid Direction	A direction which was not CS_ASCEND or CS_DESCEND was specified.
17	Invalid Field Position	A key position less than 1 was given.
18	Invalid Record Length	A fixed key position plus its length goes past the end of a fixed-length record. A variable-length record shorter than the highest fixed key position was read. A variable-length record longer than 65535 bytes was read.
19	Unknown Alignment Type	A CS_ALPHA key had k_align not CS_NOJUST, CS_JUSTLEFT, or CS_JUSTRIGHT.
20	Unknown Case Type	A CS_ALPHA key did not use CS_NOCASE or CS_CASEFOLD
21	Improper Format Declaration	A CS_NUMERIC CS_INTERNAL key was specified with field positioning. A CS_NUMERIC CS_INTERNAL k_form was named which does not exist. A key was not CS_ALPHA or CS_NUMERIC.
22	Field Type vs. Length Wrong	A CS_NUMERIC CS_INTERNAL key had an unreasonable k_len for its form.
23	Output Type Unknown	Output was not stdout, file, both or returned to caller.
24	Problem with User's Output File	An error occurred when writing data to the final output file.
25	Invalid Value	An invalid value for the given data type/task/feature has been specified.
26	Unspecified Error	Not currently returned.
27	Please Answer Yes or No	You have responded improperly to a prompt, and it requests a decision.
28	Environment Variable Undefined	Used by cosort_r() and sortcl.
29	Source Unknown	sortcl syntax error.
30	Unexpected	sortcl syntax error.
31	Wrong Combination of Items	sortcl syntax error.
32	Divide By Zero	Division by zero (mathematical error).
33	Terminated	Program execution terminated by Ctrl-C.
34	Insufficient Disk Space for Output File	Total input bytes cannot fit on disk where output file is specified.
35	Insufficient disk space or file write permission denied for work file(s)	Total input bytes exceed (maxmemory + total temp area) or no write permission for specified WORK_AREA path/filename
36	Insufficient Disk Space for Both Output File and Work File(s)	Two times total input bytes is greater than total output area, where output area overlaps work area.

Table 45: Error Values (cont.)

Value	Message	Meaning
37	Maximum Number of Sort Threads Exceeded	Trying to run more sort threads than your license allows.
38	Unspecified Error	Not currently returned.
39	Unspecified Error	Not currently returned.
40	Unspecified Error	Not currently returned.
41	Unspecified Error	Not currently returned.
42	Unspecified Error	Not currently returned.
43	Unspecified Error	Not currently returned.
44	Unspecified Error	Not currently returned.
45	Unspecified Error	Not currently returned.
46	License Violation: Incorrect Node or Invalid Key	Contact your IRI agent.
47	License Violation: Expiration Date Passed or Invalid Key	Contact your IRI agent.
48	License Violation: Invalid Key	Contact your IRI agent
49	License Error: Cannot Obtain Machine ID.	License manager is unable to identify its location. Contact your IRI agent.
50	Reserved for future use	N/A
51	Invalid Resource Variable in Resource Control File	Missing or non-writable overflow directory.
52	COSORT_HOME is Not Set in the Environment	Set the environment variable COSORT_HOME to the CoSort home directory.
53	Invalid Work Area Specified	Change or reset permission in WORK_AREA.
54	License Violation: This Application is Not Licensed	Contact your IRI agent.
55	System Error	Monitor error message.
56	csio Error	Monitor error message.
57	Initiated	Monitor event message.
58	Completed	Monitor event message.
59	Infile Opened	Monitor event message.
60	Infile Closed	Monitor event message.
61	Outfile Opened	Monitor event message.
62	Outfile Closed	Monitor event message.

Table 45: Error Values (cont.)

Value	Message	Meaning
63	Workfile Opened	Monitor event message.
64	Workfile Closed	Monitor event message.
65	Records Processed	Monitor event message.
66	Process Begins	Monitor event message.
67	Process Ends	Monitor event message.
68	Left Right	Monitor join event message.
69	Accepted Rejected	Monitor event message.
70	Too Many Errors -- Aborting	sortcl script analyzer stops after 8 syntax errors. See <i>/ERRORCOUNT</i> on page 280 to override.
71	Incomplete Command	Missing part(s) of a command. See the <i>sortcl PROGRAM</i> chapter on page 37.
72	Expecting Names	sortcl syntax error.
73	Parenthesis Count	sortcl syntax error.
74	Missing Double Quote Mark	sortcl syntax error.
75	Duplicate Name	sortcl syntax error. FILLER, as a field name, is exempt from this error.
76	Expecting	sortcl syntax error.
77	Expression Syntax	sortcl syntax error.
78	Source is Elsewhere	sortcl syntax error.
79	Abbreviation of ASCENDING	sortcl syntax error.
80	Field Length > Record Length	sortcl syntax error.
81	Overlapping Field	sortcl syntax error.
82	Illegal Character	sortcl syntax error.
83	Applies Only to Keys	sortcl syntax error.
84	Unrecognized Word	sortcl syntax error.
85	Scripts Too Deeply Nested	sortcl syntax error.
86	Circular Definition	sortcl syntax error.
87	Not an Active File	sortcl syntax error.
88	Blocking Factor Invalid	sortcl syntax error.
89	No File with this Name	sortcl syntax error.
90	Unrecognized Name in Expression	Field referenced was not specified.

Table 45: Error Values (cont.)

Value	Message	Meaning
91	No /HEADREAD or no /TAILREAD on input	Need to define a headread or tailread in the referenced input.
92	Not a Valid Option Here	Improper syntax.
93	Require Memory Amount	Tuning parameter missing from <code>cosort_r()</code> or registry.
94	Require Directory Name	Tuning parameter missing from <code>cosort_r()</code> or registry.
95	Improper Command	Not a recognized command.
96	No Such Locale on System	Unknown locale specified.
97	Cannot Set Locale on System	The requested locale routines are not available.
98	Specific Line Too Long	sortcl statement or command line limit exceeded. 16,383 bytes is the maximum line length.
99	Condition Define on a Different File	Defined on one file and used on another.
100	Reserved for future use	N/A
101	Ambiguous Reference	sortcl syntax error.
102	Invalid Record Length for this File	e.g., file size not integer multiple of fixed record length.
103	Constant in Conditional Possible Error	Invalid data type for numeric operand in a condition.
104	Error Return from CoSort	Coroutine reports an error to the caller.
105	Illegal Comparison	Cannot compare data of these different forms.
106	Divide by 0	Division by zero (mathematical error).
107	Invalid Argument	Illegal argument, such as <code>sqrt(-1)</code> or month 13.
108	Cannot Set Environment Variable	for example, <code>setlocale()</code> returns error.
109	Invalid Conversion	Cannot convert this data type.
110	Invalid Macro Referenced	Cannot convert this data type.
111	Records Not in Order	From a check action.
112	cosortrc Report	Reports the total number of sortcl errors in /DEBUG mode.
113	Last Record Incomplete	Wrong record length or missing record terminator.
114	Custom Routine Undefined	Procedure name or field name not in <code>cs_user</code> table.
115	Missing NEWLINE Added to End of Input	Warning in sort .
116	Invalid user dll specified	An invalid user DLL was specified.
117	Invalid user dll function specified	An invalid user DLL function was specified.
118	Records are in Order	Monitor event message.
119	Unable to Read File	Error in reading the specified file.

Table 45: Error Values (cont.)

Value	Message	Meaning
120	Unable to Write File	Error in writing the specified file.
121	Unable to Open File	Error in opening the specified file.
122	Unable to stat file	Unable to read status of file from the system.
123	File write permission denied	Insufficient privileges to write to the specified file / log file
124	No Such File	The specified file does not exist. May also result from the current user lacking write permission to the LOG file path.
125	Invalid File	The specified file is not a regular file.
126	Empty Input File	The specified file is empty.
127	Cannot Find a Matching File	The specified pattern did not match any file in the file system.
128	Not Supported	The feature/syntax is not supported by sortcl . Contact your IRI agent.
129	Disk Full	Insufficient space for input/temporary/output purposes.
130	Missing records (number read != number written)	A mismatch between the number of output records and the number of input records (also accounts for any filter logic). This can result from an AIO resource allocation issue. In cosortrc , set AIO OFF or increase your AIO resources.
131	AIO resources exceeded	AIO resources are insufficient and must be increased.
132-148	Reserved for future use	N/A
132	Reserved for future use	N/A
133	Process type is RANDOM	
134	Use sortcl when input process is not random	
135	Ambiguous name	
136	No external modules present	\$COSORT_HOME/lib/modules is empty.
137	Improper set range	Field size is too small for the data pulled from the set.
138	Reserved for future use	N/A
139	Open SET file	Access error on set file open.
140	Close SET file	Error while trying to close set file.
141	SET records	
142	SET out of order	
143	Non-printable ascii character found	A non-printable ascii character was found.
144	Too much commentary in SET header	
145	Reserved for future use	N/A

Table 45: Error Values (cont.)

Value	Message	Meaning
146	Reserved for future use	N/A
147	Incompatible inter join format	A error in the JOIN script.
148	Incompatible inter join separators	The field delimiters differ in the JOIN.
149	Out of memory for multi-table join	Insufficient memory to complete the multi-table join operation.
150	Reserved for future use	N/A
151	Operating System Error	Indicates an operating system error that is not otherwise covered by the one of the standard error conditions.
152	Parameter Error	Indicates that a routine was called with an illegal parameter. Check your sortcl specification.
153	Too Many Files Open	An attempt was made to open more files than the system allows open at once.
154	Unsupported Action for the Current File Mode	An operation was requested that the current file open mode does not allow.
155	Record in Use by Another Process/Task	The requested record is locked by another process/task.
156	Corrupted Index File	The indexed file is corrupt. It should be reconstructed using the appropriate host system utility.
157	Duplicate Key Detected Where it is Not Allowed	A duplicate key was detected where duplicates are not allowed.
158	Requested Record Was Not Found	The requested record was not found. This can indicate the end or beginning of the file.
159	File Handler has Undefined Status	The current file operation cannot be completed because it detected an <i>undefined</i> status for a parameter.
160	Disk Full	Insufficient space for input/temporary/output files.
161	File in Use by Another Process/Task	The file is locked by another process/task.
162	Mismatch in Record Size	Mismatch detected in the record size specifications.
163	File Type Mismatch	Trying to treat a file with a different /PROCESS type.
164	Insufficient Memory	Required memory space could not be dynamically allocated.
165	No Such File	The specified file does not exist.
166	Permission Denied	Insufficient privileges to access the specified file.
167	Requested Operation Not Supported by this Host System	The requested operation is not supported in your machine.
168	System Ran Out of Lock-Table Entries	Indicates an error when your machine ran out of lock-table entries. Try again.

Table 45: Error Values (cont.)

Value	Message	Meaning
169	Vision License Error	Invalid license file to access vision files. Make sure you have the Vision license file sortcl.vlc in the directory where you have the sortcl executable. The Vision license file can be obtained from an Acucorp representative.
170	Unknown Exception	An internal problem detected due to invalid input data or specification. Please make sure that the input data and specifications are valid. Contact IRI technical support if you cannot resolve the problem.
171	Error in the Vision Transaction System	Indicates that an error occurred in the Acucobol Vision transaction system.
172	Header information missing in input file	The input /PROCESS type you have specified requires a header record to exist, and it can not be found.
173	File read permission denied	Insufficient privileges to read from the specified file.
174	Reserved for future use	N/A
175	Invalid field length	
176	Keyword supported by CoSort but not current product	Nextform, or Rowgen, or Fieldshild.
177	Keyword multiple instances supported by CoSort but not current product	Nextform, or Rowgen, or Fieldshild.
178	[custom message]	This can vary depending on the nature of the error. Read the message for details.
180	Lib not found	Required dynamic library was not found in \$COSORT_HOME/lib/modules.
181	TeraStream-CoSort License Violation	Version special key needed for run time license.
182	Column types of LOB, GLOB, BLOB, etc are not supported	Data type not available with ODBC module.

E.2 Detailed Error and Runtime Messages

The following table contains details of the **CoSort** error and runtime messages, presented alphabetically. For a list of messages by value number, see *Table 45* on page 661.

Table 46: Error and Runtime Messages

Abbreviation of ASCENDING
A sortcl syntax error.
Accepted Rejected
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
AIO resources exceeded
AIO resources are insufficient and must be increased. The method for increasing AIO values is operating-system-dependent (see the Platform Considerations section in the CoSort Install Guide).
Ambiguous Reference
A sortcl syntax error.
Applies Only to Keys
A sortcl syntax error.
Blocking Factor Invalid
A sortcl syntax error.
Cannot Find a Matching File
The specified pattern, such as *.dat , for an input file name did not match any file in the file system.
Cannot Set Environment Variable
For example, <code>setlocale()</code> returns an error.
Cannot Set Locale on System
The requested locale routines are not available.
Circular Definition
A sortcl syntax error.
Completed
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Condition Define on a Different File
Defined on one file and used on another.

Table 46: Error and Runtime Messages

Constant in Conditional Possible Error
An invalid data type has been specified for the numeric operand in a /CONDITION statement.
Corrupted Indexed File
The indexed file is corrupt. It should be reconstructed using the appropriate host system utility.
COSORT_HOME Not in Environment
Use the appropriate command to assign the \$COSORT_HOME environment (shell) variable to your installed CoSort directory.
cosortrc Report
Reports the total number of sortcl errors in /DEBUG mode.
csio Error
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and /MONITOR on page 277.
Custom Routine Undefined
You have specified a function name to use with either /INPROCEDURE, /KEYPROCEDURE, /OUTPROCEDURE, but this function was not linked to the sortcl executable (as described in <i>CUSTOM INPUT, OUTPUT AND COMPARE PROCEDURES</i> on page 556) or the relevant structure was not updated.
Disk Full
Insufficient space for input/temporary/output purposes.
Divide by Zero
Mathematical error.
Duplicate Key Detected Where it is not Allowed
A duplicate key was detected where duplicates are not allowed.
Duplicate Name
A sortcl syntax error. FILLER, when used as a field name, is exempt from this error.
Empty Input File
A warning message that indicates that the specified file is empty.
Environment Variable Undefined
Check the data or job specifications, or the resource control file(s) in use for a missing or undefined <i>\$variable</i> .
Error in the Vision Transaction System
Indicates that an error occurred in the Vision transaction system.

Table 46: Error and Runtime Messages

Error in WORK_AREA
WORK_AREA specification in your resource control file is invalid or cannot be used. Check whether the WORK_AREA directory exists and you have read, write, file create access.
Error Return from CoSort
Coroutine reports an error to the caller.
Expecting
A sortcl syntax error.
Expecting Names
A sortcl syntax error.
Expression Syntax
A sortcl syntax error.
Field Length > Record Length
A sortcl syntax error.
Field Type vs. Length Wrong
Binary data types can only be used at integer multiples of a specific length. In this case, the length specified for the type is incorrect.
File Creation Error
sorti cannot create the indicated specification file. This might be due to improper write permission.
File Handler Has Undefined Status
The current file operation cannot be completed because it detected an <i>undefined</i> status for a parameter.
File in Use by Another Process/Task
The file is locked by another process/task.
File Read Permission Denied
Insufficient privileges to read from the specified file. Check the read privileges of the specified file.
File Type Mismatch
Trying to treat a file with a different /PROCESS type.
File Write Permission Denied
Insufficient privileges to write to the specified file. Check the write privileges of the specified file, which may be the specified log file.
Header information missing in input file

Table 46: Error and Runtime Messages

The input /PROCESS type you have specified, such as CSV or ELF, requires a header record to exist, and it can not be found.
Illegal Character
A <i>sortcl</i> syntax error.
Illegal Comparison
Cannot compare data of these different forms.
Improper Command
Not a recognized command.
Improper Format Declaration
Numeric internal fields must be on fixed boundaries; their format must be one of the types discussed in <i>DATA TYPES</i> on page 611.
Incomplete Command
Missing part(s) of a command. See the <i>sortcl PROGRAM</i> chapter on page 37.
Infile Closed
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and /MONITOR on page 277.
Infile Opened
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and /MONITOR on page 277.
Initiated
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and /MONITOR on page 277.
Insufficient Disk Space for Both Output File and Work File(s)
CoSort has found that the output area and at least one of the work areas overlap, and that the file system does not have enough free space to accommodate both the sort output size and the temporary disk space requirements of the sort. In the simplest case of a single work area overlapping with the output area, the estimated consumption of the common area is typically equal to two times the total number of input file sizes.
Insufficient Disk Space for Output File
CoSort has found that the file system mounted on the output file path does not have enough space to hold the output of the sort. The space required of that file system is typically equal to the total number of input file sizes.
Insufficient disk space or file write permission denied for work file(s)

Table 46: Error and Runtime Messages

CoSort has found that the sum of disk space available in the work areas is less than the estimated consumption of the sort. (The estimated consumption of the work areas is typically equal to the total number of input file sizes.) Or, there is no write permission for the specified WORK_AREA path/filename(s).
Insufficient Memory
Required memory space could not be dynamically allocated.
Invalid Argument
Illegal argument, such as sqrt(-1) or month 13.
Invalid Conversion
Cannot convert this data type.
Invalid Direction
Only 0 for ascending or 1 for descending is permitted.
Invalid Field Position
The acceptable field start range is $0 \leq \text{Position} \leq 65,535$.
Invalid File
The specified file is not a regular file. This could happen when you try to enter a directory name as the input file.
Invalid Format Declaration
Used for improper format definition by the user. Also, it is used when data does not conform to the definition.
Invalid Key Specification
Field positions can be specified as 0 for fixed, 1 for blank, or 2 for character delimited.
Invalid Macro Referenced
Cannot convert this data type.
Invalid Number of Keys
The acceptable range is $0 \leq \text{Number of keys} \leq 65,535$.
Invalid Record Length
<p>This is an Execution Phase error associated with variable-length records and fixed and floating keys. An error situation occurs when a fixed position key starts later than the record length. A variable-position error occurs when the record does not contain enough of the field separators, and the key cannot be found.</p> <p>Note that on the error return, the external integer, <code>cs_rec_ct</code>, contains the record number of the offending record. This is displayed by the sorti program.</p>
Invalid Record Length for this File

Table 46: Error and Runtime Messages

The file size not is not an integer multiple of the fixed-record length.
Invalid Resource Variable in Resource Control File
Your resource control file contains lines that do not specify valid resource variables.
Invalid Value
An invalid value for the given data type/task/feature has been specified. Check your sortcl specification.
Invalid Work Area specified
You have specified a directory for a work area in your cosortrc file which does not exist or for which you do not have the appropriate rights.
Last Record Incomplete
Wrong record length or missing record terminator.
Left Right
This is a join event message generated by MONITOR . See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
License Error: Cannot Obtain Machine ID
Contact your IRI agent.
License Violation: Expiration Date Passed or Invalid Key
Your CoSort evaluation period has expired. Contact your IRI agent to obtain a permanent license.
License Violation: Incorrect Node or Invalid Key
You are attempting to run CoSort on a machine that is not licensed.
License Violation: Invalid key
Recheck the license keys you received from IRI or its agent. Contact IRI if you still cannot run after re-trying the cs_setup procedure.
License Violation: This Application is Not Licensed
Contact your IRI agent.
Maximum Number of Sort Threads Exceeded
Trying to run more sort threads than your license allows. Set THREAD_MAX in your resource control file to the value that sortcl is licensed for. Usually, this is same as the number of CPUs on your machine.
Merge Invalid Number of Files
CoSort could not open all of the workfiles that it created. Try increasing the file handle limit, or adjusting THREAD_MAX and MEMORY_PERTHREAD_MAX to create fewer logical workfiles.
Mismatch in Record Size

Table 46: Error and Runtime Messages

A mismatch was detected in the record size specifications.
Missing Double Quote Mark
A sortcl syntax error.
Missing NEWLINE Added to End of Input
Warning in sort.
Missing records (number read != number written)
There is a mismatch between the number of output records and the number of input records (also accounts for any filter logic). This can result from an AIO resource allocation issue. In cosortrc , set AIO OFF or increase your AIO resources.
No File with this Name
A sortcl syntax error.
No /HEADREAD or no /TAILREAD on input
Need to define a /HEADREAD or /TAILREAD in the referenced input file to correspond to the /HEADWRITE or /TAILWRITE, respectively.
No Such File
The specified file does not exist in the file system. Can also result from the current user lacking write permission to the LOG file path.
No Such Locale on System
Unknown locale specified.
Not a Valid Option Here
Improper syntax.
Not an Active File
A sortcl syntax error.
Not Supported
The feature/syntax is not supported by sortcl . Contact your IRI agent.
Operating System Error
Indicates an operating system error that is not otherwise covered by one of the standard error conditions.
Out of memory for multi-table join
Insufficient memory to complete the multi-table join operation. Consider pre-sorting, rather than using the NOT_SORTED option where possible. Also, you can break out the multi-table join into multiple steps of two-table joins.
Outfile Closed

Table 46: Error and Runtime Messages

This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Outfile Opened
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Output Type Unknown
Only 0 for program return, 1 for file out, 2 for standard out, and 3 for both 1 and 2 are acceptable values for output Type.
Overlapping Field
A sortcl syntax error.
Parameter Error
Indicates that a routine was called with an illegal parameter. Check your sortcl specification.
Parenthesis Count
A sortcl syntax error.
Please Answer Yes or No
You have responded improperly to a prompt, and it requests a decision.
Problem With User's Output File
An I/O problem occurred while you were producing the output file. This is usually caused by running out of disk space.
Process Begins
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Process Ends
This message is generated by MONITOR. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Record in Use by Another Process/Task
The requested record is locked by another process/task.
Record Length Improper
The acceptable range is $0 \leq \text{Record Length} \leq 65,535$.
Records Are in Order
Monitor event message. A status/warning that appears in your monitor messages.
Records Not in Order
From a check action.

Table 46: Error and Runtime Messages

Records Processed
This message is generated by <code>MONITOR</code> . See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Requested Operation Not Supported by this Host System
The requested operation is not supported in your machine.
Requested Record was not Found
The requested record was not found. This might indicate the end/beginning of the file.
Require Memory Amount
Tuning parameter missing from <code>cosort_r()</code> or registry.
Require Directory Name
Tuning parameter missing from <code>cosort_r()</code> or registry.
Scripts Too Deeply Nested
A sortcl syntax error.
Source is Elsewhere
A sortcl syntax error.
Specific Line Too Long
A sortcl statement or command line limit exceeded. 16,383 bytes is the maximum line length.
System Error
This message is generated by <code>MONITOR</code> . See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
System Ran Out of Lock-Table Entries
Indicates an error when your machine ran out of lock-table entries. Try again.
Terminated
Program execution terminated by Ctrl-C.
Too Many Errors -- Aborting
The sortcl script analyzer stops after 8 syntax errors. See <i>/ERRORCOUNT</i> on page 280 to override.
Too Many Files Open
An attempt was made to open more files than the system allows open at once.
Unable to Open File
Error in opening the specified file. This could be due to resource limitations of your machine during runtime.

Table 46: Error and Runtime Messages

Unable to Read File
Error in reading the specified file. This could be due to resource limitations of your machine during runtime.
Unable to Write File
Error in writing the specified file. This could be due to resource limitations of your machine during runtime.
Uncorrected Error Condition
This error is the result of succeeding calls into <code>cosort_r()</code> after an error has already been reported. The only valid call after a reported error is to initiate a new sort or merge.
Unknown Action Requested
A Definition Phase error occurred when the requested job is not a sort, merge, or check.
Unknown Alignment Type
Only 0 for none, 1 for left, and 2 for right are acceptable values.
Unknown Case Type
Only 0 for no case conversion, or 1 for case conversion, are acceptable values.
Unknown Exception
An internal error detected due to invalid data or specification. See <i>Unknown Exception Error -- Logging</i> on page 680 for details on how this error is logged. Check whether the input data and specifications are valid. Contact IRI technical support if you cannot resolve the problem.
Unrecognized Name in Expression
A field referenced was not specified.
Unrecognized Word
A sortcl syntax error.
Unsupported Action for the Current File Mode
An operation was requested that the current file open mode does not allow.
Vision License Error
Invalid license file to access vision files. Make sure you have the Vision license file with name sortcl.vlc in the directory where you have the sortcl executable. A Vision license file can be obtained from an Acucorp representative.
Workfile Closed
This message is generated by <code>MONITOR</code> . See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.
Workfile Opened

Table 46: Error and Runtime Messages

<p>This message is generated by <code>MONITOR</code>. See <i>MONITOR_LEVEL level number</i> on page 650 and <i>/MONITOR</i> on page 277.</p>
--

E.3 Unknown Exception Error -- Logging

An unknown exception error (3) is an internal error that results from invalid data and/or specifications. When a **CoSort** job (**sort**, **sorti**, **sortcl**, or API invocation) is run, a hidden file named **.cserrlog** is created in the specified LOG area (see *Resource Control Settings* on page 647). If no LOG area is specified in your resource control file, **.cserrlog** is created in the current working directory. This file is overwritten with each subsequent job. Therefore, if an error 3 is returned when running a **sortcl** job - or invoking `sortcl_routine()` - it is recommended that you email the contents of this file to support@iri.com to assist with troubleshooting.

The **.cserrlog** report contains the following:

- /RC information (see /RC on page 276).
- major sort operations and error messages, if applicable
- parameters that you would see in a /STAT file (see /CHARSET on page 78), including the number of Logical Work Files (LWF).

The following is an example of a **.cserrlog** file generated when an input file is not found:

```
CoSort error log started: May/26/2011 16:46:21 Eastern Standard Time

CoSort Version 9.5.1 D91080301-1822 #11044.9518
Filename: ./cosort.rc
      MEMORY__MAX           50%
      THREAD_MIN            1
      THREAD_MAX            1
      BLOCKSIZE             2048KB
      MONITOR_LEVEL         5
      MIN_YEAR              70
      LOG                   cosort.log
      ON_WORKAREAS_FULL     ABORT
      ON_EMPTY_INPUT        PROCESS_WITH_ZEROS
      WORK_AREA             1
      -- f:\temp/ : 0
Initialization started... Initialization done
Command: /spec=test.scl
error (124): no such file or directory test.txt
Buffers cleanup started... Buffers cleanup done

Blocksize  Memorysize  ExtMaxMemory  RecsInMemory  Buffers  LWF
      320kb           0mb           120mb           512kb         2         0

Resources cleanup started... Resources cleanup done

CoSort error log done: Feb/26/2011 16:46:21 Eastern Standard Time
```

F ASCII COLLATING SEQUENCE

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

000	00	Nul	032	20	Bl	064	40	@	096	60	'
001	01	Soh	033	21	!	065	41	A	097	61	a
002	02	Stx	034	22	"	066	42	B	098	62	b
003	03	Etx	035	23	#	067	43	C	099	63	c
004	04	Eot	036	24	\$	068	44	D	100	64	d
005	05	Enq	037	25	%	069	45	E	101	65	e
006	06	Ack	038	26	&	070	46	F	102	66	f
007	07	Bel	039	27	'	071	47	G	103	67	g
008	08	Bs	040	28	(072	48	H	104	68	h
009	09	Ht	041	29)	073	49	I	105	69	i
010	0A	Lf	042	2A	*	074	4A	J	106	6A	j
011	0B	Vt	043	2B	+	075	4B	K	107	6B	k
012	0C	Ff	044	2C	,	076	4C	L	108	6C	l
013	0D	Cr	045	2D	-	077	4D	M	109	6D	m
014	0E	So	046	2E	.	078	4E	N	110	6E	n
015	0F	Si	047	2F	/	079	4F	O	111	6F	o
016	10	Dle	048	30	0	080	50	P	112	70	p
017	11	Dc1	049	31	1	081	51	Q	113	71	q
018	12	Dc2	050	32	2	082	52	R	114	72	r
019	13	Dc3	051	33	3	083	53	S	115	73	s
020	14	Dc4	052	34	4	084	54	T	116	74	t
021	15	Nak	053	35	5	085	55	U	117	75	u
022	16	Syn	054	36	6	086	56	V	118	76	v
023	17	Etb	055	37	7	087	57	W	119	77	w
024	18	Can	056	38	8	088	58	X	120	78	x
025	19	Em	057	39	9	089	59	Y	121	79	y
026	1A	Sub	058	3A	:	090	5A	Z	122	7A	z
027	1B	Esc	059	3B	;	091	5B	[123	7B	{
028	1C	Fs	060	3C	<	092	5C	\	124	7C	
029	1D	Gs	061	3D	=	093	5D]	125	7D	}
030	1E	Rs	062	3E	>	094	5E	^	126	7E	~
031	1F	Us	063	3F	?	095	5F	_	127	7F	Del

G EBCDIC PRINTING CHARACTERS

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

064	40	B1				192	C0	{	240	F0	0
074	4A	¢	129	81	a	193	C1	A	241	F1	1
075	4B	.	130	82	b	194	C2	B	242	F2	2
076	4C	<	131	83	c	195	C3	C	243	F3	3
077	4D	(132	84	d	196	C4	D	244	F4	4
078	4E	+	133	85	e	197	C5	E	245	F5	5
079	4F		134	86	f	198	C6	F	246	F6	6
080	50	&	135	87	g	199	C7	G	247	F7	7
090	5A	!	136	88	h	200	C8	H	248	F8	8
091	5B	\$	137	89	i	201	C9	I	249	F9	9
092	5C	*				208	D0	}			
093	5D)	145	91	j	209	D1	J			
094	5E	;	146	92	k	210	D2	K			
095	5F	^	147	93	l	211	D3	L			
096	60	-	148	94	m	212	D4	M			
097	61	/	149	95	n	213	D5	N			
106	6A		150	96	o	214	D6	O			
107	6B	,	151	97	p	215	D7	P			
108	6C	%	152	98	q	216	D8	Q			
109	6D	_	153	99	r	217	D9	R			
110	6E	>	161	A1	~	224	E0	\			
111	6F	?	162	A2	s	226	E2	S			
121	79	`	163	A3	t	227	E3	T			
122	7A	:	164	A4	u	228	E4	U			
123	7B	#	165	A5	v	229	E5	V			
124	7C	@	166	A6	w	230	E6	W			
125	7D	'	167	A7	x	231	E7	X			
126	7E	=	168	A8	y	232	E8	Y			
127	7F	"	169	A9	z	233	E9	Z			

GLOSSARY

This chapter describes industry concepts used throughout the documentation. Most definitions have been expanded to encompass **CoSort** applications.

Acucorp

Founded in the 1980s by Drake and Pamela Coker, Acucorp, Inc. is the San Diego-based ISV behind the cross-platform ACUCOBOL developer and compiler environments. Acucorp as developed a **CoSort** sort replacement option for COBOL programmers interested in faster, native sort performance. IRI has developed exclusive support for collating and converting Vision files in the **CoSort** suite. See **sortcl** and **Vision**.

Adabas

A database management systems (DBMS) from Software AG for IBM mainframes, VAX, Unix and Windows. It is an inverted list DBMS with relational capabilities. A 4GL known as Natural, plus text retrieval, GIS processing, SQL and distributed database functions are also available. Introduced in 1969, Adabas was one of the first DBMS's. See **Natural**.

address sorting

A technique that produces a file of keys and addresses which reference into source file(s). The keys are a subset of the file produced in index sorting. The address record in **CoSort** consists of an unsigned integer that is the relative offset of the source record in the file. When multiple files are used, the index record contains another byte to indicate the relative input file number.

AEP (Advanced External Procedure)

An API created by Informatica Corporation for ISVs integrating functional componentry into Informatica's PowerMart and PowerCenter ETL tools. **CoSort**'s first AEP for Informatica replaces the native sorter 'Tx' transform component to improve sort performance by up to 10X. See **Informatica**.

AES (Advanced Encryption Standard)

AES is a block cipher that is employed as a data encryption standard by the US government. It is expected to be expanded for worldwide use, as was the case with its predecessor, the Data Encryption Standard (DES). The **sortcl** program offers AES encryption using a 256-bit key. See **encryption**.

agent

A software routine that waits in the background and performs an action when a specified event occurs. Agents are also called *intelligent agents*, *personal agents* and *bots*. Your IRI agent, however, refers to the person or company providing **CoSort** license and support services to your organization.

aggregation

A *drill-down* calculation function made on several records or cells of data. SUM, AVG, MAX, MIN and COUNT are examples of aggregate functions that are used in **sortcl**, spreadsheets and database *group by* programs to produce summary information. **sortcl** can also cross-calculate and accumulate aggregating values.

algorithm

A recursive computational procedure, named after the Iranian mathematician Al-Khawarizmi, designed to reach a result after a finite number of steps (which may or may not terminate). This detailed sequence of actions performs some task such as sorting.

Amdahl's law

A theoretical formula, named after Gene Amdahl, that applies to sort performance in parallel processing environments: If F is the fraction of a sequential calculation, and (1-F) is the fraction that can be parallelized, the maximum speed-up that can be achieved by using P processors is $1/(F+(1-F)/P)$. See **parallel processing** and **SMP**.

API (Application Program Interface)

A language and message format used by an application program to communicate with the operating system or some other system or control program such as `cosort_r()` or `sortcl_routine()`. APIs are implemented by writing function calls in the program, which provide the linkage to the required subroutine for execution. Except for writing the logic that performs the actual data processing, the rest of the programming is writing the code to communicate with the operating system and other software. See **coroutine**.

ASCII (American Standard Code for Information Interchange)

Originally a 6-bit computer code representing capital English (Roman) letters and punctuation. This standard has been expanded to 7 bits that include lower case and additional symbols. Also known as the open systems version of EBCDIC, ASCII is also known as ISO 8859-1 (Latin-1), ANSI X3.4-1968, and ANSI X3.110-1983. See **EBCDIC**.

benchmark

A performance test of hardware and/or software such as **CoSort**. There are various programs that very accurately test the raw power of a single machine, the interaction in a single client/server system (one server/multiple clients) and the transactions per second in a transaction processing system. In order for product benchmarks to be fair, test conditions must remain identical. In order to be useful, the impact of benchmarked system on applications that will run concurrently must also be measured. See **TPC** and *PERFORMANCE TUNING* on page 639.

binary

A base 2 numerical system that uses any of several notational systems. It is ideally suited for computer use. Negative numbers can be internally represented in fixed-size binary systems using one's complement or two's complement notation, the latter of which is used in modern computers due to the simplicity of hardware implementation. Binary data is any kind of computer data that is not plain readable text (such as ASCII or EBCDIC).

bit

Abbreviation for binary digit, the smallest unit of computer storage and transmission, existing in only two states, 0 or 1. See **byte**.

BOM (byte order mark)

The byte order mark (BOM) is a Unicode character used to signal the endianness (byte order) of an input or output source. If the BOM is used, it appears at the start of the data. In addition to its use as a byte-order indicator, the BOM character can also indicate which Unicode representation the text is encoded in.

The BOM will be the hex characters FEFF for big-endian or FFFE for little-endian when you have a file encoded in UTF-16.

Boolean logic

The "mathematics of logic," developed by English mathematician George Boole in the mid 19th century. Its rules govern logical functions (true/false). As add, subtract, multiply and divide are the primary operations of arithmetic, AND, OR and NOT are the primary operations of Boolean logic. Boolean logic is turned into logic gates on circuit boards, and is used when defining conditions in **sortcl**.

byte

Abbreviation for binary term, a unit of computer storage and transmission representing a single character (numbers, letters, punctuation, etc.). The number of bits in a byte varies among computer systems, but is usually 8 bits long. Large amounts of memory are also indicated in byte terms. See **bit**, **gigabyte**, **kilobyte**, **megabyte**, **petabyte**, and **terabyte**.

C

A high-level programming language developed at Bell Labs that can manipulate computers at a low level, like assembly language. During the last half of the 1980s, C became the language of choice for developing commercial software because it can be compiled into machine languages for almost all computers. C is written as a series of flexible functions that call each other for processing. C was standardized by ANSI (X3J11 committee) and ISO in 1989. Variations include Turbo C, C++, and Visual C++. **CoSort** is written in C and can API calling examples are provided in C. See **C++** and **Visual C++**.

C++

An object-oriented version of C created by Bjarne Stroustrup. C++ has become popular because it combines traditional C programming with OOP capability. Smalltalk and other original OOP languages did not provide the familiar structures of conventional languages such as C and Pascal. See **C**, **OOP**, and **Visual C++**.

cla4db2 (CoSort Load Accelerator for DB2)

A native sort replacement for IBM's DB2 UDB loader utilities (EEE, ESE, v6-8), on Windows and Unix, that can double bulk load performance in DB2. See **DB2**.

clickstream

The trail of mouse clicks made by a user performing a particular operation on the computer. It often refers to linking from one page to another on the World Wide Web. High volumes of clickstream data are collected in web logs, which can be manipulated as flat files or analyzed by special eCRM software from companies like Hyperion. See **data webhouse** and **elf2ddf**.

COBOL (COmmon Business Oriented Language)

A high-level, compiled programming language that has been the primary business application language on mainframes and mini computers. Officially adopted in 1960, COBOL derived from Flomatic, a language in the mid 1950s. COBOL is verbose language structured by *identification*, *environment*, *data*, and *procedure* divisions. CoSort's *sorti* and **sortcl** programs, and API can be used from Micro Focus and ACUCOBOL programs. **CoSort** natively supports most COBOL data types, and provides a drop-in replacement for the MF COBOL sort. See **cob2ddf** and **mfcosort**.

cob2ddf (COBOL-to-data definition file)

A command line conversion utility in the **CoSort** package to translate Micro Focus (Merant) COBOL file dictionaries (copybook layout information) into **CoSort sortcl** data definition (metadata/field layout) files. See **COBOL**, **copybook**, and the *cob2ddf* chapter on page 395.

copybook

A COBOL-formatted data dictionary that defines the layouts of (usually mainframe sourced) files. See **cob2ddf**, **data dictionary**, and **metadata**.

coroutine

A generalized form of a subroutine where control continues at the point of last exit, with local data values restored. `cosort_r()` is a coroutine which uses the control capability when it interacts with the caller for services. See **API** and **subroutine**.

CoSort (Coroutine Sort)

This perpetually-licensed sort and flat-file ETL package, designed for high-performance sorting, reporting, data warehouse staging and integration, third-party sort replacement, and third-party application integrations on Unix and Windows. See **open systems**.

cosortrc (CoSort Resource Control)

A text file containing performance parameters and modifiable values to control **CoSort**'s use of system resources. On Unix, a **cosortrc** file is built during initial software setup, but on Windows, the Registry holds these values. See **COSORT_TUNER** and the *APPENDIX* chapter on page 611.

Tunable resources include:

- the number of sorting CPUs
- RAM
- I/O buffer size
- sort work areas
- runtime monitoring and logging

COSORT_TUNER

An environment variable for controlling machine resources to maximize sort speed and improve system efficiency. The **COSORT_TUNER** parameters in earlier versions of **CoSort** set the amount of memory available for sorting, I/O block size, the number of records in memory, and the location and size of overflow directories. The new **COSORT_TUNER** variable is used to specify a resource control file that contains the above parameters, plus values for multi-processing, shared memory, pre-sort disk checking, and configuration logging. See **cosortrc** and the *APPENDIX* chapter on page 611.

csv2ddf (CSV-to-data definition file)

A command line conversion utility in the **CoSort** package to translate Microsoft-standard comma-separated-values (**.csv**) file header information into **CoSort sortcl** data definition (metadata/field layout) files. See the *csv2ddf* chapter on page 401.

ctl2ddf (SQL*Loader Control File-to-data definition file)

A command-line conversion utility in the **CoSort** package to translate Oracle SQL*Loader control files (metadata for flat-file loads into the Oracle RDBMS) into **CoSort sortcl** data definition (metadata/field layout) files. See **SQL*Loader** and the *ctl2ddf* chapter on page 399.

database

One or more large structured sets of persistent data, usually associated with software to update and query the data. A simple database might be a single file containing many records, each of which contains the same set of fields where each field is a certain fixed or delimited width. A database is one component of a (sometimes relational) database management system. See **DBMS**, **FACT**, and **relational database**.

data dictionary

A set of data descriptions, or metadata structure, designed to store and share file layout information among users and applications. Examples include a **sortcl** data definition file and a COBOL copybook. In a DBMS context, a data dictionary holds the name, type, range of values, source, and authorization for access for each data element in the organization's files and databases. See **copybook** and **metadata**.

data mining

Exploring detailed business transactions by "digging through mountains of data" to uncover patterns and relationships contained within the business activity and history. Data mining can be done manually by slicing and dicing the data until a pattern becomes obvious. Or, it can be done with programs that analyze data automatically. See **OLAP**, **DSS**, **EIS**, and **data warehouse**.

DataStage

A popular ETL tool originally developed by VMark in the 1980s around the Universal database. DataStage was later sold by Ardent, then Informix, then Ascential Software, and is now sold by IBM. **CoSort** has developed a *Plug-In* for DataStage XE to speed the native sort stage's performance by up to 10X.

data warehouse

A generic term for a system for storing, retrieving, and managing large amounts of any type of data. Data warehouse tools, such as **CoSort**'s **sortcl**, which perform extraction (filtering) and manipulation/transformation (sorting/reformatting), provide useful snapshots of production/database data for decision support. See **DSS**, **EIS**, and **ODS**.

data webhouse

A data warehouse based on clickstream data, whose multifaceted tables measure various web site visitor behaviors. According to data warehouse expert Ralph Kimball, *"the highest-performance access may well be... for the webhouse to hand off the ready-to-go observation set as one or more flat files."* **CoSort**'s **elf2ddf** and **sortcl** programs respectively read and transform large web logs to produce reports and hand-offs to data mining tools.

DBA (Database Administrator)

The person responsible for a database system, particularly for defining the rules by which data is stored and accessed. The DBA is usually also responsible for database integrity, security, performance, and recovery. See **database**.

DBMS (Database Management System)

A packaged database. Popular relational examples include DB2, Oracle, SQLserver, Supra, and Sybase. **CoSort** is often used to sort large database tables and flat files off-line, and to speed repopulating/reloading of the DBMS. See **database** and **relational database**.

DB2

IBM's relational database management system, available as a licensed program on several operating systems. DB2 programmers and users can create, access, modify, and delete data in relational tables using a variety of interfaces. See **DBMS** and **CLA4DB2**.

decryption

The process of converting encrypted data back into its original form so it can be understood. The original encryption key is required to recover the contents of an encrypted field. See **encryption**.

de-identification

The process of removing or modifying identifying values within a record, such as a patient's name, telephone number, etc. This technique is used primarily in the healthcare industry to allow research, training, or other non-clinical applications to utilize real medical data, without jeopardizing patient privacy. See re-identification.

delimiter

A separator character used to indicate the beginning and end of a data field within a record. See **field**.

DOS batch file

A batch file is a file of DOS commands that are *batch* processed. That is, each command in a batch file is executed by DOS until the end of the file is reached. To create a DOS batch file, use a text editor such as Edit or Notepad. If you use a word processor, save your batch file as an ASCII text file, not as a standard document. Always include a .bat extension with your batch file name. The Windows equivalent of a DOS batch file is the Windows Scripting Host. See **shell script**.

DSS (Decision Support System)

An information and planning system that provides the ability to interrogate computers on an ad hoc basis, analyze information and predict the impact of decisions before they are made. DBMS, as well as **CoSort**'s **sortcl**, allow users to select data and derive information for reporting and analysis. However, a true DSS is a cohesive and integrated set of programs that share data and information, directly impacting management's decision-making processes. See **data warehouse**, **EIS**, **OLAP**, and **TPC-H**.

EBCDIC (Extended Binary Coded Decimal Interchange Code)

A character set adapted from punch card code in the 1960s, and is still used between printers, telex machines, and IBM mainframes. There are six different versions of EBCDIC, all of which use non-contiguous letter sequences and omit several punctuation characters used in ASCII. **CoSort** processes US EBCDIC, which has many of the same characters as ASCII, but uses different code points (see *EBCDIC PRINTING CHARACTERS* on page 681). **CoSort** also supports a data type called `ASC_IN_EBC` which contains ASCII characters collated in EBCDIC order. See **ASCII**.

8-bit clean

The treatment of 8-bit bytes as an unsigned value in the numeric range: $0 \leq \text{value} \leq 255$. The natural character mode of Unix computers is ASCII, which uses only 7 bits. If the byte's 8th bit is on, its value is interpreted as negative, which changes its ASCII order. You can collate EBCDIC and other 8-bit character sets by specifying unsigned bytes.

EIS (Executive Information System)

An information system that consolidates and summarizes ongoing transactions within the organization, and the forerunner of the modern data warehouse. An EIS provides top management with all the information it requires at all times from internal as well as external sources. If the EIS provides "what if?" manipulation capabilities like that of a DSS (decision support system), they are one in the same. See **data warehouse**, **DSS**, and **sortcl**.

elf2ddf (ELF-to-data definition file)

A command line conversion utility in the **CoSort** package to translate W3C Extended Log Format (web log) header information into **CoSort** **sortcl** data definition (metadata/field layout) files. See **data webhouse** and the *elf2ddf* chapter on page 405.

encryption

The process whereby bits of data are encoded using a mathematical algorithm based on an encryption key. The encryption process renders the data unreadable until it is decrypted. See **AES (Advanced Encryption Standard)** and **decryption**.

endian

The method Unix computers use to represent binary values. Little-endian machines (including Intel-based and DEC machines) store the value with low-order bytes at the starting address, while big-endian machines (including IBM, Motorola, and Pyramid) store high-order bytes at the starting address. Be aware of the difference when transferring binary values across platforms.

ETI (Evolutionary Technologies International)

Founded in the 1990s, ETI, Inc. is the Austin-based ISV behind an Integration Design Studio (IDS), a popular data warehouse integration tool that interfaces to, and extracts, from a variety of proprietary data sources. Code template functions in the IDS Data System Library (DSL) allow ETI users to generate **CoSort sortcl** job scripts, complete with field layouts based on metadata from ETI*Extract processes. See **ETL** and **sortcl**.

ETL (Extraction, Transformation and Loading)

The functions performed when pulling data out of one database and placing it into another of a different type. **CoSort's sortcl** program is used to routinely perform, consolidate, and/or accelerate these tasks on flat files or defined record streams within high-volume data warehouse environments. Direct **CoSort** sort replacements are available for Ascential DataStage and Informatica PowerCenter. **CoSort** calls can also be made from the ETL tools, as well as from ETI, Data Junction, Raining Data, and SAS.

extract

To perform extraction, or the result -- typically a flat file -- from an SQL query against one or more database (usually fact) tables. See **FACT**.

extraction

The first step in the ETL process. The extraction step copies the data designated for the warehouse from multiple sources like database tables.

FACT (FAst extraCT)

New IRI software that uses SQL syntax to rapidly extract database table data into a flat file or pipe. In the case of Oracle, for example, **FACT** creates this raw data, as well as the optional **sortcl** data definition file (**.ddf**) and SQL*Loader control file (**.ctl**) field layouts to facilitate a combined, high-speed ETL operation against Oracle tables. See **database**, **ETL**, **fact table**, **sortcl**, and **SQL*Loader**.

fact table

A database table, typically in a data warehouse, that contains the primary data. As the largest tables, **CoSort**'s **FACT** software is used to extract these from databases like Oracle, and **CoSort** is used to sort and transform the resulting flat file data, which can then be bulk reloaded through tools like SQL*Loader. See **database**, **data warehouse**, **ETL**, and **FACT**.

field

A named portion of a record which contains a datum. A field may be in a fixed or floating position within a record, and may be of fixed or variable length. Fields can be used as sort keys in **sort** and **sorti**. In **sortcl**, fields can also be simultaneously repositioned, resized, type-converted, and mathematically operated on, and serve as conditions for record selection.

file

A named collection of records. See **table**.

flat file

A sequential file containing only records, i.e., without a header, trailer, or proprietary structural (or hierarchical) information. Usually the contents of a database table, this is the typical (but not only) input format **CoSort** processes.

floating field

A field that has a variable starting and ending position within a record. Its location is usually determined by a delimiter, or field-separator character, such as a comma or tab.

FORTTRAN (FORmula TRANslator)

The first high-level programming language and compiler, developed in 1954 by IBM. It was originally designed to express mathematical formulas, and although FORTRAN is used occasionally for business applications, it is still the most widely used language for scientific, engineering, and mathematical problems. FORTRAN IV is an ANSI standard, but FORTRAN V has proprietary versions.

FTP (File Transfer Protocol)

A software application used to send and receive files to and from another computer system.

gigabyte

1,024 megabytes, or approximately one billion bytes. Gigabyte is often abbreviated as G or GB. See **byte** and **megabyte**.

horizontal selection

Filtering a file to extract records or bytes and reduce sorting volume. See **sortcl** and **vertical selection**.

IBM (International Business Machines)

Founded in the 1950s, the Armonk, New York-based corporation is a major supplier of information-processing products in the United States and around the world, having introduced the first personal computer and today dominating the mainframe market.

IRI is an IBM business partner, and continues to port AIX and Linux **CoSort** at IBM's request to their latest *eServers*, including the i, p, x and zSeries mainframes.

index sorting

A technique that produces an index or cross-reference to one or more source files. The index record consists only of keys and an address into the source file(s). The index file is the intermediate file produced in tag sorting.

infinite sort

A technique used in sorting which allows more records to be sorted than can fit in memory.

Informatica

Founded in 1993, the Redwood City-based ISV behind the popular PowerMart and PowerCenter solutions for data warehousing ETL and data integration. IRI is a full member of the Informatica Developer Network. See **AEP** and **ETL**.

I/O (Input/Output)

Communication between a computer and the user or the outside world. I/O is the principal bottleneck in high-volume data processing which **CoSort** minimizes or circumvents to achieve maximum sort throughput and efficiency.

iostat (I/O statistics)

A Unix command line utility that iteratively reports terminal, disk, and tape I/O activity, as well as CPU utilization.

IP (Internet Protocol) address

The address of a computer attached to a TCP/IP network. Every client and server station must have a unique IP address. Client workstations have either a permanent address or one that is dynamically assigned to them each dial-up session. IP addresses are written as four sets of numbers separated by periods; (for example, 204.171.64.2) and can be processed directly by **CoSort**.

IPC (Inter-Process Communication)

The exchange of data between one process and another, either within the same computer or over a network. It implies a protocol that guarantees a response to a request. IPC is performed automatically by programs like **CoSort**. See **IPCS** and **parallel sorting**.

IPCS (Inter-Process Communication Status)

A Unix command line utility that prints information about active inter-process communication facilities. The information that is displayed is controlled by the options supplied. Without options, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. See **IPC**.

IRI (Innovative Routines International)

Founded in 1978 as Information Resources, Inc. in New York, Florida-based IRI is the ISV behind the **CoSort** package. Developing, supporting, licensing and enhancing **CoSort** is IRI's primary focus, but with increasing attention paid to proofs-of-concept and consulting services around data warehouse ETL projects, legacy sort migrations, third-party sort replacements, database reorg, and reporting. See **CoSort**.

ISV (Independent Software Vendor)

A company that develops, licenses, sells, and supports one or more computer programs from other third-party developers.

Java

A programming language designed to generate applications that can run on all hardware platforms without modification. Developed by Sun, it has been promoted and geared heavily for Web sites and intranets. Modeled after C++, Java is interpreted into machine code at runtime, runs in small amounts of memory, and provides enhanced programming features for developers. See **J2SE**.

JCL (Job Control Language)

A syntax for mainframe batch scripts. **CoSort's** **mvs2scl** and **vse2scl** utilities translate the JCL sort statements from MVS and VSE environments to **sortcl** specifications for Unix environments.

joining

In RDBMS and **sortcl**, the matching of one table (file) against another based on some condition creating a third table or file (result) with data from the input matches. For example, a customer table can be joined with an order table creating a table for all customers who purchased a particular product. The default type of join is known as an *inner* join. It produces a resulting record if there is a matching condition. For example, matching shipments with receipts would produce only those shipments that have been received. On the other hand, an *outer* join using that example would create a record for every shipment whether or not it was received. The data for received items would be attached to the shipments, and empty, or null, fields would be attached to shipments without receipts. See **sortcl**.

J2SE

A Java-based, runtime platform that provides many features for developing Web-based Java applications, including database access (JDBC API), CORBA interface technology, and security for both local network and Internet use. J2SE is the core Java technology platform and is a competitor to the Microsoft .NET Framework.

CoSort's Java GUI requires J2SE to be installed for its cross-platform, client-server design and execute operations. See **Java**.

kilobyte

1,024 bytes. See **byte** and **megabyte**.

loading

The act of populating a database table with flat file data through a database load utility. See **SQL*Loader**.

megabyte

1,048,576 bytes, or approximately one million bytes. Megabyte is frequently abbreviated as M or MB. See **byte** and **gigabyte**.

merging

The combining of one or more sorted files into a single file. See **sorting** for the description of ordering. The keys that determine the merge are the same keys that were used to determine the original sort order.

message queue

A storage space in memory or on disk that holds incoming transmissions until the computer can process them.

metadata

Data that describes other data. **sortcl** data definitions (file layouts) are examples of metadata, as is the meta-tags that describe the content of a Web page. The term may also refer to any file or database that holds information about another database's structure, attributes, processing or changes. See **data dictionary**, **repository**, and the *sortcl PROGRAM* chapter on page 37.

mfcosort

A generic term, and library name, for direct CoSort replacements of various Micro Focus COBOL sort verbs. mfcosort can be linked at runtime to speed sort operations in Workbench, Net Express and Server Express without the need to change the COBOL source code.

Micro Focus

Founded in the U.K. in 1976, Micro Focus International, Ltd. has major facilities in California. The company is known for its "MF COBOL" programming tools that enabled both migrating from mainframe to client/server and developing on client/server platforms for the mainframe. See **COBOL** and the *COBOL TOOLS* chapter on page 453.

Microsoft

Founded in 1975 by Bill Gates and Paul Allen, the Redmond, Washington-based Microsoft Corporation is the largest software company in the world, and the developer of Windows, Open Database Connectivity (ODBC) and the SQL Server RDBMS, among many others. IRI is a member of the Microsoft Development Network (MSDN). See **Windows**.

mpstat (multi-processor statistics)

A Unix command line utility that reports CPU usage statistics (faults, interrupts, switches and waits per second) in tabular form.

multi-threading

A software model where a single program uses multiple threads that execute concurrently. On a single-CPU system, the operating system alternates between the threads, but on a multi-CPU system, the threads can operate in parallel on different CPUs. The **CoSort** engine is an example of a multi-threaded application. See **parallel sorting** and **thread**.

MVS (Multiple Virtual Storage)

Introduced in 1974, the primary operating system used on IBM mainframes (the others are VM and DOS/VSE). MVS is a batch processing-oriented operating system that manages large amounts of memory and disk space. Online operations are provided with CICS, TSO and other system software. See **mvs2scl**.

mvs2scl (MVS-to-sortcl)

A command line conversion utility in the **CoSort** package to translate MVS JCL sort parameters into **CoSort sortcl** job specification files (formerly **jcl2scl**). See **MVS** and the *mvs2scl* chapter on page 433.

Natural

A fourth-generation language from Software AG (Germany) that runs on a variety of computers from micro to mainframe. Natural applications manipulate Adabas data and can use **CoSort** directly to replace and accelerate native sort operations with no user intervention or source changes. See **Adabas** and the *sort PROGRAM* chapter on page 481.

Net Express

The current Micro Focus COBOL software environment for Windows. **CoSort** provides a drop-in replacement for the Net Express sort verb. See **mfcosort** and **Micro Focus**.

no duplicates

The condition in which records with duplicate keys are eliminated from the output of a sort or merge. Duplicates is the default condition. A `NODUPPLICATES` option is available to the **sortcl** user and programs calling `cosort_r()`, including **sorti**.

ODS (Operational Data Store)

A database designed for queries on transactional data. An ODS is often an interim or staging area for a data warehouse, but differs in that its contents are updated in the course of business, whereas a data warehouse contains static data. An ODS is designed for performance and numerous queries on small amounts of data while a data warehouse is designed for elaborate queries on large amounts of data. See **data warehouse**.

OLAP (OnLine Analytical Processing)

Decision support software that allows fast analysis of information that has been summarized into multidimensional views and hierarchies. For example, OLAP tools are used to perform trend analysis on sales and financial information. They enable users to drill down into masses of sales statistics in order to isolate the products that are the most volatile. The **sortcl** program can be used to rapidly produce sorted, multi-level summary or detail views in support of OLAP applications, including MOLAP (multidimensional), ROLAP (relational), DOLAP (database) and WOLAP (web). See **DSS**.

OOP (Object-Oriented Programming)

Programming that supports object technology. It is an evolutionary form of modular programming with more formal rules that allow pieces of software to be reused and interchanged between programs. OOP's major characteristics are (1) encapsulation, (2) inheritance, and (3) polymorphism. See **C++**.

open systems

Industry term for generally compatible computer platforms, and particularly, Unix-based computing. The goal of open systems is interoperability between hardware and software that is defined by the industry at large. Despite the Intel/Microsoft dominance in PC computing, however, open systems now also refers to Windows because so many ISVs develop "Wintel"-compatible applications. Open systems also include pervasive DBMSs and tools like **CoSort** that were developed to run on many different platforms. While open systems offer a certain freedom for future changes, it is not a problem-free environment; for example, to migrate an application from one Unix system to another, all system software components currently linked to an application must also be available for the new system.

Oracle

A popular relational database management system, available from Oracle Corporation as a licensed program on several operating systems. Oracle programmers and users can create, access, modify, and delete data in relational tables using a variety of interfaces. Pre-**CoSorting** flat files on the table's primary key can speed bulk loads into Oracle. See **FACT**, **primary key**, **relational database**, and **SQL*Loader**.

parallel processing

The simultaneous use of more than one CPU to solve a problem. See **SMP**.

parallel sorting

The simultaneous use of more than one CPU to sort. See **multi-threading**.

Pascal

A high-level programming language developed by Swiss professor Niklaus Wirth in the early 1970s and named after the French mathematician, Blaise Pascal. It is noted for its structured programming, which caused it to achieve popularity initially in academic circles. Pascal has had strong influence on subsequent languages, such as Ada, dBASE and PAL. Pascal is available in both interpreter and compiler form and has unique ways of defining variables.

petabyte

1,024 terabytes, or approximately one million gigabytes. See **byte** and **gigabyte**.

pipe

The way users can connect, from the command line, the output of one program to the input of another program without using a temporary file. A pipeline is a connection of two or more programs through pipes. See **stdin** and **stdout**.

POSIX (Portable Operating System Interface)

A set of IEEE standards designed to provide application portability between variants of Unix. IEEE 1003.1 defines a Unix-like operating system interface; IEEE 1003.2 defines the shell and utilities; and IEEE 1003.4 defines real-time extensions

primary key

The unique identifier(s) used as unique members of a relational database table.

process

Any executing program such as **sort**, **sorti**, **sortcl**, etc. A process consists of the program code (which may be shared with other processes that are executing the same program), and some private data. It may have other associated resources such as a process identifier, open files, CPU time limits, shared memory, child processes, and signal handlers. Unix operating systems can run multiple processes concurrently or in parallel.

prompt

A cue from the shell, command line, or an interactive program such as **sorti**, displayed on the terminal which indicates that it is waiting for input.

pseudonymization

The obscuring / de-identifying of actual identities through the use of field value replacement. Used for data privacy purposes. **sortcl** SET file (table look up) functionality allows you to pseudonymize values.

RDBMS (Relational Database Management System)

See **relational database**.

record

The basic component of files which contains fields in a given order. The sort process reorders records in the fields on the basis of a key field. **CoSort** can sort any number of fixed-length or variable-length records, each up to 65,535 bytes long.

re-identification

The process of using a code or some other method to restore data values that have previously been de-identified. See de-identification.

relational database

A database based on the relational model developed by E.F. Codd. A relational database allows the definition of data structures, storage and retrieval operations, and integrity constraints. In such a database, the data and relations between them are organized in tables. See **database**.

repository

A database of information about data itself (metadata repository), or applications software that includes author, data elements, inputs, processes, outputs and interrelationships. A metadata repository, such as a **sortcl** data definition file, is used to store and share file layouts and data views. A repository is used in a CASE or application development system in order to identify objects and business rules for reuse. See **metadata** and **view**.

RFA (Relative File Access)

A four-byte or five-byte integer representing the byte offset into a file, which is used by **sortcl** to randomly access a record used in a tag sort.

RM (Ryan McFarland) COBOL

A widely used COBOL application development system developed by Ryan McFarland Corporation in the 1970s, and now available from Liant Software (Austin, TX). **CoSort** processes RM COBOL data. See **COBOL** and *DATA TYPES* on page 611.

sar (system activity reporter)

A Unix command line utility that samples and displays cumulative activity counters in the operating system at specified intervals and duration. **sar** is used to monitor file access system routines, buffer activities, cache hits, I/O calls, block transfers, paging and kernel memory allocation, CPU usage, etc.

SAS System

Originally called the "Statistical Analysis System," it is an integrated set of data management and decision support tools from SAS Institute (Cary, NC). It includes a programming language and several presentation and statistical modules, as well as MOLAP, query and reporting, EIS, data mining and data visualization. SAS has written *hooks* to **CoSort** for SAS System 7 and 8 on Unix to replace the native sort routine with **CoSort**. See the *sort PROGRAM* chapter on page 481.

semaphore

The classic method for restricting access to shared resources (e.g., IPC, memory) in a multi-processing environment. Invented by Dijkstra, semaphores are protected variables whose value is the number of units of the resource which is free.

separator

See **delimiter**.

sequential file

See **flat file**.

Server Express

The current Micro Focus COBOL software environment for Unix. **CoSort** provides a drop-in replacement for the Server Express sort verb. See **mfcosort** and **Micro Focus**.

SHAR (Shell Archive)

A flattened representation of one or more files, with the unique property that it can be unflattened (the original files extracted) merely by feeding it through a standard Unix shell. The output of SHAR, known as a *shar file* or *sharchive*, can be distributed to any Unix systems, and no special unpacking software is required. Unix versions of the **CoSort** package are delivered as SHAR files. See **tar**.

shared memory

Memory in a parallel computer, usually RAM, which can be accessed by more than one processor, usually via a shared bus or network. It usually takes longer for a processor to access shared memory than to access its own private memory because of contention for the processor-to-memory connections, and because of other overhead associated with ensuring synchronized access. Computers using shared memory usually have some kind of local cache on each processor to reduce the number of accesses to shared memory.

shell

The command interpreter in Unix which works between you and the kernel to personalize your operating environment, redirect input and output, and design shorthands. **sorti** can invoke shell commands within the user session; **sortcl** can use shell variables and issue shell commands.

shell script

A file of executable Unix commands, such as `sortcl /spec=jobname.txt`, that is created in a text editor. When the file is run, each command is executed until the end of the file is reached. Shell scripts are the Unix counterpart to DOS batch files and scripts supported by the Windows Scripting Host. Once the shell script is written, it is made usable by changing its file status to *executable* with the Unix `chmod` (change mode)

command. All of **CoSort** utilities can be called within Unix shell scripts for batch operations. See **DOS batch file**.

SMP (Symmetric MultiProcessing)

Two or more similar processors connected via a high-bandwidth link and managed by one operating system, where each processor has equal access to I/O devices. SMP-enabled applications like **CoSort** can run on any or all of the processors in the system, interchangeably, at the user's and operating system's discretion. See **cosortrc** and **parallel processing**.

SORTCL (Sort Control Language)

A high-level fourth generation language (4GL) and program that defines and manipulates user data in the **sortcl** program. SORTCL supports shared data and SQL concepts. See **metadata**, **sortcl**, and **VMS**.

sortcl (sort control language)

The **CoSort** program that leverages the **SORTCL** syntax to transform data into information via selection, sorting, merging, joining, data type conversion, aggregation, reformatting and reporting -- all of which can be performed in a single pass through the data. **sortcl** is used for: ad hoc and batch summary and web-ready report production; data warehouse and ODS data integration; external RDBMS table joins; reorg sorts (to speed loads) and aggregations; and, to join and filter records. See **ETL** and the *sortcl PROGRAM* chapter on page 37.

sorti (sort interactive)

The English-language, console-based, prompting program in the **CoSort** package accepting user sort/merge job specifications for immediate execution and/or batch operation. See the *sorti PROGRAM* chapter on page 507.

sorting

Given n input records: $R1, R2, \dots, Rn$ in arbitrary order, sorting arranges the records according to given keys, producing n output records Ra, Rb, \dots, Rz with corresponding keys Ka, Kb, \dots, Kz such that $Ka \leq Kb \leq \dots \leq Kz$ read \leq as *precedes or is equal to*.

sort input

A stream of records from one or more files and from individual records which are to be sorted together. The records must have a consistent length, either a variable or fixed length, and they must have a consistent format so that the key can be found.

sort keys

One or more fields in a record that determines the position of the record in the sort output. Keys are evaluated on record pairs to determine precedence in that pair. When multiple keys are specified, comparisons continue until inequality occurs, or until there are no more keys.

sort output

The ordered sequence of records that satisfy the sorting criteria. With **CoSort**, as each record is produced, it can be written to a file or returned to the API caller.

SQL ("sequel")

Structured English Query Language, the widely-accepted fourth-generation language (4GL) introduced by IBM to allow access to data in a database.

SQL*Loader

The load utility included with the Oracle database. Pre-**CoSorting** flat files on the table's primary key can speed direct path, bulk loads into SQL*Loader. See **FACT**, **flat file**, **loading**, **Oracle**, and **ctl2ddf**.

SQL Server

A database server that uses the Structured Query Language (SQL) to accept requests for data access; also, the name of the enterprise-level, web-enabled database engine from Microsoft for its BackOffice products. Pre-**CoSorting** flat files on the table's primary key can speed bulk loads into SQL Server. See **primary key** and **relational database**.

stable sorting and merging

When the specified keys do not determine a unique record order, the output ordering is determined by input order. With **CoSort**, the merge is inherently stable (and therefore need not be specified as such), while the sort must be forced to be stable through sorting or a tie-breaking.

stderr (standard error)

A file to which a program can send output (usually error messages). Unless instructed otherwise, the shell directs this output to **stdout**. **CoSort** runtime monitoring information is sent to **stderr**.

stdin (standard input)

A file from which programs such as **sort**, **sorti**, and **sortcl** can receive input. Unless instructed otherwise, the shell uses input from the terminal, such as typed records, or input files piped into the sort. See **pipe** and **stdout**.

stdout (standard output)

A file to which a program such as **sort**, **sorti** or **sortcl** can send output. Unless otherwise instructed, the shell directs output to the terminal (i.e., the device file that represents the terminal). See **pipe** and **stdin**.

subroutine

A programming procedure, or sequence of instructions for performing a particular task. Subroutine code can be called from multiple places, even from within itself (in which case it is called recursive). The programming language implementation takes care of returning control to (just after) the calling location, usually with the support of call and return instructions at machine language level. Most languages also allow arguments to be passed to the subroutine, and one, or occasionally more, return values to be passed back. See **API** and **coroutine**.

Supra

An RDBMS from Cincom Systems, Inc. (Cincinnati, OH) that runs on IBM mainframes, the VAX, Unix and Windows. The latter versions use **CoSort** for high speed index re-population and loads. Supra includes a query language and a program that automates the database design process.

Sybase

A popular relational database management system, available from Sybase, Inc. as a licensed program on several operating systems. Sybase DBAs can create, access, modify, and delete data in relational tables using a variety of interfaces. Pre-**CoSorting** flat files on the table's primary key can speed bulk loads into Sybase. See **primary key** and **relational database**.

table

A collection of records in a database containing the same fields. Certain fields may be designated as keys, which means that searches for specific values of that field will use indexing to speed them up. Records in different tables may be linked if they have the same value in one particular field in each table. **CoSort** is often used to sort database tables or entire flat files. See **DBMS** and **file**.

tag sorting

A multi-stage sort using keys and record addresses. The input records are first reduced to keys and a record address -- usually a byte offset and, possibly, a file number. As the resultant output is produced, the address is used to access the original record as a random read. This record is then output. The technique produces a stable sort and minimizes workspace storage. Unless the record is much larger than the keys, it executes slowly because of the random accesses.

tar

Tape archive command used in Unix to create and extract file transfers to and from I/O devices such as a disk or cartridge tape. The `tar` command is used to load the **CoSort** product from media, or is used to download data files before or after sorting. See **SHAR**.

TCP/IP (Transmission Control Protocol/Internet Protocol)

A routable communications protocol developed under contract from the U.S. Department of Defense to internetwork dissimilar systems. Invented by Vinton Cerf and Bob Kahn, this de facto Unix standard is the protocol of the Internet and has become the global standard for communications. TCP provides transport functions, which ensures that the total amount of bytes sent is received correctly at the other end.

terabyte

1,024 gigabytes, or approximately one trillion bytes. See **byte** and **gigabyte**.

thread

One of several paths of execution inside a single process or context. Threaded programs allow background and foreground action to take place without the overhead of launching multiple processes or inter-process communication. Also known as *Light Weight Processes*. See **multi-threading**.

thread-safe

A computer programming concept applicable in the context of multi-threaded programs. A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time. See **thread**.

timestamp

A machine-generated macro or system variable which contains current day, time, and year information in the local format. **sortel** recognizes and processes timestamps in America, European, ISO, and Japanese format.

TPC (Transaction Processing Performance Council)

An organization devoted to benchmarking transaction processing systems. In order to derive the number of transactions that can be processed in a given time frame, TPC benchmarks measure the total performance of the system, which includes the computer, operating system, database management system and any other related components involved in the transaction processing operation. In its most recent decision support system (DSS) benchmark (TPC-H) at the 300Gb scale factor, IBM chose **CoSort** to pre-sort large tables (at a rate of 2.4Gb per minute) to speed DB2 loads and ad hoc queries. See **TPC-H**.

TPC-H (TPC Benchmark H Standard Specification Revision 1.2.1)

This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The TPC-H Benchmark represents decision support environments where users do not know which queries will be executed against a database system; hence, the ad-hoc label. Due to the ad-hoc nature, no pre-knowledge of the queries can be built into the DBMS system and the query execution times can be very long. See **DSS**.

transformation

The middle part of the ETL process and an abstract reference to sortel functionality, as the modification of source data prior to its insertion into a (data warehouse) target. See **ETL** and **sortel**.

Unicode

A superset of the ASCII character set using two bytes for each character rather than one. Able to handle 65,536 character combinations rather than just 256, Unicode can house the alphabets of most of the world's languages. ISO defines a four-byte character set for world alphabets, but also uses Unicode as a subset.

Unix

An interactive time-sharing operating system invented in 1969 by Ken Thompson at Bell Labs and Dennis Ritchie, the inventor of the C programming language. Unix, the first source-portable OS, has become multi-variant, but has a uniquely flexible and developer-friendly environment. Today, Unix is the most widely used multi-user, general-purpose operating system in the world.

UTF (Universal Transformation Format)

A method for converting 16-bit Unicode characters into 7- or 8-bit characters. UTF-7 converts to 7-bit ASCII for transmission over 7-bit mail systems, while UTF-8 converts Unicode to 8-bit bytes. See **Unicode** and 7-bit **ASCII**.

VB (Visual Basic)

A popular version of the BASIC programming language from Microsoft specialized for developing Windows applications first released in 1991. VB is used to write client front ends for client/server applications and ActiveX controls for the Web. Widely available runtime DLLs typically accompany a Visual Basic application.

vertical selection

A process for extracting for sort input only those records that meet specified criteria. The selected records would then be subject to horizontal selection. Data fields that participate in vertical selection need not be sort keys, nor even part of the sort input record. **sortcl** performs vertical selection through conditional include and omit statements. See **horizontal selection**.

view

In relational database management, a logical table to display data, created as needed. A view stores no data, but allows access to base tables on which the view is based. A view temporarily ties two or more files together so that the combined files can be displayed, printed or queried; for example, customers and orders or vendors and purchases. In an equivalent flat-file view, **sortcl** users can specify the fields to be included in data definition (metadata) and job specification files. The original files need not be permanently linked or altered, and can joined to produce another view. See **joining** and **RDBMS**.

virtual records

Records constructed dynamically using horizontal and vertical selection. Ordered virtual records are a subset of the original database that has come into existence for ad hoc purposes. Virtual records are displayed or summarized and then discarded. **sortcl**'s **/INREC** statement is used to produce a smaller, virtual record format to reduce sort work bulk and support reformatting.

Vision

The index file format developed by Drake Coker for use within ACUCOBOL. See **Acucorp** and the *COBOL TOOLS* chapter on page 453 for **sortcl** Vision support details.

Visual C++

A C and C++ development system for DOS and Windows applications from Microsoft. It includes Visual Workbench, an integrated Windows-based development environment and the Microsoft Foundation Class Library (MFC), which provide a basic framework of object-oriented code upon which to build an application. See **C** and **C++**.

VLDB (Very Large Database)

See **database**.

VMS (Virtual Memory System)

A multi-user, multitasking, virtual memory operating system for the VAX series from Digital. VMS applications run on any VAX from the MicroVAX to the largest unit. The SORTCL syntax is based on the VMS sort utility syntax.

vmstat (virtual memory statistics)

A Unix command line utility that delves into the system and reports certain statistics kept about the process, virtual memory, disk, trap and CPU activity for certain devices.

VSE (Virtual Storage Extended)

An IBM multi-user, multitasking operating system that was widely used on IBM's 43xx series mainframes. It used to be called DOS (Disk Operating System) and DOS/VSE, but due to the abundance of DOS PCs, was later renamed VSE. It continues today as VSE/ESA. See **vse2scl**.

vse2scl (VSE-to-sortcl)

A command line conversion utility in the **CoSort** package to translate VSE JCL sort parameters into **CoSort sortcl** job specification files. See **VSE** and the *vse2scl* chapter on page 445.

white space

Space or tab characters that lead and trail the critical part of the user's response to a prompt. In blank-separated fields, the field begins at the first tab or space.

Windows

A collection of operating systems created by Microsoft Corporation that allows mouse-selectable icons and menus, virtual memory management, multi-tasking, and data-sharing between applications. **CoSort** runs on Windows 2000, 2003, NT and XP.

Windows Scripting Host

A facility within Windows that executes Visual Basic and JScript scripts. The scripts can be run from the desktop using the **WSCRIPT.EXE** program or from a command line using **CSCRIPT.EXE**. It is the Windows counterpart to DOS batch files and Unix shell scripts. See **DOS batch file** and **VB**.

Workbench

A previous Micro Focus COBOL software environment for Unix and Windows. **CoSort** provides drop-in replacement for the sorts in both Workbench compiler environments. See **mfcosort** and **Micro Focus**.

W3C (World Wide Web Consortium)

An international industry consortium founded in 1994 to develop common standards for the World Wide Web. It is hosted in the U.S. by the Laboratory for Computer Science at MIT. See **elf2ddf**.

This page intentionally left blank.

We Need Your Input

IRI, Inc. upholds a standard of documentation quality that is best maintained through ongoing user feedback. If you have any comments, concerns, or suggestions regarding this documentation, please let us know.

Please indicate the title and page number(s) in the manual you are addressing. Be sure to provide your name, address (or e-mail address), and telephone number if you would like a reply from IRI.

Mailing address: CoSort Technical Publications
INNOVATIVE ROUTINES INTERNATIONAL, INC.
2194 Highway A1A, Suite 303
Melbourne, FL 32937-4932
USA

Telephone: USA +1 (321) 777-8889

Fax: USA +1 (321) 777-8886

E-Mail: support@iri.com

The comments you provide may be used by IRI to improve the quality of, or to make additions to, this document and/or the **CoSort** software.

Additional suggestions or submissions may be made or posted through the technical support area at <http://www.iri.com>.

© 1978-2016 IRI. All rights reserved. No part of this document or the **CoSort** programs may be used or copied without the express written permission of IRI.