



Centrus™ Merge/Purge Library Reference Manual

Version 1.6.6

Copyright 2000-2001, All rights reserved, Sagent Technology, Inc.
National Zip+4 Directory Copyright 2000-2001 United States Postal Service.

Trademark Notices

ZIP Code and ZIP+4 are registered trademarks of the U.S. Postal Service.

Within this manual, other trademarked names may be used. In every occurrence, the names are used editorially and to the benefit of the trademark owner, with no intent of infringement on the trademark.

USPS Notice

Sagent Technology, Inc. holds a nonexclusive license to publish and sell ZIP+4 databases on optical and magnetic media. The price of the Sagent product is neither established, controlled, nor approved by the U.S. Postal Service.

Corporate Office

800 W. El Camino Real, Suite 300
Mountain View, CA 94040
Phone: 650-493-7100
Fax: 650-815-3500

Development Office

4750 Walnut, Suite 200
Boulder, CO 80301-2538
Phone: 303-442-2838
Fax: 303-440-3523

Contents

Chapter 1:	About the Centrus Merge/Purge Library	1
	Introduction	1
	What's Special about the Centrus Merge/Purge Library?	1
	Multiple Index Key Definitions	2
	Multiple Field Match Criteria	2
	Multiple Field Match Algorithms	2
	The Use of Sliding Windows for Comparisons	2
	Match Correlation	3
	About this Manual	3
	Installing Centrus Merge/Purge	4
	Contacting Technical Support	4
Chapter 2:	Understanding Centrus Merge/Purge	5
	Introduction	5
	How Traditional Merge/Purge Works	5
	The Centrus Merge/Purge Approach	6
	Other Features of the Centrus Merge/Purge Approach	6
	How Centrus Merge/Purge Works: an Example	7
	The Traditional Approach	7
	Using Multiple Index Key Definitions	9
	Multiple Matching Fields and Algorithms	12
	Match Correlation and Duplicate Grouping	14
	Summing Up	15
Chapter 3:	Centrus Merge/Purge Components	17
	Phases	17
	Data Input Phase	19
	Record Matching Phase	20
	Duplicate Groups Phase	25
	Data Consolidation Phase	27
	Report Generation Phases	27
	Table Generation Phase	34
	Data Destinations	36
	Data Sources	36
	Functor Data Source	36
	Table Data Source	36
	Preprocessed Data Source	37
	Index Key Definitions	38
	Records	38
	Prototype Records	38
	Augmented Fields	39

Field Trimming	39
Field Match Criteria	40
Field Matching Algorithms	40
Data Lists	42
Data List Types	43
Data List Priority	43
Inter- and Intra-list Matching	43
Inclusion and Suppression Data Lists	44
Data List Service	44
Data Consolidation Criteria	45
Consolidation Example	47
Tables	48
Table Types	48
Internal Intermediate Tables	49
External Tables	50
Event Notification Facility	50
Error Handling, Logging, and Tracing	50
Error Handling	51
Error Logging	51
Error Tracing	52
Summary	53
Chapter 4: Building Centrus Merge/Purge Applications	55
Introduction	55
The Division of Labor in Centrus Merge/Purge	55
Application Program Responsibilities	55
Centrus Merge/Purge Responsibilities	56
Components of a Typical Merge/Purge Batch Application	57
Flow of a Typical Merge/Purge Batch Application	59
C Language Basics	64
How To Use Objects	64
Simple Application Sample	66
Executing Cmpsimpl	67
Flow of Cmpsimpl	68
Cmpsimpl Code	71
Chapter 5: Data Types, Constants, and Macros	73
Data Types	73
MpHnd	73
QBOOL	74
QDATESTRUCT	74
QMS_ALGORITHMS_RULE	74
QMS_ALGORITHM_TYPE	75
QMS_ALIGN_OPTION	75
QMS_CONSOL_RULE	76
QMS_CONSOL_TREAT	76

QMS_CRITERIA_RULE	77
QMS_DATLST_ACTION	78
QMS_DATLST_OUTPUT_PREF	79
QMS_DATLST_TYPE	80
QMS_DATSRC_TYPE	81
QMS_DISK_USAGE	82
QMS_ERROR_ACTION	83
QMS_ERROR_HANDLER	84
QMS_ERROR_INFO	84
QMS_ERROR_LEVEL	85
QMS_EVENT_TYPE	86
QMS_EVERY_NTHREC_FUNC	86
QMS_FIELD_TRIM	87
QMS_FILL_REC_FUNC	87
QMS_FLDEVAL_OPER	88
QMS_INDXKEY_XFORM_TYPE	88
QMS_MATCH_GROUPING	89
QMS_NAME_ORDER	89
QMS_OUTPUT_SAMPLING	89
QMS_PHASE_TYPE	90
QMS_PHASE_STATE	91
QMS_RANKING_PRIORITY	92
QMS_REPORT_TEXT_POSITION	92
QMS_REPORT_TITLE_POSITION	92
QMS_SORT_OBJECT_STATUS	93
QMS_SORT_ORDER	93
QMS_STAT_SAMPLING	93
QMS_STAT_SAMPLING_STATE	94
QMS_TABLE_INDEXES	94
QMS_VARIANT_TYPE	95
QRESULT	95
QVARSTRUCT	95
Constants and Macros	97
NULL	97
OUTPUT_*	97
QmpDeclHnd	97
QmpUtilGetMessage	98
QmpUtilGetSeverity	98
QmpUtilGetSuccessFlag	98
QmpUtilSucceeded	98
QmpUtilFailed	99
QMS_CHDIR	99
QMS_GETCWD	99
QMS_MAX	
QMS_MAX3	100
QMS_MIN	
QMS_MIN3	100
QMS_MAX_INDX_EXPRESSION_SIZE	100

QMS_MAX_INDX_NAME_SIZE	100
QMS_MAX_NAME.....	101
QMS_MAX_PATH	101
QMS_STRICTCMP.....	101
QMS_STRICTNICMP	101
QESEVERITY_SUCCEED	
QESEVERITY_FAIL.....	102
QTRUE/QFALSE	102
Chapter 6: C Function Reference	103
Objects and Handles	104
Parameters, Errors, and Return Values.....	104
QRESULT Return Codes.....	104
Function Class: QmpAlg*	105
Quick Reference	105
QmpAlgCompute	106
QmpAlgCreate	107
QmpAlgDestroy	108
Function Class: QmpCons*	109
Creating Synthesized Consolidated Records	109
Updating Duplicate Records with Consolidated Field Values	109
Attaching the Consolidation Phase Intermediate Tables	109
Quick Reference	109
QmpConsAddConsCrit	111
QmpConsAreConsCritSet	112
QmpConsClear	113
QmpConsCloseConsTbl	114
QmpConsCreate	115
QmpConsGetConsCritAt	116
QmpConsGetConsCritCnt	117
QmpConsGetModifyMasters	118
QmpConsGetModifySubords	119
QmpConsGetNthRecInterval	120
QmpConsRegEveryNthRecFunc	121
QmpConsSetModifyMasters	122
QmpConsSetModifySubords	123
QmpConsSetNthRecInterval	124
QmpConsTblClose	125
QmpConsUnRegEveryNthRecFunc	126
QmpConsUseConsTbl	127
QmpConsUseDupGrps	128
QmpConsUseTbl	129
Function Class: QmpConsCrit*	131
Notes on Consolidated Records	131
Quick Reference	131
QmpConsCritClear	133
QmpConsCritgetexprnumchar	134
QmpConsCritDestroy	135

QmpConsCritFillDesc	136
QmpConsCritFillExpr	137
QmpConsCritFillVal	138
QmpConsCritGetDataLstID	139
QmpConsCritGetExprNumChar	140
QmpConsCritGetExprStart	141
QmpConsCritGetFldHnd	142
QmpConsCritGetFldName	143
QmpConsCritGetFldNameVB	144
QmpConsCritGetOtherFldHnd	145
QmpConsCritGetOtherFldName	146
QmpConsCritGetOtherFldNameVB	147
QmpConsCritGetRule	148
QmpConsCritGetTreatment	149
QmpConsCritIsValid	150
QmpConsCritSetDataLstRule	151
QmpConsCritSetExprSubStr	152
QmpConsCritSetFldValRule	153
QmpConsCritSetGroupOrderRule	154
QmpConsCritSetGroupOtherRule	155
QmpConsCritSetNumValRule	156
QmpConsCritSetOtherFldRule	157
QmpConsCritSetSimpleRule	159
QmpConsCritSetTreatmentRule	160
QmpConsCritSetValRule	162
Function Class: QmpConsRpt*	163
Quick Reference	163
QmpConsRptAddPreProcDataSrc	164
QmpConsRptCreate	165
QmpConsRptDestroy	167
QmpConsRptGetNthRecInterval	168
QmpConsRptRegEveryNthRecFunc	169
QmpConsRptSetDupeGrpId	170
QmpConsRptSetDupeGrpScore	171
QmpConsRptSetListId	172
QmpConsRptSetNthRecInterval	173
QmpConsRptSetPrintKeys	174
QmpConsRptSetPrintSrc	175
QmpConsRptUnregEveryNthRecFunc	176
QmpConsRptUseCons	177
QmpConsRptUseDupGrp	178
Function Class: QmpCrit*	179
Field Match Criterion Transformations	179
Quick Reference	179
QmpCritAddAlg	182
QmpCritAddNameMatchAlg	183
QmpCritClear	185

QmpCritCompute	186
QmpCritCreate	187
QmpCritDestroy	189
QmpCritGetAlgCnt	190
QmpCritGetAlgWeight	191
QmpCritGetBothBlankRes	192
QmpCritGetCaseSens	193
QmpCritGetIgnoreAlpha	194
QmpCritGetIgnoreNum	195
QmpCritGetIgnorePunct	196
QmpCritGetIgnoreSpaces	197
QmpCritGetIgnoreXforms	198
QmpCritGetNameRecProto	199
QmpCritGetOneBlankRes	200
QmpCritGetRule	201
QmpCritGetStrAlign	202
QmpCritGetThreshold	203
QmpCritIsValid	204
QmpCritRemAlg	205
QmpCritSetAlgWeight	206
QmpCritSetBothBlankRes	207
QmpCritSetCaseSens	208
QmpCritSetIgnoreAlpha	209
QmpCritSetIgnoreNum	210
QmpCritSetIgnorePunct	211
QmpCritSetIgnoreSpaces	212
QmpCritSetIgnoreXforms	213
QmpCritSetNameRecProto	214
QmpCritSetOneBlankRes	216
QmpCritSetPairFullFld	217
QmpCritSetPairPartFld	218
QmpCritSetPairSubStrFld	219
QmpCritSetRule	221
QmpCritSetStrAlign	223
QmpCritSetThreshold	224
Function Class: QmpDataDest*	225
Quick Reference	225
QmpDataDestAddDataLstFilter	228
QmpDataDestCreate	229
QmpDataDestDestroy	230
QmpDataDestFillExpr	231
QmpDataDestFillStatSamp	232
QmpDataDestGetAugmentedFieldsFlag	234
QmpDataDestGetExprNumChar	235
QmpDataDestGetExprStart	236
QmpDataDestGetInclCons	237
QmpDataDestGetInclMasters	238
QmpDataDestGetInclSubords	239

QmpDataDestGetInclUniques	240
QmpDataDestGetNthRecInterval	241
QmpDataDestGetOutputTbl	242
QmpDataDestGetReplaceFlag	243
QmpDataDestGetSuppCons	244
QmpDataDestGetSuppMasters	245
QmpDataDestGetSuppSubords	246
QmpDataDestGetSuppUniques	247
QmpDataDestRegEveryNthRecFunc	248
QmpDataDestSetAugmentedFieldsFlag	249
QmpDataDestSetAugOut	250
QmpDataDestSetExprDate	251
QmpDataDestSetExprDateStruct	252
QmpDataDestSetExprFloat	253
QmpDataDestSetExprLong	254
QmpDataDestSetExprString	255
QmpDataDestSetExprSubStr	256
QmpDataDestSetExprVar	258
QmpDataDestSetExprVarStruct	260
QmpDataDestSetInclCons	261
QmpDataDestSetInclMasters	262
QmpDataDestSetInclSubords	263
QmpDataDestSetInclUniques	264
QmpDataDestSetNthRecInterval	265
QmpDataDestSetOutputTypes	266
QmpDataDestSetOutputTypesWithBitmask	267
QmpDataDestSetOutRec	268
QmpDataDestSetRecProto	269
QmpDataDestSetReplaceFlag	270
QmpDataDestSetStatSamp	271
QmpDataDestSetSuppCons	273
QmpDataDestSetSuppMasters	274
QmpDataDestSetSuppSubords	275
QmpDataDestSetSuppUniques	276
QmpDataDestUnregEveryNthRecFunc	277
Function Class: QmpDataInp*	279
Assigning Household IDs	279
Quick Reference	279
QmpDataInpAddDataSrc	281
QmpDataInpAddSerialId	282
QmpDataInpAddUserId	283
QmpDataInpClear	285
QmpDataInpCreate	286
QmpDataInpGetDataSrcAt	288
QmpDataInpGetDataSrcCnt	290
QmpDataInpGetDataSrcNameID	291
QmpDataInpGetIdxKeyCnt	292
QmpDataInpGetNthRecInterval	293

QmpDataInpGetSerialIdFldName	294
QmpDataInpGetSerialIdStart	295
QmpDataInpGetSerialIdTrim	296
QmpDataInpGetUserIdFldName	297
QmpDataInpGetUserIdPicture	298
QmpDataInpGetUserIdTrim	299
QmpDataInpIsValid	300
QmpDataInpRegEveryNthRecFunc	301
QmpDataInpRemDataSrc	302
QmpDataInpRemIndxKey	303
QmpDataInpRemSerialId	304
QmpDataInpRemUserId	305
QmpDataInpSetNthReclInterval	306
QmpDataInpUnregEveryNthRecFunc	307
QmpDataInpUseDataLstSvc	308
QmpDataInpUseIndxKey	309
Function Class: QmpDataLst*	311
Creating a Data List	311
Defining the Rules for Accepting a Record into a Data List	311
Quick Reference	311
QmpDataLstDefCreate	313
QmpDataLstDefGetAction	314
QmpDataLstDefSetAction	315
QmpDataLstDestroy	316
QmpDataLstFldCreate	317
QmpDataLstFldFillExpr	318
QmpDataLstFldSetExprDate	319
QmpDataLstFldSetExprDateStruct	320
QmpDataLstFldSetExprFloat	321
QmpDataLstFldSetExprLong	322
QmpDataLstFldSetExprString	323
QmpDataLstFldSetExprVar	324
QmpDataLstFldSetExprVarStruct	329
QmpDataLstGetCnt	330
QmpDataLstGetExprNumChar	331
QmpDataLstGetExprStart	332
QmpDataLstGetID	333
QmpDataLstGetIntraLstMat	334
QmpDataLstGetOutputPref	335
QmpDataLstGetPriority	336
QmpDataLstIsMember	337
QmpDataLstIsValid	338
QmpDataLstSetExprSubStr	339
QmpDataLstSetIntraLstMat	341
QmpDataLstSetOutputPref	342
QmpDataLstSetPriority	343
QmpDataLstSrcAddSrcID	344
QmpDataLstSrcCreate	345

QmpDataLstSrcFillSrcIDs	346
QmpDataLstSrcGetSrcIDCnt	347
Function Class: QmpDataLstSvc*	349
Quick Reference	349
QmpDataLstSvcAddDataLst	351
QmpDataLstSvcClear	352
QmpDataLstSvcCreate	353
QmpDataLstSvcDestroy	357
QmpDataLstSvcDetMem	358
QmpDataLstSvcFindDataLst	360
QmpDataLstSvcGetDataLst	361
QmpDataLstSvcGetDataLstCnt	362
QmpDataLstSvcGetDataLstID	363
QmpDataLstSvcGetDataLstNameID	364
QmpDataLstSvcGetInterLstMatByld	365
QmpDataLstSvcGetInterLstMatByName	366
QmpDataLstSvcGetRecProto	367
QmpDataLstSvclsValid	368
QmpDataLstSvcSetInterLstMatByld	369
QmpDataLstSvcSetInterLstMatByName	371
QmpDataLstSvcSetRecProto	373
Function Class: QmpDataSrc*	375
Quick Reference	375
QmpDataSrcClear	377
QmpDataSrcDestroy	378
QmpDataSrcFillExpr	379
QmpDataSrcFillStatSamp	380
QmpDataSrcFuncCreate	381
QmpDataSrcGetExprNumChar	382
QmpDataSrcGetExprStart	383
QmpDataSrcGetFillFunc	384
QmpDataSrcGetID	385
QmpDataSrcGetRecProto	386
QmpDataSrcclsNextRecUsed	387
QmpDataSrcclsValid	388
QmpDataSrcPreCreate	389
QmpDataSrcPrepare	390
QmpDataSrcSetExprDate	391
QmpDataSrcSetExprDateStruct	392
QmpDataSrcSetExprFloat	393
QmpDataSrcSetExprLong	394
QmpDataSrcSetExprString	395
QmpDataSrcSetExprSubStr	396
QmpDataSrcSetExprVar	398
QmpDataSrcSetExprVarStruct	399
QmpDataSrcSetFillFunc	400
QmpDataSrcSetRecProto	401

QmpDataSrcSetStatSamp	402
QmpDataSrcTblCreate	403
QmpDataSrcUseFldMap	405
QmpDataSrcUseTbl	406
Function Class: QmpDate*	407
Quick Reference	407
QmpDateCreate	408
QmpDateDestroy	409
QmpDateFillDataPicture	410
QmpDateFillDateStruct	411
QmpDateFillDefPrintPicture	412
QmpDateFillFormat	413
QmpDateGetDateStr	414
QmpDateGetDay	415
QmpDateGetFormat	416
QmpDateGetMonth	417
QmpDateGetMonthStr	418
QmpDateGetYear	419
QmpDateSetDateStruct	420
QmpDateSetJulian	421
QmpDateSetPicture	422
QmpDateSetRaw	423
Function Class: QmpDump*	425
Quick Reference	425
QmpDumpOpen	426
QmpDumpDump	427
QmpDumpClose	428
Function Class: QmpDupGrpRec*	429
Quick Reference	429
QmpDupGrpRecCreate	430
QmpDupGrpRecDestroy	432
QmpDupGrpRecGetDatLstID	433
QmpDupGrpRecGetDupGrpID	434
QmpDupGrpRecGetMasSubScore	435
QmpDupGrpRecGetMasSubStat	436
QmpDupGrpRecGetPrimKey	437
QmpDupGrpRecGetSrcID	438
Function Class: QmpDupGrps*	439
Rules for Choosing the Master Duplicate	439
Tips for Using the Dupe Groups Phase Effectively	439
Quick Reference	440
QmpDupGrpsAddPreProcDataSrc	443
QmpDupGrpsClear	444
QmpDupGrpsCreate	445
QmpDupGrpsFillExpression	446
QmpDupGrpsFillIDs	447
QmpDupGrpsFillMembers	448

QmpDupGrpsFillRec	449
QmpDupGrpsGetDupGrpCnt	450
QmpDupGrpsGetExpressionNumChars	451
QmpDupGrpsGetExpressionStartPos	452
QmpDupGrpsGetLargestDupGrp	453
QmpDupGrpsGetMasterCnt	454
QmpDupGrpsGetMasterDupe	455
QmpDupGrpsGetMaxDupGrpMem	456
QmpDupGrpsGetMemberCnt	457
QmpDupGrpsGetNthRecInterval	458
QmpDupGrpsGetRankingPriority	459
QmpDupGrpsGetRankingPriorityFieldName	460
QmpDupGrpsGetRankingPriorityFieldNameVB	461
QmpDupGrpsGetSortFieldName	462
QmpDupGrpsGetSortFieldNameVB	463
QmpDupGrpsGetSortOrder	464
QmpDupGrpsGetSubordCnt	465
QmpDupGrpsGetThreshold	466
QmpDupGrpsGetTreatmentType	467
QmpDupGrpsRegEveryNthRecFunc	468
QmpDupGrpsRemLowMembers	469
QmpDupGrpsRemPreProcDataSrc	470
QmpDupGrpsSetExpressionNumChars	471
QmpDupGrpsSetExpressionStartPos	472
QmpDupGrpsSetFldValPriority	473
QmpDupGrpsSetMaxDupGrpMem	474
QmpDupGrpsSetNthRecInterval	475
QmpDupGrpsSetSimplePriority	476
QmpDupGrpsSetSortOrder	477
QmpDupGrpsSetThreshold	478
QmpDupGrpsSetTreatmentPriority	479
QmpDupGrpsTblClose	480
QmpDupGrpsTempDGTblClose	481
QmpDupGrpsUnregEveryNthRecFunc	482
QmpDupGrpsUseDataLstSvc	483
QmpDupGrpsUseMatRes	484
QmpDupGrpsUseRecMat	485
QmpDupGrpsUseTbl	486
QmpDupGrpsUseTempDGTbl	487
Function Class: QmpDupsRpt*	489
Quick Reference	489
QmpDupsRptAddPreProcDataSrc	490
QmpDupsRptCreate	491
QmpDupsRptGetNthRecInterval	492
QmpDupsRptRegEveryNthRecFunc	493
QmpDupsRptSetDupeGrpId	494
QmpDupsRptSetDupeGrpScore	495
QmpDupsRptSetGrouping	496

QmpDupsRptSetListId	497
QmpDupsRptSetNthRecInterval	498
QmpDupsRptSetPrintKeys	499
QmpDupsRptSetPrintMatRes	500
QmpDupsRptSetPrintSrc	501
QmpDupsRptUnregEveryNthRecFunc	502
QmpDupsRptUseCons	503
QmpDupsRptUseDupGrp	504
Function Class: QmpFldMap*	505
Quick Reference	505
QmpFldMapClear	506
QmpFldMapCreate	507
QmpFldMapDestroy	509
QmpFldMapIsValid	510
QmpFldMapMapFldToFld	511
QmpFldMapUseRecs	512
Function Class: QmpGlb*	513
Quick Reference	513
QmpGlbFillVers	514
QmpGlbGetLog	515
QmpGlbIsLicenseDemo	516
QmpGlbIsLicenseLoaded	517
QmpGlbSetLicense	518
Function Class: QmplIdxGen*	519
QmplIdxGenClear	520
QmplIdxGenCreate	521
QmplIdxGenGetIdxKeyCnt	522
QmplIdxGenIsValid	523
QmplIdxGenRemIdxKey	524
QmplIdxGenUseIdxKey	525
Function Class: QmplIdxKey*	527
Quick Reference	528
QmplIdxKeyAddCompByHnd	529
QmplIdxKeyAddCompByName	531
QmplIdxKeyClear	533
QmplIdxKeyCreate	534
QmplIdxKeyDestroy	535
QmplIdxKeyFillExpr	536
QmplIdxKeyGetIgnoreAlpha	537
QmplIdxKeyGetIgnoreNum	538
QmplIdxKeyGetIgnorePunct	539
QmplIdxKeyGetIgnoreSpaces	540
QmplIdxKeyGetTblIdxName	541
QmplIdxKeyIsValid	542
QmplIdxKeySetIgnoreAlpha	543
QmplIdxKeySetIgnoreNum	544
QmplIdxKeySetIgnorePunct	545

QmpIdxKeySetIgnoreSpaces	546
QmpIdxKeySetRecProto	547
QmpIdxKeySetTblIdxName	548
QmpIdxKeyUsesSoundex	549
Function Class: QmpIniFil*	551
Quick Reference	551
QmpIniFilCreate	552
QmpIniFilDestroy	553
QmpIniFilReadItemValStrVB	554
QmpIniFilReadULong	556
Function Class: QmpJobRpt*	557
Quick Reference	557
QmpJobRptAddPhase	558
QmpJobRptAddPreProcDataSrc	559
QmpJobRptCreate	560
QmpJobRptFindPhase	561
QmpJobRptGetPhaseCnt	562
QmpJobRptRemPhase	563
QmpJobRptSetRecProto	564
QmpJobRptUseDataLstSvc	566
Function Class: QmpListByListRpt*	567
Quick Reference	567
QmpListByListRptAddPreProcDataSrc	568
QmpListByListRptCreate	569
QmpListByListRptGetNthRecInterval	570
QmpListByListRptRegEveryNthRecFunc	571
QmpListByListRptSetNthRecInterval	572
QmpListByListRptUnregEveryNthRecFunc	573
QmpListByListRptUseDataLstSvc	574
QmpListByListRptUseDupGrp	575
QmpListByListRptUseMatRes	576
Function Class: QmpLog*	577
Quick Reference	577
QmpLogAddErrFilename	578
QmpLogAddErrFilepointer	579
QmpLogAddTraceFilename	580
QmpLogAddTraceFilepointer	581
QmpLogClear	582
QmpLogGetAppErrHandler	583
QmpLogGetErrFatalLvl	584
QmpLogGetErrRptLvl	585
QmpLogGetIndentChar	586
QmpLogGetTraceActive	587
QmpLogGetTraceIndentIncr	588
QmpLogGetTraceRptLvl	589
QmpLogLookUpErrMsg	590
QmpLogLookUpErrMsgVB	591

QmpLogRemErrFilename	592
QmpLogRemTraceFilename	593
QmpLogSetAppErrHandler	594
QmpLogSetErrFatalLvl	595
QmpLogSetErrRptLvl	596
QmpLogSetIndentChar	597
QmpLogSetTraceActive	598
QmpLogSetTraceIndentIncr	599
QmpLogSetTraceRptLvl	600
Function Class: QmpMatRes*	601
Quick Reference	601
QmpMatResCreate	602
QmpMatResDestroy	603
Function Class: QmpMissRpt*	605
Using the Near Miss Report to Tune the Record Matching Threshold	605
Quick Reference	606
QmpMissRptAddPreProcDataSrc	608
QmpMissRptCreate	609
QmpMissRptDestroy	611
QmpMissRptGetAcceptLvl	612
QmpMissRptGetNthRecInterval	613
QmpMissRptRegEveryNthRecFunc	614
QmpMissRptSetAcceptLvl	615
QmpMissRptSetDupeGrpId	616
QmpMissRptSetDupeGrpScore	617
QmpMissRptSetListId	618
QmpMissRptSetNthRecInterval	619
QmpMissRptSetPrintKeys	620
QmpMissRptSetPrintSrc	621
QmpMissRptSetShowSubords	622
QmpMissRptUnregEveryNthRecFunc	623
QmpMissRptUseCons	624
QmpMissRptUseDupGrp	625
Function Class: QmpObj*	627
Quick Reference	627
QmpObjGetID	628
QmpObjGetName	629
QmpObjGetNameVB	630
QmpObjSetName	631
Function Class: QmpPassThru*	633
Quick Reference	633
QmpPassThruSetRecord	634
QmpPassThruUseDataInp	635
QmpPassThruUseFldMap	636
Function Class: QmpPhase*	637
Quick Reference	637
QmpPhaseDestroy	638

QmpPhaseDiskSpace	639
QmpPhaseGetState	643
QmpPhaseGetType	645
QmpPhaseSetRecProto	646
QmpPhaseStart	647
QmpPhaseStop	648
QmpPhaseUseDITR	649
Function Class: QmpRec*	651
Quick Reference	651
QmpRecAdd	653
QmpRecAddByType	654
QmpRecAddByTypePicture	656
QmpRecAddWithWidth	657
QmpRecClear	658
QmpRecClearData	659
QmpRecCopy	660
QmpRecCreate	661
QmpRecDestroy	662
QmpRecGetDataPictureByHnd	663
QmpRecGetDataPictureByName	664
QmpRecGetFldByHnd	665
QmpRecGetFldByName	666
QmpRecGetFldCnt	667
QmpRecGetFldDecimalsByHnd	668
QmpRecGetFldDecimalsByName	669
QmpRecGetFldHnd	670
QmpRecGetFldName	671
QmpRecGetFldNameVB	672
QmpRecGetFldTrimByHnd	673
QmpRecGetFldTrimByName	674
QmpRecGetFldTypeByHnd	675
QmpRecGetFldTypeByName	676
QmpRecGetFldWidthByHnd	677
QmpRecGetFldWidthByName	678
QmpRecIsValid	679
QmpRecSetCharPtrFldByHnd	680
QmpRecSetCharPtrFldByName	681
QmpRecSetDataPictureByHnd	682
QmpRecSetDataPictureByName	683
QmpRecSetDateFldByHnd	684
QmpRecSetDateFldByName	686
QmpRecSetDoubleFldByHnd	687
QmpRecSetDoubleFldByName	688
QmpRecSetFldByHnd	689
QmpRecSetFldByName	690
QmpRecSetFldTrimByHnd	691
QmpRecSetFldTrimByName	692
QmpRecSetFloatFldByHnd	693

QmpRecSetFloatFldByName	694
QmpRecSetLongFldByHnd	695
QmpRecSetLongFldByName	696
QmpRecValidFld	697
QmpRecValidHnd	698
Function Class: QmpRecMat*	699
Notes about the Dynamic Sliding Window	699
Evaluating Field Match Criteria Match Scores	699
Notes about Multi-level Matching	699
Priority of Field Comparison Operations	700
Quick Reference	700
QmpRecMatAddCrit	703
QmpRecMatAddFullLvlCrit	705
QmpRecMatAddLvlCrit	707
QmpRecMatAddPreProcDataSrc	708
QmpRecMatBuildFieldMaps	709
QmpRecMatClear	710
QmpRecMatCompute	712
QmpRecMatCreate	716
QmpRecMatFillWinSize	717
QmpRecMatFillWinStats	718
QmpRecMatGetCritAt	719
QmpRecMatGetCritCnt	720
QmpRecMatGetCritWeight	721
QmpRecMatGetGenSortPreProc	722
QmpRecMatGetIndxKeyCnt	723
QmpRecMatGetLvlCnt	724
QmpRecMatGetLvlCritAt	725
QmpRecMatGetLvlCritCnt	726
QmpRecMatGetLvlCritWeight	727
QmpRecMatGetLvlRule	728
QmpRecMatGetLvlThreshold	729
QmpRecMatGetMatRes	730
QmpRecMatGetNthRecInterval	731
QmpRecMatGetRule	732
QmpRecMatGetThreshold	733
QmpRecMatIsValid	734
QmpRecMatRegEveryNthRecFunc	735
QmpRecMatRemCrit	736
QmpRecMatRemIndxKey	737
QmpRecMatRemLvl	738
QmpRecMatRemLvlCrit	739
QmpRecMatRemPreProcDataSrc	740
QmpRecMatSetCritWeight	741
QmpRecMatSetGenSortPreProc	742
QmpRecMatSetLvlCritWeight	743
QmpRecMatSetLvlRule	744
QmpRecMatSetLvlThreshold	746

QmpRecMatSetNthRecInterval	747
QmpRecMatSetRule	748
QmpRecMatSetThreshold	749
QmpRecMatSetWinSize	750
QmpRecMatUnregEveryNthRecFunc	752
QmpRecMatUseDataLstSvc	753
QmpRecMatUseIdxKey	754
QmpRecMatUseMatRes	755
Function Class: QmpRpt*	757
Quick Reference	757
QmpRptFillStatSamp	758
QmpRptSetFilename	760
QmpRptSetLinesPerPage	761
QmpRptSetPageWidthInChars	762
QmpRptSetStatSamp	763
QmpRptSetSubtitle	765
QmpRptSetTitle	766
Function Class: QmpTbl*	767
Notes on Text Tables	767
Notes on Oracle Tables	767
Quick Reference	768
QmpTblAddNewRec	770
QmpTblAtBOF	771
QmpTblAtEOF	772
QmpTblCbCreate	773
QmpTblClearFlds	774
QmpTblClose	775
QmpTblCommitRec	776
QmpTblCreateIdx	777
QmpTblCreateRep	778
QmpTblDefineFlds	779
QmpTblDelIdx	780
QmpTblDelRec	781
QmpTblDestroy	782
QmpTblDestroyRep	783
QmpTblIDTxtCreate	784
QmpTblExists	786
QmpTblFillByFieldRec	787
QmpTblFillIdxInfo	788
QmpTblFillMappedRec	789
QmpTblFillRec	790
QmpTblIFTxtCreate	791
QmpTblGetAutoOpenIdx	793
QmpTblGetCnt	794
QmpTblGetDatabaseName	795
QmpTblGetExclusive	796
QmpTblGetFldCnt	797

QmpTblGetFldName	798
QmpTblGetFldNameVB	799
QmpTblGetFlds	800
QmpTblGetIdxCnt	801
QmpTblGetPos	802
QmpTblGetReadOnly	803
QmpTblGetTblName	804
QmpTblIsOpen.	805
QmpTblMoveBy	806
QmpTblMoveFirst	807
QmpTblMoveLast.	808
QmpTblMoveTo	809
QmpTblOpen	810
QmpTblOrclCreate	811
QmpTblOrclGetCache	812
QmpTblOrclSetCache	813
QmpTblSeekRec	814
QmpTblSeekVal.	815
QmpTblSetAutoOpenIdx	817
QmpTblSetDatabaseName	818
QmpTblSetExclusive	819
QmpTblSetIdx	820
QmpTblSetMappedRec	821
QmpTblSetReadOnly.	822
QmpTblSetRec	823
QmpTblSetTblName	824
QmpTblSetWhereClause	825
Function Class: QmpTblGen*	827
Quick Reference	827
QmpTblGenAddPreProcDataSrc.	828
QmpTblGenClear.	829
QmpTblGenCreate.	830
QmpTblGenGetDataDestCount.	831
QmpTblGenIsValid.	832
QmpTblGenRemove	833
QmpTblGenRemPreProcDataSrc	834
QmpTblGenUseCons.	835
QmpTblGenUseDataDest	836
QmpTblGenUseDataLstSvc.	837
QmpTblGenUseDupeGroup	838
Function Class: QmpTime*	839
Quick Reference	839
QmpTimeCreate	840
QmpTimeDestroy.	842
QmpTimeGetElapsedTime	844
QmpTimeGetElapsedTimeToNow.	846
QmpTimeGetStartTime	848

QmpTimeGetStopTime	850
QmpTimelsValid	852
QmpTimeStartTiming	853
QmpTimeStopTiming	855
Function Class: QmpUniqsRpt*	857
Quick Reference	857
QmpUniqsRptAddPreProcDataSrc	858
QmpUniqsRptCreate	859
QmpUniqsRptGetNthRecInterval	860
QmpUniqsRptRegEveryNthRecFunc	861
QmpUniqsRptSetNthRecInterval	862
QmpUniqsRptSetPrintKeys	863
QmpUniqsRptSetPrintSrc	864
QmpUniqsRptUnregEveryNthRecFunc	865
QmpUniqsRptUseDupGrp	866
Function Class: QmpUtil*	867
Quick Reference	867
QmpDeclHnd (Macro)	868
QmpUtilChkresultCode	869
QmpUtilFailed (Macro)	870
QmpUtilGetMessage (Macro)	871
QmpUtilGetSeverity (Macro)	872
QmpUtilGetSuccessFlag (Macro)	873
QmpUtilPutFileData	874
QmpUtilPutFileMsg	875
QmpUtilSucceeded (Macro)	876
Function Class: QmpVar*	877
Quick Reference	877
QmpVarChgType	879
QmpVarCreate	880
QmpVarDestroy	882
QmpVarFillDateStruct	883
QmpVarFillVarStruct	885
QmpVarGetDouble	886
QmpVarGetFloat	887
QmpVarGetLong	888
QmpVarGetString	889
QmpVarGetStringVB	890
QmpVarSetDateStruct	891
QmpVarSetDouble	892
QmpVarSetFloat	893
QmpVarSetLong	894
QmpVarSetString	895
QmpVarSetVarStruct	896
Appendix A: QRESULT Return Codes	897
Explanation of Return Codes	897

Success (Informational) Return Codes	899
Failure (Warning) Return Codes	899
Failure (Severe) Return Codes	900
Failure (Fatal) Return Codes	906
Last Chance Return Code	906
Appendix B: Formatting ASCII Files	907
Formatting Text Files in Windows	907
Formatting ASCII Files on UNIX	915
Editing the Format Template	916
ASCII File Properties	916
Record Field Properties	916
General Guidelines	917
Sample Format File	917
Template(fmt File	918
Glossary	919
Index	931

Chapter 1

About the Centrus Merge/ Purge Library

Introduction

The Centrus Merge/Purge library is a toolkit for the development of database merge/purge applications. It provides a powerful, flexible set of programming tools for identifying duplicate records among sets of data records. The library supports the development of both real-time applications, in which a small number of records are compared against an existing large set of records, and batch applications, in which large numbers of records are processed from one or more data sources.

The library is implemented using platform-independent standard C and C++ code. It has been ported to a number of different platforms, including Windows 32-bit platforms and many Unix platforms. It is designed to automatically scale in performance from single-processor, single-threaded platforms to multi-processor, multi-threaded platforms.

What's Special about the Centrus Merge/Purge Library?

Centrus Merge/Purge incorporates a number of techniques that set it apart from other merge/purge implementations.

Multiple Index Key Definitions

Traditional merge/purge applications order records for matching according to a single break key definition. This means that records are matched against each other in one particular order. Centrus Merge/Purge allows you to create multiple index key definitions, which creates more than one ordering of records in a job. Multiple record orderings create more possibilities for duplicate records to be placed next to each other and compared. Choosing different record sort orders also allows sophisticated matching strategies to be created.

Multiple Field Match Criteria

In Centrus Merge/Purge, you may create sophisticated rules for determining if two records match. The heart of this matching technology is the field match criterion, which is a set of rules that scores how well two record fields match. Centrus Merge/Purge allows you to define more than one field match criterion, and combine the resulting match scores into a single record match score. This means that you may match multiple record fields, and build customized rules for combining these field scores into a single match score.

Multiple Field Match Algorithms

A field match criterion may match two fields using one or more different algorithms. If you decide to use more than one matching algorithm, Centrus Merge/Purge lets you choose which algorithms to use and how to combine the algorithm scores into a single field match score.

The Use of Sliding Windows for Comparisons

In standard merge/purge applications, all of the records inside of a break group are compared. If two duplicate records are different enough to be placed in separate break groups, they will never be compared. Centrus Merge/Purge overcomes this problem by using a “sliding window” to compare adjacent records in a record list.

The sliding window is conceptually a box that surrounds a record and a specified number of following records. The first record in the window will be compared against all the other records in the window. After all of the comparisons have been made, the window is moved down the list by one record, and the matching process inside the box happens again. Centrus Merge/Purge allows you to define the size of the sliding window.

The sliding window allows matching between records that do not lie close together in the record list, and that possibly would be placed in separate break groups in other merge/purge implementations. This allows Centrus Merge/Purge to uncover potential duplicates that do not lie close together in the record list.

Match Correlation

After Centrus Merge/Purge completes the initial record matching process, it examines all the matched record pairs for additional duplicate candidates. The duplicate candidates are then placed into separate groups, with each group containing potential duplicates. Match correlation uncovers matching records that the primary record matching process may miss.

For example, imagine that the Centrus Merge/Purge record matcher discovers that record A=B, and that record B=C. The match correlation process determines that record A=C, so all three records are duplicates and are placed into a single duplicate group.

These techniques incorporated into Centrus Merge/Purge improve overall matching accuracy without a significant increase in execution time. In fact, depending on the data, the overall execution time might even be less than with traditional merge/purge techniques.

About this Manual

This manual explains the general use of the Centrus Merge/Purge C language API. The file README.TXT may be on the installation disc and would contain any modifications or corrections to this document. Always assume that the information in README.TXT supersedes the printed documentation.

For white papers and product overviews, please check out our web site at www.sagent.com.

Chapter 2 describes the main concepts and operation principles of the Centrus Merge/Purge technology.

Chapter 3 lists the main components of a Centrus Merge/Purge application and describes what they do.

Chapter 4 discusses the responsibilities of the application vs. those of the library, and the components and flow of a typical Centrus Merge/Purge batch application. It also walks you through a sample batch application.

Chapter 5 documents data types, constants, and macros in the Centrus Merge/Purge library.

Chapter 6 describes each of the functions in the Centrus Merge/Purge library.

Appendix A is the manual glossary.

Appendix B lists the library function return codes and corresponding message strings.

Appendix C discusses how ASCII data files should be formatted in Windows and UNIX.

The manual assumes that you are competent in C or a similar programming language and have some experience in application development for the Microsoft Windows and/or UNIX platforms.

Within this manual, file names appear in regular *italics* font. Language references embedded in descriptive text are shown in a monospaced Courier font:

```
x = 1; /* Just a sample */
```

Function names embedded in text are shown in **monospaced bold**; arguments are shown in *monospaced italics*.

Installing Centrus Merge/Purge

For installation information, refer to the hard copy instructions titled “Centrus Merge/Purge Installation Package Documentation and Contents”. This document is provided with the Centrus Merge/Purge product package.

Contacting Technical Support

If you have any technical questions about using Centrus Merge/Purge, contact Technical Support. If you need assistance with a problem, please provide the following information so that we can be more effective in helping you:

- Name of product
- Company name product is registered under
- Name of company you are affiliated with
- Telephone number you may be contacted at
- Development environment (if applicable)
- Thorough description of problem

Please contact Technical Support:

Sagent Technology, Inc.
4750 Walnut Street, Suite 200
Boulder CO 80301-2538
(303) 442-2838 (voice)
(303) 440-3523 (fax)
support@sagent.com
www.sagent.com

Chapter 2

Understanding Centrus Merge/ Purge

Introduction

This chapter describes the traditional approach to merge/purge technology, and shows how Centrus Merge/Purge produces superior results. It also explains how a Centrus Merge/Purge job works.

How Traditional Merge/Purge Works

All approaches to duplicate data record detection employ the same basic strategy:

- Order the data records so that records that are likely duplicates appear closely together in sequence.
- Apply record matching logic to compare the records near each other.

The challenge is two-fold: to create an ordering of the data records which is likely to place duplicate records close together, and to specify appropriate record matching criteria. The record matching logic must be “loose” enough to tolerate small differences in duplicate records, yet “tight” enough to prevent false duplicate matches. Ordering the records is necessary because we generally cannot afford to compare every data record with every other data record. That approach would require an execution time proportional to the square of the number of records to be examined. Instead, by ordering the records such that duplicate records appear close to one another, a record needs to be compared only with its nearby neighbors. This limits the total execution time required.

The specific ordering of the records is critical. If a record ordering results in true duplicate records placed such that they would never be compared, these duplicates will not be detected. This can occur if there are “minor” differences between duplicate records. For example, suppose there are two records, one with a last name of “Smith” and the other with a last name of “Snith”. All of the other fields in the records are identical. If the records are ordered alphabetically by last name, the two records might not appear closely enough in the sequence of records to be compared.

The Centrus Merge/Purge Approach

Centrus Merge/Purge technology addresses these problems by examining the data records in more than one sequence, then correlating the record matching results to identify groups of duplicates. The user controls the sequences in which records are examined by specifying multiple record ordering rules, called index key definitions. By looking at the records in more than one sequence, Centrus Merge/Purge avoids the problem illustrated above in the “Smith” versus “Snith” example. Even if the two records are not compared when sorted by last name, some other ordering, for example by address, is likely to place the records in the same comparison group.

While examining records in multiple sequences might appear to require more execution time, the technique works so well that the number of records which need to be compared within a sequence can be quite small, much smaller than if the records were examined using a single ordering. The end result is little or no increase in execution time with far superior results.

The process of comparing records in one or more index sequences is referred to as the “record matching phase”. The portion of the Centrus Merge/Purge library that carries out the record matching is referred to as a “record matcher”. Centrus Merge/Purge executes the record matcher “matching logic” once per user-specified index sequence.

Once the record matcher has finished examining the records in each index key sequence, the match results are correlated to bring matching records together into duplicate groups. This is carried out by the “dupe groups phase”. For example, the record matcher may determine that records 1 and 3 are duplicates using one index sequence. Examining the records in a different sequence, it may determine that records 3 and 47 are duplicates. The dupe groups phase deduces that records 1, 3, and 47 are all in the same dupe group. This procedure significantly enhances duplicate detection.

The dupe groups phase also builds a table of dupe groups, identifies master dupes, and performs record comparisons between the master and its subordinates.

Other Features of the Centrus Merge/Purge Approach

Centrus Merge/Purge does not use the traditional “break group” methodology for performing record comparisons. Instead, it employs a “sliding window” methodology which overcomes a number of limitations inherent in the break group approach. Break groups introduce artificial

divisions between records due to the chosen index key. With the sliding window methodology, records which are ordered close to each other are always compared, even if the traditional approach would normally assign them to different break groups. The break group and sliding window methodologies are illustrated in the following example.

How Centrus Merge/Purge Works: an Example

Here's an example of how Centrus Merge/Purge works to improve matching accuracy. Consider the following set of data records for which we would like to determine duplicates:

Table 1: Records ordered by physical record number

Record Number	Last Name	House Number	Street Name	ZIP Code
1	Smiley	123	Main	39773
2	Smith	201	Main	39773
3	Smithe	7586	Main	39774
4	Edwards	8002	Main	39774
5	Forest	17304	Maine	39775
6	Smyth	7586	Main	39774
7	Gibson	7936	Main	39774
8	Snith	201	Main	39773
9	Hall	15045	Main	39775
10	Dmth	201	Main	39773
11	Drake	346	Main	39773
12	Dillon	136	Main	39773

The records in [Table 1](#) are listed in an arbitrary order. Note that records 2 ("Smith") and 8 ("Snith") are likely duplicates, with a typo for the second letter of "Snith", and records 2 ("Smith") and 10 ("Dmth") are likely to be duplicates, with a typo for the first letter of "Dmth". Also, records 3 ("Smithe") and 6 ("Smyth") are likely duplicates if the soundex values for the last names are considered.

The Traditional Approach

Using the traditional method for ordering records for matching, we might define a key consisting of the first three characters of the ZIP Code, Street Name, and Last Name fields. Sorting records using this key would produce the ordering shown in [Table 2](#).

Table 2: Records ordered by the first three characters of ZIP Code, Street Name and Last Name

Record Number	Key Value	Last Name	House Number	Street Name	ZIP Code
12	397MAIDIL	Dillon	136	Main	39773
10	397MAIDMI	Dmith	201	Main	39773
11	397MAIDRA	Drake	346	Main	39773
4	397MAIEDW	Edwards	8002	Main	39774
5	397MAIFOR	Forest	17304	Maine	39775
7	397MAIGIB	Gibson	7936	Main	39774
9	397MAIHAL	Hall	15045	Main	39775
1	397MAISMI	Smiley	123	Main	39773
2	397MAISMI	Smith	201	Main	39773
3	397MAISMI	Smithe	7586	Main	39774
6	397MAISMY	Smyth	7586	Main	39774
8	397MAISNI	Snith	201	Main	39773

A “break group” approach would then group the records into break groups as follows:

- Records 12, 10, 11, 4, 5, 7, 9, 6, and 8 would each be in a separate break group, because the key values of these records are different. In this case, the key values are different because the first three letters of each record’s last name field are not the same.
- Records 1, 2, and 3 would be in the same break group, because their key values are identical.

If the record matching is limited to comparing records within a break group, records 2 and 10 will not be compared; neither will records 2 and 8 or records 3 and 6. Even if a different key were defined, many potential duplicate records would continue to land in different break groups. Since these suspected duplicate records are never compared, the duplicates will not be detected. While there might be *some* ordering and grouping of records that would permit each of the duplicate pairs to be compared, such an ordering and grouping cannot be determined without examining the records in advance.

Centrus Merge/Purge offers a better alternative. By using multiple index key definitions and correlating the results, we can avoid the kinds of problems illustrated above.

Using Multiple Index Key Definitions

Using the data records listed in [Table 1](#), we'll set up three index key definitions. The record matcher will compare records ordered by Last Name, then by Street Name, and finally by ZIP Code. Furthermore, we will configure the record matcher to compare records using a "sliding window" with a size of three. Instead of comparing records on a break group basis, the record matcher will move this "sliding window" in steps over the records from the first record through the last record, comparing only those records within the window boundaries.

Note: A typical Centrus Merge/Purge job would not use such simple index key definitions. A more realistic definition might be something like the first five characters of the last name plus the first three characters of the ZIP code.

The sequence of the records ordered by last name is shown in [Table 3](#):

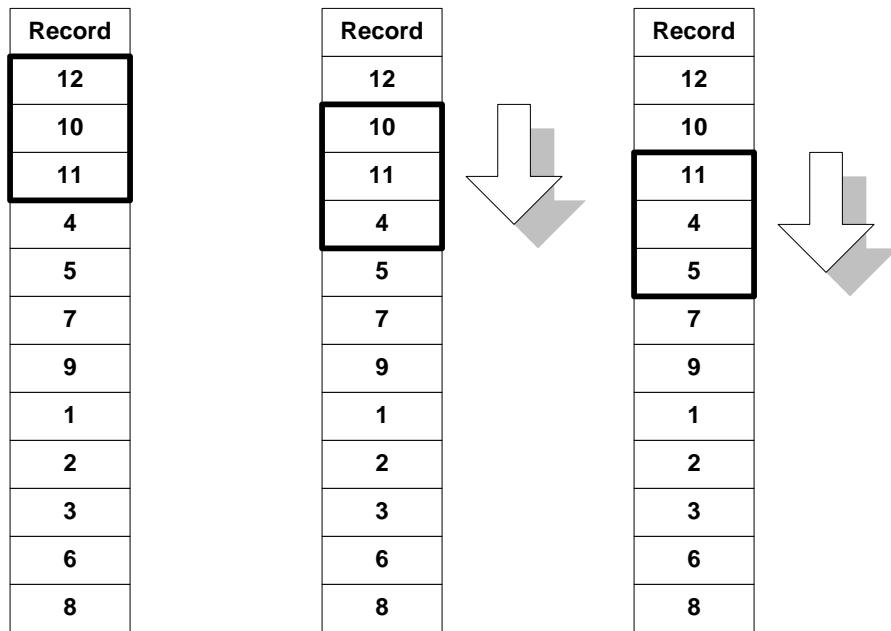
Table 3: Records ordered by Last Name

Record Number	Last Name	House Number	Street Name	ZIP Code
12	Dillon	136	Main	39773
10	Dmth	201	Main	39773
11	Drake	346	Main	39773
4	Edwards	8002	Main	39774
5	Forest	17304	Maine	39775
7	Gibson	7936	Main	39774
9	Hall	15045	Main	39775
1	Smiley	123	Main	39773
2	Smith	201	Main	39773
3	Smithe	7586	Main	39774
6	Smyth	7586	Main	39774
8	Snith	201	Main	39773

Using a sliding window size of three, the record matcher would start with the first record and compare it against the next two records. It would compare record 12 with record 10 and record 12 with record 11.

Next, it would move the sliding window down one record and compare record 10 with record 11 and record 10 with record 4. The sliding window would continue to be moved down until records 6 and 8 are compared.

Figure 1: The Sliding Window Model



The only likely match is 3 with 6. “Smithe” and “Smyth” are similar names, and all other fields in these two records are identical.

Let's examine the record comparisons for the other two record matcher index sequences. The sequence of the records ordered by Street Name is shown in [Table 4](#).

Table 4: Records Ordered by Street Name

Record Number	Last Name	House Number	Street Name	ZIP Code
1	Smiley	123	Main	39773
2	Smith	201	Main	39773
3	Smithe	7586	Main	39774
4	Edwards	8002	Main	39774
6	Smyth	7586	Main	39774
7	Gibson	7936	Main	39774
8	Snith	201	Main	39773
9	Hall	15045	Main	39775
10	Dmith	201	Main	39773
11	Drake	346	Main	39773
12	Dillon	136	Main	39773
5	Forest	17304	Maine	39775

The record matcher would discover potential matches between records 3 and 6 and between records 8 and 10.

The sequence of the records ordered by ZIP Code is shown in [Table 5](#).

Table 5: Records ordered by ZIP Code

Record Number	Last Name	House Number	Street Name	ZIP Code
12	Dillon	136	Main	39773
10	Dmith	201	Main	39773
11	Drake	346	Main	39773
1	Smiley	123	Main	39773
2	Smith	201	Main	39773
8	Snith	201	Main	39773
4	Edwards	8002	Main	39774
7	Gibson	7936	Main	39774
3	Smithe	7586	Main	39774
6	Smyth	7586	Main	39774
5	Forest	17304	Maine	39775
9	Hall	15045	Main	39775

The record matcher would discover good matches between records 2 and 8 and records 3 and 6.

The list of good matches after the record matcher has examined the records in all three index sequences would be:

- Records 2 and 8
- Records 8 and 10
- Records 3 and 6

Multiple Matching Fields and Algorithms

In our example, we matched records by combining the matching results of four separate record fields. We also used more than one algorithm when matching one of the fields, and combined the results of the multiple algorithms. Matching records using multiple fields and algorithms is one of the strengths of the Centrus Merge/Purge library.

The example job was configured to match records using four fields. Each of these fields was assigned one or more matching algorithms:

Match Field	Field Match Algorithms
Last Name	Keyboard Distance Soundex
House Number	Numeric Comparison
Street Name	String Comparison
ZIP Code	Numeric Comparison

When two records are compared, Centrus Merge/Purge CLI uses one or more algorithms assigned to each match field to calculate a match score for that field. If multiple algorithms are assigned to a field, the scores are combined in a job-specified way. If more than one match field is designated, the match scores of all the match fields are combined in a job-specified way. The result is a single match score for the entire record. If this score meets or exceeds a job-specified match score, the two records are considered matches.

In our example, the Last Name fields of two records are matched using two algorithms: keyboard distance and soundex. The average of the two algorithm scores is used as the field match score. The other three matching fields use one matching algorithm each (although not necessarily the same one). In this job, the final record match score is calculated by taking the average of the four field match scores. The following table shows exactly how this process was followed when records 2 and 8 were matched:

Matching Field	Record 2	Record	Keyboard Distance Algorithm	Soundex Algorithm	Numeric Comparison Algorithm	String Comparison Algorithm	Combination of Algorithm Scores (Field Match Score)
Last Name	Smith	Snith	97%	100%	--	--	99%
House Number	201	201	--	--	100%	--	100%
Street Name	Main	Main	--	--	--	100%	100%
ZIP Code	39773	39773	--	--	100%	--	100%
Average of Field Match Scores (Record Match Score)							100%

In this example, the record matcher threshold is set at 65%. Since the calculated record match score exceeds the threshold, records 2 and 8 are considered matches.

Records 3 and 6 are compared in the same way:

Matching Field	Record 3	Record 6	Keyboard Distance Algorithm	Soundex Algorithm	Numeric Comparison Algorithm	String Comparison Algorithm	Combination of Algorithm Scores (Field Match Score)
Last Name	Smithe	Smyth	65%	100%	--	--	83%
House Number	7586	7586	--	--	100%	--	100%
Street Name	Main	Main	--	--	--	100%	100%
ZIP Code	39774	39774	--	--	100%	--	100%
Average of Field Match Scores (Record Match Score)							96%

Since the record match score exceeds the record matcher threshold, records 3 and 6 are also considered matches.

The matching examples above use fairly simple matching criteria. By choosing different field matching algorithms and different methods of combining algorithm and field match scores, you are free to customize your matching criteria.

Match Correlation and Duplicate Grouping

The list of duplicate pair candidates shown previously is a great improvement over the results of traditional matching methods. But Centrus Merge/Purge can do even better. Using match correlation, the results from the record matcher can be examined for additional duplicate candidates. The dupe groups phase then takes the pairs of correlating records and brings the duplicate candidates together into dupe groups. In our example, the dupe groups phase would determine that records 2 and 8 match well and records 8 and 10 match well, so it would group records 2, 8, and 10 together in a dupe group. Using user-specified criteria, the dupe groups phase would also identify one record from each dupe group as the “master” dupe and compare that master against all subordinate duplicates.

The duplicate record groups created by the dupe groups phase would be:

- Records 2, 8, and 10 in dupe group 1
- Records 3 and 6 in dupe group 2

Summing Up

The following elements set Centrus Merge/Purge apart from other merge/purge implementations:

- Multiple index key definitions
- A record matcher using multiple field matching criteria and multiple field matching algorithms
- The use of a sliding windows for comparisons
- Match correlation

This technology improves overall matching accuracy without a significant increase in execution time. In fact, depending on the data, the overall execution time might even be less than with traditional merge/purge techniques.

How Centrus Merge/Purge Works: an Example

Chapter 3

Centrus Merge/Purge Components

This section lists the main components of a Centrus Merge/Purge application and describes what they do.

Phases

A Centrus Merge/Purge application organizes the various merge/purge processes into a “job”. A job is not a Centrus Merge/Purge entity, but an application-managed group of Centrus Merge/Purge components that work together to accomplish a well-defined merge/purge activity. A single application can contain more than one Centrus Merge/Purge job.

The components of a Centrus Merge/Purge job are called phases. Phases represent distinct activities in a complete Centrus Merge/Purge job. They are usually (but not always) executed sequentially. These phases are:

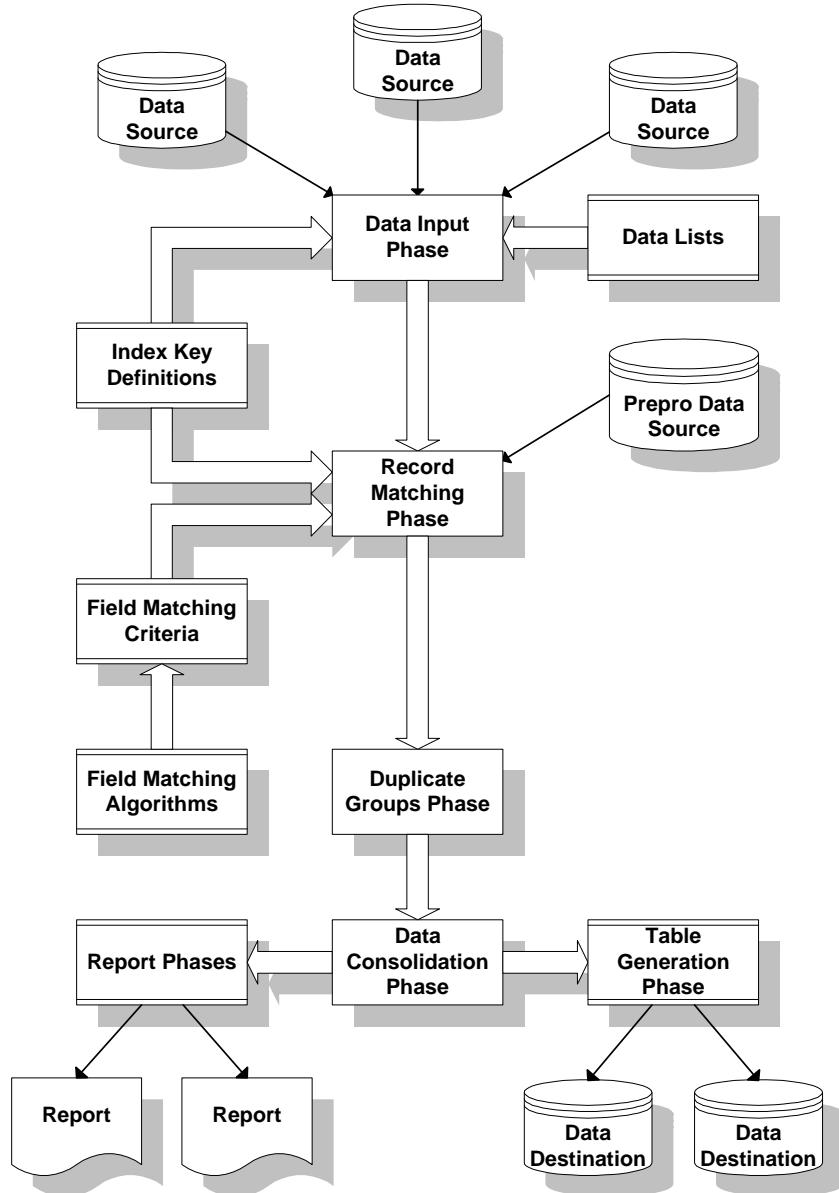
- Data input
- Record matching
- Dupe groups
- Data consolidation
- Report generation
- Table generation

The library is highly granular in nature, making it easy to incorporate only those phases actually needed for a particular job. Each phase requires inputs and creates outputs. The various inputs are “given” to the phases by the application program. Similarly, the application program gives the phases

repositories in which to store their outputs. The outputs from a phase are typically disk-based (a database table, for example), making it easy for an application to save the output from a particular phase, then use it as input to a different phase (perhaps in another application).

The basic phase relationships are diagrammed in [Figure 2](#):

Figure 2: Centrus Merge/Purge Phases



Applications interact with all of the phases in basically the same way: the application allocates them, does some amount of “setup” to initialize them, gives them resources to work with, and then “starts” them. The phases then run to completion. Generally, the bulk of an application is devoted to setting up the phases prior to running them.

Phases share certain characteristics:

- They are processes which must be started to perform their tasks.
- All phases are started via the function `QmpPhaseStart` and run to completion (if no errors occur) before returning full control to the application.
- The phases can optionally generate notification events whereby callback functions registered by the application can be invoked during phase execution. The phase that invoked the callback function can be stopped by the callback function.
- Each phase records its starting and stopping times and can be asked for the start time, stop time, total elapsed time, and the current elapsed time (which can only be obtained with a callback function).

A job can contain a subset of the available phases to accomplish all or part of a merge/purge operation. In fact, it would be very reasonable to create several Centrus Merge/Purge jobs to be run separately, with each job doing a part of the full Centrus Merge/Purge operation through the execution of a subset of available phases.

A typical batch application contains all of the phases shown in [Figure 2](#). A real-time application typically uses only the record matching and dupe groups phases. A very simple real-time application might use only the record Matching Phase.

Data Input Phase

The data input phase reads in data records from one or more data sources in preparation for the record matching phase. This is the first phase in most Centrus Merge/Purge applications, since the records must be input into the system before they can be indexed, matched, or correlated.

Users may create any number of data sources of type `QMS_DATSRC_TYPE_FUNCTOR` or `QMS_DATSRC_TYPE_TABLE`, and add them to the data input phase. Once the data input phase is started using the `QmpPhaseStart` function, it will cycle through its list of data sources. For each data source, the data input phase will gather records until the data source returns a value of `QFALSE`. This means either that there are no more records in the data source, or if statistical sampling is being used, the data source is finished providing records. When the data input phase receives a value of `QFALSE` from a data source, the phase starts gathering records from the next data source. As records are retrieved from a data source, they are assigned data list membership and added to the Data Input Table Repository (DITR).

As the data input phase writes newly-input records to the internal DITR, it adds several additional (augmented) fields to the records. These fields hold values used for internal library calculations. For example, if an index sequence on the soundex of the last name is desired, a new field with the soundex value of the last name must be created. This field is added to records when they are stored in the DITR.

Data Pass-through

To reduce unnecessary processing, it is possible to read into the DITR only those record fields that are needed for indexing, matching, or data consolidation. In data pass-through the other unused fields are identified to the library so that they are not read in by the data input phase. If the values of the pass-through fields are required in an output table, the library will get the necessary data from the source records in the data sources.

Data pass-through reduces the amount of data flowing through all the phases leading up to table generation and improves performance.

Record Matching Phase

Once the Centrus Merge/Purge application has acquired the data records, it is ready to start the record matching phase. This phase performs all record comparisons in order to detect duplicate records. The fundamental approach to duplicate record detection is to order the records so that duplicate records appear closely together, and then to compare records with their near neighbors.

Record ordering is important in Centrus Merge/Purge because the record matcher only compares records that are within an application-specified distance of each other. If a particular record ordering does not place duplicate records sufficiently close together, the record matcher will not identify them as duplicates. This can occur if there are even minor differences between duplicate records that would cause them to inadvertently be placed far from each other in record order. It is therefore very important to define record orderings that have a high probability of placing duplicate records near each other.

The record matcher is designed this way because it is generally too inefficient and time-consuming to compare every record with every other record. Instead, by ordering the records so that duplicate records appear close together, comparisons between records can be limited to a small set of records at a time, which reduces total execution time.

To avoid missing duplicates because they may not always appear close together in one ordering, the record matcher may examine the records in more than one sequence. The application uses an index key definition to define the sequencing logic for record ordering. Multiple index key definitions are used to define multiple record ordering schemes. Before matching, the record matcher sorts the DITR into individual tables, with the ordering of each table defined by an index key definition.

The record matcher uses its matching logic once per index key definition (sorted table). Conceptually, the record matcher moves a “sliding window” through each ordering of the records, comparing each record in the sliding window to every other record in that window. The sliding window can be set to a fixed record size, or it can be set to grow (as needed) up to an application-specified upper limit. This dynamic sliding window needs to expand if a large number of records with the same index key value come through the system. This allows these records to be compared in the same sliding window.

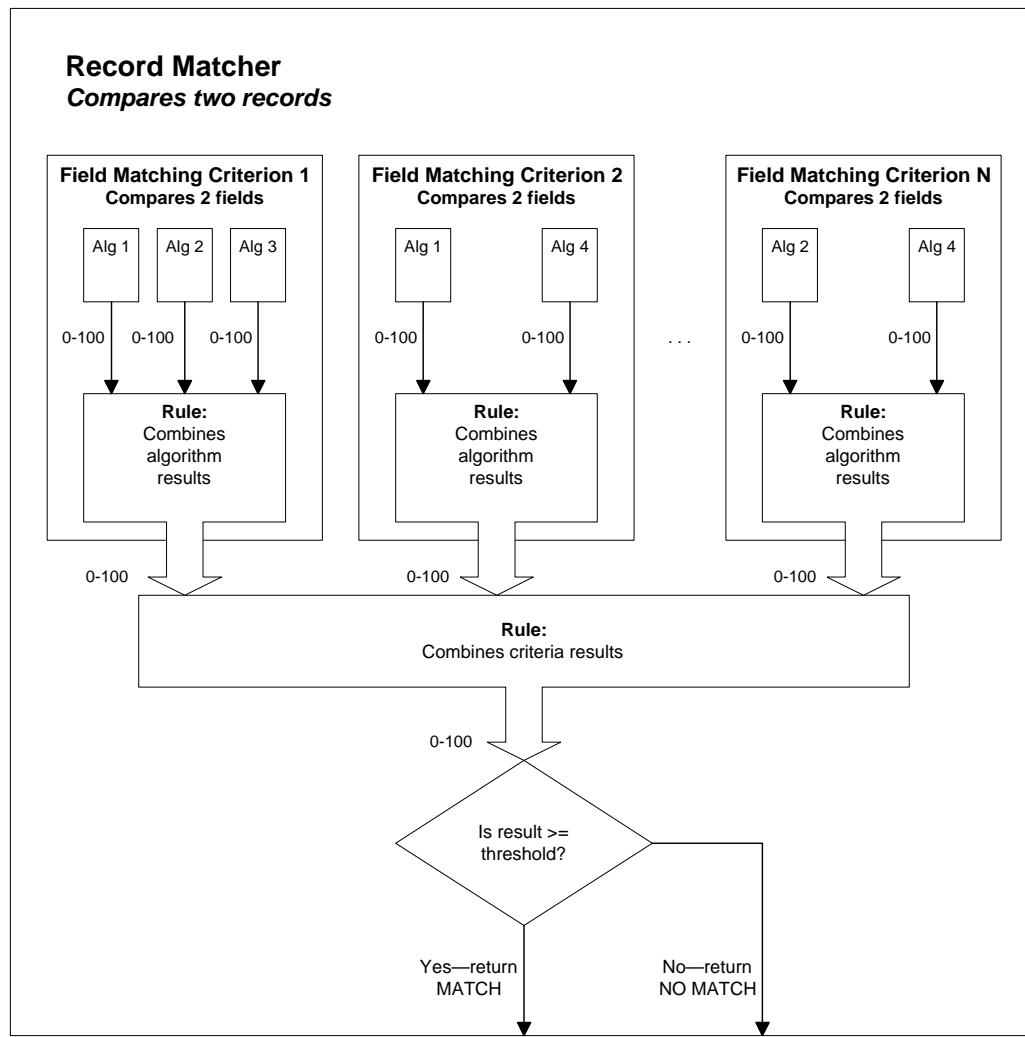
The record matcher comparison logic is based on its collection of field matching criteria. The application creates one or more field matching criteria and adds them to a collection of criteria maintained inside the record matcher. These field matching criteria specify exactly which fields in the records to compare, and what algorithms to use in performing the field comparisons.

Basic Record Matching

In the basic record matching process, the record matcher combines the match scores of its field matching criteria in an application-specified way to create a final record match score. If the record match score equals or exceeds an application-specified matching threshold, the two records are considered a match.

[Figure 3](#) illustrates how the basic record matching process is used to compare two records:

Figure 3: The basic record matching process



Multi-level Record Matching

Centrus Merge/Purge also allows you to create “levels” of matching rules. If the matching rule on the first level yields a match, the matching process stops. If the first level rule doesn’t yield a match, the matching rule on the second level is used. If a match is found, the process stops; if not, the matching rule on the next level is used. This process continues until one of the levels yields a match, or all of the matching rule levels are exhausted.

Multi-level record matching is ideal for first matching high-confidence data with fast algorithms, and then matching lower-confidence data later (if necessary) with slower algorithms.

Example

For example, imagine that you would like to match records using four fields:

- PhoneNumber
- LastName
- ZIPCode
- Address

You have the highest confidence in the phone number data, and comparing these fields with the string algorithm is very fast, so you prefer to match on this field first. Since you have high confidence in this field, if the phone number fields match, it is not necessary to match the other fields.

If the phone number fields don't match well, then you want to match on the other fields. Because you have a lower level of confidence in this data, you want to combine the match scores of these fields into a single score.

You would solve this problem with basic matching by setting a record matcher threshold of around 45%, and creating four field match criteria, whose scores are combined using a weighted average:

Field	Weight
PhoneNumber	50%
LastName	20%
ZIPCode	20%
Address	10%

In this scheme, a phone number match alone would be enough to exceed the match threshold. Good matches in the LastName, ZIPCode, and Address fields would also collectively ensure a match. However, in basic record matching, there is no provision to direct which fields are matched first, so all the fields must be compared (which slows down the matching).

In multi-level matching, you would solve this problem by creating two matching levels:

Level	Field	Match Algorithm
0	PhoneNumber	String
1	LastName	Edit distance & character frequency
1	ZIPCode	String
1	Address	Numeric and string

In this scheme, level zero matches only the PhoneNumber field using the string algorithm (which is very fast). If a match is found, the process stops and the next pair of records is matched. If not, the lower-confidence fields in level one are matched, and their match scores combined into a single record match score. This process ensures that the high-confidence data is always matched first and quickly. Lower-confidence data, compared with slower algorithms, is matched later, and only if necessary.

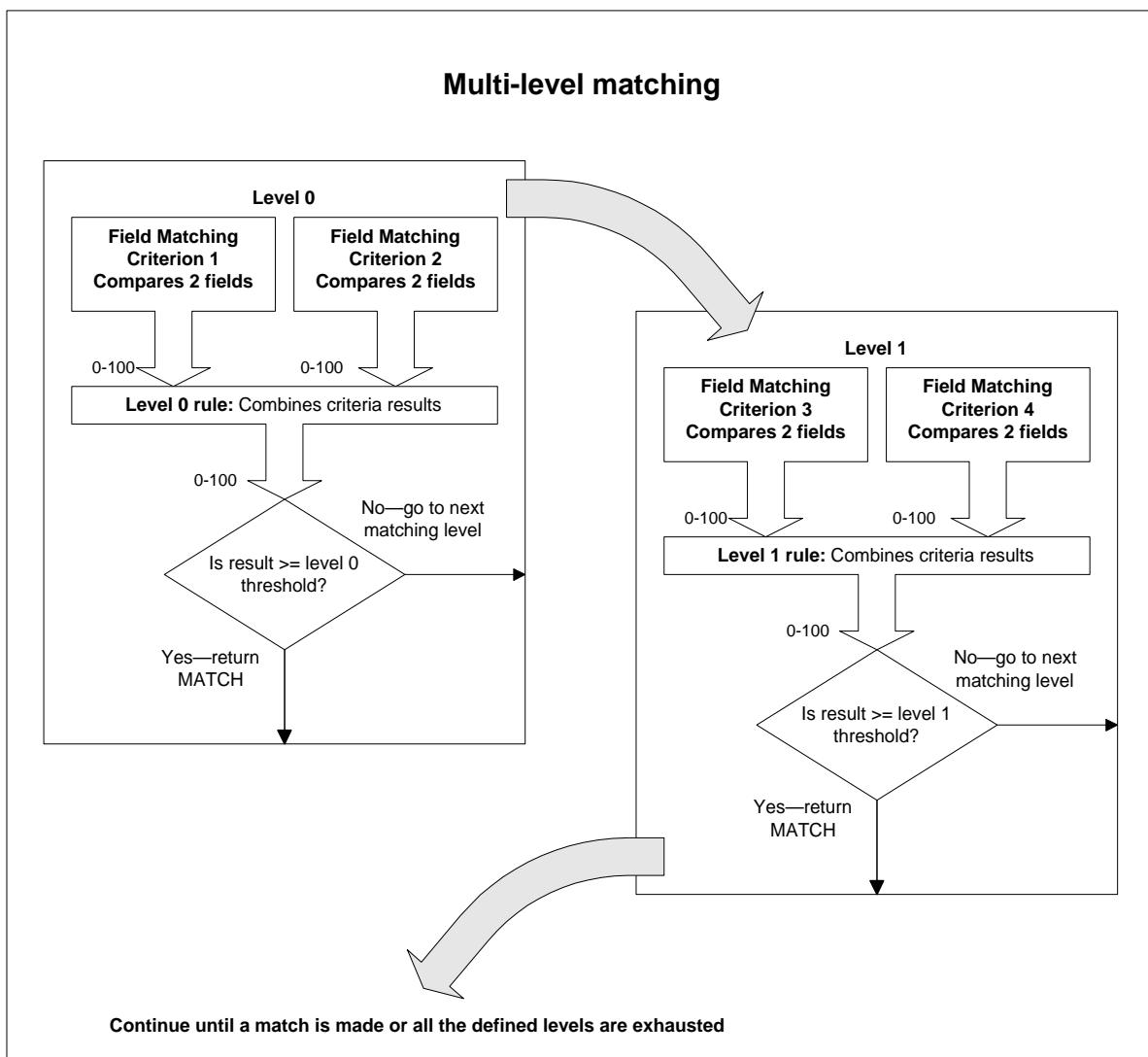
The Multi-level Matching Process

In the multi-level matching process, the application creates a set of field match criteria for each matching level. These criteria establish the rules for matching two records at that level. The application adds the criteria to the record matcher and sets their level. The application must also set, for each level, the rule for combining criteria match scores. Finally, the application sets a separate record matching threshold for each level.

When the record matcher executes, it matches each pair of records using the criteria in the first level. The criteria scores are combined using the rule established for that level, and the final record match score is compared with the threshold set for that level. If the score equals or exceeds the threshold, the records are considered a match and the process stops. If not, the records are compared using the criteria in the next higher level. This process continues until one of the matching levels yields a match, or all of the matching levels are exhausted.

[Figure 4](#) illustrates how multi-level match rules are used to compare two records:

Figure 4: The multi-level matching process



Creating effective field matching criteria is an important part of the record matching process. The record matching logic embodied in the field matching criteria must accommodate some degree of differences between records that should be considered duplicates without inadvertently identifying incorrect duplicates.

Duplicate Groups Phase

During the dupe groups phase, the results of the record matcher are examined to determine the actual members of the dupe groups. In this process, the dupe groups phase iterates through the match result table, finding relationships between different match result records. The formula for forming dupe groups is if A=B, and B=C, then A=C. Therefore, records A, B and C would all be part of the same dupe group.

For example, let's assume we had the following two match result records:

- record 5 matches record 10
- record 10 matches record 50

The dupe groups phase will create a dupe group with records 5, 10 and 50.

The dupe groups phase does not guarantee that all dupe group records compare favorably against all other records in the group, but only that each record compared very well against at least one other record in the dupe group.

The dupe groups phase builds two tables: a dupe groups table and a temporary dupe groups table (tempDGT). See [“Dupe Groups Table” on page 49](#) and [“TempDGT” on page 49](#) for discussions of these tables.

The dupe groups phase uses a threshold that determines which records in the match result table are accepted into the dupe groups table. This dupe groups threshold is set by calling `QmpDupGrpsSetThreshold`. If the threshold is set, the score of a match result table record must be equal to or greater than the threshold to be accepted into the dupe groups table. If the application does not set the dupe groups threshold, all records from the match result table are accepted into the dupe groups table.

For the match result table records accepted into the dupe groups table, the dupe groups phase calculates which of the records in a dupe group is the master duplicate, and which are the subordinate duplicates. The dupe groups phase also fills in the score for how the master matches each of the subordinates. Some of these comparisons have already been done by the record matcher. However, if a subordinate duplicate has not already been matched against the master duplicate, the dupe groups phase makes the comparison, so that eventually all of the subordinate duplicates are matched against the master.

Finally, the dupe groups phase may optionally remove subordinate duplicates that matched poorly with the dupe group master. This is done by calling `QmpDupGrpsRemLowMembers`. A subordinate duplicate might not match the master duplicate well if the two records were not directly matched before the dupe groups phase. If all subordinate duplicates are removed from a dupe group, the entire dupe group is added to the unique records pool.

For example, imagine that the record matcher phase matches records A and B, and records B and C, and the dupe groups phase places all three records into the same dupe group. The dupe groups phase then determines record A is the master duplicate. At this point, we know that A=B and B=C. We assume that A=C, but at this point, the two records have not been directly compared. The dupe groups phase will make this match, and if it turns out that the match score between A and C is below the threshold set by `QmpDupGrpsRemLowMembers`, record C will be dropped from the dupe group.

Data Consolidation Phase

Data consolidation is the process of extracting field data from the records in a dupe group and writing that data to another record. For each field in the consolidated record, the user chooses the rule (called a consolidation criterion) that determines what the consolidated field value will be. Some criteria choose a field value directly from a dupe group record; others calculate the field value using several dupe group record fields. See “[Data Consolidation Criteria](#)” on page 45 for a more detailed discussion.

Consolidated records come in three flavors:

Consolidated Record Type	Description
Synthesized	For each dupe group, a new consolidated record is constructed. A synthesized consolidated record contains all of the fields in the prototype record, and all of these fields are updated with the appropriate consolidated value. A synthesized record for each dupe group is always constructed.
Master duplicate	If this option is chosen, the fields in a dupe group's preexisting master duplicate are updated with consolidated values.
Subordinate duplicate	If this option is chosen, the fields in a dupe group's preexisting subordinate duplicates are updated with consolidated values.

When the data consolidation phase is executed, synthesized consolidated records are always created (one synthesized record for each dupe group). You may also direct master duplicates to be updated with consolidated field values, as well as direct subordinate duplicates to be updated with consolidated field values. When duplicate records are updated, no new fields are added; the fields already present are updated with the values placed in the synthesized consolidated record.

The data consolidation phase uses two application-provided intermediate tables. See “[Synthesized Consolidated Records Table](#)” on page 50, and “[Updated Duplicates Consolidation Table](#)” on page 50 for a discussion of these tables.

Report Generation Phases

The report generation phases are used to report the results of the previous Centrus Merge/Purge phases. The different report phase types are:

- Data consolidation
- Duplicate records
- Job summary

- List-by-list
- Near miss
- Unique records

A report phase is the final phase in any Centrus Merge/Purge job, and is optional. On the other hand, a Centrus Merge/Purge job could be built that does nothing but generate reports. Applications should include a report generation phase only when a report is needed, and may include one or more different report phase types. Reports are quickly generated, and require little computational overhead.

Statistical Sampling

You may control which records are included in your reports with statistical sampling. This technique allows you to quickly filter output records in a controlled way. By including a small subset of records in a report, you can evaluate the output before committing to a long processing run.

Statistical sampling allows you to filter records into a report in the following ways:

Sampling Method	Example
Maximum number of records	Include 1,000 records in the report.
Include record X to record Y	Include record 850 to record 10,000 in the report.
Individual sampling (include every Nth record)	Filter every 5th record into the report.
Group sampling (include a group of M records starting at every Nth record)	At every 5th record, include a group of two records. In this example, records are included in the pattern 5, 6, 10, 11, 15, 16, 20, 21, ...
Maximum number of dupe groups	Include the records from a maximum of 50 dupe groups.
Include the records in dupe group X to dupe group Y	Include the records from dupe group 10 to dupe group 20.
Dupe group sampling (include the records in every Nth dupe group)	Include the records from every 10th dupe group.

Statistical sampling for reports is optional. If statistical sampling is used for a report, a statistical sampling section will appear in that report, as well as in the job summary report.

Data Consolidation Report Phase

The consolidated records report lists each dupe group's consolidated record, as well as the group's other duplicates. Each consolidated record is identified by a 'C' in the M/S column. The report also displays the consolidation criterion specified for each record field.

Notes on consolidated records:

- The primary key, source ID, and score values are always 0.
- The dupe group ID is the same as that of the dupe group from which the consolidated record was constructed.
- A consolidated record is automatically assigned to the default data list (whose list ID is 0).

The following is an example of a consolidated records report:

Centrus Merge/Purge version 1.1.0							Page: 1	
Sagent Technology, Inc.								
Consolidated Records Report								
Consolidated Records Report for a Threshold of 90								
Generated for Make-A-Wish Foundation								
Date/Time: September 10, 1998 11:50 AM								
<hr/>								
Summary: Consolidated Records Report								
Number of Records				: 4400				
Number of Masters				: 923				
Number of Subordinates				: 1085				
Number of Uniques				: 2392				
Dupe Group Threshold				: 91				
Consolidation Criteria by field:								
Field: LASTNAME								
Rule: Choose highest ranking non-blank field								
Field: ZIPCODE								
Rule: Choose most common value for this field								
Treatment: Treat as string								
Field: DATE								
Rule: Choose based on a field value expression								
Field Expression: DATE < 30000101								
M/S	DupeID	Score	ListID	Key	SrcID	LASTNAME	ZIPCODE	DATE
M	1	100	2	2	1	ALMARALES	03304	19980101
S	1	100	2	3	1	ALMARALES	03304	19980101
S	1	100	2	3722	1	ALMARALES	03304	19980101
C	1	0	0	0	0	ALMARALES	03304	19980101
M	2	100	1	6	1	AMOLE	038483003	19980101
S	2	100	1	2056	1	AMOLE	038483003	19980101
C	2	0	0	0	0	AMOLE	038483003	19980101
M	3	100	1	7	1	ANSAY	045650147	19980101
S	3	100	1	8	1	ANSAY	045650147	19980101
S	3	100	1	3723	1	ANSAY	045650147	19980101
C	3	0	0	0	0	ANSAY	045650147	19980101
M	4	100	1	12	1	ARIZZI	050469760	19980101
S	4	100	1	13	1	ARIZZI	050469760	19980101
C	4	0	0	0	0	ARIZZI	050469760	19980101

Duplicate Records Report Phase

A duplicates report contains duplicate record information. The duplicate records in the report may be grouped by dupe group, or by dupe group and further ranked by master/subordinate status.

Here is an example of a duplicates report ordered by dupe group:

Centrus Merge/Purge version 1.0.0 Sagent Technology, Inc. Duplicate Report Ordered by Dupe Group						Page: 1
Rpt-Duplicates-90.log: Duplicates Report for a Threshold of 90						
Generated for Make-A-Wish Foundation						
Date/Time: May 22, 1998 9:36 AM						
<hr/>						
Summary: By Dupe Group Report						
Number of Records: 42510						
Number of Masters: 13001						
Number of Subordinates: 13042						
Number of Uniques: 16467						
<hr/>						
DupeID	ListID	Key	SrcID	FIRSTNAME	LASTNAME	
1	2	1	1	ALVETTA	AKSTIN	
1	2	29996	1	ALVETTA	AKSTIN	
2	1	2	1	ALVILDA	ALBERICO	
2	1	3	1	ALVILDA	ALBERICO	
2	1	4	1	ALVUILDA	ALBERIC	

Here is an example of a duplicates report ordered by master ranking:

Centrus Merge/Purge version 1.0.0 Sagent Technology, Inc. Duplicate Report by Master/Subordinate Ranking						Page: 1	
Rpt-Duplicates-90.log: Duplicates Report for a Threshold of 90							
Generated for Make-A-Wish Foundation							
Date/Time: May 22, 1998 10:13 AM							
<hr/>							
Summary: Master/Subordinate Report							
Number of Records: 42510							
Number of Masters: 13001							
Number of Subordinates: 13042							
Number of Uniques: 16467							
M/S	DupeID	Score	ListID	Key	SrcID	FIRSTNAME	LASTNAME
M	1	100	2	1	1	ALVETTA	AKSTIN
S	1	100	2	29996	1	ALVETTA	AKSTIN
M	2	100	1	1	2	ALVILDA	ALBERICO
S	2	98	1	4	1	ALVUILDA	ALBERIC
S	2	100	1	2	1	ALVILDA	ALBERICO
S	2	100	1	3	1	ALVILDA	ALBERICO

Job Summary Report Phase

The job summary report provides a snapshot of the settings used to produce a processing run. It provides a summary for each phase you require (including start time, end time, and elapsed time), as well as a display of the relevant settings in effect during the processing run.

Here is an excerpt from a job summary report:

```

Centrus Merge/Purge version 1.6.0                                         Page:  1
Sagent Technology, Inc.
Job Summary Report
My Report Title
My Report Subtitle
Date/Time: April 20, 1999    4:06 PM

-----
Summary for the Data Input Phase: Data Input 1
  Date: 04/20/1999
  Start time   (h:m:s): 16:06:24
  End time     (h:m:s): 16:06:24
  Elapsed time (h:m:s): 0:00:00
  Number of Records: 979
  Summary of the Data Sources:
    Data Source 1: My First Data Source
      Table Name : datainp16.dbf
      Input count : 1000
      Statistical Sampling :
        Maximum Records : 0
        Start Record   : 0
        End Record     : 0
        Interval       : 0
        Group size     : 0
      Field Evaluator expression      : STATE != TX
        starting at           : 1
        number of characters : 0
  Prototype record summary :
    name      type   width  decimals trim
    FIRSTNAME STRING  15      0        BOTH
    LASTNAME  STRING  15      0        BOTH
    ADDRESS   STRING  35      0        BOTH
    CITY      STRING  20      0        BOTH
    STATE     STRING  2       0        BOTH
    ZIPCODE   STRING  9       0        BOTH
    DATE      DATE    8       0        BOTH mm-dd-yy
    SerialID  LONG   10      0        BOTH
    UserID    STRING  40      0        BOTH
    User defined Householding : UserID
      User defined picture : ADDRESS+ZIPCODE
      trimming             : BOTH
    Serially defined Householding : SerialID
      starting position   : 1
      trimming             : BOTH
  Summary for the Record Matcher Phase: Record Matcher 1
  Date: 04/20/1999
  Start time   (h:m:s): 16:06:24
  End time     (h:m:s): 16:06:30
  Elapsed time (h:m:s): 0:00:06
  Pre Processed Data Source   : tableprepro
    Source Id          : 2
    Number of records : 16
  Record Matching Threshold : 65%
  Dynamic Sliding Window is being used.
    Default sliding window size: 6
    Maximum sliding window size: 10
    Times sliding window grew : 0
    Times sliding window did not grow: 2937

```

```

Maximum size sliding window grew: 6
Record Matching Rule      : a weighted average of all criteria
Criteria Levels Count    : 1
Field Criterion Level 1:
Field Criterion hits     : 14925
Field Criterion matches   : 1676
Field Criterion #1 of 3:
Criterion name           : Address
First Field                : ADDRESS
Second Field               : ADDRESS
Rule                      : an average of all algorithms
Weighting                 : 40%
Threshold                 : 0%
Case Sensitivity          : No
Ignore Whitespace          : Yes
Ignore Punctuation         : Yes
One Blank Match            : 0
Both Blank Match           : 0
Alignment                  : never align strings
Substring field 1          : No
Substring field 2          : No
Algorithm count             : 1
Algorithm #1 type          : Numeric and String comparison

```

List-by-list Report Phase

The list-by-list report categorizes matches by data list. The lists are arranged by name forming an n x n matrix, where n is the number of lists defined during the processing run. A field value represents the number of matches that involved records from both data lists.

For example, in the following list-by-list report, there were two intra-list matches between records in the Reference Data List. There were 988 matches between records in the Reference Data List and TU Data lists. There were 11 matches between the Reference Data List and Outside Data lists.

Centrus Merge/Purge version 1.6.0 Sagent Technology, Inc.		Page: 1			
List by List Report					
MITI List By List Report					
My Report Subtitle					
Date/Time: May 18, 1999 11:47 AM					

Summary: List By List Report					
Number of Records : 2000					
Number of Masters : 519					
Number of Subordinates : 528					
Number of Uniques : 953					
Default Reference Data List TU Data Outside Data					
Default	0	0	0		
Reference Data List	0	2	988		
TU Data	0	988	3		
Outside Data	0	11	11		

Summary: List Match Summary					
List Name	Input Count	IntraList Matches	InterList Matches	Total Matches	Percent of Input
Default	0	0	0	0	0.00
Reference Data List	500	2	499	501	99.20
TU Data	500	3	501	504	99.20
Outside Data	1000	26	11	37	5.50
Totals	2000	31	1011	1042	52.10

Near Miss Report Phase

The near miss report is a specialized duplicates report, and was designed to simplify the process of selecting a proper matching threshold. It lists dupe groups that have at least one member whose dupe group score is below a set threshold (called the acceptance level). All records listed in the report are valid matches (i.e. each record had a record matching score equal to or exceeding the record matcher threshold).

Notes:

- The near miss report supports record and dupe group statistical sampling.
- Some of the scores listed in the report may be below the record matching threshold. This is because the listed scores are subordinate/master match scores, not record-against-record scores generated by the record matcher. All records included in the near miss report matched another record at the record matching threshold or higher. When the dupe groups table is built, every subordinate in a dupe group is matched against the master duplicate (an operation not necessarily performed by the record matcher). The direct subordinate/master comparisons reveal subordinate duplicates that belong in the dupe group, but do not compare well with the master duplicate.
- The way duplicates are displayed in the report is fixed. The near miss report is always formatted as a master/subordinate report.
- In the M/S (master/subordinate) column, subordinate duplicates that score below the acceptance level are indicated by a '->' (arrow) character.

Below is an example of a near miss report. The address, city, state, zipcode, and date fields have been omitted for clarity:

Centrus Merge/Purge version 1.1.0							Page: 1
Sagent Technology, Inc.							
NearMiss Report by Master/Subordinate Ranking							
My Report Title							
My Report Subtitle							
Date/Time: September 11, 1998 9:45 AM							
<hr/>							
Summary: Near Miss Report							
Number of Records : 1016							
Number of Masters : 107							
Number of Subordinates : 302							
Number of Uniques : 607							
Record Match Threshold : 65							
Acceptance Level : 70							
M/S	DupeID	Score	ListID	Key	SrcID	FIRSTNAME	LASTNAME
M	2	100	2	2	8	ANTHON	ALMARALE
S	2	100	1	2	1	ANTHON	ALMARALE
S	2	100	1	3	1	ANTHON	ALMARALE
S	2	82	1	4	1	ATHON	ALMRALE
->	2	67	1	5	1	ANTHO	ALPARALE
M	17	100	1	42	1	CYNDIE	BENDEN
S	17	100	1	43	1	CYNDIE	BENDEN
S	17	78	1	44	1	CYNYIE	BENDN
->	17	68	1	45	1	CYNDIW	KENDEN
M	26	100	1	87	1	FREDRICK	BRODOSI
S	26	100	1	88	1	FREDRICK	BRODOSI

Phases

->	26	68	1	89	1	FRERICK	BODOSI
S	26	88	1	90	1	FREDRICKP	BRODOSMI
M	29	100	1	102	1	GUME	CAMEZ
S	29	100	1	103	1	GUME	CAMEZ
S	29	80	1	104	1	GCME	CAMEZ
->	29	66	1	105	1	UGME	ACMEZ
M	38	100	1	146	1	LICO	COMMARERI
S	38	73	1	147	1	LINETTA	CONTRELLA
S	38	73	1	148	1	LINETTA	CONTRELLA
->	38	67	1	149	1	LINETT	CONTRELLA
->	38	67	1	150	1	LINBTTA	CNOTRELLA

Unique Records Report Phase

The uniques report lists unique records (records that did not match other records) found in the processing run. Uniques come in handy when you would like to find records that don't match a reference list. For example, you might want to know how many names on a rented list are not already in a reference list. The uniques report not only tells you how many records don't match (are unique), but lists them as well.

The uniques report consists of a header section, a summary section, and the records themselves.

Centrus Merge/Purge version 1.0.0 Sagent Technology, Inc. Uniques Report	Page: 1			
<hr/>				
Rpt-Uniques-90.log: Uniques Report for a Threshold of 90				
Generated for Make-A-Wish Foundation				
Date/Time: May 22, 1998 9:36 AM				
<hr/>				
Summary: Uniques Report				
Number of Records: 42510				
Number of Masters: 13001				
Number of Subordinates: 13042				
Number of Uniques: 16467				
<hr/>				
Source	Key	Data List	FIRSTNAME	LASTNAME
1	5	1	ALVLIDA	LABERCCO
1	6	2	ALVILDAR	ALBERIC
1	10	2	ANHON	AJLMARALES

Table Generation Phase

Table generation allows you to capture the results of a Centrus Merge/Purge run in an output file. The table generation phase allows filtering of output records based on the following record characteristics:

- Unique/duplicate status
- Master/subordinate duplicate status
- Field value
- Specific data list membership
- Type of data list record belongs to (inclusion or suppression)

- Consolidated records
- Statistical sampling

For example, an output table could be created containing all master duplicates with incomes greater than 50,000 dollars. Another table might contain all uniques from the current subscriber data list. A third table might contain all suppression uniques from the subscriber data list, whose age is less than 25.

Statistical Sampling

Statistical sampling allows you to quickly filter output records in a controlled way. By including a small subset of records in an output table, you may evaluate the output before committing to a long processing run. Statistical sampling is useful for refining a job's settings without the overhead of processing an entire record set. Statistical sampling for output tables is optional. If it is used, a statistical sampling section will appear in the job summary report.

Statistical sampling allows you to filter records into an output table in the following ways:

Sampling Method	Example
Maximum number of records	Include 1,000 records in the table.
Include record X to record Y	Include record 850 to record 10,000 in the table.
Individual sampling (include every Nth record)	Filter every 5th record into the table.
Group sampling (include a group of M records starting at every Nth record)	At every 5th record, include a group of two records. In this example, records are included in the pattern 5, 6, 10, 11, 15, 16, 20, 21, . . .
Maximum number of dupe groups	Include the records from a maximum of 50 dupe groups.
Include the records in dupe group X to dupe group Y	Include the records from dupe group 10 to dupe group 20.
Dupe group sampling (include the records in every Nth dupe group)	Include the records from every 10th dupe group.

Filtering Output Fields

Centrus Merge/Purge allows you to specify which record fields are written to a specific output table. You may also direct the library to include or not include augmented fields in a specific output table.

Data Destinations

A data destination object encapsulates a physical output table in the same way that a data source surrounds a physical input table. The application creates a data destination object for every output table it wants to create, and will access the table only through the corresponding data destination object. A data destination defines the physical format of the output file, as well as the types of records that are put into it (uniques, duplicates, subordinate duplicates, consolidated records, etc.). Once the application creates a data destination object, it hands the object to the table generation phase, which will use the object to create the table.

Data Sources

A data source is a Centrus Merge/Purge object that fully encapsulates a physical source of input records. The application must create a data source object for every physical source of records. Data sources interact with data lists so that individual records in the data sources can be broken out or assigned to different “logical” data lists. The Centrus Merge/Purge library will always access records through the corresponding data source object. An application may have as many data sources as it wants. The application gives a data source to the data input phase, record matching phase, and other phases that use it.

Data sources come in three types: functor, table, and preprocessed.

Functor Data Source

The functor data source requires that the application write a callback function that the data input phase calls to get the records from the source of records. This callback fills up a supplied empty record with the field values for the next record in the data source, and then returns a value of “true” back to the data input phase.

When there are no more records left in the data source, the callback function (functor) returns “false”, indicating that all records have been given. The functor data source is used only by the data input phase.

Table Data Source

The table data source requires that the application “wrap” the source of records in one of the table objects supplied by Centrus Merge/Purge. The application gives the table object to the data source, and then Centrus Merge/Purge can interact with the table directly, rather than calling a functor to access the records. The table data source is used by only by the data input phase.

Preprocessed Data Source

The preprocessed data source is similar to the table data source. However, it is highly constrained in its content and format. The following list describes the properties of the preprocessed data source:

- It contains separate, sorted tables. A preprocessed data source will always contain one table sorted in S1K1 order, as well as N separate tables (one per index key definition that an application would normally define for jobs using this data source). The sort order of these extra tables is determined by the index key definitions.
- The S1K1 table has all of the same fields (name, type, and width) as the functor and table data sources. In addition, the S1K1 table is augmented with the required augmented fields that are normally added in the data input phase to the DITR. These are the augmented field contents:

Field Name	Type	Width	Comments
QQSRCID	long	5	Value must be the same as the preprocessed data source ID. Every source ID value must be the same number, unique from any other data source IDs. Calling <code>QmpDataSrcPrepare</code> will ensure that this value is the same as the preprocessed data source ID.
QQPRIKEY	long	11	Must have unique integers in it.
QQDATLST	long	5	Must be “current”. The application may call <code>QmpDataSrcPrepare</code> to update this field (assign data list membership).

The S1K1 table orders records by the source ID and primary key. It is used by the dupe groups phase and other post-record matching phases as a repository for looking up a record’s source.

- The other tables defined by the index key definitions contain only match fields and a derived key value field.

The preprocessed data source is not used by the data input phase. It is used by the record matching, report generation, table generation, and dupe groups phases.

You can use the table generation phase to create output tables for use as preprocessed data sources. For example, the results of one inclusion masters run might be used in subsequent runs to purge a file of unwanted duplicates. In another example, a suppression list could be generated to act as a preprocessed suppression list in subsequent runs.

Index Key Definitions

Index key definitions are user-supplied specifications of how the input records should be sequenced for the purpose of record matching. They define the desired index sequences that the user believes would group similar records together. An index key definition is roughly analogous to a “break key definition” in a traditional break group-oriented merge/purge process. The user creates one index key definition per desired index sequence. Each index key definition embodies an “expression” involving one or more record fields by which data records should be sequenced. These expressions are based on fields, subfields, and transformations of those fields (such as soundex).

The client application has the option of using multiple keys to sequence the input records (which is fortunate, because Centrus Merge/Purge identifies duplicate records most accurately when examining the data records in more than one sequence). This makes the matching technology especially robust. If at least one index key of the several used is very good, this compensates for poor results from a less useful index key. The better the index keys are at sequencing “true” duplicates together, the more successful the record matching will be.

Records

Record objects serve as data source-independent containers for data, and perform several important roles in a Centrus Merge/Purge application. They ferry input data processed by the library as well as define data field structures for the library.

Prototype Records

One of the application program’s responsibilities is to establish a common record format. This common format is used to read records from data sources, process records internally, and write record fields to output tables and reports. The application establishes this record format by creating a prototype record object and defining the object’s fields. Defining a field includes specifying the field name, type (numeric, character, date, etc.), and perhaps the data picture, decimals, or width. It is also the application program’s responsibility to hand the prototype record to the various phases that need to see it.

Note: In order for output tables and reports to display record date fields in the four-digit CCYY format, you must define date fields in the record prototype to use that format. This is an important precaution to take in order to avoid Year 2000 date-related problems.

Augmented Fields

The Centrus Merge/Purge library adds augmented fields to records stored internally. These fields hold values used for internal library calculations. The augmented fields include:

Field Name	Table Where Field Is Added	Description
QQSRCID	DITR	Source ID of the input record.
QQPRIKEY	DITR	Primary key of the input record.
QQDATLST	DITR	Data list ID of the input record.
“QQSX” + first six letters of transformed field name	DITR	Soundex transformation of an existing field. There may be more than one soundex field in a record.
QQDUPGRP	tempDGT	Dupe group ID of the input record.
QQSTATUS	tempDGT	M if the record is a master duplicate, S if a subordinate duplicate.
QQSCORE	tempDGT	The match score between a subordinate duplicate and the dupe group master duplicate.

The application may choose whether to include augmented fields with records written to output tables.

Soundex Augmented Fields

Soundex fields are added to the DITR so that the calculated soundex values may be used several times in later phases, rather than recalculated every time they are needed.

A soundex augmented field has a length of 6, and is a string field. If a soundex value has fewer than 6 digits, it is preceded with enough zeros to pad it to a length of 6.

For example, the soundex value for “Jackson” is “110080”. The soundex value for “Clarke” is “081403”. If this soundex field is used in an index, the “081403” will occur before “110080”.

Field Trimming

Centrus Merge/Purge supports field trimming to reduce unnecessary processing. In field trimming, blank characters in strings are deleted from the left or right of field values. You may specify which sides of a field, if any, you want to trim. The data input phase supports trimming when reading in record fields to the DITR. The table generation phase supports trimming when writing record fields to output tables.

Field Match Criteria

The field match criterion is an object that specifies how a field in one record should be compared with a field in another record. In a Centrus Merge/Purge application, a field match criterion is defined for each set of record fields to be matched. Each criterion calculates a field match score. The record matcher then uses the collection of field match scores to calculate a match score for the entire record.

A field match criterion consists of a pair of names of record fields to compare, what portion of the fields' values to compare, and how to perform the comparison. While a typical criterion defines a comparison between the same field in each record, cross field comparisons may also be specified.

A field match criterion uses one or more field match algorithms to compare the fields. If more than one algorithm is used, the application must specify how the algorithm match scores are combined into a single field match score. See the next section, “[Field Matching Algorithms](#),” for a discussion of the available algorithms and how multiple algorithm scores may be combined.

Field Matching Algorithms

A field match criterion uses one or more matching algorithms to compare fields. The library supplies the following algorithms:

Algorithm	Description
Character frequency	This algorithm assesses the frequency of occurrence of each character in each field and compares the overall frequencies between the fields. The results range from 0%, meaning none of the same characters appear in common between the field values, and 100%, meaning the same characters with the same count appear in both fields. The algorithm is particularly good at detecting character transpositions.
Edit distance	This algorithm determines how many characters need to be changed to convert one field value to the other field value. The result is the percentage of characters that do not have to be changed. The results can range from 0%, meaning all of the characters need to be changed, to a result of 100%, meaning none of the characters need to be changed.
Keyboard distance	This algorithm is similar to the edit distance algorithm, except that mismatched characters are penalized by how far apart on a keyboard are the desired and actual characters.

Algorithm	Description
Name matching	<p>This is a sophisticated algorithm that uses a name database to match different expressions of the same name. Matching features include:</p> <ul style="list-style-type: none"> Conjunctive matching ("Cindy and Marsha Brady" = "Cindy Brady") Alias matching ("Chuck Brown" = "Charlie Brown") Ability to specify order of names in file (first/last, last/first, or unknown order) <p>It has a number of matching options, such as:</p> <ul style="list-style-type: none"> Maiden matching ("Susan Black-White" = "Sue Black") Initial name matching ("Sue Black" = "S. Black") Match hygiene ("Dr. Susan White, Ph.D" = "Sue White") Initial alias matching ("Bill Brown" = "W. Brown") <p>The name matching algorithm will match both uni-field names contained in one string (e.g. "FirstName LastName"), or multi-field names split into several strings (e.g. "FirstName" field and "LastName" field).</p>
Numeric	This algorithm simply compares the fields' numeric values. They either match perfectly (100%), or not at all (0%).
Numeric string	This algorithm was created to compare address lines. It first matches numeric data in the string with the numeric algorithm. If the numeric data matches at 100%, the alphabetic data is matched using the edit distance and character frequency algorithms. The final match score is calculated as follows:
Final match score =	$\frac{100 + \left(\frac{\text{edit dist score} + \text{character freq score}}{2} \right)}{2}$
Soundex	This algorithm computes the soundex value of the fields' values and compares them. The fields either match perfectly (100%) or not at all (0%).
String comparison	This algorithm performs a simple string comparison. If the strings match perfectly, a match score of 100% is returned, otherwise a 0% score is returned. This algorithm is good for fields containing highly accurate values, such as a standardized ZIP code.

One or more of these algorithms may be added to a field match criterion. If more than one algorithm is added to a criterion, the match scores of the algorithms may be combined in one of the following ways:

- Use the average of the algorithm match scores
- Use the weighted average of the algorithm match scores
- Use the maximum algorithm match score
- Use the minimum algorithm match score

- Return a match score of 100% if all algorithm scores are greater than or equal to a threshold, otherwise return a score of 0%
- Return a match score of 100% if any algorithm score is greater than or equal to a threshold, otherwise return a score of 0%

When using the weighted average of algorithm match scores, you must also specify a weight for each algorithm. The total of the weights of the algorithms being used must be 100. For example, if you want to use both the soundex and edit distance algorithms on a last name field, and you want the edit distance score to be three times more important than the soundex score, assign a weight of 75 to the edit distance algorithm and 25 to the soundex algorithm.

Data Lists

While it is often useful to process records according to the data source they originated from, it is also useful to put records into other groupings for processing. The data list object fulfills this need by maintaining a list of records. Records are assigned to a data list based upon a set of application-defined rules. Each record in a job will be assigned to exactly one data list. Once all of the records have been assigned to a list, the phases may consult them to make record processing decisions. Some applications of data lists include:

- Filtering records to output tables based on data list membership
- Using a record's data list membership to determine the master dupe in a dupe group
- Directing that the records in two data lists should or should not be matched (inter-list matching)

A data list object is the specification for a single data list. When a data list object is created, the application must also set the list's associated properties. The data list properties include:

- The data list type
- The rules for assigning a record to the data list
- Priority
- Inter- and intra-list matching
- Output preference (inclusion or suppression)

Data List Types

Source

In this list type, each physical data source has its own data list. A simple example would be to assign data list 1, called “InHouse”, to a physical source of records that a company owns, and data list 2, called “Rented”, to a physical source of records that is being purchased from an outside broker.

A source data list may include several data sources in a single data list.

Fieldval

A fieldval data list uses an application-defined expression to evaluate a particular field of an input record. If the record’s field value causes the expression to be true, the fieldval data list claims the record as a member. The field being evaluated must be defined in the record prototype.

“State = Colorado” would be an example of such an expression.

Default

Any record not captured by one of the above list types is put into a default data list.

See [“Defining the Rules for Accepting a Record into a Data List” on page 311](#) for a discussion of this topic.

Data List Priority

The data list priority value is used to determine the master duplicate in a dupe group. If the application directs the master duplicate to be chosen by data list priority, the dupe that belongs to the highest priority data list becomes the master duplicate.

A “high” priority is one that has a low numeric value (for example, priority 1 is “higher” than priority 2). The highest priority that may be set by the application is 1. Priority 0 is reserved for internal use. The default priority (which is also the lowest possible priority) is 999.

Applications cannot set the priority of a suppression list. If an application changes a non-suppression list to a suppression list, the list’s priority is automatically changed to 0. If a suppression list is changed to a non-suppression list, the application must reset it to the desired priority level.

Inter- and Intra-list Matching

The application may specify if records belonging to the same data list should be compared (intra-list matching). It may also specify if records in two separate data lists should be compared (inter-list matching).

For example, imagine that you have two rented lists that are matched against an in-house reference list. The Centrus Merge/Purge library allows you to suppress matching between the two rented lists.

Inclusion and Suppression Data Lists

Data lists have an output preference property, which allows them to be classified as “inclusion” or “suppression” lists. This property is used by the table generation phase to determine which records to include in tables. Note that flagging a data list as “inclusion” will not automatically cause the records from that list to be output, nor will flagging a data list as “suppression” cause the records from the suppression list to be held back. Actual output is controlled with inclusion and suppression parameters passed to the table generation phase.

A suppression data list is automatically given a priority of 0; this value can not be changed by the application. Inclusion data lists may be assigned priority values from 1 through 999 (the default). An inclusion data list may be changed to type suppression, and vice versa.

Data List Service

The data list service is an object created by the application that manages and coordinates data list activities for clients of data lists, such as the data input phase and the record matching phase.

Clients use the data list service for different purposes:

Client	How it uses the data list service
Data input phase	Initially assign new records to a data list
Record matching phase	Determine if two records from the same data list should be compared
Dupe groups, Table generation, and Reports phases	Discover the properties of specific data lists that contain the records they are processing
Data consolidation phase	Consolidate records by data list
Preprocessed data source	Assign reference data records to data lists at the beginning of an application

Specifically, the data list service provides the following services:

- Manages a collection of data lists
- Queries the data lists in its service to determine which list “owns” a record
- Allows the application to set intra- and inter-list matching properties (will records in the same or different data lists be matched)

The application adds individual data lists to the data list service one at a time. Once the data list service has been populated, it is ready to use. As records are input by the data input phase, they are given to the data list service. The service evaluates the records and returns the ID of the data list which claimed the record as a member.

The data list service allows each of its contained data lists to examine the record. The data list service gives the record to the data lists in the order that the data lists were given to the data list service, waiting to let the default data list examine the record after all other registered data lists have had a chance at it. Once a record is assigned to a data list, subsequent data list rules are not evaluated (i.e. the first data list to claim a record as its member wins).

Data Consolidation Criteria

In data consolidation, the user must select the rules for choosing field values from dupe group records to write to the consolidated record. These rules are called data consolidation criteria. For each field in the prototype record, the user must select either zero or one of these consolidation criteria. For example, if the prototype record has eight fields, the user would add up to eight consolidation criteria to the data consolidation phase.

The following is a list of the consolidation criteria:

Criterion	Description
Master	The consolidated record copies field "X" from the master record, regardless of its content (even if it is blank). If the application chooses <i>not</i> to specify a consolidation criterion for a field, this criterion is automatically applied.
Non-blank	The consolidated record copies field "X" from the highest-ranking dupe with a non-blank field value. If the master dupe has a non-blank field value, it is used. Otherwise, the highest-ranking subordinate dupe with a non-blank field value is used. If all the records in the dupe group have a blank field "X", then the consolidated record will have a blank field "X", also.
Largest	The consolidated record takes the field "X" value from the duplicate with the highest field "X" value.
Smallest	The consolidated record takes the field "X" value from the duplicate with the smallest non-blank field "X" value.
Rollup	Add up all field "X" values in a dupe group and place that sum in the consolidated record field "X".
Average	Average all field "X" values in a dupe group and place that average in the consolidated record field "X".

Criterion	Description
Largest other	<p>In this criterion, some other field "Y" will be examined. Whichever dupe has the largest value in field "Y" will contribute the field "X" value.</p> <p>This criterion is appropriate to use when you want find the most recent address. For example, you might choose an address field from a record with the largest (i.e., most recent) date field.</p>
Smallest other	<p>In this criterion, some other field "Y" will be examined. Whichever dupe has the smallest value in field "Y" will contribute the field "X" value.</p> <p>This criterion is appropriate to use when you want to find the oldest member of a household. For example, you might choose a name field from a dupe with the smallest (i.e., oldest) "date of birth" field.</p>
Fldval	<p>In this criterion, some other field "Y" will be examined. The first dupe group record whose field "Y" value makes the associated inequality or equation true contributes the field "X" value. The master dupe is the first examined; after that, the subordinate dupes are examined in order of match score.</p> <p>This criterion is appropriate to use when you want to carry through sales figures for a certain region. For example, you could choose a sales revenue field value from a record with a region field equal to "Western".</p>
Data list	<p>The consolidated record's field "X" value is chosen from the highest scoring dupe which belongs to a specified data list.</p>
Most common	<p>The consolidated record's field "X" value is the most common, non-blank field "X" value among all the dupes in the dupe group.</p>
Val	<p>The application supplies the value that is directly placed into a consolidated record field.</p>
Num value	<p>This rule evaluates a field "X"/operator/constant expression, and counts the number of records in a dupe group that make the expression true. This record count becomes the consolidated field value.</p>
Num dupes	<p>Puts the number of duplicates in the dupe group into a specified field.</p>
Shortest	<p>Field "X" is examined in every dupe for the shortest non-blank value (in bytes). This value is written to the consolidated record's field "X". The field "X" value is treated as a string.</p>
Longest	<p>Field "X" is examined in every dupe for the longest value (in bytes). This value is written to the consolidated record's field "X". The field "X" value is treated as a string.</p>

Criterion	Description
Group order	The criterion specifies exactly which dupe will contribute a field "X" value. For this criterion, all dupes in a group are enumerated, with the master dupe assigned value 1, and the subordinate dupes enumerated according to their match score with the master (the lower the match, the higher the value). See the example in the next section.
Group order other	In this criterion, some other field "Y" will be examined. The criterion specifies exactly which dupe will contribute a field "Y" value, which is then placed in field "X". For this criterion, all dupes in a group are enumerated, with the master dupe assigned value 1, and the subordinate dupes enumerated according to their match score with the master (the lower the match, the higher the value). See the example below.

Consolidation Example

Imagine that you have four records in a dupe group with the following fields:

Group Order	FirstName	LastName	FirstName	FirstName	FirstName
			1	2	3
Master	1	Peter	Brady		
Subordinate	2	Greg	Brady		
Subordinate	3	Marsha	Brady		
Subordinate	4	Cindy	Brady		

Notice that the last three fields are not populated; they are reserved for use in the consolidated record. Subordinate 2 has the highest master/subordinate match score; subordinate 4 has the lowest score.

Tables

If you wanted to create the following consolidated record using the group order and group order other criteria,

FirstName	LastName	FirstName 1	FirstName 2	FirstName 3
Peter	Brady	Greg	Marsha	Cindy

you would assign the consolidation criteria to the fields in the following way:

	FirstName	LastName	FirstName 1	FirstName 2	FirstName 3
	Peter	Brady	Greg	Marsha	Cindy
Consolidation rule	Group order	Group order	Group order other	Group order other	Group order other
Group order number (duplicate number)	1	1	2	3	4
Field that value is taken from	N/A	N/A	FirstName	FirstName	FirstName

Tables

Centrus Merge/Purge offers a utility table object that applications can (and must) use for a number of different purposes. Tables are used for data input, data output, intermediate storage, and to communicate information between phases in a job. Because of Centrus Merge/Purge's built-in table mechanism, applications do not need to rely on an external database for table or record support.

Tables are created and destroyed through Centrus Merge/Purge library functions. After the application creates a table, it gives the table to the various clients (objects) that need to use it.

Table Types

Centrus Merge/Purge supports the following table types:

- CodeBase
- Oracle
- Text

Internal Intermediate Tables

DITR

The DITR is an internal representation of input records gathered by the data input phase. It contains records read in during data input from all data sources of type functor and table (but not of type preprocessed). An internal DITR record includes all fields from the prototype record and augmented fields added by the data input phase.

Match Result Repository

The match result repository contains the results of record comparisons performed by the record matcher. Applications create a match result repository object (which contains a table object) and give it to the record matcher to fill with record comparison results. When the record matcher is finished, the match result repository contains the complete list of matching records. The dupe groups phase uses this repository of record matching information to build its dupe groups table.

Dupe Groups Table

The dupe groups table stores the internal representation of the dupe groups data, and contains similar information to that found in the match result repository. A dupe groups table record does not contain source record fields; it only contains the following augmented fields:

- QQSRCID
- QQPRIKEY
- QQDUPGRP
- QQSTATUS
- QQSCORE

After the dupe groups phase is completed, the dupe groups table is sorted in source ID/primary key order. This ordering is useful for finding unique records.

TempDGT

The tempDGT (temporary dupe groups table) contains all of the records in the dupe groups table. The tempDGT records contain the source record fields as well as the augmented fields found in the corresponding DITR or preprocessed data source. Records in the tempDGT are sorted in dupe group order.

Since the tempDGT contains all the dupe group records with all of the source and augmented fields, it is consulted when making dupe group master/subordinate matches. Having a set of complete dupe group records sorted in dupe group order is also very handy for creating dupe group tables and reports.

Synthesized Consolidated Records Table

This table is used by the data consolidation phase to hold synthesized (new) consolidated records.

Updated Duplicates Consolidation Table

This table is used by the data consolidation phase to update duplicate records with consolidated field values.

External Tables

Data Source (Table or Preprocessed)

A table or preprocessed data source uses a table as a source of input records.

Data Destination

Each data destination represents an individual table that is created and filled by the table generation phase. A data destination may use any of the table types.

Event Notification Facility

The Centrus Merge/Purge library provides an event notification facility which the application can use to track the progress of a phase. At an application-specified record interval, an event message named `QMS_EVENT_EVERYNTHRECORD` is generated. The application registers functions with the Centrus Merge/Purge library to be invoked by the library when certain events occur. All of the phases except the list-by-list report and job summary report phases can produce the `QMS_EVENT_EVERYNTHRECORD` event. When registering a function to be called when the `QMS_EVENT_EVERYNTHRECORD` event occurs, the application also specifies the record count interval ("nth" record).

Error Handling, Logging, and Tracing

Errors generated by Centrus Merge/Purge fall into the following levels (represented by enumerated type values):

- `QMS_ERROR_INFORMATIVE`
- `QMS_ERROR_WARNING`
- `QMS_ERROR_SEVERE`
- `QMS_ERROR_FATAL`

Error Handling

An application may specify which error level should halt application execution. By default, QMS_ERROR_FATAL errors cause the application to terminate.

If a run-time error occurs, a message is sent to the error processing system inside Centrus Merge/Purge. This message is of type QRESULT and consists of an 8-digit decimal value. The QRESULT values, which are also used as library function return codes, are examined in “[QRESULT Return Codes](#)” on [page 897](#).

Before processing an error message, the error processing system checks to see if the application has defined an error handling function. This error handling function is set using the library function `QmpLogSetAppErrorHandler`. The error handler accepts a single argument of type `QMS_ERROR_INFO*` and returns a value of type `QMS_ERROR_ACTION`. The `QMS_ERROR_INFO` structure is a rich structure that contains several pieces of information about the current error, including the `QRESULT` value. The `QMS_ERROR_INFO` structure is documented on [page 84](#).

The application error handler takes any action necessary based on the error and then returns control to the Centrus Merge/Purge error handling system. For example, the error handler may look up the error text associated with the `QRESULT` value using `QmpLogLookupErrMsg`. Further action by the error handling system is based on the result returned by the application error handler:

- If `QMS_ERROR_ACTION_HANDLED` is returned, the Centrus Merge/Purge error handling system does no further processing and returns control to the function that generated the error.
- If `QMS_ERROR_ACTION_CONTINUE` is returned, the error is processed normally (in addition to any processing done by the application error handler).
- If `QMS_ERROR_ACTION_TERMINATE` is returned, the error handling system attempts to terminate the application.

Using the default behavior of the error handling system, or configuring the error handling system using the `QmpLog*` functions, should meet most users’ error handling needs. However, interactive applications may benefit by using an error handler to give the user a chance to correct an error before processing continues.

Error Logging

An application may specify at what levels errors should be reported via the logging facility, and where error reporting messages will be sent. Destinations may include any number of files and/or system outputs (such as `stdout` and `stderr`). By default, errors with a severity of `QMS_ERROR_WARNING` or higher are written to the file `error.log` and sent to `stderr`.

If the application turns on error logging and tracing, it is useful to send error tracing to the same destinations used for error logging. This combines error and tracing messages in the same destination.

The following is an example of an error log file:

```
***** SEVERE ERROR *****
*** Failure: *** 22000003: Severe - bad input
*** class CQmsIndexKey::AddKeyComponent
*** File: S:\cmp\Core\QIndxKey.cpp
*** Line 171
AddKeyComponent failed
***** SEVERE ERROR *****
*** Failure: *** 22000003: Severe - bad input
*** class CQmsIndexKey::AddKeyComponent
*** File: S:\cmp\Core\QIndxKey.cpp
*** Line 171
AddKeyComponent failed
***** SEVERE ERROR *****
*** Failure: *** 22000003: Severe - bad input
*** IndxKey::QmpIndxKeyAddCompByHnd
*** File: S:\cmp\Util\CWrapper.cpp
*** Line 1060
QmpIndxKeyAddCompByHnd failed
***** SEVERE ERROR *****
*** Failure: *** 22000003: Severe - bad input
*** IndxKey::QmpIndxKeyAddCompByHnd
*** File: S:\cmp\Util\CWrapper.cpp
*** Line 1060
QmpIndxKeyAddCompByHnd failed
```

Error Tracing

In addition to the error handling and logging system, the Centrus Merge/Purge library also has extensive tracing capabilities. Tracing is generally used only for debugging purposes. The application may turn on or off the execution tracing facility and set the level of tracing detail, ranging from level 0 through level 6.

The higher the level, the more detailed and voluminous the tracing information generated. For example, trace level 0 turns off tracing, while level 6 generates an enormous amount of output. The default trace level is 0.

The application controls where execution tracing messages are sent. Destinations may include any number of files and/or system outputs (such as stdout and stderr). By default, all trace output is sent to the file trace.log and to the standard output stream.

If the application turns on error logging and tracing, it is useful to send error tracing to the same destinations used for error logging. This combines error and tracing messages in the same destination.

Note: The higher the level of tracing, the more overhead the application incurs, and the slower the application will run.

Summary

A Centrus Merge/Purge application organizes the various merge/purge processes into a job. A job is split up into phases that represent distinct activities. Phases in turn need to be configured by the application, and use other object resources.

Summary

Chapter 4

Building Centrus Merge/Purge Applications

Introduction

This chapter discusses the responsibilities of the application vs. those of the library, and the components and flow of a typical Centrus Merge/Purge batch application. It also walks you through a sample batch application.

The Division of Labor in Centrus Merge/Purge

There is a natural division of responsibility and expertise between the user's application (which knows the data best) and Centrus Merge/Purge (which understands how to compare records best). It follows that there will be some activities and information which are the responsibility of the application program, and others that are owned by Centrus Merge/Purge.

Application Program Responsibilities

The application program handles these tasks:

- **Creates data sources.**
- **Creates data lists and the data list service.**

- **Specifies the index key definitions.** The application provides these index key definitions to the data input and record matching phases, which use these definitions to perform their functions. It is to the application's advantage to specify multiple index key definitions, because this will allow the record matcher to examine the input records in several index sequences.
- **Specifies the field matching criteria and algorithms.**
- **Specifies how the results of the algorithms and criteria will be combined.**
- **Inputs the records to be processed.** In batch applications, the application program passes the records to be processed to the Centrus Merge/Purge data input phase. If any record cleanup or normalization is required, the application program does it. In real-time applications, the application program passes the record(s) to be checked for duplication to the Centrus Merge/Purge library.
- **Creates data destinations and registers them** with the table generation phase.
- **Defines which reports to generate.**
- **Defines how dupe group records will be consolidated.**

Centrus Merge/Purge Responsibilities

Centrus Merge/Purge handles these tasks:

- **Gathers input records** from the client applications during data input phase for batch applications; accepts individual records for comparison during real-time applications.
- **Generates index key values** for any record using the application-supplied index key definitions.
- **Performs duplicate record detection** via the record matcher, which examines the records in the orders indicated by the index key definitions.
- **Performs match correlation** on the results via the dupe groups phase, which deduces dupe group membership.
- **Builds a table of dupe groups** and identifies the master and subordinate duplicates.
- **Consolidates dupe group records.**
- **Generates specified reports.**
- **Generates output tables.**

Components of a Typical Merge/Purge Batch Application

A typical merge/purge batch job uses the following components:

- Record prototype (one required)
- Intermediate tables (all required)
 - DITR
 - Match result repository
 - Dupe groups
 - tempDGT
- Index keys (one or more required)
 - Requires the record prototype
- Data lists
 - One default data list is required
 - Other data lists are optional
- Data list service (one required)
 - Requires data lists
 - Requires record prototype
- Data sources (one or more required)
- Field matching algorithms (one or more required)
- Field matching criteria (one or more required)
 - Requires algorithms
- Match result object (one required)
 - Requires match result repository
- Data destinations (one or more required if table generation phase is used)
- Data input phase (one required)
 - Requires record prototypes
 - Requires DITR
 - Requires index keys
 - Requires data list service
 - Requires data sources
- Record matching phase (one required)
 - Requires the record prototype
 - Requires the DITR

- Requires the match result repository
- Requires index keys
- Requires the match result object
- Requires the field matching criteria
- Requires the preprocessed data source (if used)
- Dupe groups phase (one required)
 - Requires the DITR
 - Requires the match result repository
 - Requires the data list service
 - Requires the match result object
 - Requires the record matching phase
 - Requires the preprocessed data source (if used)
 - Requires an application-created dupe groups table
 - Requires the tempDGT
- Table generation phase (optional)
 - Requires the record prototype
 - Requires the data list service
 - Requires dupe groups phase
 - Requires data destinations
 - Requires the preprocessed data source (if used)
- Duplicates report phase (optional)
 - Requires the record prototype
 - Requires the DITR
 - Requires the dupe groups phase
 - Requires the preprocessed data source (if used)
 - Requires consolidation phase (if duplicates are updated by the consolidation phase)
- Uniques report phase (optional)
 - Requires the record prototype
 - Requires the DITR
 - Requires the dupe groups phase
 - Requires the preprocessed data source (if used)

- List-by-list report phase (optional)
 - Requires the record prototype
 - Requires the data list service
 - Requires the dupe groups phase
 - Requires the match result object
 - Requires the preprocessed data source (if used)
- Job summary report phase (optional)
 - Requires the DITR
 - Requires the data list service
 - Requires any phases to be reported

Flow of a Typical Merge/Purge Batch Application

A typical merge/purge batch application takes the following steps:

1. Declare variables (use the `QmpDeclHnd` macro to allocate and initialize handles to `NULL`)
2. Configure logging and tracing (optional)
3. Create the record prototype
 - Include fields that are needed for ordering and matching
 - Also include fields that you want to appear in reports and output tables
 - Set passthrough field (optional)
 - Set field trimming (optional)
 - Add household ID field to records (optional)
4. Create the intermediate tables
 - DITR
 - Match result table
 - Create the match result object that uses this table
 - Dupe groups table
 - tempDGT
 - Consolidation table

5. Create the index keys
 - Create one key for each ordering of the data
 - Set additional index key properties
 - Attach the record prototype to the index keys
 - You must create at least one key (two or three keys are a good idea)
6. Create data lists
 - You must create a default data list; others are optional
7. Set the properties of the data lists
 - Set the action of the default data list
 - Set the data source IDs for source data lists
 - Set the expression for field value data lists
 - Set the priority for all data lists
 - Set optional flags, including output preference (inclusion/suppression) and intra-list matching (yes/no)
8. Create and set the properties of the data list service
 - Attach the record prototype to the data list service
 - Add the default data list—it must be added to the data list service before any other data list
 - Add other data lists in the order that you want their conditions evaluated
9. Create the Phases
 - Data input phase
 - Set field evaluation expression (if used)
 - Record matcher phase
 - Requires a criterion rule to combine criteria
 - Requires a match threshold
 - All of these settings may be changed later
 - Dupe groups phase
 - Requires the dupe groups table—can be changed later
 - Requires the tempDGT
 - Consolidation phase (optional)
 - Table generation phase (optional)
 - Report phases (all optional)
 - Job summary report
 - Dupe groups report

- Uniques report
 - List-by-list Report
- 10.** Add the record prototype to all phases except the dupe groups and job summary report phases using `QmpPhaseSetRecProto`
 - 11.** Add the DITR to all phases using the `Qmp*UseDITR` functions
 - 12.** Add the index keys to the:
 - Data input phase
 - Record matcher phase
 - **Note:** *The record prototype must be added to the index keys before they can be added to the phases.*
 - 13.** Add the data list service to the:
 - Data input phase
 - Record matcher phase
 - Dupe groups phase
 - Table generation phase
 - Job summary report
 - List-by-list summary report
 - 14.** Create the data sources
 - Requires data source IDs and types
 - Create tables for preprocessed and table data sources
 - Point to functions for functor data sources
 - Add tables and functions
 - Add record prototype to functor and table data sources
 - Prepare the preprocessed data source (if used)
 - Requires the data list service
 - 15.** Add the preprocessed data source (if used) to the:
 - Data input phase
 - Record matcher phase
 - Dupe groups phase
 - Table generation phase
 - Dupes report
 - Uniques report
 - List-by-list report
 - 16.** Configure the data input phase
 - Add the data sources

- 17. Create match algorithms**
 - Create one each of each type of algorithm you want to use
- 18. Create match criteria**
 - Requires rule and threshold—both can be changed later. Threshold is ignored if the rule is not QMS_ALGORITHMS_ADD or QMS_ALGORITHMS_OR.
 - Create one criterion for each pair of fields to be compared
- 19. Configure match criteria**
 - Add the fields to be compared (can be full or partial fields)
 - Add the algorithms to be used
 - Requires a weight that is only used if the criterion's rule is QMS_ALGORITHMS_WEIGHTED_AVERAGE—can be changed later
 - (Optional) set additional properties
 - Blank results
 - Alignment
 - Blank removal
 - Case sensitivity
 - Ignore numbers
 - Ignore letters
- 20. Configure record matcher phase**
 - Add criteria
 - Add match result object
 - Set sliding window size
- 21. Configure dupe groups phase**
 - Add record matcher phase
 - Add match result object
- 22. Configure data consolidation phase**
 - Create consolidation criteria
 - Add consolidation criteria
 - Add consolidation table
- 23. Create data destination tables**

- 24. Create and configure data destinations**
 - Requires data destination tables
 - Add the record prototype
 - Set flags
 - Inclusion/suppression
 - Master/subordinate/unique/consolidation
 - Add filter expression (optional)
 - Add output record; the order of fields within this record determines the order in the data destination
- 25. Configure the table generation phase**
 - Add dupe groups phase
 - Add data destinations
- 26. Configure reports**
 - Set file names
 - Set optional properties as needed
 - Title
 - Subtitle
 - Configure page
 - Duples report properties
 - Add dupe groups phase
 - Set grouping
 - Optional columns
 - Dupe group ID
 - Dupe group score
 - Source ID
 - List ID
 - Primary key
 - Match result columns
 - Consolidation report properties
 - Add dupe groups phase
 - Set grouping
 - Optional columns
 - Dupe group ID
 - Dupe group score
 - Source ID

- List ID
- Primary key
- Match result columns
- Uniques report properties
 - Add dupe groups phase
 - Optional columns
 - Primary key
 - Source ID
- Job summary report properties
 - Add phases to be monitored
- List-by-list report properties
 - Add dupe groups phase
 - Add match result phase

27. Run phases with `QmpPhaseStart`
28. Destroy all objects
29. Exit

It is important to realize that these steps represent a “generic” application, and that not every step needs to be completed in exactly this order. However, following this blueprint will result in a program that is easier to understand.

C Language Basics

“CmpAppl.h” is the only header file that needs to be explicitly included in the C application. This file includes the key C library header files, as well as the other necessary Centrus Merge/Purge header files for definitions, types, function prototypes, etc. Detailed information can be found in [“Data Types, Constants, and Macros”](#) beginning on [page 73](#).

All of the C function calls for the Centrus Merge/Purge library accept a variable of type `QRESULT` in which a return code is placed. The return code can be tested against the constant `QRESULT_YES` for success. Refer to [“QRESULT Return Codes”](#) beginning on [page 897](#) for further information on result code values.

How To Use Objects

Centrus Merge/Purge applications written in C manipulate the various components in the library using handles. A Centrus Merge/Purge handle is of type `MpHnd`, and may point to any object that can be instantiated. When the application calls a library function to create an object, it receives a handle to the created object.

Figure 5: Centrus Merge/Purge Object Hierarchy

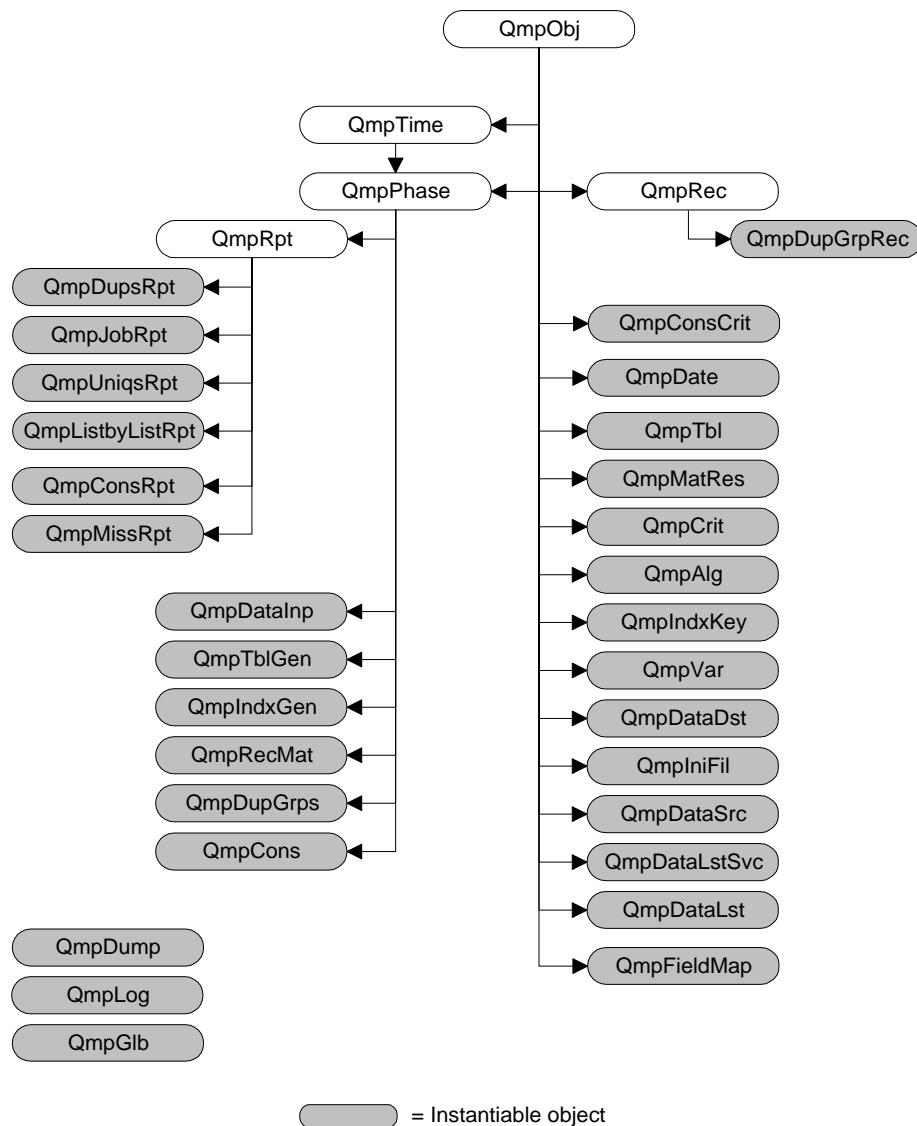


Figure 5 shows the parent/child relationships between Centrus Merge/Purge objects. In general, objects at the “tips” of the tree (objects that have no children) are the only ones that can be instantiated. For example, you can’t create an object of type `QmpRpt`, but you can create a `QmpDupsRpt` object.

Any function that accepts an instantiable parent object handle will also accept an instantiable child object handle. For example, the function `QmpTimeStartTiming` will accept a `QmpTime` handle, created by calling `QmpTimeCreate`. Since `QmpRecMat` is an instantiable object that is a child of `QmpTime`, `QmpTimeStartTiming` may also be called using a `QmpRecMat` handle.

For safety, handles should always be initialized to NULL upon declaration. For example:

```

MpHnd hTbl = NULL;
MpHnd hDataInput = NULL;
  
```

Allocating and initializing handles to `NULL` may also be accomplished by using the `QmpDeclHnd` macro:

```
QmpDeclHnd( hDataInput );
```

It is good practice to destroy objects immediately when they are no longer needed. It is also good practice to assign a `NULL` value to the handles of destroyed objects. An exception is raised when a `NULL` handle is dereferenced; assigning `NULL` values yourself will make it apparent if the application uses the handle of a destroyed object down the line. Note that an exception is raised for dereferencing an invalid handle as well as a `NULL`-initialized one.

Simple Application Sample

The following is a sample application named `cmpsimpl` that takes a single database table as input and generates a set of reports and a set of output tables. This sample application is available on the Centrus Merge/Purge CD-ROM.

The following list summarizes important application details:

- The table `cmpsimpl.dbf` is the data source.
- The data fields in the source table are:
 - Address1
 - City
 - FirstName
 - LastName
 - State
 - ZipCode
- Three data lists are created:
 - FieldVal—captures all records with a zip codes > 50000
 - Source—captures all records from the data source
 - Default—any record assigned here stops execution. All records should be caught by the first two data lists.
- Three passes over the data are made using the following orderings:
 - By Address1
 - By LastName
 - By ZipCode

- Field matching is performed on the following fields:
 - Address1 using the “Match By Edit Distance” algorithm on the first 10 characters
 - FirstName using the “Match By Edit Distance” and “Match By Soundex” algorithms (the highest-scoring algorithm is used)
 - LastName using the “Match By Edit Distance” and “Match By Soundex” algorithms (the highest-scoring algorithm is used)
 - ZipCode using the “Match By Edit Distance” algorithm on the first 5 characters
- The following reports are generated:
 - Duplicate records
 - Unique records
 - List-by-list
 - Job summary
- The following tables are generated:
 - Inclusion uniques
 - Suppression uniques*
 - Inclusion masters*
 - Suppression masters
 - Inclusion subordinates*
 - Suppression subordinates

* Using the *cmpsimpl.dbf* input table, these output tables are empty (but the files are still generated)

Executing *Cmpsimpl*

Cmpsimpl is totally self-contained and takes no command-line parameters. The executable name is *cmpsimpl.exe*. Once the program is started, *cmpsimpl* expects to find *cmpsimpl.dbf* in the default installation directory. It will then generate the set of output tables and reports in the current or working directory.

Flow of *Cmpsiml*

The following table breaks down *cmpsimpl.c* into its component tasks. The function that contains each task is also listed.

Task	Cmpsiml.c Function
Declare function prototypes (done in <i>cmpsimpl.h</i>)	
Declare and initialize handles to global Merge/Purge objects (done at beginning of file)	
Declare global variables (done at beginning of file)	
Initialize the application	Initialize
Get system wide log object	Initialize
Create data list service object	Initialize
Create data list objects	Initialize
Create data input object	Initialize
Create data source object	Initialize
Initialize pointer to callback function used to get data from data source	Initialize
Create record matcher object	Initialize
Create dupe groups table object	Initialize
Create dupe groups object	Initialize
Create inclusion uniques table object	Initialize
Create suppression uniques table object	Initialize
Create inclusion masters table object	Initialize
Create suppression masters table object	Initialize
Create inclusion subordinates table object	Initialize
Create suppression subordinates table object	Initialize
Create and set inclusion uniques data destination object	Initialize
Create and set suppression uniques data destination object	Initialize
Create and set inclusion masters data destination object	Initialize
Create and set suppression masters data destination object	Initialize
Create and set inclusion subordinates data destination object	Initialize
Create and set suppression subordinates data destination object	Initialize
Create table generator object	Initialize

Task	Cmpsiml.c Function
Create field matching criterion objects	<code>Initialize</code>
Create matching algorithm objects	<code>Initialize</code>
Create match result table object	<code>Initialize</code>
Create match result object	<code>Initialize</code>
Create job summary report object	<code>Initialize</code>
Create duplicates report object	<code>Initialize</code>
Create uniques report object	<code>Initialize</code>
Create list-by-list report object	<code>Initialize</code>
Create prototype record object	<code>Initialize</code>
Create index key objects	<code>Initialize</code>
Create DITR table	<code>Initialize</code>
Set Nth record interval callback function pointer	<code>Initialize</code>
Set matcher threshold	<code>Initialize</code>
Create input table object	<code>Initialize</code>
Create intermediate record object	<code>Initialize</code>
Set up the allocated Merge/Purge objects.	<code>Setup</code>
Set log indent character	<code>Setup</code>
Set the license file	<code>Setup</code>
Set up trace logging	<code>SetupJobTraceLogging</code>
Remove old dump log file so we don't append to it	<code>SetupJobTraceLogging</code>
Add a trace file and ostream	<code>SetupJobTraceLogging</code>
Turn tracing on or off	<code>SetupJobTraceLogging</code>
Set up error logging	<code>SetupJobErrorLogging</code>
Default error output is to error.log and to stderr, so no need to change it	<code>SetupJobErrorLogging</code>
Set up the structure for the prototype record	<code>SetupRecord</code>
Set up the index keys	<code>SetupIndexKeys</code>
Set up the data input phase	<code>SetupDataInput</code>
Configure the data list service so that the fieldval data list captures all records with zipcodes > 50000, the source data list captures all records from data source 1, and the default data list stops execution. This should work so that all records are caught by the two normal data lists.	<code>SetupDataInput</code>
Set output preference for data list, where: QMS_DATLST_OUTPUT_PREF_INCLUDE = 0 (this is an "inclusion" list) QMS_DATLST_OUTPUT_PREF_SUPPRESS = 1 (this is a "suppression" list)	<code>SetupDataInput</code>

Task	Cmpsiml.c Function
Configure the data source	<code>SetupDataInput</code>
Give data input the data list service and the three indexes of interest	<code>SetupDataInput</code>
Give the data input phase the prototype record	<code>SetupDataInput</code>
Delete the internal data records CodeBase table	<code>SetupDataInput</code>
Register the Nth record callback function and set the interval to 100	<code>SetupDataInput</code>
Set up field matching criteria	<code>SetupMatchCriteria</code>
Set up the match criterion for matching on the first 10 characters of the address	<code>SetupMatchCriteria</code>
Set up the match criterion for matching on FirstName	<code>SetupMatchCriteria</code>
Set up the match criterion for matching on LastName	<code>SetupMatchCriteria</code>
Set up the match criterion for matching on the first 5 digits of the ZipCode	<code>SetupMatchCriteria</code>
Set up record matching phase	<code>SetupRecordMatcher</code>
Destroy the physical representation of the record matcher results file	<code>SetupRecordMatcher</code>
Set miscellaneous properties	<code>SetupRecordMatcher</code>
Add criteria	<code>SetupRecordMatcher</code>
Indicate tables and other resources to use	<code>SetupRecordMatcher</code>
Indicate which index keys to use in examining the records	<code>SetupRecordMatcher</code>
Give the record matcher the prototype record	<code>SetupRecordMatcher</code>
Register the Nth record callback function and set the interval to 100	<code>SetupRecordMatcher</code>
Set up table generation phase	<code>SetupTableGenerator</code>
Give the DITR to the table generation phase	<code>SetupTableGenerator</code>
Give the table generator the dupe groups object	<code>SetupTableGenerator</code>
Give the table generator the data list service object	<code>SetupTableGenerator</code>
Give the data destinations to the table generation phase	<code>SetupTableGenerator</code>
Set up the input database	<code>SetupDatabase</code>
Open the input table	<code>SetupDatabase</code>
Set up the intermediate record to match the input table schema	<code>SetupDatabase</code>
Set up the (global) field handles to the intermediate record (for use by the fill function later)	<code>SetupDatabase</code>

Task	Cmpimpl.c Function
Set up reports	<code>SetupReports</code>
Run the application	<code>Run</code>
Start data input and record matching phases	<code>Run</code>
Change the settings for error and trace logging	<code>Run</code>
Set up dupe groups	
Run dupe groups	<code>Run</code>
Start the table generation phase	<code>Run</code>
Dump the match result and dupe groups objects to a file. (This is not necessary in a normal application. One application of this is to compare the dumped data with that of a previous run for testing code changes that might affect matching results.)	<code>DoDumps</code>
Print all reports	<code>PrintReports</code>
Print a duplicates report	<code>PrintReports</code>
Print a uniques report	<code>PrintReports</code>
Print a list-by-list report	<code>PrintReports</code>
Print a job summary report	<code>PrintReports</code>
Destroy all allocated Merge/Purge objects	<code>CleanUp</code>

Cmpimpl Code

The *cmpimpl* executable file, source code, and sample input table *cmpimpl.dbf* are provided on the Merge/Purge library CD-ROM. For more information, refer to the “readme” file *cmpimpl.txt* (also contained on the library CD-ROM).

Chapter 5

Data Types, Constants, and Macros

This chapter documents all data types, constants, and macros belonging to the Centrus Merge/Purge library C language API.

Data Types

MpHnd

Description

The “handle” type for all Centrus Merge/Purge objects accessed through the C API. This handle is an opaque reference to the underlying C++ object which actually implements the true Centrus Merge/Purge object.

Summary

```
typedef struct TAG_MpHnd{ int i; } *MpHnd;
```

QBOOL

Description

A boolean type used in Centrus Merge/Purge. The value is either QTRUE or QFALSE.

Summary

```
typedef unsigned char QBOOL;
```

QDATESTRUCT

Description

The string contained in the structure holds a date value plus one byte for the NULL character.

Summary

```
typedef struct _qmsdate {
    char ccyyymmdd[9];
} QDATESTRUCT;
```

QMS_ALGORITHMS_RULE

Description

This enumeration defines the available ways that the results of algorithms can be combined during the processing of a field match criterion. They describe how a criterion evaluates the results of its contained collection of field matching algorithms. Algorithm results are always a score from 0 (no match) to 100 (perfect match). Because a field match criterion can contain multiple algorithms, the application can instruct the criterion how to combine the results of each of those algorithms when coming up with a score for the entire criterion.

- “Average” just takes the arithmetic average of all the algorithm results.
- “Weighted Average” weights each algorithms with an application-supplied weight. The total of all algorithm weights must add up to 100.
- “Max” uses only the maximum score from any of the algorithms.
- “Min” uses the minimum score from any of the algorithms.
- “And” requires that all algorithms meet an application-supplied criterion threshold.
- “Or” requires that at least one algorithm meet an application-supplied criterion threshold.

Summary

```
typedef enum QMS_ALGORITHMS_RULE {
    QMS_ALGORITHMS_UNDEFINED,
    QMS_ALGORITHMS_AVERAGE,
    QMS_ALGORITHMS_WEIGHTED_AVERAGE,
    QMS_ALGORITHMS_MAX,
    QMS_ALGORITHMS_MIN,
    QMS_ALGORITHMS_AND,
    QMS_ALGORITHMS_OR
} QMS_ALGORITHMS_RULE;
```

QMS_ALGORITHM_TYPE

Description

This enumeration contains the types of field match algorithms. The function `QmpAlgCreate` uses this type to create algorithm objects.

Summary

```
typedef enum QMS_ALGORITHM_TYPE {
    QMS_ALGORITHM_TYPE_CHARFREQ,
    QMS_ALGORITHM_TYPE_EDITDIST,
    QMS_ALGORITHM_TYPE_KEYDIST,
    QMS_ALGORITHM_TYPE_SOUNDDEX,
    QMS_ALGORITHM_TYPE_STRING,
    QMS_ALGORITHM_TYPE_NUMERIC,
    QMS_ALGORITHM_TYPE_NUMERICSTRING,
    QMS_ALGORITHM_TYPE_NAME
} QMS_ALGORITHM_TYPE;
```

QMS_ALIGN_OPTION

Description

Enumeration of String alignment options. These rules describe how a field matching criterion aligns incoming strings from two records. Algorithms that utilize “distance” concepts (edit distance and keyboard distance) benefit from using pre-aligned strings.

Summary

```
typedef enum QMS_ALIGN_OPTION {
    QMS_ALIGN_ASNEEDED,
    QMS_ALIGN_NEVER,
    QMS_ALIGN_ALWAYS
} QMS_ALIGN_OPTION;
```

QMS_CONSOL_RULE

Description

This type is used in data consolidation and defines the different rules a consolidation criterion can use to consolidate a field.

Summary

```
typedef enum QMS_CONSOL_RULE {
    QMS_CONSOL_RULE_UNDEFINED,
    QMS_CONSOL_RULE_NONE,
    QMS_CONSOL_RULE_MASTER,
    QMS_CONSOL_RULE_NON_BLANK,
    QMS_CONSOL_RULE_ROLLUP,
    QMS_CONSOL_RULE_AVERAGE,
    QMS_CONSOL_RULE_NUM_DUPES,
    QMS_CONSOL_RULE_SHORTEST,
    QMS_CONSOL_RULE_LONGEST,
    QMS_CONSOL_RULE_LARGEST,
    QMS_CONSOL_RULE_SMALLEST,
    QMS_CONSOL_RULE_MOST_COMMON,
    QMS_CONSOL_RULE_LARGEST_OTHER,
    QMS_CONSOL_RULE_SMALLEST_OTHER,
    QMS_CONSOL_RULE_FLDVAL,
    QMS_CONSOL_RULE_NUM_VALUE,
    QMS_CONSOL_RULE_DATLST,
    QMS_CONSOL_RULE_VAL,
    QMS_CONSOL_RULE_GROUP_ORDER,
    QMS_CONSOL_RULE_GROUP_ORDER_OTHER
} QMS_CONSOL_RULE;
```

QMS_CONSOL_TREAT

Description

This type is used during data consolidation. It is applied to certain types of consolidation criteria to give them information on how a field should be interpreted during consolidation.

Summary

```
typedef enum QMS_CONSOL_TREAT {
    QMS_CONSOL_TREAT_UNDEFINED,
    QMS_CONSOL_TREAT_STRING,
    QMS_CONSOL_TREAT_NUMBER,
    QMS_CONSOL_TREAT_DATE
} QMS_CONSOL_TREAT;
```

QMS_CRITERIA_RULE

Description

This enumeration defines the available ways that the results of criteria can be combined during the record matching phase. They describe how the record matcher evaluates the results of its contained collection of field matching criteria. Criteria results are always a score from 0 (no match) to 100 (perfect match). Because a record matcher can contain multiple criteria (usually one criterion per “match field”), the application can instruct the record matcher how to combine the results of each of those criteria when coming up with a score for the entire record-to-record comparison.

- “Average” just takes the arithmetic average of all the criteria results.
- “Weighted Average” weights each criterion with an application-supplied weight. The total of all criterion weights must add up to 100.
- “Max” uses only the maximum score from any of the criteria.
- “Min” uses the minimum score from any of the criteria.
- “And” requires that all criteria meet an application-supplied criterion threshold.
- “Or” requires that at least one criterion meet an application-supplied record matcher threshold.

Summary

```
typedef enum QMS_CRITERIA_RULE {
    QMS_CRITERIA_UNDEFINED,
    QMS_CRITERIA_AVERAGE,
    QMS_CRITERIA_WEIGHTED_AVERAGE,
    QMS_CRITERIA_MAX,
    QMS_CRITERIA_MIN,
    QMS_CRITERIA_AND,
    QMS_CRITERIA_OR
} QMS_CRITERIA_RULE;
```

QMS_DATLST_ACTION

Description

This enumeration is the type of the answer a data list gives the data list server when asked if a record is a member of the data list. The application can set the “action” for the default data list, which is always required to exist in the data list service, and if accompanied by other data lists, is always the last one to have a chance to claim the record as a member. The allowable actions for it are “YES”, “IGNORE”, and “STOP”.

- Yes - This means that the data list has accepted as a member a record that it encountered. A source, fieldval, or default data list can support this action.
- Ignore - The data list suggests that the record be completely ignored. Only a default data list can support this action.
- Stop execution - The data list suggests that execution of the phase be discontinued. Only a default data list can support this action.

Other values in this enumeration are:

- None - not a legal value for a data list to have.
- No - This means that the data list has not accepted as a member a record that it encountered. A source or fieldval data list can support this action. It would return this record if (in the case of a source-type data list) the record came from a source different from any given to the source-type data list. It would return this value (in the case of a fieldval-type data list) if the evaluation of the specified field did not match the expression which the fieldval-type data list is using to judge records.
- Retry - Tells the data list service to start over again at the first data list in its collection to determine the membership of the current record. Only a extension data list can support this action (accessible through C++ interface only).

You can set the “action” property of the default data list to “yes” (accept any record it encounters), “ignore” (ignore the record entirely), or “stop execution” which is a suggestion to stop the execution of the phase. During the data input phase, if the data list service returns an “ignore” to data input for the current record being input, that record will not appear in the DITR. If the data list service returns “stop execution” to the data input phase, the data input phase will stop reading records and gracefully finish.

See [QmpDataLstSvcDetMem](#), [QmpDataLstIsMember](#),
[QmpDataLstGetAction](#), [QmpDataLstSetAction](#).

Summary

```
typedef enum QMS_DATLST_ACTION {
    QMS_DATLST_ACTION_NONE,
    QMS_DATLST_ACTION_YES,
    QMS_DATLST_ACTION_NO,
    QMS_DATLST_ACTION_RETRY,
    QMS_DATLST_ACTION_IGNORE_RECORD,
    QMS_DATLST_ACTION_STOP_EXECUTE
} QMS_DATLST_ACTION;
```

QMS_DATLST_OUTPUT_PREF

Description

This enumerator defines the possibilities for output preference for a Data List. Data lists have an output preference property that can take one of these two values. Typically, a data list with an output property of “inclusion” is one whose records you would want to see output to a table. A data list with an output property of “suppression” is typically one that you would want to see suppressed from being output. A standard usage of it would be to associate all records from a reference file with a suppression data list, and all records from a “raw” data source with an inclusion data list. You would then create a data destination which asked for inclusion uniques, and inclusion masters. This would result in an output file that was purged of all duplicates, plus it was purged of any records that matched a record in the suppression reference table.

By definition, a data list with an output preference of “suppression” has a higher priority than any other “inclusion” data list, and the same priority as any other “suppression” data list. So if a dupe group contains a record from a suppression data list, that record will be the master duplicate.

See [QmpDataLstGetOutputPref](#), [QmpDataLstSetOutputPref](#),
[QmpDataDestSetOutputTypes](#),
[QmpDataDestSetOutputTypesWithBitmask](#),
[QmpDataDestSetInclUniques](#), [QmpDataDestSetInclMasters](#),
[QmpDataDestSetInclSubords](#), [QmpDataDestGetInclUniques](#),
[QmpDataDestGetInclMasters](#), [QmpDataDestGetInclSubords](#).

Summary

```
typedef enum QMS_DATLST_OUTPUT_PREF {
    QMS_DATLST_OUTPUT_PREF_INCLUDE,
    QMS_DATLST_OUTPUT_PREF_SUPPRESS
} QMS_DATLST_OUTPUT_PREF;
```

QMS_DATLST_TYPE

Description

This enumeration contains the possible types of data lists. The data lists are gathered together in a data list service which coordinates their activities. The data list service has several clients – among them are the data input phase which uses it to initially categorize the input records into data lists, and the **QmpDataSrcPrepare** function, which uses it to update the data list membership of a preprocessed data source. The types of data lists are:

- Default data list is the only required data list in the data list service. If you have a data list service, it must at least contain a default data list. The purpose of the default data list is to be a “last chance” catch-all data list for records that no other data list has claimed. You can set the “action” property of this data list to “yes” (accept any record it encounters), “ignore” (ignore the record entirely), or “stop execution” which is a suggestion to stop the execution of the phase. During the data input phase, if the data list service returns an “ignore” to data input for the current record being input, that record will not appear in the DITR. If the data list service returns “stop execution” to the data input phase, the data input phase will stop reading records and gracefully finish.
- Source data list – the application specifies one or more data source ids. This data list will accept any record which comes from a data source which matches its collection of data source ids.
- FieldVal data list – the application specifies a simple expression on a single field. Any record examined by this data list which has a value in the field which makes the expression true will be accepted by this data list. As an example, “Age > 21”.
- Expression data list – not currently supported, but intended to allow more complex expressions on multiple fields.

See **QmpDataLstCreate**.

Summary

```
typedef enum QMS_DATLST_TYPE {
    QMS_DATLST_TYPE_DEFAULT,
    QMS_DATLST_TYPE_SOURCE,
    QMS_DATLST_TYPE_FIELDVAL,
    QMS_DATLST_TYPE_EXPRESSION
} QMS_DATLST_TYPE;
```

QMS_DATSRC_TYPE

Description

This enumeration data type defines the types of data sources an application can create.

- The functor data type is one which the application gives a callback function to. The Centrus Merge/Purge library invokes this callback for each input record from the data source. The callback keeps filling an supplied I/O record with the values with the next record in the data source. It returns QTRUE as long as there are more records, and QFALSE when it runs out of records.
- The table data source is one which the application gives a Centrus Merge/Purge table object to. The data input phase will use normal Centrus Merge/Purge table interfaces to read the values.
- The preprocessed data source is similar to the table data source. It will always contain one table sorted in S1K1 order, as well as N separate tables (one per index key definition that an application would normally define for jobs using this data source). The sort order of these extra tables is determined by the index key definitions. A preprocessed data source is not read in through the data input phase, but first enters the Centrus Merge/Purge system during record matching. It bypasses data input because it already has everything in it that the data input phase would normally add. This type of data source is commonly used for reference tables of existing information, and is run against less well known data coming in through a functor or table data source. See “[Preprocessed Data Source](#)” on page 37 for more information.

See [QmpDataSrcCreate](#).

Summary

```
typedef enum QMS_DATSRC_TYPE {
    QMS_DATSRC_TYPE_NONE,
    QMS_DATSRC_TYPE_FUNCTOR,
    QMS_DATSRC_TYPE_TABLE,
    QMS_DATSRC_TYPE_PREPRO
} QMS_DATSRC_TYPE;
```

QMS_DISK_USAGE

Description

This structure is passed to `QmpPhaseDiskSpace`, which uses it to store the disk space requirements of phases. The client is responsible for initializing the structure before using.

Summary

```
typedef struct _QMS_DISK_USAGE {
    long lDitrRecCount;
    long lDitrRecWidth;
    long lPreProRecCount;
    long lSoundexCount;
    long lTotalUsage;
    long lDataInputUsage;
    long lMatchUsage;
    long lMatchRes5Usage;
    long lDupeTable5Usage;
    long lTempDgt5Usage;
    long lDupeConsUsage;
    long lConsUsage;
    long lTableGenUsage;
    long lDupeRptUsage;
    long lMissRptUsage;
    long lConsRptUsage;
    long lUniqRptUsage;
    long lListRptUsage;
    long lJobRptUsage;
} QMS_DISK_USAGE;
```

Structure Member	Explanation
IDitrRecCount	Count of records in the DITR
IDitrRecWidth	Width of records in the DITR
IPreProRecCount	Count of records in preprocessed data source
ISoundexCount	Count of indexes using soundex
ITotalUsage	Total predicted disk usage
The following are per phase disk usage subtotals:	
IDataInputUsage	Predicted disk space usage for data input
IMatchUsage	Predicted disk space usage for record matching
IMatchRes5Usage	Predicted disk space usage for match result at 5%
IDupeTable5Usage	Predicted disk space usage for dupe table at 5%
ITempDgt5Usage	Predicted disk space usage for tempDGT at 5%
IDupeConsUsage	Predicted disk space usage for dupe groups consolidation
IConsUsage	Predicted disk space usage for Data consolidation

Structure Member	Explanation
ITableGenUsage	Predicted disk space usage for table generation
IDupeRptUsage	Predicted disk space usage for dupes report
IMissRptUsage	Predicted disk space usage for near miss report
IConsRptUsage	Predicted disk space usage for consolidated report
IUniqRptUsage	Predicted disk space usage for uniques report
IListRptUsage	Predicted disk space usage for listbylist report
IJobRptUsage	Predicted disk space usage for job summary report

QMS_ERROR_ACTION

Description

Error action enumeration type. These are the values that the application-supplied error handler can return to the error processing system in the Centrus Merge/Purge library when it is invoked. The application-supplied error handler is called by the error processing system before any other error processing logic is executed. This is done to allow the application to have an opportunity to examine the error, and potentially do something about it.

Note that the application error handler will not be able to make the error go away. However, it can take corrective action, enabling the application to then re-try the function which originally generated the error. The type of actions are:

- Handled – The application error handler has determined that the error has been dealt with adequately. The Centrus Merge/Purge error processing system will do no further reporting on THIS error. However, the nature of error handling in the Centrus Merge/Purge library is that if an error occurs deep inside a function, errors are reported at that deep level, and at each succeeding level above it as it backs out of the error situation. The application error handler would be invoked for each one of these until the library has completed its unwinding of the stack. If the application wishes to hide all of these error messages, it could return QMS_ERR_ACTION_HANDLED for each one of them.
- Continue – The application error handler has determined that this is a legitimate error, and that the Centrus Merge/Purge error processing system should deal with it in the normal fashion.
- Terminate - The application error handler has determined that this is a very serious error. Regardless of the current “error fatality level” that the application has set, the Centrus Merge/Purge error handling system should terminate.

Summary

```
typedef enum QMS_ERROR_ACTION {
    QMS_ERROR_ACTION_HANDLED,
    QMS_ERROR_ACTION_CONTINUE,
    QMS_ERROR_ACTION_TERMINATE
} QMS_ERROR_ACTION;
```

QMS_ERROR_HANDLER

Description

Typedef for application-supplied callback for processing errors from the error logging facility. Applications can install a single one of these to intercept errors that occur in the library before normal error reporting occurs.

The error handler function takes a pointer to the QMS_ERROR_INFO structure as a parameter. Refer to “[QMS_ERROR_INFO](#)” in this chapter for a description of the structure.

Returns QMS_ERROR_ACTION.

Summary

```
typedef QMS_ERROR_ACTION (* QMS_ERROR_HANDLER ) (
    QMS_ERROR_INFO* in_pErrorInfo );
```

QMS_ERROR_INFO

Description

The QMS_ERROR_INFO structure contains error information and is passed to the application-supplied error handler function. Refer to “[QMS_ERROR_HANDLER](#)” in this chapter for a description of how this structure is used with an error handler function.

Summary

```
typedef struct _QMS_ERROR_INFO {
    QRESULTqres;
    char    szClassName[ QMS_MAX_NAME ];
    char    szFuncName[ QMS_MAX_NAME ];
    long    lLineNum;
    char    szFile[ QMS_MAX_NAME ];
    QBOOL   bIsWrapper;
    char    szCustomMsg[ QMS_MAX_NAME ];
    char    szNotUsed[ 2048 ];
} QMS_ERROR_INFO;
```

Structure	
Member	Explanation
qres	The result code
szClassName	The class name (or for C wrapper, the functional area)
szFuncName	The function name
ILineNum	The line number of the error
szFile	The file in which the error occurred
blsWrapper	True if generated by the wrapper, false if not
szCustomMsg	A custom message (the standard message can be “looked up”)
szNotUsed	Reserved for future use

QMS_ERROR_LEVEL

Description

These are the error reporting and fatality levels. The application has separate control (through the `QmpLog*` functions) for the error reporting and error fatality levels. Error reporting level means the severity that an error must have before it is output to the registered error logs. Error severity level means the severity that an error must have before it causes the system to terminate. If `QMS_ERROR_NONE` is specified for either level, then no error reporting will occur, and no generated Centrus Merge/Purge error will cause the system to initiate a controlled shut down.

See `QmpLogGetErrFatalLvl`, `QmpLogSetErrFatalLvl`,
`QmpLogGetErrRptLvl`, `QmpLogSetErrRptLvl`.

Summary

```
typedef enum QMS_ERROR_LEVEL {
    QMS_ERROR_INFORMATIVE ,
    QMS_ERROR_WARNING ,
    QMS_ERROR_SEVERE ,
    QMS_ERROR_FATAL ,
    QMS_ERROR_NONE
} QMS_ERROR_LEVEL;
```

QMS_EVENT_TYPE

Description

This enumeration defines the types of events that are available in the Centrus Merge/Purge library. The event most commonly requested by applications is QMS_EVENT_EVERYNTHRECORD. The other events are internally generated and are used by utility classes in the table generator.

Summary

```
typedef enum QMS_EVENT_TYPE {
    QMS_EVENT_UNDEFINED,
    QMS_EVENT_EVERYNTHRECORD1,
    QMS_EVENT_UNIQUE_RECORDS,
    QMS_EVENT_MASTER_RECORDS,
    QMS_EVENT_SUBORD_RECORDS,
    QMS_EVENT_CONSOL_RECORDS
} QMS_EVENT_TYPE;
```

QMS_EVERY_NTHREC_FUNC

Description

Typedef for application-supplied event handler for processing every Nth record” type events from the various phases.

Returns void.

Summary

```
typedef void (* QMS_EVERY_NTHREC_FUNC ) (
    unsigned longin_uID,
    char*     in_szName,
    long      in_lRecCnt
);
```

Structure Member	Explanation
in_uID	Object ID of originator of event
in_szName	Name of originator of event
in_lRecCnt	Nth record count

QMS_FIELD_TRIM

Description

This enumerated type controls trimming properties for field values read in from a data source or written to a data destination. “Trimming” means deleting blank characters (for strings).

Summary

```
typedef enum QMS_FIELD_TRIM {
    QMS_FIELD_TRIM_NONE,
    QMS_FIELD_TRIM_LEFT,
    QMS_FIELD_TRIM_RIGHT,
    QMS_FIELD_TRIM_BOTH
} QMS_FIELD_TRIM;
```

Structure Member	Explanation
QMS_FIELD_TRIM_NONE	Don't trim.
QMS_FIELD_TRIM_LEFT	Trim to the left of the field value.
QMS_FIELD_TRIM_RIGHT	Trim to the right of the field value.
QMS_FIELD_TRIM_BOTH	Trim both sides of the field value.

QMS_FILL_REC_FUNC

Description

Typedef for application-supplied event handler for reading next record into a Functor style data source. This function MUST be implemented by the application and is set in the functor data source.

Returns `QTRUE` when a new record is successfully gotten; `QFALSE` when there are no more records to process.

Summary

```
typedef QBOOL (* QMS_FILL_REC_FUNC ) (
    MpHnd io_hRec,
    MpHnd in_hCaller
);
```

Structure Member	Explanation
io_hRec	Handle of record to fill
in_hCaller	Handle of the record matcher or match correlator

QMS_FLDEVAL_OPER

Description

This enumeration is used in functions that set field evaluation expressions. This defines the “comparator” between the field and the constant in an expression. See `QmpDataDestSetExprLong`, `QmpDataDestSetExprFloat`, `QmpDataDestSetExprString`, `QmpDataLstSetExprLong`, `QmpDataLstSetExprFloat`, `QmpDataLstSetExprString`, `QmpDataSrcSetExprLong`, `QmpDataSrcSetExprFloat`, `QmpDataSrcSetExprString`.

Summary

```
typedef enum QMS_FLDEVAL_OPER {
    QMS_FLDEVAL_OPER_NONE,
    QMS_FLDEVAL_OPER_LESS,
    QMS_FLDEVAL_OPER_EQUAL,
    QMS_FLDEVAL_OPER_GREATER,
    QMS_FLDEVAL_OPER_LESSEQUAL, /* not currently used */
    QMS_FLDEVAL_OPER_GREATEREQUAL,/* not currently used */
    QMS_FLDEVAL_OPER_NOTLESS,
    QMS_FLDEVAL_OPER_NOTEQUAL,
    QMS_FLDEVAL_OPER_NOTGREATER,
} QMS_FLDEVAL_OPER;
```

QMS_INDXKEY_XFORM_TYPE

Description

This enumeration lists the types of index key field transformations that can be used in `QmpIdxKeyAddCompByName` and `QmpIdxKeyAddCompByHnd`. The identity transformation does nothing to the fields being used in the index. The soundex transformation applies the Soundex algorithm to the fields before using them in the index.

Summary

```
typedef enum QMS_INDXKEY_XFORM_TYPE {
    QMS_INDXKEY_XFORM_IDENTITY,
    QMS_INDXKEY_XFORM_SOUNDEX
} QMS_INDXKEY_XFORM_TYPE;
```

QMS_MATCH_GROUPING

Description

This enumeration contains the possible ways to print groups of duplicates in a duplicates report.

- “By dupe group” will cause the report to just group the output records by their dupe group IDs.
- “By master ranking groups” groups by dupe group ID, and also identifies the master record and the subordinate records.

See [QmpDupsRptSetGrouping](#).

Summary

```
typedef enum QMS_MATCH_GROUPING {
    QMS_GROUP_MATCH_BY_DUP_GROUP,
    QMS_GROUP_MATCH_BY_MASTER_RANKING
} QMS_MATCH_GROUPING;
```

QMS_NAME_ORDER

Description

This enumerated type is used to inform the name match algorithm of the order of the names it will be matching.

Summary

```
typedef enum QMS_NAME_ORDER {
    QMS_NAME_ORDER_NONE,
    QMS_NAME_ORDER_FIRSTLAST,
    QMS_NAME_ORDER_LASTFIRST
} QMS_NAME_ORDER;
```

QMS_OUTPUT_SAMPLING

Description

QMS_OUTPUT_SAMPLING is filled by a client and passed to a data destination object. It defines the pattern of records to use during table output. In every member, a value of 0 means “this part not specified.”

Summary

```
typedef struct _QMS_OUTPUT_SAMPLING {
    long lMaxRecords;
    long lFirstRecordNum;
    long lLastRecordNum;
    long lInterval;
    long lGroupSize;
    long lMaxDupeGroups;
    long lDuplicatesFirst;
    long lDuplicatesLast;
    long lDuplicatesInterval;
} QMS_OUTPUT_SAMPLING;
```

Structure	
Member	Explanation
lMaxRecords	Maximum number of records to output
lFirstRecordNum	Record number of first record to output
lLastRecordNum	Record number of last record to return output
lInterval	Sampling interval (return every “nth” record)
lGroupSize	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record)
lMaxDupeGroups	Maximum number of dupe groups to output
lDuplicatesFirst	First dupe group to output
lDuplicatesLast	Last record to output
lDuplicatesInterval	Dupe group sampling interval

QMS_PHASE_TYPE

Description

These are the types of phases that can be created in a Centrus Merge/Purge job. The phase creation functions are `QmpDataInpCreate`, `QmpIndxGenCreate`, `QmpRecMatCreate`, `QmpDupGrpsCreate`, `QmpTblGenCreate`, `QmpDupsRptCreate`, `QmpUniqsRptCreate`, `QmpListByListRptCreate`, `QmpMissRptCreate`, and `QmpJobRptCreate`. The function to get the phase type is `QmpPhaseGetType`.

Summary

```
typedef enum QMS_PHASE_TYPE {
    QMS_PHASE_TYPE_NONE,
    QMS_PHASE_TYPE_MATCH,
    QMS_PHASE_TYPE_REPORT,
    QMS_PHASE_TYPE_DATAINPUT,
    QMS_PHASE_TYPE_TABLEINDEX,
    QMS_PHASE_TYPE_DUPEGROUPS,
    QMS_PHASE_TYPE_TABLEGENERATOR,
    QMS_PHASE_TYPE_UNIQUE_REPORT,
    QMS_PHASE_TYPE_DUPS_REPORT,
    QMS_PHASE_TYPE_NEARMISS_REPORT,
    QMS_PHASE_TYPE_CONSOL_REPORT,
    QMS_PHASE_TYPE_LISTBYLIST_REPORT,
    QMS_PHASE_TYPE_CONSOLIDATION,
    QMS_PHASE_TYPE_SUMM_REPORT
} QMS_PHASE_TYPE;
```

QMS_PHASE_STATE

Description

This enumeration lists the possible states a phase can be in during a Centrus Merge/Purge job. Before a phase has started, it is in the “idle” state. While it is running it is in the “started” state. After it has finished, it is “done”. If it encounters an error during execution, and stops prematurely, it is “failed”. If the application calls `QmpPhaseStop`, it enters a “stopping” state, and when it finally stops (the next convenient time it can), it goes into the “stopped” state. The report phases can enter a “ready” state if they are capable of being started. See `QmpPhaseGetState`.

Summary

```
typedef enum QMS_PHASE_STATE {
    QMS_PHASE_STATE_NONE,
    QMS_PHASE_STATE_IDLE,
    QMS_PHASE_STATE_READY,
    QMS_PHASE_STATE_DONE,
    QMS_PHASE_STATE_FAILED,
    QMS_PHASE_STATE_STARTING,
    QMS_PHASE_STATE_STARTED,
    QMS_PHASE_STATE_STOPPING,
    QMS_PHASE_STATE_STOPPED
} QMS_PHASE_STATE;
```

QMS_RANKING_PRIORITY

Description

This type is used to set the ranking priority for dupe groups master selection.

Summary

```
typedef enum QMS_RANKING_PRIORITY {
    QMS_RANKING_PRIORITY_DATALIST
    QMS_RANKING_PRIORITY_FLDVAL
    QMS_RANKING_PRIORITY_LARGEST
    QMS_RANKING_PRIORITY_SMALLEST
    QMS_RANKING_PRIORITY_RANDOM
} QMS_RANKING_PRIORITY;
```

QMS_REPORT_TEXT_POSITION

Description

This enumeration defines the options for placing text on a report. This is not currently being used in the Centrus Merge/Purge API.

Summary

```
typedef enum QMS_REPORT_TEXT_POSITION {
    QMS_REPORT_TEXT_UPPER_LEFT,
    QMS_REPORT_TEXT_UPPER_CENTER,
    QMS_REPORT_TEXT_UPPER_RIGHT,
    QMS_REPORT_TEXT_LOWER_LEFT,
    QMS_REPORT_TEXT_LOWER_CENTER,
    QMS_REPORT_TEXT_LOWER_RIGHT
} QMS_REPORT_TEXT_POSITION;
```

QMS_REPORT_TITLE_POSITION

Description

This enumeration defines the location of titles on a report. This is not currently being used in the Centrus Merge/Purge API.

Summary

```
typedef enum QMS_REPORT_TITLE_POSITION {
    QMS_REPORT_TITLE_LEFT,
    QMS_REPORT_TITLE_CENTER,
    QMS_REPORT_TITLE_RIGHT
} QMS_REPORT_TITLE_POSITION;
```

QMS_SORT_OBJECT_STATUS

Description

When an application asks for the next record from a sorted record object, the object returns QMS_STAT_SORT_STATE_OK if another record is available, or QMS_STAT_SORT_STATE_EMPTY if there are no more records in the sorted object.

Summary

```
typedef enum QMS_SORT_OBJECT_STATUS {
    QMS_STAT_SORT_STATE_EMPTY,
    QMS_STAT_SORT_STATE_OK
} QMS_SORT_OBJECT_STATUS;
```

QMS_SORT_ORDER

Description

This type is used to sort dupes in a dupe group before the master dupe is selected.

Summary

```
typedef enum QMS_SORT_ORDER {
    QMS_SORT_ORDER_NONE
    QMS_SORT_ORDER_ASCENDING
    QMS_SORT_ORDER_DESCENDING
} QMS_SORT_ORDER;
```

QMS_STAT_SAMPLING

Description

The QMS_STAT_SAMPLING structure is used to set the statistical sampling properties that are used for providing data source objects with information on which records to use from the data source. It applies to both the functor and the table types of data sources, but not the preprocessed data source. It defines the pattern of records to return to the Data Input phase. For all members of this structure, a value of 0 means “this aspect of statistical sampling is not specified and should not be used”. The contents of the structure mean:

- Maximum number of records to input from a data source – the data source will stop providing records to the Data Input phase when this limit has been hit.
- First record number - record number of first record to return from a data source. All earlier records will be skipped.

- Last record number - record number of last record to return from a data source. All subsequent records will be skipped.
- Interval - sampling interval (return every “nth” record).
- Group size - size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record)

See `QmpDataSrcFillStatSamp` and `QmpDataSrcSetStatSamp`.

Summary

```
typedef struct _QMS_STAT_SAMPLING {
    long lMaxRecords;
    long lFirstRecordNum;
    long lLastRecordNum;
    long lInterval;
    long lGroupSize;
} QMS_STAT_SAMPLING;
```

QMS_STAT_SAMPLING_STATE

Description

This type enumerates the various states that statistical sampling can assume during input or output sampling.

Summary

```
typedef enum QMS_STAT_SAMPLING_STATE {
    QMS_STAT_SAMPLING_STATE_UNDEFINED,
    QMS_STAT_SAMPLING_STATE_ACCEPT,
    QMS_STAT_SAMPLING_STATE_MAX_EXCEEDED,
    QMS_STAT_SAMPLING_STATE_BEFORE_FIRST,
    QMS_STAT_SAMPLING_STATE_AFTER_LAST,
    QMS_STAT_SAMPLING_STATE_NOT_IN_GROUP
} QMS_STAT_SAMPLING_STATE;
```

QMS_TABLE_INDEXES

Description

`QmpTblFillIdxInfo` fills an array of these structures for an application, with each array entry representing an index that exists on the table.

Summary

```
typedef struct _QMS_TABLE_INDEXES {
    char szIndexExpression[QMS_MAX_INDX_EXPRESSION_SIZE];
    char szIndexName[QMS_MAX_INDX_NAME_SIZE];
} QMS_TABLE_INDEXES;
```

QMS_VARIANT_TYPE

Description

This enumeration defines the types of values a Centrus Merge/Purge variant object can contain. See the set of `QmpVar*` functions in the C wrapper. Also see `QmpRecAddByType`, `QmpRecGetFldTypeByHnd`, and `QmpRecGetFldTypeByName`.

Summary

```
typedef enum QMS_VARIANT_TYPE {
    QMS_VARIANT_EMPTY,
    QMS_VARIANT_BOOLEAN,
    QMS_VARIANT_CHAR,
    QMS_VARIANT_DOUBLE,
    QMS_VARIANT_FLOAT,
    QMS_VARIANT_INT,
    QMS_VARIANT_LONG,
    QMS_VARIANT_STRING,
    QMS_VARIANT ULONG,
    QMS_VARIANT USHORT,
    QMS_VARIANT_VOIDSTAR,
    QMS_VARIANT_DATE
} QMS_VARIANT_TYPE;
```

QRESULT

Description

The type of all error codes. This type is used for the last parameter (the result code) of all Centrus Merge/Purge C functions.

Summary

```
typedef long QRESULT;
```

QVARSTRUCT

Description

The QVARSTRUCT is a public C structure that an application can use to initialize a Centrus Merge/Purge variant object. The variant objects, themselves, are used to set and get information from Centrus Merge/Purge record objects. See `QmpVarSetVarStruct` and `QmpVarGetVarStruct`.

Summary

```
typedef struct _varstruct {
    QMS_VARIANT_TYPE varType;
    union {
        QBOOL          QBOOLValue;
        char           charValue;
        double         doubleValue;
        float          floatValue;
        int            intValue;
        long           longValue;
        char*          szStringValue;
        unsigned long  ulongValue;
        unsigned short ushortValue;
        void*          pVoidValue;
        QDATESTRUCT    dateValue;
    } uVal;
} QVARSTRUCT;
```

Constants and Macros

NULL

Description

A NULL or zero value used to initialize pointers. It is defined on Microsoft platforms, but possibly not elsewhere.

Summary

```
#ifndef NULL
#define NULL 0
#endif
```

OUTPUT_*

Description

These constants are used as input to the data destination object to configure it to receive certain types of records from the table generator. Each data destination can register for any combination of these record types.

Summary

```
#define OUTPUT_NONE          0x00
#define OUTPUT_INCL_UNIQUES  0x01
#define OUTPUT_INCL_MASTERS  0x02
#define OUTPUT_INCL_SUBORDS  0x04
#define OUTPUT_SUPP_UNIQUES  0x08
#define OUTPUT_SUPP_MASTERS  0x10
#define OUTPUT_SUPP_SUBORDS 0x20
#define OUTPUT_INCL_CONSOL   0x40
#define OUTPUT_SUPP_CONSOL   0x80
```

QmpDeclHnd

Description

Defines a safe way to allocate a handle for the C wrapper. Even with all of the safeguards Centrus Merge/Purge takes to make sure handles are okay when they are passed to the C API functions, they still must be initialized to NULL before use.

Sample usage: `QmpDeclHnd(hRecMat);`

Summary

```
#define QmpDeclHnd(_hnd_) \
MpHnd _hnd_ = NULL
```

QmpUtilGetMessage

Description

Isolates and returns the value of the message digits of the QRESULT value.

Summary

```
#define QmpUtilGetMessage( _qr_ ) ( ( _qr_ ) % 1000 )
```

QmpUtilGetSeverity

Description

Isolates and return the value of the severity digit of the QRESULT value.

Summary

```
#define QmpUtilGetSeverity( _qr_ ) ( ( ( _qr_ ) / 1000000 ) %  
10 )
```

QmpUtilGetSuccessFlag

Description

Isolates and returns the value of the success flag digit of the QRESULT value.

Summary

```
#define QmpUtilGetSuccessFlag( _qr_ ) ( ( _qr_ ) / 10000000 )
```

QmpUtilSucceeded

Description

Tests whether the return code indicates success.

Summary

```
#define QmpUtilSucceeded( _qr_ ) ( ((long)( _qr_ ) / 10000000)  
== 1 )
```

QmpUtilFailed

Description

Tests whether the return code indicates failure.

Summary

```
#define QmpUtilFailed( _qr_ ) ( ((long)(_qr_) / 10000000) == 2 )
```

QMS_CHDIR

Description

A portable macro for changing the current directory. It has different implementations on the several platforms supported by Centrus Merge/Purge.

Summary

```
#define QMS_CHDIR <platform dependent>
```

QMS_GETCWD

Description

A portable macro for getting the current working directory. It has different implementations on the several platforms supported by Centrus Merge/Purge.

Summary

```
#define QMS_GETCWD <platform dependent>
```

QMS_MAX

QMS_MAX3

QMS_MIN

QMS_MIN3

Description

Defines QMS “min” and “max” macros. These are defined in windef.h for windows, but Centrus Merge/Purge needs its own version for portability.

Summary

```
#ifndef QMS_MAX
#define QMS_MAX(_a_,_b_) (((_a_) > (_b_)) ? (_a_) : (_b_))
#endif
#ifndef QMS_MAX3
#define QMS_MAX3(_a_,_b_,_c_) (((_a_) > (_b_)) ?
    QMS_MAX((_a_),(_c_)) : QMS_MAX((_b_),(_c_)))
#endif
#ifndef QMS_MIN
#define QMS_MIN(_a_,_b_) (((_a_) < (_b_)) ? (_a_) : (_b_))
#endif
#ifndef QMS_MIN3
#define QMS_MIN3(_a_,_b_,_c_) (((_a_) < (_b_)) ?
    QMS_MIN((_a_),(_c_)) : QMS_MIN((_b_),(_c_)))
#endif
```

QMS_MAX_INDX_EXPRESSION_SIZE

Description

The maximum size of an index expression.

Summary

```
#define QMS_MAX_INDX_EXPRESSION_SIZE 2048
```

QMS_MAX_INDX_NAME_SIZE

Description

The maximum size of an index name.

Summary

```
#define QMS_MAX_INDX_NAME_SIZE 256
```

QMS_MAX_NAME

Description

The maximum length of a name (class name, function name, message name, etc.). Use it when creating a buffer to hold the name of something.

Summary

```
#define QMS_MAX_NAME 256
```

QMS_MAX_PATH

Description

Maximum path length. This is used for numerous paths and strings in the system.

Summary

```
#define QMS_MAX_PATH 256
```

QMS_STRICTCMP

Description

A portable case insensitive string comparison macro. It has different implementations on the several platforms supported by Centrus Merge/Purge.

Summary

```
#define QMS_STRICTCMP <platform dependent>
```

QMS_STRNICMP

Description

A portable case-insensitive string comparison using “n” characters macro. It has different implementations on the several platforms supported by Centrus Merge/Purge.

Summary

```
#define QMS_STRNICMP <platform dependent>
```

QSEVERITY_SUCCEED **QSEVERITY_FAIL**

Description

These constants define success and failure.

Summary

```
#define QSEVERITY_SUCCEED1  
#define QSEVERITY_FAIL2
```

QTRUE/QFALSE

Description

Centrus Merge/Purge's simple boolean types. See “[QBOOL](#)” on page 74.

Summary

```
#define QTRUE (QBOOL) 1  
#define QFALSE (QBOOL) 0
```

Chapter 6

C Function Reference

The Centrus Merge/Purge API provides a C interface to the powerful C++ classes and methods that underlie Sagent's Merge/Purge technology. This chapter describes in detail each of the public functions available through the API. The format for the syntax of each function is:

```
ReturnType FunctionName ( ArgumentType identifier, ...  
                           QRESULT* out_pResult )
```

where:

- **ReturnType** is the return value associated with the function.
- **FunctionName** is the name of the function.
- **ArgumentType** is the type of the parameter that follows.
- *identifier* is the parameter name.
- *out_pResult* is an error return code.

Function names, such as **QmpDataInpCreate**, are formed by appending abbreviations for the relevant C++ class and method to the prefix **Qmp**. Functions with the same name prefix are grouped together into a class, with the class name being the function prefix followed by an asterisk. For example, the function class **QmpAlg*** includes all functions that start with the prefix “QmpAlg”. In this chapter, individual function explanations are grouped by class.

Argument identifiers indicate whether an argument is input or output, its type, and its meaning. For example, the identifier *in_bCaseSens* is an input argument of type **QBOOL**, and sets the case sensitivity of the function.

Objects and Handles

The Centrus Merge/Purge API uses pointers as handles to Merge/Purge library objects. Each object type—such as an algorithm, criterion, or phase—has a “create” and a “destroy” method. The “create” functions return an object handle to the client program. If you forget to free an object with the object’s “destroy” method, a memory leak results. The client application must call “destroy” for every “create”.

Parameters, Errors, and Return Values

Each function that operates on an object accepts the object handle as the first parameter. Functions return useful values, rather than a return code. The *out_pResult* return argument is the final parameter in each function. The value of this parameter can be tested against the constant `QRESULT_YES` to ascertain success or failure of the function.

QRESULT Return Codes

The complete list of `QRESULT` return codes values, and messages can be found starting on [page 897](#). These codes are defined in the file `CmpErMsg.h`. You can also add custom error codes to this header file. A number of macros, such as `QmpUtilSucceeded` and `QmpUtilFailed`, exist that can be used to check the value of return codes. These two macros can be found in the file `CmpDefs.h`.

See “[QRESULT Return Codes](#)” on page 897 for a discussion of how to interpret the `QRESULT` values.

Function Class: QmpAlg*

FUNCTIONS FOR MANIPULATING FIELD MATCHING ALGORITHMS.

Quick Reference

Function	Description	Page
QmpAlgCompute	Compares two strings using a specified algorithm.	106
QmpAlgCreate	Creates an algorithm.	107
QmpAlgDestroy	Destroys an algorithm.	108

QmpAlgCompute

COMPARES TWO STRINGS USING A SPECIFIED ALGORITHM.

Syntax

```
int QmpAlgCompute ( MpHnd in_hAlg, MpHnd in_hVar1, MpHnd
                     in_hVar2, QRESULT* out_pResult );
```

in_hAlg
Handle to algorithm. *Input*.

in_hVar1
Handle to first field variant. *Input*.

in_hVar2
Handle to second field variant. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the match result score if successful, -1 if there is an error.

Notes

This function compares two strings using the specified algorithm and returns the match result score as a value from 0 (no match) to 100 (exact match).

See Also

`QmpAlgCreate`, `QmpVar*` functions.

Example

```
iMatchScore = QmpAlgCompute (hAlg, hVariant1, hVariant2,
                             &Result );
```

QmpAlgCreate

CREATES AN ALGORITHM.

Syntax

```
MpHnd QmpAlgCreate ( QMS_ALGORITHM_TYPE in_Type, QRESULT*  
    out_pResult );
```

in_Type

Type of matching algorithm to create. *Input*.

Valid enums are:

QMS_ALGORITHM_TYPE_CHARFREQ	Character frequency
QMS_ALGORITHM_TYPE_EDITDIST	Edit distance
QMS_ALGORITHM_TYPE_KEYDIST	Keyboard distance
QMS_ALGORITHM_TYPE_SOUNDDEX	Soundex
QMS_ALGORITHM_TYPE_STRING	String comparison
QMS_ALGORITHM_TYPE_NUMERIC	Numeric
QMS_ALGORITHM_TYPE_NUMERICSTRING	Numeric string
QMS_ALGORITHM_TYPE_NAME	Name matching

out_pResult

Result code. *Output*.

Return Value

Returns a handle to the algorithm if successful, NULL if there is an error.

Notes

This function creates the field-matching algorithm object specified in the *in_Type* argument. If the creation function fails, an error code is returned in the *out_pResult* parameter.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

See Also

[QmpAlgDestroy](#)

Example

```
/* Create some field match algorithms */  
QmpDeclHnd(hAlgEditDist);  
QmpDeclHnd(hAlgSoundex);  
QmpDeclHnd(hAlgString);  
  
hAlgEditDist = QmpAlgCreate(QMS_ALGORITHM_TYPE_EDITDIST, &qres  
    );  
hAlgSoundex = QmpAlgCreate(QMS_ALGORITHM_TYPE_SOUNDDEX,   &qres  
    );  
hAlgString = QmpAlgCreate(QMS_ALGORITHM_TYPE_STRING,  &qres );
```

QmpAlgDestroy

DESTROYS AN ALGORITHM

Syntax

```
void QmpAlgDestroy ( MpHnd in_hAlg, QRESULT* out_pResult );
```

in_hAlg
Handle to algorithm. *Input.*

out_pResult
Result code. *Output.*

Return Value

None.

Notes

This function destroys the specified algorithm object.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

See Also

`QmpAlgCreate`

Example

```
QmpAlgDestroy(hCHARFREQ, &qres);  
QmpAlgDestroy(hEDITDIST, &qres);  
QmpAlgDestroy(hKEYDIST, &qres);  
QmpAlgDestroy(hSOUNDEX, &qres);  
QmpAlgDestroy(hSTRING, &qres);  
QmpAlgDestroy(hNUMERIC, &qres);
```

Function Class: QmpCons*

DATA CONSOLIDATION PHASE FUNCTIONS.

Creating Synthesized Consolidated Records

Synthesized (new) consolidated records are always created when the data consolidation phase is executed. You do not have to do anything with them, but they are always constructed.

Updating Duplicate Records with Consolidated Field Values

To update a dupe group's master dupe with the field values from the synthesized consolidated record, call `QmpConsSetModifyMasters`. To update all the subordinate dupes in a dupe group with the synthesized record's field values, call `QmpConsSetModifySubords`.

Attaching the Consolidation Phase Intermediate Tables

Both the synthesized consolidated records table and the updated duplicates consolidation table must be passed to the data consolidation phase before the phase may be executed.

The handle to the synthesized consolidated records table is passed to the data consolidation phase in `QmpConsCreate`. The handle to the updated duplicates consolidation table is passed to the data consolidation phase by calling `QmpConsUseConsTbl`.

Quick Reference

Function	Description	Page
<code>QmpConsAddConsCrit</code>	Adds a data consolidation criterion to the data consolidation phase.	111
<code>QmpConsAreConsCritSet</code>	Determines if the application has set any consolidation criteria in the data consolidation phase.	112
<code>QmpConsClear</code>	Clears a data consolidation phase.	113
<code>QmpConsCloseConsTbl</code>	Closes the updated duplicates consolidation table.	114
<code>QmpConsCreate</code>	Creates a data consolidation phase.	115
<code>QmpConsGetConsCritAt</code>	Gets the data consolidation criterion at a particular field index position.	116

Function	Description	Page
QmpConsGetConsCritCnt	Gets the number of consolidation criteria in the data consolidation phase (including default criteria).	117
QmpConsGetModifyMasters	Returns the modify masters flag from the data consolidation phase.	118
QmpConsGetModifySubords	Returns the modify subordinates flag from the data consolidation phase.	119
QmpConsGetNthReclInterval	Gets data consolidation phase's Nth record event interval.	120
QmpConsRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	121
QmpConsSetModifyMasters	Sets the modify masters flag in the data consolidation phase.	122
QmpConsSetModifySubords	Sets the modify subordinates flag in the data consolidation phase.	123
QmpConsSetNthReclInterval	Sets data consolidation phase Nth record event interval.	124
QmpConsTblClose	Closes the synthesized consolidated records table.	125
QmpConsUnRegEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	126
QmpConsUseConsTbl	Gives the updated duplicates consolidation table to the data consolidation phase.	127
QmpConsUseDupGrps	Give the dupe groups object to the data consolidation phase.	128
QmpConsUseTbl	Gives the synthesized consolidated records table to the data consolidation phase.	129

QmpConsAddConsCrit

ADDS A DATA CONSOLIDATION CRITERION TO THE DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsAddConsCrit( MpHnd in_hCons, MpHnd in_hConsCrit,
                         QRESULT* out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_hConsCrit

Handle to a data consolidation criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

A consolidation criterion is a rule that determines what a consolidated record field value will be. Each field in the prototype record may have a user-defined consolidation criterion attached. If the user does not attach a consolidation criterion to a field, a default criterion will be attached.

See Also

QmpConsAreConsCritSet, **QmpConsGetConsCritAt**,
QmpConsGetConsCritCnt.

Example

```
QmpConsAddConsCrit ( hCons, hConsCrit, &qres );
```

QmpConsAreConsCritSet

DETERMINES IF THE APPLICATION HAS SET ANY CONSOLIDATION CRITERIA IN THE DATA CONSOLIDATION PHASE.

Syntax

```
QBOOL QmpConsAreConsCritSet( MpHnd in_hCons, QRESULT*  
    out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if the application has set consolidation criteria, else QFALSE.

Notes

QmpConsAreConsCritSet is a convenience function that frees the application from remembering if it has set record consolidation criteria.

See Also

QmpConsAddConsCrit, **QmpConsGetConsCritAt**,
QmpConsGetConsCritCnt.

Example

```
bConsSet = QmpConsAreConsCrit ( hCons, &qres );
```

QmpConsClear

CLEAR A DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsClear( MpHnd in_hCons, QRESULT* out_pResult );
```

in_hCons
Handle to the data consolidation phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the specified data consolidation object back to its initial state. All attached consolidation criteria are set to “master” status.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpConsClear ( hCons, &qres );
```

QmpConsCloseConsTbl

CLOSES THE UPDATED DUPLICATES CONSOLIDATION TABLE.

Syntax

```
void QmpConsCloseConsTbl( MpHnd in_hCons, QRESULT* out_pResult  
);
```

in_hCons

Handle to the data consolidation phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function should only be called after a call to `QmpConsClear()`;

`QmpConsCloseConsTbl` is used to close the updated duplicates consolidation table. `QmpConsClear` must be called before calling `QmpConsCloseConsTbl`, so that all references to this table in the data consolidation object are erased.

You would call `QmpConsCloseConsTbl` to close the table so that another application may use the table, or as the last step before destroying the table.

See Also

`QmpConsClear`.

Example

```
QmpConsCloseConsTbl ( hCons, &qres );
```

QmpConsCreate

CREATES A DATA CONSOLIDATION PHASE.

Syntax

```
MpHnd QmpConsCreate( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl

Handle to application-created table to be used as the synthesized consolidated records table. This *must* be a CodeBase table. *Input*.

out_pResult

Result code. *Output*.

Return Value

A handle to the newly created data consolidation phase.

Notes

The data consolidation phase uses a table for generating new consolidated records. This synthesized consolidated records table is initially assigned to the phase in its creation function **QmpConsCreate**.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

```
QmpConsCreate ( hTbl, &qres );
```

QmpConsGetConsCritAt

GETS THE DATA CONSOLIDATION CRITERION AT A PARTICULAR FIELD INDEX POSITION.

Syntax

```
MpHnd QmpConsGetConsCritAt( MpHnd in_hCons, int in_iIndex,  
    QRESULT* out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_iIndex

Index number of the criterion to be retrieved. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the handle to the consolidation criterion if successful, or NULL if there is an error.

Notes

QmpConsGetConsCritAt fetches a consolidation criterion using the criterion's index value. This index value is the same as the index of the corresponding consolidated record field. Indexes are zero-based.

See Also

QmpConsGetConsCritCnt.

Example

```
hConsCrit = QmpConsGetConsCritAt ( hCons, 1, &qres );
```

QmpConsGetConsCritCnt

GETS THE NUMBER OF CONSOLIDATION CRITERIA IN THE DATA CONSOLIDATION PHASE (INCLUDING DEFAULT CRITERIA).

Syntax

```
long QmpConsGetConsCritCount( MpHnd in_hCons, QRESULT*  
                               out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the number of consolidation criteria if successful, or -1 if there is an error. This value is also the prototype record field count.

See Also

[QmpConsGetConsCritAt](#).

Example

```
lCount = QmpConsGetConsCritCnt ( hCons, &qres );
```

QmpConsGetModifyMasters

RETURNS THE MODIFY MASTERS FLAG FROM THE DATA CONSOLIDATION PHASE.

Syntax

```
QBOOL QmpConsGetModifyMasters( MpHnd in_hCons, QRESULT*  
    out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

A boolean value representing the current status of the modify masters flag in the data consolidation phase.

Notes

When the modify masters flag is set to QTRUE, the fields in master duplicate records will be updated with consolidated values.

Example

```
bModMasters = QmpConsGetModifyMasters ( hCons, &gres );
```

QmpConsGetModifySubords

RETURNS THE MODIFY SUBORDINATES FLAG FROM THE DATA CONSOLIDATION PHASE.

Syntax

```
QBOOL QmpConsGetModifySubords( MpHnd in_hCons, QRESULT*  
    out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

A boolean value representing the current status of the modify subordinates flag in the data consolidation phase.

Notes

When the modify subordinates flag is set to QTRUE, the fields in subordinate duplicate records will be updated with consolidated values.

Example

```
bModSubords = QmpConsGetModifySubords ( hCons, &qres );
```

QmpConsGetNthRecInterval

GETS DATA CONSOLIDATION PHASE'S NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpConsGetNthRecInterval( MpHnd in_hCons, QRESULT*  
                               out_pResult );
```

in_hCons
Handle to the data consolidation phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns data consolidation phase's Nth record event interval if successful, or -1 if there is an error.

Notes

The data consolidation phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an "occasional" notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the data consolidation phase processes 100 records, it will send out a notification to *all* registered clients.

See Also

[QmpConsSetNthRecInterval](#).

Example

```
lInterval = QmpConsGetNthRecInterval ( hCons, &qres );
```

QmpConsRegEveryNthRecFunc

REGISTER THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpConsRegEveryNthRecFunc( MpHnd in_hCons,
                                QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hCons
```

Handle to the data consolidation phase. *Input*.

in_Func

Pointer to the callback function. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The data consolidation phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

See Also

[QmpConsUnRegEveryNthRecFunc](#).

Example

```
QmpConsRegEveryNthRecFunc ( hCons, CallbackFunc, &qres );
```

QmpConsSetModifyMasters

SETS THE MODIFY MASTERS FLAG IN THE DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsSetModifyMasters( MpHnd in_hCons, QBOOL  
    in_bModifyMasters, QRESULT* out_pResult );
```

in_hCons
Handle to the data consolidation phase. *Input*.

in_bModifyMasters
Boolean value to set modify masters flag. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

When the modify masters flag is set to QTRUE, the fields in master duplicate records will be updated with consolidated values.

Example

```
QmpConsSetModifyMasters ( hCons, QTRUE, &qres );
```

QmpConsSetModifySubords

SETS THE MODIFY SUBORDINATES FLAG IN THE DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsSetModifySubords( MpHnd in_hCons, QBOOL  
    in_bModifySubords, QRESULT* out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_bModifySubords

Boolean value to set modify subordinates flag. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

When the modify subordinates flag is set to QTRUE, the fields in subordinate duplicate records will be updated with consolidated values.

Example

```
QmpConsSetModifySubords ( hCons, QTRUE, &qres );
```

QmpConsSetNthRecInterval

SETS DATA CONSOLIDATION PHASE NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpConsSetNthRecInterval( MpHnd in_hCons, long  
                               in_lInterval, QRESULT* out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_lInterval

Data consolidation phase Nth record event interval. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets the data consolidation phase's Nth record event interval.

The data consolidation phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an "occasional" notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the data consolidation phase processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

[QmpConsGetNthRecInterval](#).

Example

```
QmpConsSetNthRecInterval ( hCons, 500, &qres );
```

QmpConsTblClose

CLOSES THE SYNTHESIZED CONSOLIDATED RECORDS TABLE.

Syntax

```
void QmpConsTblClose( MpHnd in_hCons, QRESULT* out_pResult );
```

in_hCons
Handle to the data consolidation phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpConsTblClose is used to close the synthesized consolidated records table. **QmpConsClear** must be called before calling **QmpConsTblClose**, so that all references to this table in the data consolidation object are erased.

You would call **QmpConsTblClose** to close the table so that another application may use the table, or as the last step before destroying the table.

See Also

QmpConsClear.

Example

```
QmpConsTblClose ( hCons, &qres );
```

QmpConsUnRegEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpConsUnRegEveryNthRecFunc( MpHnd in_hCons,
                                   QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hCons
Handle to the data consolidation phase. *Input*.

in_Func
Pointer to the callback function. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpConsUnRegEveryNthRecFunc unregisters the specified event handler from the data consolidation phase, so that the event handler will no longer be notified of every Nth record processed.

The data consolidation phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

See Also

[QmpConsRegEveryNthRecFunc](#).

Example

```
QmpConsUnRegEveryNthRecFunc (hCons, CallbackFunc, &qres);
```

QmpConsUseConsTbl

GIVES THE UPDATED DUPLICATES CONSOLIDATION TABLE TO THE DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsUseConsTbl( MpHnd in_hCons, MpHnd in_hTbl,
    QRESULT* out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_hTbl

Handle to application-created table to be used as the updated duplicates consolidation table. This *must* be a CodeBase table. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The data consolidation phase requires a table for appending duplicate records with consolidation fields. This updated duplicates consolidation table is given to the consolidation phase by calling **QmpConsUseConsTbl**.

See Also

QmpConsCreate, **QmpConsClear**, **QmpConsUseTbl**.

Example

```
QmpConsUseConsTbl ( hCons, hTbl, &qres );
```

QmpConsUseDupGrps

GIVE THE DUPE GROUPS OBJECT TO THE DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsUseDupGrps( MpHnd in_hCons, MpHnd in_hDupGrps,  
    QRESULT* out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_hDupGrps

Handle to the dupe groups object. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Example

```
QmpConsUseDupGrps ( hCons, hDupGrps, &qres );
```

QmpConsUseTbl

GIVES THE SYNTHESIZED CONSOLIDATED RECORDS TABLE TO THE DATA CONSOLIDATION PHASE.

Syntax

```
void QmpConsUseTbl( MpHnd in_hCons, MpHnd in_hTbl, QRESULT*  
    out_pResult );
```

in_hCons

Handle to the data consolidation phase. *Input*.

in_hTbl

Handle to an application-created table to be used as the synthesized consolidated records table. This *must* be a CodeBase table. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The data consolidation phase uses a table for generating new consolidated records. This synthesized consolidated records table is initially assigned to the phase in its creation function **QmpConsCreate**.

If you wish to assign another synthesized consolidated records table to the data consolidation phase, you must clear the current data consolidation object by calling **QmpConsClear**, and then call **QmpConsUseTbl**.

The synthesized consolidated records table *must* be a CodeBase table.

See Also

QmpConsCreate, **QmpConsClear**.

Example

```
QmpConsUseTbl ( hCons, hTbl, &qres );
```


Function Class: QmpConsCrit*

DATA CONSOLIDATION CRITERION FUNCTIONS.

Notes on Consolidated Records

- The primary key, source ID, and score values are always 0.
- The dupe group ID is the same as that of the dupe group from which the consolidated record was constructed.
- A consolidated record is automatically assigned to the default data list (whose list ID is 0). The inclusion/suppression setting for the default data list is applied to the consolidated record. For example, if the default data list is set to be a suppression list, the consolidated record is a suppression record.

Quick Reference

Function	Description	Page
QmpConsCritClear	Clears a data consolidation criterion.	133
QmpConsCritGetExprNumChar	Creates a data consolidation criterion.	134
QmpConsCritDestroy	Destroys a data consolidation criterion.	135
QmpConsCritFillDesc	Generates a string for the client containing the consolidation criterion description.	136
QmpConsCritFillExpr	Fills a client-owned string with a consolidation criterion's field value expression.	137
QmpConsCritFillVal	Fills a client-owned variant with a consolidation criterion's variant value.	138
QmpConsCritGetDataLstID	Gets data consolidation rule's data list ID.	139
QmpConsCritGetExprNumChar	Returns the character count of a fldval data consolidation criterion's substring.	140
QmpConsCritGetExprStart	Returns the start position of a fldval data consolidation criterion's substring.	141
QmpConsCritGetFldHnd	Gets the handle of a field a consolidation criterion is attached to.	142
QmpConsCritGetFldName	Gets the name of a field a consolidation criterion is attached to.	143
QmpConsCritGetFldNameVB	Gets the name of a field a consolidation criterion is attached to (Visual Basic version).	144
QmpConsCritGetOtherFldHnd	Gets the field handle of the other field this criterion uses.	145

Function	Description	Page
QmpConsCritGetOtherFldName	Gets the field name of the other field this criterion uses.	146
QmpConsCritGetOtherFldNameVB	Gets the field name of the other field this criterion uses (Visual Basic version).	147
QmpConsCritGetRule	Gets data consolidation rule which the criterion will use.	148
QmpConsCritGetTreatment	Gets the treatment type of the field this consolidation criterion applies to.	149
QmpConsCritIsValid	Determines if a data consolidation criterion is valid or not.	150
QmpConsCritSetDataLstRule	Sets the QMS_CONSOL_RULE_DATLST rule for a consolidation criterion.	151
QmpConsCritSetExprSubStr	Sets expression substring for a fldval data consolidation criterion.	152
QmpConsCritSetFldValRule	Sets the QMS_CONSOL_RULE_FLDVAL rule for a consolidation criterion.	153
QmpConsCritSetGroupOrderRule	Sets the QMS_CONSOL_RULE_GROUP_ORDER rule for a consolidation criterion.	154
QmpConsCritSetGroupOtherRule	Sets the QMS_CONSOL_RULE_GROUP_ORDER_OTH ER rule for a consolidation criterion.	155
QmpConsCritSetNumValRule	Sets the NUM_VALUE rule for a consolidation criterion.	156
QmpConsCritSetOtherFldRule	Sets the QMS_CONSOL_RULE_LARGEST_OTHER and QMS_CONSOL_RULE_SMALLEST_OTHER rules for a consolidation criterion.	157
QmpConsCritSetSimpleRule	Sets the master, non-blank, rollup, average, num dupes, shortest, and longest rules for a consolidation criterion.	159
QmpConsCritSetTreatmentRule	Sets the QMS_CONSOL_RULE_LARGEST, QMS_CONSOL_RULE_SMALLEST, and QMS_CONSOL_RULE_MOST_COMMON rules for a consolidation criterion.	160
QmpConsCritSetValRule	Sets the QMS_CONSOL_RULE_VAL rule for a consolidation criterion.	162

QmpConsCritClear

CLEAR A DATA CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritClear ( MpHnd in_hConsCrit, QRESULT*
    out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpConsCritClear clears the data consolidation criterion, leaving it in a usable state but with no specified settings. After calling this function, the criterion rule becomes QMS_CONSOL_RULE_UNDEFINED.

Example

```
MpHnd hconscrit;
QRESULT presult;

QmpConsCritClear(hconscrit, &presult);
```

QmpConsCritgetexprnumchar

CREATES A DATA CONSOLIDATION CRITERION.

Syntax

```
MpHnd QmpConsCritCreate ( QRESULT* out_pResult );  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to created data consolidation criterion if successful, or `NULL` if there is an error.

Notes

A newly-created consolidation criterion is of type `QMS_CONSOL_RULE_UNDEFINED`. The application must set the consolidation type using one of the `QmpConsCritSet*Rule` functions before this object may be used.

Example

See example on [page 430](#).

QmpConsCritDestroy

DESTROYS A DATA CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritDestroy ( MpHnd in_hConsCrit, QRESULT*  
    out_pResult );
```

in_hConsCrit
Handle to data consolidation criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function destroys the specified data consolidation criterion.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

See Also

QmpConsCritCreate.

Example

```
MpHnd hconscrit;  
QRESULT presult;  
  
QmpConsCritDestroy ( hconscrit, &presult );
```

QmpConsCritFillDesc

GENERATES A STRING FOR THE CLIENT CONTAINING THE CONSOLIDATION CRITERION DESCRIPTION.

Syntax

```
void QmpConsCritFillDesc ( MpHnd in_hConsCrit, char*  
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

io_szBuffer
Client-allocated description buffer. *Input, output*.

in_lSize
Size of client-allocated description buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

A narrative description of the consolidation criterion is placed into *io_szBuffer*.

Example

```
MpHnd hconscript;  
char szbuffer[1024];  
QRESULT presult;  
  
QmpConsCritFillDesc(hconscript, szbuffer, sizeof(szbuffer),  
    &presult);
```

QmpConsCritFillExpr

FILLS A CLIENT-OWNED STRING WITH A CONSOLIDATION CRITERION'S FIELD VALUE EXPRESSION.

Syntax

```
void QmpConsCritFillExpr ( MpHnd in_hConsCrit, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
io_szBuffer
    Client-allocated expression buffer. Input, output.
in_lSize
    Size of client-allocated expression buffer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The field value expression contained in the data consolidation criterion is placed into *io_szBuffer*.

This function is valid only for a consolidation criterion of type QMS_CONSOL_RULE_FLDVAL.

A criterion using a rule of QMS_CONSOL_RULE_FLDVAL selects a field “X” value from a record with a field “Y” value which is greater/less/equal to some fixed value. The expression that states this field “Y”, the operation, and the constant value can be represented as a string. **QmpConsCritFillExpr** returns this string.

Example

```
MpHnd hconscrit;
char szbuffer[1024];
QRESULT presult;

QmpConsCritFillExpr ( hconscrit, szbuffer, sizeof(szbuffer),
    &presult );
```

QmpConsCritFillVal

FILLS A CLIENT-OWNED VARIANT WITH A CONSOLIDATION CRITERION'S VARIANT VALUE.

Syntax

```
void QmpConsCritFillVal ( MpHnd in_hConsCrit, MpHnd io_Value,
                          QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
io_Value
    Handle to client-allocated variant. Input, output.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function is valid only for a consolidation criterion of type QMS_CONSOL_RULE_VAL.

The QMS_CONSOL_RULE_VAL rule means that the application supplies the value that is placed into a consolidated record field. This value is stored in a variant. **QmpConsCritFillVal** gets the value of this consolidation criterion variant.

See Also

QmpConsCritSetValRule, **QmpVar*** functions.

Example

```
MpHnd hconscriit;
MpHnd hVariant;
QRESULT presult;

QmpConsCritFillVal ( hconscriit, hVariant, &presult );
```

QmpConsCritGetDataLstID

GETS DATA CONSOLIDATION RULE'S DATA LIST ID.

Syntax

```
long QmpConsCritGetDataLstID ( MpHnd in_hConsCrit, QRESULT*  
    out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns data list ID used in the criterion rule if successful, -1 if there is an error.

Notes

This function is valid only for a consolidation criterion of type QMS_CONSOL_RULE_DATLST.

A criterion using a rule of QMS_CONSOL_RULE_DATLST selects a field “X” value from the highest-ranking record from a particular data list. **QmpConsCritFillExpr** returns the ID of this data list.

Example

```
long lRuleDataLstID;  
MpHnd hconscrit;  
QRESULT presult;  
  
lRuleDataLstID = QmpConsCritGetDataLstID ( hconscrit,  
    &presult );
```

QmpConsCritGetExprNumChar

RETURNS THE CHARACTER COUNT OF A FLDVAL DATA CONSOLIDATION CRITERION'S SUBSTRING.

Syntax

```
int QmpConsCritGetExprNumChar( MpHnd in_hConsCrit, QRESULT*  
                               out_pResult );  
  
in_hConsCrit  
    Handle to consolidation criterion. Input.  
out_pResult  
    Result code. Output.
```

Return Value

Returns the field evaluation substring number of characters.

Notes

This function is valid only for the consolidation rule
QMS_CONSOL_RULE_FLDVAL.

In the fldval data consolidation rule, a field value is used as part of an expression of the type FieldValue Operator Constant.

`QmpConsCritSetExprSubStr` specifies a substring of the field value to be used in the expression. `QmpConsCritGetExprNumChar` returns the character count of this substring.

Example

```
iCharCount = QmpConsCritGetExprNumChar( in_hConsCrit,  
                                         &out_pResult );
```

QmpConsCritGetExprStart

RETURNS THE START POSITION OF A FLDVAL DATA CONSOLIDATION CRITERION'S SUBSTRING.

Syntax

```
int QmpConsCritGetExprStart(MpHnd in_hConsCrit, QRESULT*
    out_pResult );
in_hConsCrit
    Handle to data consolidation criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the field evaluation substring start position.

Notes

This function is valid only for the consolidation rule
QMS_CONSOL_RULE_FLDVAL.

In the fldval data consolidation rule, a field value is used as part of an expression of the type FieldValue Operator Constant. **QmpConsCritSetExprSubstr** specifies a substring of the field value to be used in the expression. **QmpConsCritGetExprStart** returns the start position of this substring.

Example

```
iStartPos = QmpConsCritGetExprStart(in_hConsCrit,
    &out_pResult );
```

QmpConsCritGetFldHnd

GETS THE HANDLE OF A FIELD A CONSOLIDATION CRITERION IS ATTACHED TO.

Syntax

```
int QmpConsCritGetFldHnd ( MpHnd in_hConsCrit, QRESULT*  
                           out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the handle of a field associated with a consolidation criterion if successful, -1 if there is an error.

Notes

This function applies to all consolidation rule types.

Example

```
int iFieldHnd;  
MpHnd hconscrit;  
QRESULT presult;  
  
iFieldHnd = QmpConsCritGetFldHnd ( hconscrit, presult );
```

QmpConsCritGetFldName

GETS THE NAME OF A FIELD A CONSOLIDATION CRITERION IS ATTACHED TO.

Syntax

```
const char* QmpConsCritGetFldName ( MpHnd in_hConsCrit,
                                    QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the name of a field associated with a consolidation criterion if successful, NULL if there is an error.

Notes

This function applies to all consolidation rule types.

Example

```
MpHnd hconscrit;
QRESULT presult;
const char *pszfldname;

pszfldname = QmpConsCritGetFldName(hconscrit, &presult);
```

QmpConsCritGetFldNameVB

GETS THE NAME OF A FIELD A CONSOLIDATION CRITERION IS ATTACHED TO (VISUAL BASIC VERSION).

Syntax

```
void QmpConsCritGetFldNameVB ( MpHnd in_hConsCrit, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
io_szBuffer
    Field name buffer. Input, output.
in_lSize
    Size of application-allocated buffer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Because Visual Basic can't receive char pointers, **QmpConsCritGetFldNameVB** accepts a preallocated buffer which it fills with the field name.

Example

```
MpHnd hconscrit;
QRESULT presult;
char szbuffer[1024];

QmpConsCritGetFldNameVB(hconscrit, szbuffer,
    sizeof(szbuffer), &presult);
```

QmpConsCritGetOtherFldHnd

GETS THE FIELD HANDLE OF THE OTHER FIELD THIS CRITERION USES.

Syntax

```
int QmpConsCritGetOtherFldHnd ( MpHnd in_hConsCrit, QRESULT*  
    out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the index of the other field associated with a consolidation criterion if successful, -1 if there is an error.

Notes

This function applies to consolidation rule types
QMS_CONSOL_RULE_LARGEST_OTHER,
QMS_CONSOL_RULE_SMALLEST_OTHER, QMS_CONSOL_RULE_NUM_VALUE,
and QMS_CONSOL_RULE_GROUP_ORDER_OTHER.

Some consolidation rules use two fields. The first field is the one whose value is being determined. The second field's value is evaluated to determine the value of the first field. **QmpConsCritGetOtherFldHnd** returns the value of the second field.

See Also

QmpConsCritGetFldHnd

Example

```
MpHnd hconscript;  
QRESULT presult;  
int iootherfldhnd;  
  
iotherfldhnd = QmpConsCritGetOtherFldHnd(hconscript,  
    &presult);
```

QmpConsCritGetOtherFldName

GETS THE FIELD NAME OF THE OTHER FIELD THIS CRITERION USES.

Syntax

```
const char* QmpConsCritGetOtherFldName ( MpHnd in_hConsCrit,
                                         QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the name of the other field associated with a consolidation criterion if successful, NULL if there is an error.

Notes

This function applies to consolidation rule types

QMS_CONSOL_RULE_LARGEST_OTHER,
QMS_CONSOL_RULE_SMALLEST_OTHER, QMS_CONSOL_RULE_NUM_VALUE,
and QMS_CONSOL_RULE_GROUP_ORDER_OTHER.

Some consolidation rules use two fields. The first field is the one whose value is being determined. The second field's value is evaluated to determine the value of the first field. **QmpConsCritGetOtherFldName** returns the name of the second field.

See Also

[QmpConsCritGetFldName](#)

Example

```
MpHnd hconscript;
QRESULT presult;
const char *pszotherfldname;

pszotherfldname = QmpConsCritGetOtherFldName(hconscript,
                                             &presult);
```

QmpConsCritGetOtherFldNameVB

GETS THE FIELD NAME OF THE OTHER FIELD THIS CRITERION USES (VISUAL BASIC VERSION).

Syntax

```
void QmpConsCritGetOtherFldNameVB ( MpHnd in_hConsCrit, char*  
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

io_szBuffer
Other field name buffer. *Input, output*.

in_lSize
Size of application-allocated buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function applies to consolidation rule types
QMS_CONSOL_RULE_LARGEST_OTHER and
QMS_CONSOL_RULE_SMALLEST_OTHER.

The consolidation rules QMS_CONSOL_RULE_LARGEST_OTHER and
QMS_CONSOL_RULE_SMALLEST_OTHER use two fields. The first field is the one
whose value is being determined. The second field's value is evaluated to
determine the value of the first field. **QmpConsCritGetOtherFldNameVB**
puts the name of the second field into *io_szBuffer*.

See Also

[QmpConsCritGetOtherFldName](#)

Example

```
MpHnd hconscrit;  
QRESULT presult;  
char szotherfldname[1024];  
  
QmpConsCritGetOtherFldNameVB(hconscrit, szotherfldname,  
    sizeof(szotherfldname), &presult);
```

QmpConsCritGetRule

GETS DATA CONSOLIDATION RULE WHICH THE CRITERION WILL USE.

Syntax

```
QMS_CONSOL_RULE QmpConsCritGetRule ( MpHnd in_hConsCrit,  
    QRESULT* out_pResult );
```

in_hConsCrit

Handle to consolidation criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns data consolidation rule if successful, QMS_CONSOL_RULE_UNDEFINED if there is an error.

Notes

This function is valid for all consolidation rule types.

QmpConsCritGetRule gets the rule assigned to a data consolidation criterion.

Example

```
MpHnd hconscrit;  
QRESULT presult;  
QMS_CONSOL_RULE rule;  
  
rule = QmpConsCritGetRule(hconscrit, &presult);
```

QmpConsCritGetTreatment

GETS THE TREATMENT TYPE OF THE FIELD THIS CONSOLIDATION CRITERION APPLIES TO.

Syntax

```
QMS_CONSOL_TREAT QmpConsCritGetTreatment ( MpHnd
    in_hConsCrit, QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns field treatment type if successful, QMS_CONSOL_TREAT_UNDEFINED if there is an error.

Notes

This function applies to the following consolidation rule types:

QMS_CONSOL_RULE_LARGEST
QMS_CONSOL_RULE_SMALLEST
QMS_CONSOL_RULE_MOST_COMMON
QMS_CONSOL_RULE_LARGEST_OTHER
QMS_CONSOL_RULE_SMALLEST_OTHER

During data consolidation, computations may be performed on duplicate record fields in order to generate the field value of the consolidated record. The library must know how to interpret these fields (as strings, integers, dates, etc.) in order to do these calculations correctly.

QmpConsCritGetTreatment returns a field's treatment type (how the field will be interpreted for data consolidation calculations).

Example

```
MpHnd hconscrit;
QRESULT presult;
QMS_CONSOL_TREAT treat;

treat = QmpConsCritGetTreatment(hconscrit, &presult);
```

QmpConsCritIsValid

DETERMINES IF A DATA CONSOLIDATION CRITERION IS VALID OR NOT.

Syntax

```
QBOOL QmpConsCritIsValid ( MpHnd in_hConsCrit, QRESULT*  
    out_pResult );
```

in_hConsCrit

Handle to consolidation criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if the criterion is valid, else QFALSE.

Example

```
MpHnd hconscrit;  
QBOOL bisvalid;  
QRESULT presult;  
  
bisvalid = QmpConsCritIsValid(hconscrit, &presult);
```

QmpConsCritSetDataLstRule

SETS THE QMS_CONSOL_RULE_DATLST RULE FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetDataLstRule ( MpHnd in_hConsCrit,
                                QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
                                in_szFldName, long in_lDataLstId, QRESULT* out_pResult );
in_hConsCrit
    Handle to consolidation criterion. Input.
in_Rule
    Consolidation rule. QMS_CONSOL_RULE_DATLST is the only valid value.
    Input.
in_hRec
    Handle to prototype record. Input.
in_szFldName
    Field name this criterion applies to. Input.
in_lDataLstId
    Data list ID. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the QMS_CONSOL_RULE_DATLST consolidation rule for a consolidation criterion.

Example

```
MpHnd hconscrit;
MpHnd hrec;
QRESULT presult;

QmpConsCritSetDataLstRule(hconscrit, QMS_CONSOL_RULE_DATLST,
                           hrec, "TotalSales", 2, &presult);
```

QmpConsCritSetExprSubStr

SETS EXPRESSION SUBSTRING FOR A FLDVAL DATA CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetExprSubStr( MpHnd in_hConsCrit, int
                               in_iCharStartPos, int in_iNumChars, QRESULT* out_pResult );
```

in_hConsCrit
Handle to data consolidation criterion. *Input*.

in_iCharStartPos
Start position of substring in first field. *Input*.

in_iNumChars
Number of substring characters in first field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

In the fldval data consolidation rule, a field value is used as part of an expression of the type *FieldValue Operator Constant*. This function specifies a substring of the field value to be used in the expression.

Example

```
QmpConsCritSetExprSubStr( in_hConsCrit, 3, 4, &out_pResult );
```

QmpConsCritSetFldValRule

SETS THE QMS_CONSOL_RULE_FLDVAL RULE FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetFldValRule ( MpHnd in_hConsCrit,
    QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
    in_szFldName, QMS_FLDEVAL_OPER in_Operator, MpHnd in_hVar,
    QRESULT* out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

in_Rule
Consolidation rule. QMS_CONSOL_RULE_FLDVAL is the only valid value.
Input.

in_hRec
Handle to prototype record. *Input*.

in_szFldName
Name of field this criterion applies to.

in_Operator
Operator used for field evaluation comparison. *Input*.

in_hVar
Handle of variant containing constant value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the QMS_CONSOL_RULE_FLDVAL consolidation rule for a consolidation criterion.

Example

```
QmpDeclHnd(hconscript);
QmpDeclHnd(hrec);
QRESULT presresult;
QmpDeclHnd(hvar); /* variant */

hvar = QmpVarCreate(&presresult);
QmpVarSetLong(hvar, 50000, &presresult);
QmpConsCritSetFldValRule(hconscript, QMS_CONSOL_RULE_FLDVAL,
    hrec, "TotalSales", QMS_FLDEVAL_OPER_GREATER, hvar,
    &presresult);
```

QmpConsCritSetGroupOrderRule

SETS THE QMS_CONSOL_RULE_GROUP_ORDER RULE FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetGroupOrderRule( MpHnd in_hConsCrit,
    QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
    in_szFldName, int in_iOrder, QRESULT* out_pResult );

in_hConsCrit
    Handle to consolidation criterion. Input.

in_Rule
    Consolidation rule. QMS_CONSOL_RULE_GROUP_ORDER is the only valid value. Input.

in_hRec
    Handle to record prototype. Input.

in_szFldName
    Field this criterion applies to. Input.

in_iOrder
    Group order of field to select. Input.

out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The QMS_CONSOL_RULE_GROUP_ORDER rule selects the value for *in_szFldName* from the dupe group record specified by the *in_iOrder* number. If *in_iOrder* is greater than the number of records in the dupe group, an empty value is returned.

Example

```
QmpConsCritSetGroupOrderRule( hConsCrit,
    QMS_CONSOL_RULE_GROUP_ORDER, hRec, "Lname", 4, &qres );
```

QmpConsCritSetGroupOtherRule

SETS THE QMS_CONSOL_RULE_GROUP_ORDER_OTHER RULE FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetGroupOtherRule( MpHnd in_hConsCrit,
                                QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
                                in_szFldName, int in_iOrder, const char* in_szOtherFldName,
                                QRESULT* out_pResult );
```

in_ConsCrit
Handle to consolidation criterion. *Input*.

in_Rule
Consolidation rule. QMS_CONSOL_RULE_GROUP_ORDER_OTHER is the only valid value. *Input*.

in_hRec
Handle to record prototype. *Input*.

in_szFldName
Field this criterion applies to. *Input*.

in_iOrder
Group order of field to select. *Input*.

in_szOtherFldName
Name of other field to examine. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The QMS_CONSOL_RULE_GROUP_ORDER_OTHER rule selects the value contained in the *in_szOtherFldName* field from the dupe group record specified by the *in_iOrder* number. The value is placed in the field specified by *in_szFldName*. If *in_iOrder* is greater than the number of records in the dupe group, an empty value is returned.

Example

```
QmpConsCritSetGroupOtherRule( hConsCrit,
                               QMS_CONSOL_RULE_GROUP_ORDER_OTHER, hRec, "Fname1", 4,
                               "FName", &qres );
```

QmpConsCritSetNumValRule

SETS THE NUM_VALUE RULE FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetNumValRule( MpHnd in_ConsCrit,
    QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
    in_szDstFldName, const char* in_szSrcFldName,
    QMS_FLDEVAL_OPER in_Operator, MpHnd in_hVar, QRESULT*
    out_pResult );
```

in_ConsCrit
Handle to data consolidation criterion. *Input*.

in_Rule
Consolidation rule. QMS_CONSOL_RULE_NUM_VALUE is the only valid enum. *Input*.

in_hRec
Handle to record prototype. *Input*.

in_szDstFldName
Name of field in which to store results. *Input*.

in_szSrcFldName
Name of field to evaluate. *Input*.

in_Operator
Operator used for field evaluation comparison. *Input*.

in_hVar
Handle of variant containing constant value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The Num value rule evaluates the source field/operator/variant expression, and counts the number of records in a dupe group that make the expression true. This record count is put into the destination field.

Example

```
QmpConsCritSetNumValRule( hConsCrit,
    QMS_CONSOL_RULE_NUM_VALUE, hRec, "NameCount", "Lname",
    QMS_FLDEVAL_OPER_EQUAL, "Smith", &qres );
```

QmpConsCritSetOtherFldRule

SETS THE QMS_CONSOL_RULE_LARGEST_OTHER AND
QMS_CONSOL_RULE_SMALLEST_OTHER RULES FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetOtherFldRule ( MpHnd in_hConsCrit,
                                QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
                                in_szFldName, QMS_CONSOL_TREAT in_TreatmentType, const
                                char* in_szOtherFldName, QRESULT* out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

in_Rule
Consolidation rule. QMS_CONSOL_RULE_LARGEST_OTHER and
QMS_CONSOL_RULE_SMALLEST_OTHER are the only valid values.
Input.

in_hRec
Handle to prototype record. *Input*.

in_szFldName
Name of field this criterion applies to. *Input*.

in_TreatmentType
Treatment type of the other field (how to interpret the other field type for
data consolidation calculations). *Input*.
Valid enums are:

QMS_CONSOL_TREAT_STRING
QMS_CONSOL_TREAT_NUMBER
QMS_CONSOL_TREAT_DATE

in_szOtherFldName
Name of the other field to examine. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the QMS_CONSOL_RULE_LARGEST_OTHER and
QMS_CONSOL_RULE_SMALLEST_OTHER consolidation rules for a consolidation
criterion.

Example

```
MpHnd hconscript;
MpHnd hrec;
QRESULT presult;

QmpConsCritSetOtherFldRule(hconscript,
    QMS_CONSOL_RULE_LARGEST_OTHER, hrec, "TotalSales",
    QMS_CONSOL_TREAT_NUMBER, "YearsEmployed", &presult);
```

QmpConsCritSetSimpleRule

SETS THE MASTER, NON-BLANK, ROLLUP, AVERAGE, NUM DUPES, SHORTEST, AND LONGEST RULES FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetSimpleRule ( MpHnd in_hConsCrit,
    QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
    in_szFldName, QRESULT* out_pResult );
```

in_hConsCrit

Handle to consolidation criterion. *Input*.

in_Rule

Consolidation rule. *Input*.

The following are the only valid enums:

QMS_CONSOL_RULE_MASTER
QMS_CONSOL_RULE_NON_BLANK
QMS_CONSOL_RULE_ROLLUP
QMS_CONSOL_RULE_AVERAGE
QMS_CONSOL_RULE_NUM_DUPES
QMS_CONSOL_RULE_SHORTEST
QMS_CONSOL_RULE_LONGEST

in_hRec

Handle to prototype record. *Input*.

in_szFldName

Name of field this criterion applies to.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Example

```
MpHnd hconscrit;
MpHnd hrec;
QRESULT presult;

QmpConsCritSetSimpleRule(hconscrit, QMS_CONSOL_RULE_ROLLUP,
    hrec, "TotalSales", &presult);
```

QmpConsCritSetTreatmentRule

SETS THE QMS_CONSOL_RULE_LARGEST, QMS_CONSOL_RULE_SMALLEST, and QMS_CONSOL_RULE_MOST_COMMON RULES FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetTreatmentRule ( MpHnd in_hConsCrit,
    QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
    in_szFldName, QMS_CONSOL_TREAT in_TreatmentType, QRESULT*
    out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

in_Rule
Consolidation rule. *Input*.
The following are the only valid enums:

QMS_CONSOL_RULE_LARGEST
QMS_CONSOL_RULE_SMALLEST
QMS_CONSOL_RULE_MOST_COMMON

in_hRec
Handle to prototype record. *Input*.

in_szFldName
Name of field this criterion applies to.

in_TreatmentType
Treatment type of the field (how to interpret the field type for data consolidation calculations). *Input*.
Valid enums are:

QMS_CONSOL_TREAT_STRING
QMS_CONSOL_TREAT_NUMBER
QMS_CONSOL_TREAT_DATE

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Example

```
MpHnd hconscript;
MpHnd hrec;
QRESULT presult;

QmpConsCritSetTreatmentRule(hconscript,
    QMS_CONSOL_RULE_LARGEST, hrec, "TotalSales",
    QMS_CONSOL_TREAT_NUMBER, &presult);
```

QmpConsCritSetValRule

SETS THE QMS_CONSOL_RULE_VAL RULE FOR A CONSOLIDATION CRITERION.

Syntax

```
void QmpConsCritSetValRule ( MpHnd in_hConsCrit,
    QMS_CONSOL_RULE in_Rule, MpHnd in_hRec, const char*
    in_szFldName, MpHnd in_hVar, QRESULT* out_pResult );
```

in_hConsCrit
Handle to consolidation criterion. *Input*.

in_Rule
Consolidation rule; QMS_CONSOL_RULE_VAL is the only valid value. *Input*.

in_hRec
Handle to prototype record. *Input*.

in_szFldName
Name of field this criterion applies to. *Input*.

in_hVar
Handle to variant containing the value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

See Also

[QmpConsCritFillVal](#).

Example

```
MpHnd hconscrit;
MpHnd hrec;
MpHnd hVar;
QRESULT presult;

QmpConsCritSetValRule ( hconscrit, QMS_CONSOL_RULE_VAL,
    hrec, "TotalSales", hVar, &presult );
```

Function Class: QmpConsRpt*

CONSOLIDATED RECORDS REPORT FUNCTIONS.

Quick Reference

Function	Description	Page
QmpConsRptAddPreProcDataSrc	Specifies the preprocessed data source to be used by the consolidated records report phase.	164
QmpConsRptCreate	Creates a consolidated records report.	165
QmpConsRptDestroy	Destroys a consolidated records report.	167
QmpConsRptGetNthRecInterval	Gets consolidated records report Nth record event interval.	168
QmpConsRptRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	169
QmpConsRptSetDupeGrpId	Specifies whether to print dupe group column IDs in the report.	170
QmpConsRptSetDupeGrpScore	Specifies whether to print dupe group column scores in the report.	171
QmpConsRptSetListId	Specifies whether to print the data list ID column in the report.	172
QmpConsRptSetNthRecInterval	Sets consolidated records report Nth record event interval.	173
QmpConsRptSetPrintKeys	Specifies whether to print the record primary key column in the report.	174
QmpConsRptSetPrintSrc	Specifies whether to print a record source ID column in the report.	175
QmpConsRptUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	176
QmpConsRptUseCons	Gives the data consolidation phase to a consolidation report.	177
QmpConsRptUseDupGrp	Specifies the dupe groups object to use in the consolidated records report.	178

QmpConsRptAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE CONSOLIDATED RECORDS REPORT PHASE.

Syntax

```
void QmpConsRptAddPreProcDataSrc ( MpHnd in_hConsRpt, MpHnd
                                   in_hDataSrc, QRESULT* out_pResult );
```

in_hConsRpt
Handle to consolidated records report object. *Input*.

in_hDataSrc
Handle to preprocessed data source to be used. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data sources of type
QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the consolidated records report phase.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

Example

See example on [page 165](#).

QmpConsRptCreate

CREATES A CONSOLIDATED RECORDS REPORT.

Syntax

```
MpHnd QmpConsRptCreate (QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns handle to consolidated records report if successful, or NULL if there is an error.

Notes

If the creation function fails, an error code is returned in the *out_pResult* parameter. Application programs should always test the result code for success.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

```
QmpDeclHnd( hConsReport );
hConsReport = QmpConsRptCreate( &qres );

/* Print a Consolidated records report */
QmpRptSetLinesPerPage( hConsReport, -1, &qres );
QmpRptSetPageWidthInChars( hConsReport, 79, &qres );
QmpConsRptSetPrintKeys( hConsReport, QTRUE, &qres );
QmpConsRptSetPrintSrc( hConsReport, QTRUE, &qres );
QmpConsRptUseDupGrp( hConsReport, hDupeGroups, &qres );
QmpConsRptUseCons( hConsReport, hConsolidation, &qres );
QmpPhaseSetRecProto( hConsReport, hProtoRec, &qres );
sprintf( szStringPtr, "Rpt-Consol-%d.log", iThreshHold );
sprintf( pszTempPathBuffer, "%s%s%s", pszOutputPathBuffer, pszDelim,
        szStringPtr );
remove(pszTempPathBuffer);
QmpRptSetFilename( hConsReport, pszTempPathBuffer, &qres );
sprintf( szTempPtr, ": Consolidated Records Report for a Threshold of %d",
        iThreshHold );
strcat( szStringPtr, szTempPtr );
QmpRptSetTitle( hConsReport, szStringPtr, &qres );
QmpRptSetSubtitle( hConsReport, "Generated for Make-A-Wish Foundation", &qres );
)
QmpPhaseUseDITR( hConsReport, hDITR, &qres );
if ( bEveryNth ) {
    QmpConsRptRegEveryNthRecFunc( hConsReport, pEveryNthRecordClient, &qres );
    QmpConsRptSetNthRecInterval( hConsReport, 200, &qres );
}
if ( bUseSampling ) {
    OutputSampling.lMaxRecords = 1000; /* max number of records to output */
    OutputSampling.lFirstRecordNum = 1; /* first record to output */
    OutputSampling.lLastRecordNum = 1000; /* last record to output */
    OutputSampling.lInterval = 0; /* Sampling interval for output */
    OutputSampling.lGroupSize = 0; /* size of group beginning at each */
    /* output interval */
    OutputSampling.lMaxDupeGroups = 5; /* max number of Dupe Groups to */
    /* output. */
}
```

QmpConsRptCreate

```
    OutputSampling.lDupesFirst = 1;      /* first Dupe Group to output. */
    OutputSampling.lDupesLast = 20;       /* last DupeGroup to output. */
    putSampling.lDupesInterval = 2;       /* Dupe Group sampling interval. */
    QmpRptSetStatSamp( hConsReport, &OutputSampling, &qres );
}
printf( "Starting Near Miss Report phase:\n" );
if ( bUsePrePro ) {
    QmpConsRptAddPreProcDataSrc( hConsReport, hPreProDataSrc, &qres );
}
QmpPhaseStart( hConsReport, &qres );
QmpPhaseDestroy( hConsReport, &qres );
```

QmpConsRptDestroy

DESTROYS A CONSOLIDATED RECORDS REPORT.

Syntax

```
void QmpConsRptDestroy ( MpHnd in_hConsRpt, QRESULT*  
    out_pResult );
```

in_hConsRpt

Handle to consolidated records report. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function destroys the specified consolidated records report.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

Example

```
MpHnd hConsolRpt;  
QRESULT qres;  
  
QmpConsRptDestroy( hConsolRpt, &qres );
```

QmpConsRptGetNthRecInterval

GETS CONSOLIDATED RECORDS REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpConsRptGetNthRecInterval ( MpHnd in_hConsRpt,
                                  QRESULT* out_pResult );
in_hConsRpt
    Handle to consolidated records report. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns consolidated records report Nth record event interval if successful, or -1 if there is an error.

Notes

This function gets the consolidated records report phase Nth record event interval.

The consolidated records report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the consolidated records report phase processes 100 records, it will send out a notification to *all* registered clients.

Example

```
MpHnd hConsolRpt;
QRESULT qres;
long lCount;

lCount = QmpConsRptGetNthRecInterval( hConsolRpt, &qres );
```

QmpConsRptRegEveryNthRecFunc

REGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpConsRptRegEveryNthRecFunc ( MpHnd in_hConsRpt,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hConsRpt
Handle to consolidated records report. Input.
in_Func
Pointer to event handler. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The consolidated records report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

See example on [page 165](#).

QmpConsRptSetDupeGrpId

SPECIFIES WHETHER TO PRINT DUPE GROUP COLUMN IDs IN THE REPORT.

Syntax

```
void QmpConsRptSetDupeGrpId( MpHnd in_hConsRpt, QBOOL  
    in_bDupeGrpId, QRESULT* out_pResult );  
  
in_hConsRpt  
    Handle to consolidated records report object. Input.  
  
in_bDupeGrpId  
    If QTRUE, print dupe group IDs. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

This function specifies whether to print dupe group column IDs in the report. The dupe group ID of a consolidated record is the same as that of the dupe group it was constructed from.

Example

```
MpHnd hConsolRpt;  
QRESULT qres;  
  
QmpConsRptSetDupeGrpId( hConsolRpt, QTRUE, &qres );
```

QmpConsRptSetDupeGrpScore

SPECIFIES WHETHER TO PRINT DUPE GROUP COLUMN SCORES IN THE REPORT.

Syntax

```
void QmpConsRptSetDupeGrpScore( MpHnd in_hConsRpt, QBOOL  
    in_bDupeGrpScore, QRESULT* out_pResult );
```

in_hConsRpt

Handle to consolidated records report object. *Input*.

in_bDupeGrpScore

If QTRUE, print dupe group scores. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The dupe group score is the match score between a master duplicate and subordinate duplicate. All the subordinates in a dupe group are scored against the group master. The dupe group score of a consolidated record is always 0.

Example

```
MpHnd hConsolRpt;  
QRESULT qres;  
  
QmpConsRptSetDupeGrpScore( hConsolRpt, QTRUE, &qres );
```

QmpConsRptSetListId

SPECIFIES WHETHER TO PRINT THE DATA LIST ID COLUMN IN THE REPORT.

Syntax

```
void QmpConsRptSetListId( MpHnd in_hConsRpt, QBOOL in_bListId,
                           QRESULT* out_pResult );
in_hConsRpt
    Handle to consolidated records report object. Input.
in_bListId
    If QTRUE, print data list IDs. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

A record's data list ID field records which data list the record belongs to. The data list ID of a consolidated record is always 0.

Example

```
MpHnd hConsolRpt;
QRESULT qres;

QmpConsRptSetListId ( hConsolRpt, QFALSE, &qres );
```

QmpConsRptSetNthRecInterval

SETS CONSOLIDATED RECORDS REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpConsRptSetNthRecInterval ( MpHnd in_hConsRpt, long
                                   in_lInterval, QRESULT* out_pResult );
```

in_hConsRpt
Handle to consolidated records report. *Input*.

in_lInterval
Report Nth record event interval. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the consolidated records report Nth record event interval.

The consolidated records report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the consolidated records report phase processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

Example

See example on [page 165](#).

QmpConsRptSetPrintKeys

SPECIFIES WHETHER TO PRINT THE RECORD PRIMARY KEY COLUMN IN THE REPORT.

Syntax

```
void QmpConsRptSetPrintKeys (MpHnd in_hConsRpt, QBOOL  
    in_bPrintKeys, QRESULT* out_pResult );
```

in_hConsRpt

Handle to consolidated records report object. *Input*.

in_bPrintKeys

If QTRUE, print record primary keys. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The primary key is a number (generally a record number) that is unique to a data source. It is usually the counter value when the records are read in by the data input phase. If a job has one data source, the primary key uniquely identifies a record. If a job has multiple data sources, the primary key and source ID values uniquely identify a record.

The primary key value of a consolidated record is always 0.

Example

See example on [page 165](#).

QmpConsRptSetPrintSrc

SPECIFIES WHETHER TO PRINT A RECORD SOURCE ID COLUMN IN THE REPORT.

Syntax

```
void QmpConsRptSetPrintSrc ( MpHnd in_hConsRpt, QBOOL  
    in_bPrintSources, QRESULT* out_pResult );
```

in_hConsRpt
Handle to consolidated records report object.

in_bPrintSources
If QTRUE, print record source IDs.

out_pResult
Result code.

Return Value

None.

Notes

Consolidated records always have a source ID value of 0.

Example

See example on [page 165](#).

QmpConsRptUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpConsRptUnregEveryNthRecFunc ( MpHnd in_hConsRpt,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hConsRpt
    Handle to consolidated records report. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function unregisters the specified event handler from the consolidated records report, so that the event handler will no longer be notified of every Nth record processed.

The consolidated records report can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
MpHnd hConsolRpt;
QMS_EVERY_NTHREC_FUNC pEveryNthRecConsRpt;
QRESULT qres;

QmpConsRptUnregEveryNthRecFunc( hConsolRpt,
                                    pEveryNthRecConsRpt, &qres );
```

QmpConsRptUseCons

GIVES THE DATA CONSOLIDATION PHASE TO A CONSOLIDATION REPORT.

Syntax

```
void QmpConsRptUseCons( MpHnd in_hConsRpt, MpHnd in_hCons,
                         QRESULT* out_pResult );
```

in_hConsRpt
Handle to consolidation report. *Input*.

in_hCons
Handle to data consolidation phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Example

```
QmpConsRptUseCons ( hConsRpt, hCons, &qres );
```

QmpConsRptUseDupGrp

SPECIFIES THE DUPE GROUPS OBJECT TO USE IN THE CONSOLIDATED RECORDS REPORT.

Syntax

```
void QmpConsRptUseDupGrp( MpHnd in_hConsRpt, MpHnd  
                           in_hdUpGrps, QRESULT* out_pResult );  
  
in_hConsRpt  
    Handle to consolidated records report object. Input.  
  
in_hdUpGrps  
    Handle to dupe groups object. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

The dupe groups phase determines dupe group membership and master/subordinate status. When directed by the application, it also constructs consolidated records.

Example

```
MpHnd hConsolRpt;  
MpHnd hDupGrps;  
QRESULT qres;  
  
QmpConsRptUseDupGrp( hConsolRpt, hDupGrp, &qres );
```

Function Class: QmpCrit*

FIELD MATCH CRITERION FUNCTIONS.

Field Match Criterion Transformations

A field match criterion may specify that certain operations (transformations) be performed on the fields' values before or while applying the matching algorithms. The following Centrus Merge/Purge criterion transformations are listed in order of precedence (if multiple transformations are applied to a criterion):

1. Remove whitespace—all blank characters are removed prior to comparisons.
2. Remove punctuation—all punctuation characters are removed prior to comparisons. Punctuation characters are defined as non-space and non-alphanumeric.
3. Remove letters—all letters are removed prior to comparisons.
4. Remove digits—all digits (numbers) are removed prior to comparisons.
5. Check if one or both of the match fields are blank.
6. Align strings—an attempt to line up matching sub-strings is made prior to the comparison. For example, for the values of “Smith” and “Ms. Smith”, leading blanks would be added to “Smith” such that the “S” becomes the fifth character, as it is in “Ms. Smith”.
7. Perform substringing.
8. Ignore case—upper and lower case is ignored during string comparisons.

Quick Reference

Function	Description	Page
QmpCritAddAlg	Adds an algorithm to a criterion.	182
QmpCritAddNameMatchAlg	Adds the name matching algorithm to a criterion.	183
QmpCritClear	Clears a criterion.	185
QmpCritCompute	Invokes the Criterion compute method to compare two records.	186
QmpCritCreate	Creates a Criterion.	187
QmpCritDestroy	Destroys a criterion.	189
QmpCritGetAlgCnt	Gets the number of algorithms in a criterion.	190
QmpCritGetAlgWeight	Gets the weighting for an algorithm used in a criterion.	191
QmpCritGetBothBlankRes	Gets the “both blank” match result setting.	192

Function	Description	Page
QmpCritGetCaseSens	Gets the criterion's case sensitivity setting.	193
QmpCritGetIgnoreAlpha	Gets the criterion "ignore letters" setting.	194
QmpCritGetIgnoreNum	Gets "ignore Numeric" criterion property.	195
QmpCritGetIgnorePunct	Gets "ignore punctuation" field match criterion property value.	196
QmpCritGetIgnoreSpaces	Gets the "criterion ignore white space" setting.	197
QmpCritGetIgnoreXforms	Gets ignore transformations property.	198
QmpCritGetNameRecProto	Gets the name record prototype for the name matching algorithm.	199
QmpCritGetOneBlankRes	Gets the criterion's "one blank" match result setting.	200
QmpCritGetRule	Gets the rule the field match criterion uses to combine algorithm scores.	201
QmpCritGetStrAlign	Gets the criterion's string alignment setting.	202
QmpCritGetThreshold	Gets the threshold value for a criterion.	203
QmpCritIsValid	Tests whether a criterion is valid.	204
QmpCritRemAlg	Removes the specified algorithm from the criterion.	205
QmpCritSetAlgWeight	Sets the weighting for an algorithm used in the criterion.	206
QmpCritSetBothBlankRes	Sets the criterion's "both blank" match result setting.	207
QmpCritSetCaseSens	Sets the criterion's case sensitivity.	208
QmpCritSetIgnoreAlpha	Sets a criterion's "ignore letters" setting.	209
QmpCritSetIgnoreNum	Sets "ignore Numeric" criterion property.	210
QmpCritSetIgnorePunct	Sets "ignore punctuation" field match criterion property.	211
QmpCritSetIgnoreSpaces	Sets criterion's "ignore white space" setting.	212
QmpCritSetIgnoreXforms	Sets ignore transformations property.	213
QmpCritSetNameRecProto	Attaches a record prototype containing the name fields to the match criterion.	214
QmpCritSetOneBlankRes	Sets the criterion's "one blank" match result setting.	216
QmpCritSetPairFullFld	Establishes the pair of record fields to be compared by the criterion (no substringing).	217
QmpCritSetPairPartFld	Establishes the pair of record fields to be compared by the criterion (partial substringing).	218
QmpCritSetPairSubStrFld	Establishes the pair of record fields to be compared by the criterion (full substringing).	219
QmpCritSetRule	Sets the rule the criterion uses to combine algorithm scores.	221

Function	Description	Page
QmpCritSetStrAlign	Sets the string alignment property for the criterion.	223
QmpCritSetThreshold	Sets the threshold for the criterion.	224

QmpCritAddAlg

ADDS AN ALGORITHM TO A CRITERION.

Syntax

```
void QmpCritAddAlg ( MpHnd in_hCrit, MpHnd in_hAlg, int
                      in_iWeight, QRESULT* out_pResult );
in_hCrit
    Handle to criterion. Input.
in_hAlg
    Handle to algorithm. Input.
in_iWeight
    Optional weight to assign to the algorithm. If assigned, this value must be
    between 0 and 100. The default value is 0. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure a field matching criterion for each field pair you want to compare. To add more than one algorithm to a criterion, repeat the call to **QmpCritAddAlg**, specifying additional algorithms in the *in_hAlg* argument. The same algorithm cannot be added more than once.

A weight needs to be supplied via the *in_iWeight* argument only if the algorithms for the criterion are to have their results combined using the weighted average rule (see **QmpCritCreate**), although there is no harm in supplying a weight when it's not necessary. If the algorithm results are to be combined using the weighted average rule, then the total of all the algorithm weights for the criterion must total 100.

See Also

[QmpCritRemAlg](#)

Example

```
/* Set up a field match criterion for matching on */
/* last name */
QmpCritAddAlg( hCritLName, hAlgEditDist, 0, &qres );
QmpCritAddAlg( hCritLName, hAlgSoundex, 0, &qres );
QmpCritSetPairFullFld( hCritLName, hRec, "LastName",
                         "LastName", &qres );
```

QmpCritAddNameMatchAlg

ADDS THE NAME MATCHING ALGORITHM TO A CRITERION.

Syntax

```
void QmpCritAddNameMatchAlg( MpHnd in_hCrit, MpHnd in_hAlg,
    const char* in_pszNameData, MpHnd in_NameProtoRec,
    QMS_NAME_ORDER in_NameOrder, QBOOL in_bMaidenMatch, QBOOL
    in_bInitialMatch, QBOOL in_bMatchHygiene, QBOOL
    in_bInitialAlias, int in_iWeight, QRESULT* out_pResult );
```

in_hCrit

Handle to field match criterion. *Input*.

in_hAlg

Handle to name match algorithm. *Input*.

in_pszNameData

Location of name database that contains raw names, aliases, prefixes, titles, etc. *Input*.

in_NameProtoRec

Handle to name match prototype record. *Input*.

in_NameOrder

Order that names will appear in the name prototype record. If the names are in no set order, or you don't know what the order is, use the value QMS_NAME_ORDER_NONE, and the name matching algorithm will try and determine name order itself. *Input*.

Valid enums are:

QMS_NAME_ORDER_NONE

QMS_NAME_ORDER_FIRSTLAST

QMS_NAME_ORDER_LASTFIRST

in_bMaidenMatch

Enables or disables maiden name matching. For example, if set to QTRUE, "Susan Black-White" would match "Sue Black". *Input*.

in_bInitialMatch

Enables or disables initial name matching. For example, if set to QTRUE, "Sue Black" would match "S. Black". *Input*.

in_bMatchHygiene

Enables or disables name match hygiene. For example, if set to QTRUE, "Dr. Susan White, Ph.D" would match "Sue White". *Input*.

in_bInitialAlias

Enables or disables alias initial matching. For example, if set to QTRUE, "Bill Brown" would match "W. Brown". *Input*.

in_iWeight

Sets the algorithm weight (if algorithm scores are combined using a weighted average). *Input*.

out_pResult
 Result code. *Output.*

Return Value

None.

Notes

QmpCritAddNameMatchAlg adds a name matching algorithm of type QMS_ALGORITHM_TYPE_NAME to the field match criterion. The function arguments set all the properties available for name matching.

When using the name matching algorithm, the ignore transformations property for field match criteria should be QTRUE.

See the "Notes" section of **QmpCritSetNameRecProto** for a discussion of the name prototype record.

Name matching will be speediest when you know the first name/last name order of your data and set the appropriate QMS_NAME_ORDER_* enumerated type. Name matching speed is reduced if you specify QMS_NAME_ORDER_NONE and make the library determine name order for itself.

See Also

QmpCritGetIgnoreXforms, **QmpCritSetIgnoreXforms**,
QmpCritAddAlg.

Example

```

QBOOL bIgnore;
hMatchByName = QmpAlgCreate( QMS_ALGORITHM_TYPE_NAME,
    &qres );

char pszWordDataPath[QMS_MAX_PATH] = { " " };

/* go get the name data */
QmpIniFilReadItemValStrVB( hInifil, pszWordDataPath, sizeof(
    pszWordDataPath ), "[SETUP]", "WordDataPath", "", &qres );

QmpCritAddNameMatchAlg(
    hNameMatchCriterion,
    hMatchByName,           /* handle to algorithm */
    pszWordDataPath,        /* path to name database */
    hNameMatchRec,          /* record contains name fields */
    QMS_NAME_ORDER_NONE,   /* name order */
    QTRUE,                 /* maiden name matching */
    QFALSE,                /* initials matching */
    QTRUE,                 /* match hygiene */
    QFALSE,                /* initial alias matching */
    0, &qres );            /* weight and return */

```

QmpCritClear

CLEAR A CRITERION.

Syntax

```
void QmpCritClear ( MpHnd in_hCrit, QRESULT* out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the specified criterion to its original state, allowing it to be redefined.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpCritClear (hCrit, pResult);
```

QmpCritCompute

INVOKES THE CRITERION COMPUTE METHOD TO COMPARE TWO RECORDS.

Syntax

```
int QmpCritCompute ( MpHnd in_hCrit, MpHnd in_hRec1, MpHnd
                     in_hRec2, QRESULT* out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

in_hRec1
Handle to first record. *Input*.

in_hRec2.
Handle to second record. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the match result score if successful, -1 if there is an error.

Notes

This function compares two records using the specified field matching criterion. A field matching criterion consists of the specification of a pair of fields to be matched and the types of field matching comparisons to be performed on those fields.

See Also

[QmpCritCreate](#).

Example

```
QmpCritCompute (hCrit, hRec1, hRec2, pResult );
```

QmpCritCreate

CREATES A CRITERION.

Syntax

```
MpHnd QmpCritCreate ( QMS_ALGORITHMS_RULE in_AlgorithmsRule,
    int in_iThreshold, QRESULT* out_pResult );
```

in_AlgorithmsRule

Algorithm rule to use. *Input*.

Valid enums are:

QMS_ALGORITHMS_AVERAGE	Use the average of all algorithms. Default.
QMS_ALGORITHMS_WEIGHTED_AVERAGE	Use the weighted average of all algorithms.
QMS_ALGORITHMS_MAX	Use the algorithm that provided the highest score.
QMS_ALGORITHMS_MIN	Use the algorithm that provided the lowest score.
QMS_ALGORITHMS_AND	All algorithms must achieve a "threshold" score.
QMS_ALGORITHMS_OR	At least one algorithm must achieve a "threshold score".

in_iThreshold

Threshold for match; only required if *in_AlgorithmsRule* is QMS_ALGORITHMS_AND or QMS_ALGORITHMS_OR. Must be an integer between 0 and 100. The default value is 75. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the handle to the criterion if successful, NULL if there is an error.

Notes

QmpCritCreate is used to create a field match criterion and specify how algorithms are evaluated by the criterion. If the creation function fails, an error code is returned in the *out_pResult* parameter.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

If either of the algorithm rules QMS_ALGORITHMS_AND or QMS_ALGORITHMS_OR are used, the overall match score returned is 0 if the threshold is not met, and 100 if the threshold is met.

See Also

QmpCritAddAlg, **QmpCritDestroy**

Example

```
/* Create some field match criteria */
hCritAddr = QmpCritCreate( hJob, QMS_ALGORITHMS_AVERAGE, 0,
    &qres );
hCritLName = QmpCritCreate( hJob, QMS_ALGORITHMS_MAX, 0, &qres
    );
hCritZipCode = QmpCritCreate( hJob, QMS_ALGORITHMS_AVERAGE, 0,
    &qres );
hCritZip4 = QmpCritCreate( hJob, QMS_ALGORITHMS_AVERAGE, 0,
    &qres );
```

QmpCritDestroy

DESTROYS A CRITERION.

Syntax

```
void QmpCritDestroy ( MpHnd in_hCrit, QRESULT* out_pResult );  
in_hCrit  
Handle to criterion. Input.  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

This function destroys the specified criterion.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

See Also

QmpCritCreate.

Example

```
QmpCritDestroy (hCritZip4, &Result);
```

QmpCritGetAlgCnt

GETS THE NUMBER OF ALGORITHMS IN A CRITERION.

Syntax

```
long QmpCritGetAlgCnt ( MpHnd in_hCrit, QRESULT* out_pResult  
);
```

in_hCrit
Handle to criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the number of algorithms in a criterion if successful, -1 if there is an error.

Notes

This function returns the number of algorithms in a criterion.

See Also

[QmpCritCreate](#), [QmpCritAddAlg](#).

Example

```
QmpCritGetAlgCnt (hCrit, pResult );
```

QmpCritGetAlgWeight

GETS THE WEIGHTING FOR AN ALGORITHM USED IN A CRITERION.

Syntax

```
int QmpCritGetAlgWeight ( MpHnd in_hCrit, MpHnd in_hAlg,
    QRESULT* out_pResult );
in_hCrit
    Handle to criterion. Input.
in_hAlg
    Handle to algorithm. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the current weight for the specified algorithm if successful, -1 if there is an error.

Notes

Criteria with more than one algorithm may have weights assigned to algorithms. The value of each weight must be between 0 and 100, and the total of the weights of all algorithms must equal 100. The default weight value is 0.

See Also

[QmpCritSetAlgWeight](#).

Example

```
algorithm_weight = QmpCritGetAlgWeight (hCrit, hAlg, pResult
    );
```

QmpCritGetBothBlankRes

GETS THE “BOTH BLANK” MATCH RESULT SETTING.

Syntax

```
int QmpCritGetBothBlankRes(MpHnd in_hCrit, QRESULT*  
                           out_pResult );  
  
in_hCrit  
        Handle to criterion. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns the “both blank” result setting if successful, -1 if there is an error. The default setting is 0.

Notes

This function returns the application-specified “both fields blank” match result setting for the specified criterion. The default setting is 0 (i.e., when both fields are blank, they do NOT match, and a score of “0” is returned). A setting of 0 prevents pairs of blank fields from being scored as a match.

See Also

[QmpCritSetBothBlankRes](#), [QmpCritSetOneBlankRes](#),
[QmpCritGetOneBlankRes](#).

Example

```
match_result_setting = QmpCritGetBothBlankRes (hCrit, pResult  
                                              );
```

QmpCritGetCaseSens

GETS THE CRITERION'S CASE SENSITIVITY SETTING.

Syntax

```
QBOOL QmpCritGetCaseSens ( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns **QTRUE** if the criterion is case sensitive, **QFALSE** (the default) if not case sensitive. Returns **QFALSE** if there is an error.

Notes

Use this function to see if the specified criterion considers case in determining whether fields match.

See Also

[QmpCritSetCaseSens](#).

Example

```
case_sensitivity_setting = QmpCritGetCaseSens (hCrit, pResult  
);
```

QmpCritGetIgnoreAlpha

GETS THE CRITERION “IGNORE LETTERS” SETTING.

Syntax

```
QBOOL QmpCritGetIgnoreAlpha ( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit

Handle to field match criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if criterion ignores letters, QFALSE (the default) if the criterion doesn't ignore letters or if there is an error.

Notes

Use this function to get the current “ignore letters” setting for a criterion. If the setting is QTRUE, letters in fields will be ignored when matching them.

For example, if the “ignore letters” setting is QTRUE, “200 Main”, “200 First”, and “200 Second” would all be matches.

See Also

[QmpCritSetIgnoreAlpha](#).

Example

```
bIgnoreAlpha = QmpCritGetIgnoreAlpha  
    (hFirstNameMatchCriterion, &qres );
```

QmpCritGetIgnoreNum

GETS “IGNORE NUMERIC” CRITERION PROPERTY.

Syntax

```
QBOOL QmpCritGetIgnoreNum ( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit

Handle to field match criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

QTRUE if field numbers are ignored; QFALSE if field numbers are considered in a match.

Notes

The “ignore numeric” property directs the Centrus Merge/Purge library to either ignore or respect numbers in the fields being compared. If set to QTRUE, all numbers are removed from the strings before the comparison is made.

See Also

QmpCritSetIgnoreNum.

Example

```
QmpDeclHnd( hCritAddress );
QRESULT qres;
QBOOL bIgnoreNumeric;

bIgnoreNumeric = QmpCritGetIgnoreNum( hCritAddress, &qres );
```

QmpCritGetIgnorePunct

GETS “IGNORE PUNCTUATION” FIELD MATCH CRITERION PROPERTY VALUE.

Syntax

```
QBOOL QmpCritGetIgnorePunct ( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit
Handle to criterion. *Input*.
out_pResult
Result code. *Output*.

Return Value

`QTRUE` if field punctuation is ignored; `QFALSE` if field punctuation is considered in a match.

Notes

This ignore punctuation property directs the Centrus Merge/Purge library to either ignore or respect punctuation in the fields being compared. If set to `QTRUE`, all punctuation characters are removed from the string before the comparison is made. A punctuation character is any printing character which is not a space, number, or alphabetic character between ‘a’ and ‘z’ or ‘A’ and ‘Z’. This property is useful when matching string fields containing abbreviations or titles such as “Dr.”, “St.”, “MD.”, or “P.O.”.

See Also

`QmpCritSetIgnorePunct`.

Example

```
QmpDeclHnd( hCritAddress );
QRESULT qres;
QBOOL bIgnorePunct;

bIgnorePunct = QmpCritGetIgnorePunct( hCritAddress, &qres );
```

QmpCritGetIgnoreSpaces

GETS THE “CRITERION IGNORE WHITE SPACE” SETTING.

Syntax

```
QBOOL QmpCritGetIgnoreSpaces ( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns `QTRUE` if criterion ignores white space, `QFALSE` (the default) if the criterion doesn't ignore white space or if there is an error.

Notes

Use this function to get the current “ignore white space” setting for a criterion. If the setting is set to `QTRUE`, white space in fields will be ignored when matching them.

For example, if the “ignore white space” setting is `QTRUE`, “MergePurge”, “Merge Purge”, and “ Merge Purge ” would all be matches.

See Also

`QmpCritSetIgnoreSpaces`

Example

```
bIgnoreWhiteSpace = QmpCritGetIgnoreSpaces  
    (hFirstNameMatchCriterion, &qres );
```

QmpCritGetIgnoreXforms

GETS IGNORE TRANSFORMATIONS PROPERTY.

Syntax

```
QBOOL QmpCritGetIgnoreXforms( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit
Handle to field match criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if specified criterion is configured to ignore field criterion transformations, else QFALSE.

Notes

QmpCritGetIgnoreXforms reports the current property setting. If you are using the name match algorithm, this property should be QTRUE.

See Also

QmpCritSetIgnoreXforms.

Example

```
QBOOL bIgnore;  
hMatchByName = QmpAlgCreate( QMS_ALGORITHM_TYPE_NAME, &qres );  
  
bIgnore = QmpCritGetIgnoreXforms( hNameMatchCriterion, &qres );  
  
QmpCritSetIgnoreXforms( hNameMatchCriterion, QTRUE, &qres );  
bIgnore = QmpCritGetIgnoreXforms( hNameMatchCriterion, &qres );
```

QmpCritGetNameRecProto

GETS THE NAME RECORD PROTOTYPE FOR THE NAME MATCHING ALGORITHM.

Syntax

```
void QmpCritGetNameRecProto( MpHnd in_hCrit, MpHnd io_hRec,
                           QRESULT* out_pResult );
in_hCrit
    Handle to field match criterion. Input.
io_hRec
    Handle to name prototype record. Input, output.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpCritGetNameRecProto gets the current name record prototype defined for the name matching algorithm.

See Also

QmpCritSetNameRecProto.

Example

```
hNameMatchRec = QmpRecCreate( &qres );
hCurrentNameMatchRec = QmpRecCreate( &qres );

iFirstName = QmpRecAddWithWidth( hNameMatchRec, szFirstName,
                                 50, &qres );
iLastName = QmpRecAddWithWidth( hNameMatchRec, szLastName, 50,
                                &qres );

hLNameMatchCriterion = QmpCritCreate( QMS_ALGORITHMS_AVERAGE,
                                      90, &qres);

QmpCritSetNameRecProto( hLNameMatchCriterion, hNameMatchRec,
                        &qres );
QmpCritGetNameRecProto( hLNameMatchCriterion,
                           hCurrentNameMatchRec, &qres );
```

QmpCritGetOneBlankRes

GETS THE CRITERION'S "ONE BLANK" MATCH RESULT SETTING.

Syntax

```
int QmpCritGetOneBlankRes( MpHnd in_hCrit, QRESULT*  
                           out_pResult );  
  
in_hCrit  
        Handle to criterion. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns the "one blank" result setting if successful, or -1 if there is an error. The default setting is 0.

Notes

This function returns the application-specified "one field blank" match result setting for the specified criterion. The default setting is 0 (i.e., when one field in the pair of fields being compared is blank, the pair of fields does NOT match, and a score of "0" is returned). A setting of 0 prevents a blank field from matching a non-blank field.

See Also

[QmpCritSetOneBlankRes](#), [QmpCritSetBothBlankRes](#),
[QmpCritGetBothBlankRes](#).

Example

```
one_blank_match_setting = QmpCritGetOneBlankRes ( hCrit,  
                                                pResult );
```

QmpCritGetRule

GETS THE RULE THE FIELD MATCH CRITERION USES TO COMBINE ALGORITHM SCORES.

Syntax

```
QMS_ALGORITHMS_RULE QmpCritGetRule ( MpHnd in_hCrit, QRESULT*  
    out_pResult );
```

in_hCrit

Handle to criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the rule a criterion uses to evaluate algorithms if successful, or QMS_ALGORITHMS_UNDEFINED if there is an error. Valid enums are:

QMS_ALGORITHMS_AVERAGE	Use the average of all algorithms. Default .
QMS_ALGORITHMS_WEIGHTED_AVERAGE	Use the weighted average of all algorithms.
QMS_ALGORITHMS_MAX	Use the algorithm that provided the highest score.
QMS_ALGORITHMS_MIN	Use the algorithm that provided the lowest score.
QMS_ALGORITHMS_AND	All algorithms must achieve a "threshold" score.
QMS_ALGORITHMS_OR	At least one algorithm must achieve a "threshold score".

Notes

Use this function to see which rule the specified criterion uses to combine algorithm scores.

See Also

[QmpCritSetRule](#).

Example

```
algorithm_rule = QmpCritGetRule (hCrit, pResult );
```

QmpCritGetStrAlign

GETS THE CRITERION'S STRING ALIGNMENT SETTING.

Syntax

```
QMS_ALIGN_OPTION QmpCritGetStrAlign ( MpHnd in_hCrit,
                                     QRESULT* out_pResult );
in_hCrit
Handle to criterion. Input.
out_pResult
Result code. Output.
```

Return Value

Returns the current setting for the string alignment property if successful, or QMS_ALIGN_ASNEEDED if there is an error. Valid enums are:

QMS_ALIGN_ASNEEDED	Align strings if criterion's algorithms require aligned strings. This setting will work <i>only</i> for the keyboard distance algorithm. Default .
QMS_ALIGN_NEVER	Never align strings.
QMS_ALIGN_ALWAYS	Always align strings.

Notes

This function returns the current string alignment setting, which determines how a criterion aligns incoming strings from record pairs. Algorithms that utilize “distance” concepts, such as keyboard distance, benefit from using prealigned strings.

See Also

[QmpCritSetStrAlign](#).

Example

```
alignment_option = QmpCritGetStrAlign (hCrit, pResult );
```

QmpCritGetThreshold

GETS THE THRESHOLD VALUE FOR A CRITERION.

Syntax

```
int QmpCritGetThreshold ( MpHnd in_hCrit, QRESULT* out_pResult
    );
```

in_hCrit

Handle to criterion. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the current threshold value for the specified criterion if successful, or -1 if there is an error. This may be any integer between 0 and 100. The default value is 75.

Notes

Threshold values are only used if a criterion employs the QMS_ALGORITHMS_AND rule (all algorithms must achieve a threshold score) or the QMS_ALGORITHMS_OR rule (at least one algorithm must achieve a threshold score) in its matching rule.

See Also

QmpCritCreate, **QmpCritSetRule**, **QmpCritGetRule**,
QmpCritSetThreshold.

Example

```
criterion_threshold_value = QmpCritGetThreshold (hCrit,
    pResult );
```

QmpCritIsValid

TESTS WHETHER A CRITERION IS VALID.

Syntax

```
QBOOL QmpCritIsValid ( MpHnd in_hCrit, QRESULT* out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the criterion is valid, else QFALSE. If there is an error, QFALSE is returned.

Notes

This function tests whether a criterion is valid. An invalid field match criterion is improperly configured and can not be used in a field comparison.

For example, if a weighted average is being used to combine the criterion algorithm results, but the weights don't sum to 100, this would cause the criterion to fail.

Example

```
is_criteria_valid = QmpCritIsValid (hCrit, pResult );
```

QmpCritRemAlg

REMOVES THE SPECIFIED ALGORITHM FROM THE CRITERION.

Syntax

```
MpHnd QmpCritRemAlg ( MpHnd in_hCrit, MpHnd in_hAlg, QRESULT*  
    out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

in_hAlg
Handle to algorithm. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the handle of the removed algorithm if successful, or NULL if there is an error.

Notes

This function removes the specified algorithm from the criterion.

See Also

[QmpCritAddAlg](#)

Example

```
removed_algorithm = QmpCritRemAlg (hCrit, hAlg, pResult );
```

QmpCritSetAlgWeight

SETS THE WEIGHTING FOR AN ALGORITHM USED IN THE CRITERION.

Syntax

```
void QmpCritSetAlgWeight ( MpHnd in_hCrit, MpHnd in_hAlg, int
                           in_iWeight, QRESULT* out_pResult );
in_hCrit
    Handle to record. Input.
in_hAlg
    Handle to algorithm. Input.
in_iWeight
    Weight to be assigned to algorithm. Must be an integer between 0 and 100.
    The default value is 0. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Criteria with more than one algorithm may have weights assigned to algorithms. The value of each weight must be between 0 and 100, and the total of the weights of all algorithms must equal 100.

See Also

[QmpCritGetWeight](#)

Example

```
/* Set the weights for the algorithms in one criteria */
QmpCritSetAlgWeight (hCrit, hAlg1, 40, pResult );
QmpCritSetAlgWeight (hCrit, hAlg2, 20, pResult );
QmpCritSetAlgWeight (hCrit, hAlg3, 20, pResult );
QmpCritSetAlgWeight (hCrit, hAlg4, 20, pResult );
```

QmpCritSetBothBlankRes

SETS THE CRITERION'S "BOTH BLANK" MATCH RESULT SETTING.

Syntax

```
void QmpCritSetBothBlankRes( MpHnd in_hCrit, int
    in_iBothBlankRes, QRESULT* out_pResult );
in_hCrit
    Handle to criterion. Input.
in_iBothBlankRes
    Blank result, 0 to 100. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function specifies the "both fields blank" match result setting for the specified criterion. The default setting is 0 (i.e., when both fields are blank, they do NOT match, and a score of "0" is returned). A setting of 0 prevents pairs of blank fields from being scored as a match.

The application may use this function to give record matching the "benefit of the doubt" when fields are missing (blank), rather than penalizing the overall match result due to blank fields.

See Also

[QmpCritGetBothBlankRes](#), [QmpCritSetOneBlankRes](#),
[QmpCritGetOneBlankRes](#).

Example

```
QmpCritSetBothBlankRes (hCrit, iBothBlankRes, pResult );
```

QmpCritSetCaseSens

SETS THE CRITERION'S CASE SENSITIVITY.

Syntax

```
void QmpCritSetCaseSens ( MpHnd in_hCrit, QBOOL in_bCaseSens,
                           QRESULT* out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

in_bCaseSens
Turn case sensitivity on (QTRUE) or off (QFALSE). Default is QFALSE. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function determines whether the specified criterion considers case when comparing fields.

See Also

[QmpCritGetCaseSens](#).

Example

```
QmpCritSetCaseSens (hCrit, QTRUE, pResult );
```

QmpCritSetIgnoreAlpha

SETS A CRITERION'S "IGNORE LETTERS" SETTING.

Syntax

```
void QmpCritSetIgnoreAlpha ( MpHnd in_hCrit, QBOOL
    in_bIgnoreAlpha, QRESULT* out_pResult );
in_hCrit
    Handle to field match criterion. Input.
in_bIgnoreAlpha
    "Ignore letters" setting. QTRUE is on, QFALSE (the default) is off. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to set the "ignore letters" setting for a criterion. If the setting is QTRUE, letters in fields will be ignored when matching them.

For example, if the "ignore letters" setting is QTRUE, "200 Main", "200 First", and "200 Second" would all be matches.

See Also

[QmpCritGetIgnoreAlpha](#).

Example

```
QmpCritSetRule( hFirstNameMatchCriterion,
    QMS_ALGORITHMS_AVERAGE, &qres );
QmpCritSetAlgWeight(hFirstNameMatchCriterion, hMatchBySoundex,
    80, &qres );
QmpCritSetThreshold(hFirstNameMatchCriterion, 90, &qres );
QmpCritSetCaseSens(hFirstNameMatchCriterion, QTRUE, &qres );
QmpCritSetIgnoreAlpha(hFirstNameMatchCriterion, QTRUE, &qres
);
QmpCritSetOneBlankRes(hFirstNameMatchCriterion, 50, &qres );
QmpCritSetBothBlankRes(hFirstNameMatchCriterion, 50, &qres );
QmpCritSetStrAlign(hFirstNameMatchCriterion,
    QMS_ALIGN_ASNEEDED, &qres );
```

QmpCritSetIgnoreNum

SETS “IGNORE NUMERIC” CRITERION PROPERTY.

Syntax

```
void QmpCritSetIgnoreNum ( MpHnd in_hCrit, QBOOL  
    in_bIgnoreNum, QRESULT* out_pResult );
```

in_hCrit
Handle to field match criterion. *Input*.

in_bIgnoreNum
“Ignore numeric” property flag. If QTRUE, field numbers are be ignored; if QFALSE (the default), field numbers considered in a match. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The “ignore numeric” property directs the Centrus Merge/Purge library to either ignore or respect numbers in the fields being compared. If set to QTRUE, all numbers are removed from the strings before the comparison is made.

See Also

[QmpCritGetIgnoreNum](#).

Example

```
QmpDeclHnd( hCritAddress );  
QRESULT qres;  
  
QmpCritSetIgnoreNum( hCritAddress, QTRUE, &qres );
```

QmpCritSetIgnorePunct

SETS “IGNORE PUNCTUATION” FIELD MATCH CRITERION PROPERTY.

Syntax

```
void QmpCritSetIgnorePunct ( MpHnd in_hCrit, QBOOL
    in_bIgnorePunct, QRESULT* out_pResult );
```

in_hCrit

Handle to criterion. *Input*.

in_bIgnorePunct

Ignore punctuation property flag. QTRUE directs field punctuation to be ignored; QFALSE directs field punctuation to be considered in a match. Default is QFALSE. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This ignore punctuation property directs the Centrus Merge/Purge library to either ignore or respect punctuation in the fields being compared. If set to QTRUE, all punctuation characters are removed from the string before the comparison is made. A punctuation character is any printing character which is not a space, number, or alphabetic character between ‘a’ and ‘z’ or ‘A’ and ‘Z’. This property is useful when matching string fields containing abbreviations or titles such as “Dr.”, “St.”, “MD.”, or “P.O.”.

See Also

[QmpCritGetIgnorePunct](#).

Example

```
QmpDeclHnd( hCritAddress );
QRESULT qres;

QmpCritSetIgnorePunct( hCritAddress, QTRUE, &qres );
```

QmpCritSetIgnoreSpaces

SETS CRITERION'S "IGNORE WHITE SPACE" SETTING.

Syntax

```
void QmpCritSetIgnoreSpaces ( MpHnd in_hCrit, QBOOL
    in_bIgnoreSpaces, QRESULT* out_pResult );
```

in_hCrit

Handle to criterion. *Input*.

in_bIgnoreSpaces

"Ignore white space" setting. QTRUE is on, QFALSE is off. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Use this function to set the "ignore white space" setting for a criterion. The default setting is QFALSE. If the setting is set to QTRUE, white space in fields will be ignored when matching them.

For example, if the "ignore white space" setting is QTRUE, "MergePurge", "Merge Purge", and " Merge Purge " would all be matches.

See Also

[QmpCritGetIgnoreSpaces](#)

Example

```
QmpCritSetRule( hFirstNameMatchCriterion,
    QMS_ALGORITHMS_AVERAGE, &qres );
QmpCritSetAlgWeight(hFirstNameMatchCriterion, hMatchBySoundex,
    80, &qres );
QmpCritSetThreshold(hFirstNameMatchCriterion, 90, &qres );
QmpCritSetCaseSens(hFirstNameMatchCriterion, QTRUE, &qres );
QmpCritSetIgnoreSpaces(hFirstNameMatchCriterion, QTRUE, &qres
);
QmpCritSetOneBlankRes(hFirstNameMatchCriterion, 50, &qres );
QmpCritSetBothBlankRes(hFirstNameMatchCriterion, 50, &qres );
QmpCritSetStrAlign(hFirstNameMatchCriterion,
    QMS_ALIGN_ASNEEDED, &qres );
```

QmpCritSetIgnoreXforms

SETS IGNORE TRANSFORMATIONS PROPERTY.

Syntax

```
void QmpCritSetIgnoreXforms( MpHnd in_hCrit, QBOOL
    in_bIgnoreTransforms, QRESULT* out_pResult );
in_hCrit
    Handle to field match criterion. Input.
in_bIgnoreTransforms
    Ignore transformations setting. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpCritSetIgnoreXforms is a convenience function that allows you to enable or disable all the field match transformations that have been previously set.

Field match transformations are incompatible with the name matching algorithm. For instance, if the field match criteria property “ignorewhitespace” is used with the name matching algorithm, comparing “Bob Smith” and “Robert Smith” will result in comparing “BobSmith” and “RobertSmith”. This comparison returns a negative match (value 0).

See Also

QmpCritGetIgnoreXforms.

Example

```
QBOOL bIgnore;

hMatchByName = QmpAlgCreate( QMS_ALGORITHM_TYPE_NAME, &qres );
bIgnore = QmpCritGetIgnoreXforms( hNameMatchCriterion, &qres
);
QmpCritSetIgnoreXforms( hNameMatchCriterion, QTRUE, &qres );
bIgnore = QmpCritGetIgnoreXforms( hNameMatchCriterion, &qres
);
```

QmpCritSetNameRecProto

ATTACHES A RECORD PROTOTYPE CONTAINING THE NAME FIELDS TO THE MATCH CRITERION.

Syntax

```
void QmpCritSetNameRecProto( MpHnd in_hCrit, MpHnd in_hRec,
                             QRESULT* out_pResult );
in_hCrit
    Handle to field match criterion. Input.
in_hRec
    Handle to name prototype record. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

A field match criterion using the name matching algorithm requires a special prototype record (the name matching prototype record) to operate. This record, which is created by the application, contains the name fields which are needed by the name matching algorithm (such as the first, last, and middle name fields).

The `in_NameOrder` parameter in `QmpCritAddNameMatchAlg` specifies the order that the algorithm expects the name fields to appear in the name prototype record. For example, if `in_NameOrder = QMS_NAME_ORDER_LASTFIRST`, then the name prototype record must contain the last name field first, followed by the first name field.

If you are matching single-field names (both the first and last names are contained in a single field), include the single field in the name prototype record and set the `in_NameOrder` parameter appropriately. For example, if you have a single name match field containing a string of the format “LastName FirstName”, include the field in the name prototype record and set `in_NameOrder` to `QMS_NAME_ORDER_LASTFIRST`.

See Also

`QmpCritGetNameRecProto`.

Example

```
hNameMatchRec = QmpRecCreate( &qres );

iFirstName = QmpRecAddWithWidth( hNameMatchRec, szFirstName,
                                50, &qres );
iLastName = QmpRecAddWithWidth( hNameMatchRec, szLastName, 50,
                                &qres );

hLNameMatchCriterion = QmpCritCreate( QMS_ALGORITHMS_AVERAGE,
                                      90, &qres);

QmpCritSetNameRecProto( hLNameMatchCriterion, hNameMatchRec,
                            &qres );
```

QmpCritSetOneBlankRes

SETS THE CRITERION'S "ONE BLANK" MATCH RESULT SETTING.

Syntax

```
void QmpCritSetOneBlankRes( MpHnd in_hCrit, int  
                           in_iOneBlankRes, QRESULT* out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

in_iOneBlankRes
Blank result, 0 to 100. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the "one field blank" match result setting for the specified criterion. The default setting is 0 (i.e., when one field in a pair of fields is blank, the pair does NOT match, and a score of "0" is returned). A setting of 0 prevents a blank field from "matching" a non-blank field. The application may change this setting to different levels so that scores between 0 and 100 are returned when one of the two fields are blank.

See Also

[QmpCritGetOneBlankRes](#), [QmpCritSetBothBlankRes](#),
[QmpCritGetBothBlankRes](#).

Example

```
QmpCritSetOneBlankRes (hCrit, 100, pResult );
```

QmpCritSetPairFullFld

ESTABLISHES THE PAIR OF RECORD FIELDS TO BE COMPARED BY THE CRITERION (NO SUBSTRINGING).

Syntax

```
void QmpCritSetPairFullFld ( MpHnd in_hCrit, MpHnd in_hRec,
                           const char* in_szFldName1, const char* in_szFldName2,
                           QRESULT* out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

in_hRec
Handle to record. *Input*.

in_szFldName1
Name of the first field. *Input*.

in_szFldName2
Name of the second field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function specifies two fields to compare. The full fields are compared, with no substringing. If your application needs to perform a comparison using a substring of the fields, use **QmpCritSetPairPartFld** or **QmpCritSetPairSubStrFld**.

See Also

QmpCritSetPairPartFld, **QmpCritSetPairSubStrFld**.

Example

```
/* Set up a field match criterion for matching on */
/* last name */
QmpCritAddAlg( hCritLName, hAlgEditDist, 0, &qres );
QmpCritAddAlg( hCritLName, hAlgSoundex, 0, &qres );
QmpCritSetPairFullFld( hCritLName, hRec, "LastName",
                         "LastName", &qres );
```

QmpCritSetPairPartFld

ESTABLISHES THE PAIR OF RECORD FIELDS TO BE COMPARED BY THE CRITERION (PARTIAL SUBSTRINGING).

Syntax

```
void QmpCritSetPairPartFld ( MpHnd in_hCrit, MpHnd in_hRec,
    const char* in_szFldName1, int in_iCharStartPos1, const
    char* in_szFldName2, int in_iCharStartPos2, QRESULT*  

    out_pResult );
```

in_hCrit
Handle to criterion. *Input*.

in_hRec
Handle to record. *Input*.

in_szFldName1
Name of the first field. *Input*.

in_iCharStartPos1
Character start position of the first field. Must be 1 or greater. *Input*.

in_szFldName2
Name of the second field. *Input*.

in_iCharStartPos2
Character start position of the second field. Must be 1 or greater. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is used to specify fields to compare using partial substringing. You specify the fields to be compared and the character position at which to start the comparison. Comparison continues to the end of the string. If you need to specify an internal substring that begins after the first letter and ends before the last character, use the function **QmpCritSetPairSubStrFld**.

The starting character of a field is considered to have an index of 1. For example, in the following field, character 1 is "J":

J O N E S

See Also

QmpCritSetPairFullFld, **QmpCritSetPairSubStrFld**.

Example

```
QmpCritSetPairPartFld (hCrit, hRec, "Streetaddress", 5,
    "Fulladdress", 7, pResult );
```

QmpCritSetPairSubStrFld

ESTABLISHES THE PAIR OF RECORD FIELDS TO BE COMPARED BY THE CRITERION (FULL SUBSTRINGING).

Syntax

```
void QmpCritSetPairSubStrFld ( MpHnd in_hCrit, MpHnd in_hRec,
    const char* in_szFldName1, int in_iCharStartPos1, int
    in_iNumChars1, const char* in_szFldName2, int
    in_iCharStartPos2, int in_iNumChars2, QRESULT* out_pResult
);

in_hCrit
    Handle to criterion. Input.
in_hRec
    Handle to record. Input.
in_szFldName1
    Name of the first field. Input.
in_iCharStartPos1
    Character start position of the first field. Must be 1 or greater. Input.
in_iNumChars1
    Number of characters in first field. Must be 1 or above. Input.
in_szFldName2
    Name of the second field. Input.
in_iCharStartPos2
    Character start position of the second field. Must be 1 or greater. Input.
in_iNumChars2
    Number of characters in second field. Must be 1 or above. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function is used to specify fields to compare using full substringing. You specify the fields to be compared, the starting character position, and how many characters to use in the comparison.

The starting character of a field is considered to have an index of 1. The number of characters from the starting position includes the starting character. For example, in the following field:

J O N E S

- Character 1 is “J”
- The values *in_iCharStartPos* = 1 and *in_iNumChars* = 1 define the first character “J”
- The values *in_iCharStartPos* = 1 and *in_iNumChars* = 3 define the string “JON”

The values *in_iCharStartPos* = 1 and *in_iNumChars* = 1000 define the string “JONES” (An *in_iNumChars* value greater than the length of the field defines the entire field)

See Also

QmpCritSetPairPartFld, **QmpCritSetPairFullFld**.

Example

```
/* setup the match criterion for matching on the first */
/* 10 characters of the Address */
QmpCritAddAlg( hAddressCriterion, hMatchByEditDist, 0, pResult
);
QmpCritSetPairSubStrFld( hAddressCriterion, hSourceRecord,
    "AddrLine1", 1, 10, "AddrLine1", 1, 10, pResult );
QmpCritSetRule( hAddressCriterion, QMS_ALGORITHMS_AVERAGE,
    pResult );
QmpCritSetStrAlign( hAddressCriterion, QMS_ALIGN_NEVER, pResult
);
```

QmpCritSetRule

SETS THE RULE THE CRITERION USES TO COMBINE ALGORITHM SCORES.

Syntax

```
void QmpCritSetRule ( MpHnd in_hCrit, QMS_ALGORITHMS_RULE
    in_Rule, QRESULT* out_pResult );
```

in_hCrit

Handle to criterion. *Input*.

in_Rule

Algorithm rule. *Input*.

Valid enums are:

QMS_ALGORITHMS_AVERAGE	Use the average of all algorithms. Default .
QMS_ALGORITHMS_WEIGHTED_AVERAGE	Use the weighted average of all algorithms.
QMS_ALGORITHMS_MAX	Use the algorithm that provided the highest score.
QMS_ALGORITHMS_MIN	Use the algorithm that provided the lowest score.
QMS_ALGORITHMS_AND	All algorithms must achieve a "threshold" score.
QMS_ALGORITHMS_OR	At least one algorithm must achieve a "threshold score".

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets the rule the specified criterion uses to combine algorithm scores.

See Also

[QmpCritGetRule](#)

Example

```
/* setup the match criterion for matching on FirstName */
QmpCritAddAlg( hFNameMatchCriterion, hMatchByEditDist, 0,
    pResult );
QmpCritAddAlg( hFNameMatchCriterion, hMatchBySoundex, 0,
    pResult );
QmpCritSetPairFullFld( hFNameMatchCriterion, hSourceRecord,
    "FirstName", "FirstName", pResult );
QmpCritSetRule( hFNameMatchCriterion, QMS_ALGORITHMS_MAX,
    pResult );
```

QmpCritSetRule

```
QmpCritSetStrAlign( hFNameMatchCriterion, QMS_ALIGN_NEVER,  
                     pResult );
```

QmpCritSetStrAlign

SETS THE STRING ALIGNMENT PROPERTY FOR THE CRITERION.

Syntax

```
void QmpCritSetStrAlign ( MpHnd in_hCrit, QMS_ALIGN_OPTION  
    in_StrAlign, QRESULT* out_pResult );
```

in_hCrit

Handle to criterion. *Input*.

in_StrAlign

String alignment. *Input*.

Valid enums are:

<code>QMS_ALIGN_ASNEEDED</code>	Align strings if criterion's algorithms require aligned strings. This setting will work <i>only</i> for the keyboard distance algorithm. Default .
---------------------------------	---

<code>QMS_ALIGN_NEVER</code>	Never align strings.
------------------------------	----------------------

<code>QMS_ALIGN_ALWAYS</code>	Always align strings.
-------------------------------	-----------------------

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function defines the string alignment setting for the specified criterion. The string alignment setting determines how the criterion aligns incoming strings from two records. Algorithms that utilize “distance” concepts, such as keyboard distance, benefit from using prealigned strings.

See Also

[QmpCritGetStrAlign](#)

Example

```
/* setup the match criterion for matching on FirstName */  
QmpCritAddAlg( hFNameMatchCriterion, hMatchByEditDist, 0,  
    pResult );  
QmpCritAddAlg( hFNameMatchCriterion, hMatchBySoundex, 0,  
    pResult );  
QmpCritSetPairFullFld( hFNameMatchCriterion, hSourceRecord,  
    "FirstName", "FirstName", pResult );  
QmpCritSetRule( hFNameMatchCriterion, QMS_ALGORITHMS_MAX,  
    pResult );  
QmpCritSetStrAlign( hFNameMatchCriterion, QMS_ALIGN_NEVER,  
    pResult );
```

QmpCritSetThreshold

SETS THE THRESHOLD FOR THE CRITERION.

Syntax

```
void QmpCritSetThreshold ( MpHnd in_hCrit, int in_iThreshold,  
    QRESULT* out_pResult );
```

in_hCrit

Handle to criterion. *Input*.

in_iThreshold

Sets threshold value for specified criterion. This may be any integer between 0 and 100; the default value is 75. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Threshold values are only used if a criterion employs the QMS_ALGORITHMS_AND rule (all algorithms must achieve a threshold score) or the QMS_ALGORITHMS_OR rule (at least one algorithm must achieve a threshold score) in its matching rule.

See Also

[QmpCritCreate](#), [QmpCritSetRule](#), [QmpCritGetRule](#).

Example

```
QmpCritSetThreshold (hCrit, 100, pResult );
```

Function Class: QmpDataDest*

DATA DESTINATION FUNCTIONS.

Quick Reference

Function	Description	Page
QmpDataDestAddDataLstFilter	Adds a data list ID filter.	228
QmpDataDestCreate	Creates a data destination object.	229
QmpDataDestDestroy	Destroys a data destination object.	230
QmpDataDestFillExpr	Fills expression string from specified data destination.	231
QmpDataDestFillStatSamp	Fills a structure with the current statistical sampling properties of a data destination.	232
QmpDataDestGetAugmentedFieldsFlag	Gets the data destination augmented fields flag.	234
QmpDataDestGetExprNumChar	Returns the field evaluation substring character count.	235
QmpDataDestGetExprStart	Returns the field evaluation substring start position.	236
QmpDataDestGetInclCons	Gets the output inclusion consolidated records flag.	237
QmpDataDestGetInclMasters	Gets the output inclusion master duplicates flag.	238
QmpDataDestGetInclSubords	Gets the output inclusion subordinate duplicates flag.	239
QmpDataDestGetInclUniques	Gets the output inclusion uniques flag.	240
QmpDataDestGetNthRecInterval	Gets data destination Nth record event interval.	241
QmpDataDestGetOutputTbl	Gets the output table object associated with the data destination.	242
QmpDataDestGetReplaceFlag	Gets the data destination replace flag.	243
QmpDataDestGetSuppCons	Gets the output suppression consolidated records flag.	244
QmpDataDestGetSuppMasters	Gets the output suppression master duplicates flag.	245
QmpDataDestGetSuppSubords	Gets the output suppression subordinate duplicates flag.	246
QmpDataDestGetSuppUniques	Gets the data destination output suppression uniques flag.	247
QmpDataDestRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	248

Function	Description	Page
QmpDataDestSetAugmentedFieldsFlag	Sets the data destination augmented fields flag.	249
QmpDataDestSetAugOut	Sets the output augmented fields property for a data destination.	250
QmpDataDestSetExprDate	Sets date-type expression in specified data destination.	251
QmpDataDestSetExprDateStruct	Sets a QDATESTRUCT-type expression in specified data destination.	252
QmpDataDestSetExprFloat	Sets float-type expression in specified data destination.	253
QmpDataDestSetExprLong	Sets long-type expression in specified data destination.	254
QmpDataDestSetExprString	Sets string-type expression in specified data destination.	255
QmpDataDestSetExprSubStr	Sets expression substring for a data destination.	256
QmpDataDestSetExprVar	Sets expression in specified data destination to a variant object value.	258
QmpDataDestSetExprVarStruct	Sets expression in specified data destination to a QVARSTRUCT value.	260
QmpDataDestSetInclCons	Sets the output inclusion consolidated records flag.	261
QmpDataDestSetInclMasters	Sets the output inclusion master duplicates flag.	262
QmpDataDestSetInclSubords	Sets the output inclusion subordinate duplicates flag.	263
QmpDataDestSetInclUniques	Sets the output inclusion uniques flag.	264
QmpDataDestSetNthRecInterval	Sets data destination Nth record event interval.	265
QmpDataDestSetOutputTypes	Sets the output type flags from a string.	266
QmpDataDestSetOutputTypesWithBitmap	Sets the output type flags from a mask.	267
QmpDataDestSetOutRec	Sets the output prototype record.	268
QmpDataDestSetRecProto	Sets the prototype record via an application-owned record.	269
QmpDataDestSetReplaceFlag	Sets the data destination replace flag.	270
QmpDataDestSetStatSamp	Sets the statistical sampling properties of a data destination.	271
QmpDataDestSetSuppCons	Sets the output suppression consolidated records flag.	273
QmpDataDestSetSuppMasters	Sets the output suppression master duplicates flag.	274

Function	Description	Page
QmpDataDestSetSuppSubords	Sets the output suppression subordinate duplicates flag.	275
QmpDataDestSetSuppUniques	Sets the data destination output suppression uniques flag.	276
QmpDataDestUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	277

QmpDataDestAddDataLstFilter

ADDS A DATA LIST ID FILTER.

Syntax

```
void QmpDataDestAddDataLstFilter( MpHnd in_hDataDest, long  
                                in_lDataLstId, QRESULT* out_pResult );
```

in_DataDest
Handle to data destination object. *Input*.

in_lDataLstId
Data list ID. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

A data list filter allows the user to filter the records in the output stream by data list. For example, you might have a data source in which you've assigned certain records (such as records with an "income" value greater than 70,000) to a data list. By setting a data list filter on your data source, only records within the data list (i.e., records with incomes greater than 70,000) will be included in the output. If other "filters" are defined, these also operate on the record, determining whether the record is written to the output file. If no data list filter is defined, all data lists are included in the output, provided they pass all other output filtering.

This function adds the specified data list to the array of data lists to be included in the specified data destination. Multiple calls to this function populate the array.

Example

```
QmpDataDestAddDataLstFilter (hDataDestSuppUniques, 1, pResult  
);
```

QmpDataDestCreate

CREATES A DATA DESTINATION OBJECT.

Syntax

```
MpHnd QmpDataDestCreate( MpHnd in_hOutputTable, QRESULT*
    out_pResult );
in_hOutputTable
    Handle to output table. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns handle to data destination if successful, or `NULL` if there is an error.

Notes

This function creates a data destination for use in outputting selected records. This function returns the handle used in all subsequent calls to `QmpDataDest` functions.

An application must create a data destination object for each output table to be created. Once the data destination object is created, it may be configured to define the types of records the output table will receive (for example, uniques, duplicates, or subordinate duplicates).

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

`QmpDataDestAddDataLstFilter`, `QmpDataDestSetOutputTypes`,
`QmpDataDestSetOutputTypesWithBitmask`,
`QmpDataDestSetReplaceFlag`, `QmpDataDestSetInclUniques`,
`QmpDataDestSetSuppUniques`, `QmpDataDestSetInclMasters`,
`QmpDataDestSetSuppMasters`, `QmpDataDestSetInclSubords`,
`QmpDataDestSetSuppSubords`.

Example

```
hDataDestInclUniques = QmpDataDestCreate ( hUniquesTable,
    pResult );
```

QmpDataDestDestroy

DESTROYS A DATA DESTINATION OBJECT.

Syntax

```
void QmpDataDestDestroy( MpHnd in_DataDest, QRESULT*
    out_pResult );
```

in_DataDest
Handle to data destination. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This destroys the data destination identified by the *in_DataDest* handle.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

Example

```
QmpTblGenDestroy( hTableGen, &qres );
QmpDataDestDestroy( hDataDestUniques, &qres );
QmpDataDestDestroy( hDataDestMasters, &qres );
QmpDataDestDestroy( hDataDestSubords, &qres );
QmpDataDestDestroy( hDataDestSupp, &qres );
```

QmpDataDestFillExpr

FILLS EXPRESSION STRING FROM SPECIFIED DATA DESTINATION.

Syntax

```
void QmpDataDestFillExpr( MpHnd in_hDataDest, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
io_szBuffer
    Expression buffer. Input, Output.
in_lSize
    Size of client-allocated expression buffer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a field value against a set value. If the expression is evaluated to be true, the record will be output to the data destination.

QmpDataDestFillExpr puts a data destination's filtering expression into an application-allocated buffer. The expression is constructed from a record field name, a comparison operator of type **QMS_FLDEVAL_OPER**, and a string constant to compare the field value against.

See Also

QmpDataDestSetExprString

Example

```
QmpDataDestFillExpr (hDataDest, szBuffer, lBufferSize, pResult
);
```

QmpDataDestFillStatSamp

FILLS A STRUCTURE WITH THE CURRENT STATISTICAL SAMPLING PROPERTIES OF A DATA DESTINATION.

Syntax

```
void QmpDataDestFillStatSamp ( MpHnd in_hDataDest,
    QMS_OUTPUT_SAMPLING* out_StatSampling, QRESULT* out_pResult
);
```

in_hDataDest

Handle to data destination. *Input*.

out_StatSampling

Statistical sampling structure. *Output*.

Structure members are:

lMaxRecords	Maximum number of records to include.
lFirstRecordNum	Record number of first record to include.
lLastRecordNum	Record number of last record to include.
lInterval	Sampling interval (return every “nth” record).
lGroupSize	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record).
lMaxDupeGroups	Maximum number of dupe groups to include.
lDupesFirst	First dupe group to include.
lDupesLast	Last dupe group to include.
lDupesInterval	Dupe group sampling interval.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpDataDestFillStatSamp fills a structure of type **QMS_OUTPUT_SAMPLING** with the statistical sampling properties currently set for a data destination.

Statistical sampling allows you to quickly filter output records in a controlled way. By including a small subset of records in a report or output table, you can evaluate the output before committing to a long processing run.

Statistical sampling for data destinations is optional. Call **QmpDataDestSetStatSamp** to turn statistical sampling on for a data destination and to set the sampling properties. If statistical sampling is used for a data destination, a statistical sampling section will appear in the job summary report.

A data destination will receive up to the maximum number of records set by statistical sampling. This limit might be set by the `lMaxRecords` property or by the difference between the `lFirstRecordNum` and `lLastRecordNum` values. If a data destination is configured to receive 1,000 records from a 1,000,000 record data source, after 1,000 records are written the job will begin processing the next data destination.

Example

```

OutputSampling.lMaxRecords      = 2000;
OutputSampling.lFirstRecordNum  = 0;
OutputSampling.lLastRecordNum   = 0;
OutputSampling.lInterval        = 0;
OutputSampling.lGroupSize       = 0;
OutputSampling.lMaxDupeGroups  = 0;
OutputSampling.lDuplicatesFirst = 0;
OutputSampling.lDuplicatesLast = 0;
OutputSampling.lDuplicatesInterval = 0;
QmpDataDestSetStatSamp( hDataDestSupp, &OutputSampling, &qres );
QmpDataDestFillStatSamp( hDataDestSupp, &TestSampling, &qres );
if ( TestSampling.lMaxRecords != OutputSampling.lMaxRecords ||
     TestSampling.lFirstRecordNum != OutputSampling.lFirstRecordNum ||
     TestSampling.lLastRecordNum != OutputSampling.lLastRecordNum ||
     TestSampling.lInterval != OutputSampling.lInterval )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Output sampling is not correct"
);

```

QmpDataDestGetAugmentedFieldsFlag

GETS THE DATA DESTINATION AUGMENTED FIELDS FLAG.

Syntax

```
QBOOL QmpDataDestGetAugmentedFieldsFlag ( MpHnd in_hDataDest,
                                         QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
out_pResult
    Result code.
```

Return Value

If QTRUE, records are output to the table with their augmented fields. If QFALSE, the records are output without them. The default value is QTRUE.

Notes

QmpDataDestGetAugmentedFieldsFlag gets the flag that determines whether records are written to a data destination with their augmented fields.

See “[Augmented Fields](#)” on page 39 for a description of these fields.

Some of the augmented fields will be meaningless in certain cases (the dupe group fields in a unique record, for instance). If the flag is set to QTRUE, all augmented fields are written to the data destination, even fields with meaningless values.

See Also

[**QmpDataDestSetAugmentedFieldsFlag**](#).

Example

```
QmpDeclHnd( hDataDest );
QBOOL bAugmentedFields;
QRESULT qres;

bAugmentedFields = QmpDataDestGetAugmentedFieldsFlag( hDataDest, &qres );
```

QmpDataDestGetExprNumChar

RETURNS THE FIELD EVALUATION SUBSTRING CHARACTER COUNT.

Syntax

```
int QmpDataDestGetExprNumChar ( MpHnd in_hDataDest, QRESULT*  
                               out_pResult );  
  
in_hDataDest  
    Handle to data destination. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns number of characters in substring.

Notes

QmpDataDestGetExprNumChar returns the length of the field substring defined by **QmpDataDestSetExprSubStr**.

See Also

QmpDataDestSetExprSubStr.

Example

```
/* return the field evaluation substring start position */  
  
/* return the object's starting position */  
iFldExprStart = QmpDataDestGetExprStart( hDataDest, &qres );  
if ( iFldExprStart != 1 )  
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );  
  
/* return the object's number of characters */  
iFldExprNumChars = QmpDataDestGetExprNumChar( hDataDest, &qres );  
if ( iFldExprNumChars != 0 )  
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );  
  
QmpDataDestSetExprSubStr( hDataDest, 2, 4, &qres );  
  
/* return the object's starting position */  
iFldExprStart = QmpDataDestGetExprStart( hDataDest, &qres );  
if ( iFldExprStart != 2 )  
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );  
  
/* return the object's number of characters */  
iFldExprNumChars = QmpDataDestGetExprNumChar( hDataDest, &qres );  
if ( iFldExprNumChars != 4 )  
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataDestGetExprStart

RETURNS THE FIELD EVALUATION SUBSTRING START POSITION.

Syntax

```
int QmpDataDestGetExprStart ( MpHnd in_hDataDest, QRESULT*
    out_pResult );
in_hDataDest
    Handle to data destination. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the substring starting position.

Notes

QmpDataDestGetExprStart returns the starting position of the field substring defined by **QmpDataDestSetExprSubstr**.

See Also

QmpDataDestSetExprSubStr.

Example

```
/* return the field evaluation substring start position */
/* return the object's starting position */
iFldExprStart = QmpDataDestGetExprStart( hDataDest, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataDestGetExprNumChar( hDataDest, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataDestSetExprSubStr( hDataDest, 2, 4, &qres );

/* return the object's starting position */
iFldExprStart = QmpDataDestGetExprStart( hDataDest, &qres );
if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataDestGetExprNumChar( hDataDest, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataDestGetInclCons

GETS THE OUTPUT INCLUSION CONSOLIDATED RECORDS FLAG.

Syntax

```
QBOOL QmpDataDestGetInclCons( MpHnd in_hDataDest, QRESULT*  
    out_pResult );
```

in_hDataDest
Handle to data destination. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if specified data destination is configured to output consolidated records belonging to an inclusion list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output consolidated records belonging to an inclusion list.

Though a consolidated record is created with a data list ID of 0 (see “[Function Class: QmpConsRpt*](#)” on page 163), during table generation the record is analyzed and placed in the proper data list. Note that it will never belong to a data source list (because it has a data source ID of 0).

Example

See example on [page 261](#).

QmpDataDestGetInclMasters

GETS THE OUTPUT INCLUSION MASTER DUPLICATES FLAG.

Syntax

```
QBOOL QmpDataDestGetInclMasters( MpHnd in_hDataDest, QRESULT*  
                               out_pResult );
```

in_hDataDest
Handle to data destination. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if specified data destination is configured to output master duplicate records belonging to an inclusion list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output master duplicate records belonging to an inclusion list.

Example

```
/* gets the output inclusion uniques flag */  
bInclUniques = QmpDataDestGetInclUniques( hDataDestUniques,  
                                         &qres );  
  
/* gets the output suppression uniques flag */  
bSuppUniques = QmpDataDestGetSuppUniques( hDataDestUniques,  
                                         &qres );  
  
/* gets the output inclusion master duplicates flag */  
bInclMasters = QmpDataDestGetInclMasters( hDataDestUniques,  
                                         &qres );  
  
/* gets the output suppression master duplicates flag */  
bSuppMasters = QmpDataDestGetSuppMasters( hDataDestUniques,  
                                         &qres );  
  
/* gets output inclusion subordinate duplicates flag */  
bInclSubords = QmpDataDestGetInclSubords( hDataDestUniques,  
                                         &qres );
```

QmpDataDestGetInclSubords

GETS THE OUTPUT INCLUSION SUBORDINATE DUPLICATES FLAG.

Syntax

```
QBOOL QmpDataDestGetInclSubords( MpHnd in_DataDest, QRESULT*  
    out_pResult );
```

in_DataDest
Handle to data destination. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns **QTRUE** if specified data destination is configured to output subordinate duplicate records belonging to an inclusion list, else **QFALSE**.

Notes

Use this function to check whether the specified data destination is configured to output subordinate duplicate records belonging to an inclusion list.

See Also

QmpDataDestSetInclSubords

Example

```
/* gets the output inclusion uniques flag */
bInclUniques = QmpDataDestGetInclUniques( hDataDestUniques,
    &qres );

/* gets the output suppression uniques flag */
bSuppUniques = QmpDataDestGetSuppUniques( hDataDestUniques,
    &qres );

/* gets the output inclusion master duplicates flag */
bInclMasters = QmpDataDestGetInclMasters( hDataDestUniques,
    &qres );

/* gets the output suppression master duplicates flag */
bSuppMasters = QmpDataDestGetSuppMasters( hDataDestUniques,
    &qres );

/* gets output inclusion subordinate duplicates flag */
bInclSubords = QmpDataDestGetInclSubords( hDataDestUniques,
    &qres );
```

QmpDataDestGetInclUniques

GETS THE OUTPUT INCLUSION UNIQUES FLAG.

Syntax

```
QBOOL QmpDataDestGetInclUniques( MpHnd in_DataDest, QRESULT*
                                out_pResult );
in_DataDest
    Handle to data destination. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns QTRUE if specified data destination is configured to output unique records belonging to an inclusion list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output unique records belonging to an inclusion list.

See Also

[QmpDataDestSetInclUniques](#)

Example

```
/* gets the output inclusion uniques flag */
bInclUniques = QmpDataDestGetInclUniques( hDataDestUniques,
                                           &qres );

/* gets the output suppression uniques flag */
bSuppUniques = QmpDataDestGetSuppUniques( hDataDestUniques,
                                            &qres );

/* gets the output inclusion master duplicates flag */
bInclMasters = QmpDataDestGetInclMasters( hDataDestUniques,
                                            &qres );

/* gets the output suppression master duplicates flag */
bSuppMasters = QmpDataDestGetSuppMasters( hDataDestUniques,
                                            &qres );

/* gets output inclusion subordinate duplicates flag */
bInclSubords = QmpDataDestGetInclSubords( hDataDestUniques,
                                            &qres );
```

QmpDataDestGetNthRecInterval

GETS DATA DESTINATION NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpDataDestGetNthRecInterval ( MpHnd in_hDataDest,
                                    QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns data destination Nth record event interval if successful, or -1 if there is an error.

Notes

This function gets the data destination Nth record event interval.

The data destination can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the data destination processes 100 records, it will send out a notification to *all* registered clients.

See Also

QmpDataDestRegEveryNthRecFunc, **QmpDataDestSetNthRecInterval**,
QmpDataDestUnregEveryNthRecFunc

Example

```
lNthRecordInterval = QmpDataDestGetNthRecInterval (hDataDest,
                                                 pResult );
```

QmpDataDestGetOutputTbl

GETS THE OUTPUT TABLE OBJECT ASSOCIATED WITH THE DATA DESTINATION.

Syntax

```
MpHnd QmpDataDestGetOutputTbl( MpHnd in_hDataDest, QRESULT*  
          out_pResult );
```

in_hDataDest

Handle to data destination object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to output table associated with the data destination if successful, or NULL if there is an error.

Notes

Gets the output table object associated with the specified data destination.

Example

```
hOutputTbl = QmpDataDestGetOutputTbl(hDataDest, pResult);
```

QmpDataDestGetReplaceFlag

GETS THE DATA DESTINATION REPLACE FLAG.

Syntax

```
QBOOL QmpDataDestGetReplaceFlag( MpHnd in_hDataDest, QRESULT*
```

```
    out_pResult );
```

in_hDataDest

Handle to data destination. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns `QTRUE` if specified data destination is configured to overwrite any existing table, `QFALSE` if data is to be appended to an existing table. `QFALSE` is also returned if an error occurs.

Notes

Gets the data destination replace flag. This flag determines whether a data destination is configured to overwrite any existing table, or append data to the table.

See Also

[QmpDataDestSetReplaceFlag](#)

Example

```
hTblDestSupp = QmpTblCbCreate( szDatabaseName,
                               pszTempPathBuffer, &qres );
hDataDestSupp = QmpDataDestCreate( hTblDestSupp, &qres );
QmpDataDestSetOutputTypesWithBitmask( hDataDestSupp,
                                      OUTPUT_SUPP_UNIQUES|OUTPUT_SUPP_MASTERS|OUTPUT_SUPP_SUBORDS,
                                      &qres );
bReplaceFlag = QmpDataDestGetReplaceFlag( hDataDestSupp,
                                           &qres );
QmpDataDestSetReplaceFlag( hDataDestSupp, QTRUE, &qres );
bReplaceFlag = QmpDataDestGetReplaceFlag( hDataDestSupp,
                                           &qres );
ulMaxRecords = QmpDataDestGetMaxRec( hDataDestSupp, &qres );
```

QmpDataDestGetSuppCons

GETS THE OUTPUT SUPPRESSION CONSOLIDATED RECORDS FLAG.

Syntax

```
QBOOL QmpDataDestGetSuppCons( MpHnd in_hDataDest, QRESULT*  
    out_pResult );
```

in_hDataDest

Handle to data destination object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if specified data destination is configured to output consolidated records belonging to a suppression list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output consolidated records belonging to a suppression list.

Example

See example on [page 261](#).

QmpDataDestGetSuppMasters

GETS THE OUTPUT SUPPRESSION MASTER DUPLICATES FLAG.

Syntax

```
QBOOL QmpDataDestGetSuppMasters( MpHnd in_hDataDest, QRESULT*  
                                out_pResult );
```

in_hDataDest

Handle to data destination object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if specified data destination is configured to output master duplicate records belonging to a suppression list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output master duplicate records belonging to a suppression list.

See Also

[QmpDataDestSetSuppMasters](#)

Example

```
/* gets the output inclusion uniques flag */  
bInclUniques = QmpDataDestGetInclUniques( hDataDestUniques,  
                                         &qres );  
  
/* gets the output suppression uniques flag */  
bSuppUniques = QmpDataDestGetSuppUniques( hDataDestUniques,  
                                         &qres );  
  
/* gets the output inclusion master duplicates flag */  
bInclMasters = QmpDataDestGetInclMasters( hDataDestUniques,  
                                         &qres );  
  
/* gets the output suppression master duplicates flag */  
bSuppMasters = QmpDataDestGetSuppMasters( hDataDestUniques,  
                                         &qres );  
  
/* gets output inclusion subordinate duplicates flag */  
bInclSubords = QmpDataDestGetInclSubords( hDataDestUniques,  
                                         &qres );
```

QmpDataDestGetSuppSubords

GETS THE OUTPUT SUPPRESSION SUBORDINATE DUPLICATES FLAG.

Syntax

```
QBOOL QmpDataDestGetSuppSubords( MpHnd in_hDataDest, QRESULT*
                                out_pResult );
in_hDataDest
    Handle to data destination object. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns QTRUE if specified data destination is configured to output subordinate duplicate records belonging to a suppression data list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output subordinate duplicate records belonging to a suppression data list.

See Also

[QmpDataDestSetSuppSubords](#)

Example

```
/* gets the output inclusion uniques flag */
bInclUniques = QmpDataDestGetInclUniques( hDataDestUniques,
                                           &qres );

/* gets the output suppression uniques flag */
bSuppUniques = QmpDataDestGetSuppUniques( hDataDestUniques,
                                            &qres );

/* gets the output inclusion master duplicates flag */
bInclMasters = QmpDataDestGetInclMasters( hDataDestUniques,
                                            &qres );

/* gets the output suppression master duplicates flag */
bSuppMasters = QmpDataDestGetSuppMasters( hDataDestUniques,
                                            &qres );

/* gets the output inclusion subordinate duplicates */
/* flag */
bInclSubords = QmpDataDestGetInclSubords( hDataDestUniques,
                                            &qres );

/* gets the output suppression subordinate duplicates flag */
bSuppSubords = QmpDataDestGetSuppSubords( hDataDestUniques,
                                              &qres );
```

QmpDataDestGetSuppUniques

GETS THE DATA DESTINATION OUTPUT SUPPRESSION UNIQUES FLAG.

Syntax

```
QBOOL QmpDataDestGetSuppUniques( MpHnd in_hDataDest, QRESULT*  
    out_pResult  
    in_hDataDest  
        Handle to data destination object. Input.  
    out_pResult  
        Result code. Output.
```

Return Value

Returns QTRUE if specified data destination is configured to output unique records belonging to a suppression data list, else QFALSE.

Notes

Use this function to check whether the specified data destination is configured to output unique records belonging to a suppression data list.

See Also

[QmpDataDestSetSuppUniques](#)

Example

```
/* gets the output inclusion uniques flag */  
bInclUniques = QmpDataDestGetInclUniques( hDataDestUniques,  
    &qres );  
  
/* gets the output suppression uniques flag */  
bSuppUniques = QmpDataDestGetSuppUniques( hDataDestUniques,  
    &qres );  
  
/* gets the output inclusion master duplicates flag */  
bInclMasters = QmpDataDestGetInclMasters( hDataDestUniques,  
    &qres );  
  
/* gets the output suppression master duplicates flag */  
bSuppMasters = QmpDataDestGetSuppMasters( hDataDestUniques,  
    &qres );  
  
/* gets output inclusion subordinate duplicates flag */  
bInclSubords = QmpDataDestGetInclSubords( hDataDestUniques,  
    &qres );
```

QmpDataDestRegEveryNthRecFunc

Registers the event handler to be notified of every Nth record processed.

Syntax

```
void QmpDataDestRegEveryNthRecFunc ( MpHnd in_hDataDest,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hDataDest
Handle to data destination object. Input.
in_Func
Pointer to event handler. Input.
out_pResult
Result code.
```

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The data destination can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

The default value of the data destination Nth record interval is 1.

See Also

[QmpDataDestUnregEveryNthRecFunc](#),
[QmpDataDestSetNthRecInterval](#)

Example

```
if ( bEveryNth ) {
    QmpDataDestRegEveryNthRecFunc( hDataDestUniques,
                                   pEveryNthRecordClient, &qres );
    QmpDataDestSetNthRecInterval( hDataDestUniques, 200, &qres );
    QmpDataDestRegEveryNthRe_cFunc( hDataDestMasters,
                                   pEveryNthRecordClient, &qres );
    QmpDataDestSetNthRecInterval( hDataDestMasters, 200, &qres );
    QmpDataDestRegEveryNthRecFunc( hDataDestSubords, &qres );
    QmpDataDestSetNthRecInterval( hDataDestSubords, 200, &qres );
    QmpDataDestRegEveryNthRecFunc( hDataDestSupp,
                                   pEveryNthRecordClient, &qres );
    QmpDataDestSetNthRecInterval( hDataDestSupp, 200, &qres );
}
```

QmpDataDestSetAugmentedFieldsFlag

SETS THE DATA DESTINATION AUGMENTED FIELDS FLAG.

Syntax

```
void QmpDataDestSetAugmentedFieldsFlag ( MpHnd in_hDataDest,
                                         QBOOL in_bAugmentedFields, QRESULT* out_pResult );
in_hDataDest
Handle to data destination object. Input.
in_bAugmentedFields
If QTRUE, records are output to the table with their augmented fields. If QFALSE, the records are output without them. The default value is QTRUE. Input.
out_pResult
Result code.
```

Return Value

None.

Notes

QmpDataDestSetAugmentedFieldsFlag sets the flag that determines whether records are written to a data destination with their augmented fields.

See “[Augmented Fields](#)” on page 39 for a description of these fields.

Some of the augmented fields will be meaningless in certain cases (the dupe group fields in a unique record, for instance). If the flag is set to QTRUE, all augmented fields are written to the data destination, even fields with meaningless values.

See Also

[QmpDataDestGetAugmentedFieldsFlag](#).

Example

```
QmpDeclHnd( hDataDest );
QRESULT qres;

QmpDataDestSetAugmentedFieldsFlag( hDataDest, QTRUE, &qres );
```

QmpDataDestSetAugOut

SETS THE OUTPUT AUGMENTED FIELDS PROPERTY FOR A DATA DESTINATION.

Syntax

```
void QmpDataDestSetAugOut ( MpHnd in_DataDest, QBOOL in_bFlag,
                           QRESULT* out_pResult );
in_DataDest
    Handle to the data destination object. Input.
in_bFlag
    Property setting. QTRUE (the default) writes augmented fields to the
    specified data destination. QFALSE suppresses augmented fields from
    being written.
    Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpDataDestSetAugOut determines whether internal augmented fields are written to a specific data destination. Augmented fields are placed last in an output record, in a predefined order.

If you plan to generate a table which will be used as a preprocessed data source, you must include augmented fields. Otherwise, the these fields are optional.

See Also

[QmpDataDestSetOutRec](#).

Example

```
QmpDataDestSetOutRec( hDataDestCons, hOutputRec, &qres );
QmpDataDestSetAugOut( hDataDestCons, QFALSE, &qres );
```

QmpDataDestSetExprDate

SETS DATE-TYPE EXPRESSION IN SPECIFIED DATA DESTINATION.

Syntax

```
void QmpDataDestSetExprDate( MpHnd in_hDataDest, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, MpHnd
    in_hDate, QRESULT* out_pResult );
in_DataDest
    Handle to data destination object. Input.
in_szFieldName
    Record field name. Input.
in_Operator
    Operator used for field evaluation comparison. Input.
in_hDate
    Handle to date object to compare field against. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a field value against a set value. If the expression is evaluated to be true, the record will be output to the data destination. If other filters are defined they also operate on the record.

This function allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type QMS_FLDEVAL_OPER, and a date object to compare the field value against. The date object contains a date value.

Example

See example on [page 258](#).

QmpDataDestSetExprDateStruct

SETS A QDATESTRUCT-TYPE EXPRESSION IN SPECIFIED DATA DESTINATION.

Syntax

```
void QmpDataDestSetExprDateStruct( MpHnd in_hDataDest, const
    char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator,
    QDATESTRUCT* in_pDateStruct, QRESULT* out_pResult );
```

in_DataDest

Handle to data destination object. *Input*.

in_szFieldName

Record field name. *Input*.

in_Operator

Operator used for field evaluation comparison. *Input*.

in_pDateStruct

Pointer to the QDATESTRUCT structure containing a date value. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a field value against a constant value. If the expression is evaluated to be true, the record will be output to the data destination. If other filters are defined they also operate on the record.

This function allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type QMS_FLDEVAL_OPER, and a QDATESTRUCT date structure to compare the field value against. The structure contains a date value.

Example

See example on [page 258](#).

QmpDataDestSetExprFloat

SETS FLOAT-TYPE EXPRESSION IN SPECIFIED DATA DESTINATION.

Syntax

```
void QmpDataDestSetExprFloat( MpHnd in_hDataDest, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, float
    in_fValue, QRESULT* out_pResult );
```

in_hDataDest
Handle to data destination object. *Input*.

in_szFieldName
Record field name. *Input*.

in_Operator
Operator used for field evaluation comparison. *Input*.

in_fValue
Float constant to compare against. *Input*.

out_pResult

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a float-type field value against a set value. If the expression is evaluated to be true, the record will be output to the data destination.

QmpDataDestSetExprFloat allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type `QMS_FLDEVAL_OPER`, and a float constant to compare the field value against.

See Also

`QmpDataDestSetExprString`, `QmpDataDestSetExprLong`

Example

```
QmpDataDestSetExprFloat(hDataDestInclUniques, "ZipCode",
    QMS_FLDEVAL_OPER_GREATER, (float)50000, pResult );
```

QmpDataDestSetExprLong

SETS LONG-TYPE EXPRESSION IN SPECIFIED DATA DESTINATION.

Syntax

```
void QmpDataDestSetExprLong( MpHnd in_hDataDest, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, long
    in_lValue, QRESULT* out_pResult );
```

in_hDataDest

Handle to data destination object. *Input*.

in_szFieldName

Record field name. *Input*.

in_Operator

Operator used for field evaluation comparison. *Input*.

in_lValue

Long constant to compare against. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a long-type field value against a set value. If the expression is evaluated to be true, the record will be output to the data destination.

QmpDataDestSetExprLong allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type **QMS_FLDEVAL_OPER**, and a long constant to compare the field value against.

See Also

QmpDataDestSetExprString, **QmpDataDestSetExprFloat**

Example

```
QmpDataDestSetExprLong( hDataDestInclUniques, "ZipCode",
    QMS_FLDEVAL_OPER_GREATER, (long)50000, pResult );
```

QmpDataDestSetExprString

SETS STRING-TYPE EXPRESSION IN SPECIFIED DATA DESTINATION.

Syntax

```
void QmpDataDestSetExprString( MpHnd in_hDataDest, const
    char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator, const
    char* in_szValue, QRESULT* out_pResult );
```

in_DataDest

Handle to data destination object. *Input*.

in_szFieldName

Record field name. *Input*.

in_Operator

Operator used for field evaluation comparison. *Input*.

in_szValue

String constant to compare against. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a string-type field value against a set value. If the expression is evaluated to be true, the record will be output to the data destination. If other filters are defined they also operate on the record.

QmpDataDestSetExprString allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type **QMS_FLDEVAL_OPER**, and a string constant to compare the field value against.

See Also

QmpDataDestFillExpr, **QmpDataDestSetExprLong**,
QmpDataDestSetExprFloat

Example

```
QmpDataDestSetExprString( hDataDestInclUniques, "ZipCode",
    QMS_FLDEVAL_OPER_GREATER, "50000", pResult );
```

QmpDataDestSetExprSubStr

SETS EXPRESSION SUBSTRING FOR A DATA DESTINATION.

Syntax

```
void QmpDataDestSetExprSubStr ( MpHnd in_hDataDest, int
                               in_iCharStartPos, int in_iNumChars, QRESULT* out_pResult );
in_hDataDest
Handle to data destination. Input.
in_iCharStartPos
Start position. Input.
in_iNumChars
Number of characters to consider. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

`QmpDataDestSetExprSubStr` adds a substring component to the expression created by one of the `QmpDataDestSetExpr*` functions. Specifically, it modifies the expression to evaluate a substring of the field. For example, in the expression “`FirstName = ‘Fred’`”, this function makes it possible to evaluate a substring of the `FirstName` field.

This function is valid for any of the data destination expression types (date, float, variant, etc.). If the field is not of type string, the value is converted to a string. Be aware of how the field type in question represents its values internally, and how that will affect its conversion to a string.

See Also

`QmpDataDestGetExprStart`, `QmpDataDestGetExprNumChar`.

Example

```
/* return the field evaluation substring start position */

/* return the object's starting position */
iFldExprStart = QmpDataDestGetExprStart( hDataDest, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataDestGetExprNumChar( hDataDest, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataDestSetExprSubStr( hDataDest, 2, 4, &qres );

/* return the object's starting position */
iFldExprStart = QmpDataDestGetExprStart( hDataDest, &qres );
if ( iFldExprStart != 2 )
```

```
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );  
/* return the object's number of characters */  
iFldExprNumChars = QmpDataDestGetExprNumChar( hDataDest, &qres );  
if ( iFldExprNumChars != 4 )  
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataDestSetExprVar

SETS EXPRESSION IN SPECIFIED DATA DESTINATION TO A VARIANT OBJECT VALUE.

Syntax

```
void QmpDataDestSetExprVar( MpHnd in_hDataDest, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, MpHnd
    in_hVar, QRESULT* out_pResult );
```

in_hDataDest
Handle to data destination object. *Input*.

in_szFieldName
Record field name. *Input*.

in_Operator
Operator used for field evaluation comparison. *Input*.
Valid enums are:

QMS_FLDEVAL_OPER_NONE
QMS_FLDEVAL_OPER_LESS
QMS_FLDEVAL_OPER_EQUAL
QMS_FLDEVAL_OPER_GREATER
QMS_FLDEVAL_OPER_LESSEQUAL
QMS_FLDEVAL_OPER_GREATEREQUAL
QMS_FLDEVAL_OPER_NOTLESS
QMS_FLDEVAL_OPER_NOTEQUAL
QMS_FLDEVAL_OPER_NOTGREATER

in_hVar
Handle of variant containing constant value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a field value against a variant constant. If the expression is evaluated to be true, the record will be output to the data destination.

This function allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type QMS_FLDEVAL_OPER, and a variant constant to compare the field value against.

Example

```

/* create a date, and move the date information into a date structure and */
/* into a variant structure */
hVar = QmpVarCreate( &qres );
hDate = QmpDateCreate( &qres );
QmpDateSetPicture( hDate, "jan 01, 3000", "mmm dd, ccyy", &qres );
QmpDateFillDateStruct( hDate, &dateStruct, &qres );
QmpVarSetDateStruct( hVar, &dateStruct, &qres );

/* Set/Fill data destination expressions using the variant structure */
/* and the date structure */
QmpDataDestSetRecProto( hDataDestMasters, hProtoRec, &qres );
QmpDataDestSetExprVar( hDataDestMasters, szDate, QMS_FLDEVAL_OPER_LESS,
    hVar, &qres );
QmpVarFillVarStruct( hVar, &varStruct, &qres );
QmpDataDestSetExprVarStruct( hDataDestMasters, szLastName,
    QMS_FLDEVAL_OPER_LESS, &varStruct, &qres );
QmpDataDestSetExprDate( hDataDestMasters, szDate, QMS_FLDEVAL_OPER_EQUAL,
    hDate, &qres );
QmpDataDestSetExprDateStruct( hDataDestMasters, szDate,
    QMS_FLDEVAL_OPER_EQUAL, &dateStruct, &qres );
QmpDataDestFillExpr( hDataDestMasters, szBuffer, sizeof( szBuffer ), &qres );

```

QmpDataDestSetExprVarStruct

SETS EXPRESSION IN SPECIFIED DATA DESTINATION TO A QVARSTRUCT VALUE.

Syntax

```
void QmpDataDestSetExprVarStruct( MpHnd in_hDataDest, const
    char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator,
    QVARSTRUCT* in_pVarStruct, QRESULT* out_pResult );
```

in_hDataDest

Handle to data destination object. *Input*.

in_szFieldName

Record field name. *Input*.

in_Operator

Operator used for field evaluation comparison. *Input*.

in_pVarStruct

Pointer to the QVARSTRUCT structure containing a constant value. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Records can be filtered into a data destination by comparing a field value against a constant. If the expression is evaluated to be true, the record will be output to the data destination.

This function allows the application to specify this filtering expression for a data destination. The expression is constructed from a record field name, a comparison operator of type QMS_FLDEVAL_OPER, and a constant in a QVARSTRUCT structure to compare the field value against. A QVARSTRUCT is a C structure that is analogous to a variant object. It can be filled and manipulated directly by the client.

Example

See example on [page 258](#).

QmpDataDestSetInclCons

SETS THE OUTPUT INCLUSION CONSOLIDATED RECORDS FLAG.

Syntax

```
void QmpDataDestSetInclCons( MpHnd in_hDataDest, QBOOL
    in_bFlag, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_bFlag
    QTRUE if output table should include consolidated records, else QFALSE.
    Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output consolidated records belonging to an inclusion list.

Though a consolidated record is created with a data list ID of 0 (see “[Function Class: QmpConsRpt*](#)” on page 163), during table generation the record is analyzed and placed in the proper data list. Note that it will never belong to a data source list (because it has a data source ID of 0).

Example

```
hTblCons = QmpTblCbCreate( "", "consolidated", &qres );
hDataDestCons = QmpDataDestCreate( hTblCons, &qres );
QmpDataDestSetInclCons( hDataDestCons, QTRUE, &qres );
QmpDataDestSetSuppCons( hDataDestCons, QTRUE, &qres );
if ( QmpDataDestGetInclCons( hDataDestCons, &qres ) != QTRUE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data destination is not setup
        correctly" );
if ( QmpDataDestGetSuppCons( hDataDestCons, &qres ) != QTRUE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data destination is not setup
        correctly" );
```

QmpDataDestSetInclMasters

SETS THE OUTPUT INCLUSION MASTER DUPLICATES FLAG.

Syntax

```
void QmpDataDestSetInclMasters( MpHnd in_DataDest, QBOOL
    in_bFlag, QRESULT* out_pResult );
in_DataDest
    Handle to data destination object. Input.
in_bFlag
    QTRUE if output table should include master duplicate records, else
    QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output master duplicate records belonging to an inclusion list.

Example

```
/* test setting the output flags */

/* sets the output suppression uniques flag */
QmpDataDestSetSuppUniques( hDataDestUniques, QFALSE, &qres );

/* sets the output inclusion master duplicates flag */
QmpDataDestSetInclMasters( hDataDestUniques, QFALSE, &qres );

/* sets the output suppression master duplicates flag */
QmpDataDestSetSuppMasters( hDataDestUniques, QFALSE, &qres );

/* sets output inclusion subordinate duplicates flag */
QmpDataDestSetInclSubords( hDataDestUniques, QFALSE, &qres );

/* sets output suppression subordinate duplicates flag */
QmpDataDestSetSuppSubords( hDataDestUniques, QFALSE, &qres );
```

QmpDataDestSetInclSubords

SETS THE OUTPUT INCLUSION SUBORDINATE DUPLICATES FLAG.

Syntax

```
void QmpDataDestSetInclSubords( MpHnd in_hDataDest, QBOOL
    in_bFlag, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_bFlag
    QTRUE if output table should include subordinate duplicate records, else
    QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output subordinate duplicate records belonging to an inclusion list.

Example

```
/* test setting the output flags */

/* sets the output suppression uniques flag */
QmpDataDestSetSuppUniques( hDataDestUniques, QFALSE, &qres );

/* sets the output inclusion master duplicates flag */
QmpDataDestSetInclMasters( hDataDestUniques, QFALSE, &qres );

/* sets the output suppression master duplicates flag */
QmpDataDestSetSuppMasters( hDataDestUniques, QFALSE, &qres );

/* sets output inclusion subordinate duplicates flag */
QmpDataDestSetInclSubords( hDataDestUniques, QFALSE, &qres );

/* sets output suppression subordinate duplicates flag */
QmpDataDestSetSuppSubords( hDataDestUniques, QFALSE, &qres );
```

QmpDataDestSetInclUniques

SETS THE OUTPUT INCLUSION UNIQUES FLAG.

Syntax

```
void QmpDataDestSetInclUniques ( MpHnd in_hDataDest, QBOOL
                               in_bFlag, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_bFlag
    QTRUE if output table should include unique records, else QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output unique records belonging to an inclusion list.

See Also

[QmpDataDestGetInclUniques](#)

Example

```
hSubordsTable = QmpTblCbCreate( "", "InclSubord", pResult );
hSuppSubordsTable = QmpTblCbCreate( "", "SuppSubord", pResult
);
hDataDestInclUniques = QmpDataDestCreate( hUniquesTable,
                                         pResult );
QmpDataDestSetInclUniques( hDataDestInclUniques, QTRUE,
                             pResult );
hDataDestSuppUniques = QmpDataDestCreate( hSuppUniquesTable,
                                         pResult );
QmpDataDestSetSuppUniques( hDataDestSuppUniques, QTRUE,
                             pResult );
hDataDestInclMasters = QmpDataDestCreate( hMastersTable,
                                         pResult );
```

QmpDataDestSetNthRecInterval

SETS DATA DESTINATION NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpDataDestSetNthRecInterval ( MpHnd in_hDataDest, long
                                in_lInterval, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_lInterval
    data destination Nth record event interval. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the data destination Nth record event interval.

The data destination can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the data destination processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

[QmpDataDestGetNthRecInterval](#), [QmpDataDestRegEveryNthRecFunc](#),
[QmpDataDestUnregEveryNthRecFunc](#)

Example

```
if ( bEveryNth ) {
    QmpDataDestRegEveryNthRecFunc( hDataDestUniques,
        pEveryNthRecordClient, &qres );
    QmpDataDestSetNthRecInterval( hDataDestUniques, 200, &qres );
    QmpDataDestRegEveryNthRecFunc( hDataDestMasters,
        pEveryNthRecordClient, &qres );
    QmpDataDestSetNthRecInterval( hDataDestMasters, 200, &qres );
    QmpDataDestRegEveryNthRecFunc( hDataDestSubords, &qres );
    QmpDataDestSetNthRecInterval( hDataDestSubords, 200, &qres );
    QmpDataDestRegEveryNthRecFunc( hDataDestSupp,
        pEveryNthRecordClient, &qres );
    QmpDataDestSetNthRecInterval( hDataDestSupp, 200, &qres );
}
```

QmpDataDestSetOutputTypes

SETS THE OUTPUT TYPE FLAGS FROM A STRING.

Syntax

```
void QmpDataDestSetOutputTypes( MpHnd in_hDataDest, const
```

```
    char* in_szFlags, QRESULT* out_pResult
```

```
in_hDataDest
```

Handle to data destination object. *Input*.

```
in_szFlags
```

String version of output types. *Input*.

Permitted string values are:

"INCL_UNIQUE"	Inclusion uniques
"INCL_MASTER"	Inclusion master duplicates
"INCL_SUBORD"	Inclusion subordinate duplicates
"INCL_CONSOL"	Inclusion consolidated records.
"SUPP_UNIQUE"	Suppression uniques
"SUPP_MASTER"	Suppression master duplicates
"SUPP_SUBORD"	Suppression subordinate duplicates
"SUPP_CONSOL"	Suppression consolidated records.

The “|” operator will concatenate flags.

```
out_pResult
```

Result code. *Output*.

Return Value

None.

Notes

QmpDataDestSetOutputTypes sets the data destination record type output flags with a string.

Example

```
QmpDataDestSetOutputTypes( hDataDestSubords, "INCL_SUBORD",
    &qres );
```

QmpDataDestSetOutputTypesWithBitmask

SETS THE OUTPUT TYPE FLAGS FROM A MASK.

Syntax

```
void QmpDataDestSetOutputTypesWithBitmask( MpHnd
    in_hDataDest, unsigned int in_uiFlags, QRESULT* out_pResult
);
```

in_hDataDest

Handle to data destination object. *Input*.

in_uiFlags

Bitmask version of output types. *Input*.

Available bit mask values are:

OUTPUT_NONE	None defined
OUTPUT_INCL_UNIQUE	Inclusion unique records
OUTPUT_INCL_MASTERS	Inclusion master duplicates
OUTPUT_INCL_SUBORDS	Inclusion subordinate duplicates
OUTPUT_SUPP_UNIQUE	Suppression unique records
OUTPUT_SUPP_MASTERS	Suppression master duplicates
OUTPUT_SUPP_SUBORDS	Suppression subordinate duplicates
OUTPUT_INCL_CONSOL	Inclusion consolidated duplicates
OUTPUT_SUPP_CONSOL	Suppression consolidated duplicates

The “ | ” operator will concatenate bit masks.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpDataDestSetOutputTypesWithBitmask sets the data destination record output type flags using a bit mask.

Example

```
QmpDataDestSetOutputTypesWithBitmask (OutputDest, (unsigned
int)~0, &Result);
```

QmpDataDestSetOutRec

SETS THE OUTPUT PROTOTYPE RECORD.

Syntax

```
void QmpDataDestSetOutRec ( MpHnd in_DataDest, MpHnd in_hRec,
                           QRESULT* out_pResult );
```

in_DataDest
Handle to the data destination. *Input*.

in_hRec
Handle to the output record prototype. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDataDestSetOutRec uses a prototype record to specify the fields, and the order of the fields, output to a data destination. You may populate the output prototype record with a subset of the fields specified in the normal or pass-through prototype record.

See Also

QmpDataDestSetAugOut.

Example

```
QmpDataDestSetOutRec( hDataDestCons, hOutputRec, &qres );
QmpDataDestSetAugOut( hDataDestCons, QFALSE, &qres );
```

QmpDataDestSetRecProto

SETS THE PROTOTYPE RECORD VIA AN APPLICATION-OWNED RECORD.

Syntax

```
void QmpDataDestSetRecProto ( MpHnd in_hDataDest, MpHnd  
    in_hRec, QRESULT* out_pResult );
```

in_hDataDest
Handle to data destination object. *Input*.

in_hRec
Handle to the prototype record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDataDestSetRecProto uses an application-owned record to set the data fields of the data destination object. This is done to enable the data destination to handle records in a given job.

Example

```
QmpDataDestSetMaxRec( hDataDestInclUniques, 2500, pResult );  
QmpDataDestSetRecProto( hDataDestInclUniques, hSourceRecord,  
    pResult );
```

QmpDataDestSetReplaceFlag

SETS THE DATA DESTINATION REPLACE FLAG.

Syntax

```
void QmpDataDestSetReplaceFlag( MpHnd in_hDataDest, QBOOL
    in_bReplace, QRESULT* out_pResult
in_hDataDest
    Handle to data destination. Input.
in_bReplace
    QTRUE if output table should be overwritten, QFALSE if data should be
    appended to the file. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function determines whether any existing output table in the data destination should be overwritten, or if data should be appended to the output table.

See Also

[QmpDataDestGetReplaceFlag](#)

Example

```
hTblDestSupp = QmpTblCbCreate( szDatabaseName,
    pszTempPathBuffer, &qres );
hDataDestSupp = QmpDataDestCreate( hTblDestSupp, &qres );
QmpDataDestSetOutputTypesWithBitmask( hDataDestSupp,
    OUTPUT_SUPP_UNIQUES|OUTPUT_SUPP_MASTERS|OUTPUT_SUPP_SUBORDS,
    &qres );
bReplaceFlag = QmpDataDestGetReplaceFlag( hDataDestSupp,
    &qres );
QmpDataDestSetReplaceFlag( hDataDestSupp, QTRUE, &qres );
bReplaceFlag = QmpDataDestGetReplaceFlag( hDataDestSupp,
    &qres );
ulMaxRecords = QmpDataDestGetMaxRec( hDataDestSupp, &qres );

/* sets the maximum number of records allowed */
QmpDataDestSetMaxRec( hDataDestSupp, 2000, &qres );
ulMaxRecordsCheck = QmpDataDestGetMaxRec( hDataDestSupp, &qres
)
```

QmpDataDestSetStatSamp

SETS THE STATISTICAL SAMPLING PROPERTIES OF A DATA DESTINATION.

Syntax

```
void QmpDataDestSetStatSamp ( MpHnd in_hDataDest,
    QMS_OUTPUT_SAMPLING* in_StatSampling, QRESULT*  

    out_pResult );
```

in_hDataDest
Handle to a data destination. *Input*.

in_StatSampling
Statistical sampling structure. If a QMS_OUTPUT_SAMPLING member is set to 0, that member will not be used. *Input*.

Structure members are:

lMaxRecords	Maximum number of records to include.
lFirstRecordNum	Record number of first record to include.
lLastRecordNum	Record number of last record to include.
lInterval	Sampling interval (return every “nth” record). This <i>must</i> be greater than lGroupSize.
lGroupSize	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record).
lMaxDupeGroups	Maximum number of dupe groups to include.
lDupesFirst	First dupe group to include.
lDupesLast	Last dupe group to include.
lDupesInterval	Dupe group sampling interval.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDataDestSetStatSamp turns on statistical sampling for a data destination and sets the sampling properties using an application-supplied structure of type **QMS_OUTPUT_SAMPLING**. The structure members must be set before the function is called.

Statistical sampling allows you to quickly filter output records in a controlled way. By including a small subset of records in a report or output table, you can evaluate the output before committing to a long processing run.

Statistical sampling for data destinations is optional. Call **QmpDataDestFillStatSamp** to get the current statistical sampling properties of a data destination. If statistical sampling is used for a data destination, a statistical sampling section will appear in the job summary report.

A data destination will receive up to the maximum number of records set by statistical sampling. This limit might be set by the `lMaxRecords` property or by the difference between the `lFirstRecordNum` and `lLastRecordNum` values. If a data destination is configured to receive 1,000 records from a 1,000,000 record data source, after 1,000 records are written the job will begin processing the next data destination.

If you want to use either the interval or group size sampling properties, both properties must be set to a value greater than zero. It does not make sense to set a group size greater than an interval; doing so will cause an error.

Example

```

QMS_OUTPUT_SAMPLING OutputSampling; /* statistical sampling structure */
QmpDeclHnd( hTblSurvivors ); /* handle to Masters table */
QmpDeclHnd( hDataDestSurvivors ); /* handle to Masters Data Destination */
strcpy( szDatabaseName, "" );
sprintf( pszTempPathBuffer, "%s%sSurvivors", pszOutputPathBuffer, pszDelim );
hTblSurvivors = QmpTblCbCreate( szDatabaseName, pszTempPathBuffer, &qres );
hDataDestSurvivors = QmpDataDestCreate( hTblSurvivors, &qres );
QmpDataDestSetInclMasters( hDataDestSurvivors, QTRUE, &qres );
QmpDataDestSetInclUniques( hDataDestSurvivors, QTRUE, &qres );
OutputSampling.lMaxRecords = 4000; /* max number of records to output */
OutputSampling.lFirstRecordNum= 0; /* first record to output */
OutputSampling.lLastRecordNum= 0; /* last record to output */
OutputSampling.lInterval = 0; /* Sampling interval for output */
OutputSampling.lGroupSize = 0; /* size of group beginning at each */
/* output interval */
OutputSampling.lMaxDupeGroups= 0; /* max number of Dupe Groups to output */
OutputSampling.lDuplicatesFirst = 0; /* first Dupe Group to output. */
OutputSampling.lDuplicatesLast = 0; /* last DupeGroup to output. */
OutputSampling.lDuplicatesInterval= 10; /* Dupe Group sampling interval. */
QmpDataDestSetStatSamp( hDataDestSurvivors, &OutputSampling, &qres );

/* Give the data destinations to the table generation phase */
QmpTblGenUseDataDest( hTableGen, hDataDestSurvivors, &qres );
lDestCount = QmpTblGenGetDataDestCount( hTableGen, &qres );
printf( "Starting Table Generation phase:\n" );

/* give the consolidation table to table generation */
QmpTblGenUseCons( hTableGen, hConsolidation, &qres );
QmpPhaseStart( hTableGen, &qres );

/* Destroy table generation */
QmpPhaseDestroy( hTableGen, &qres );
QmpDataDestDestroy( hDataDestSurvivors, &qres );
QmpTblClose( hTblSurvivors, &qres );
QmpTblDestroy( hTblSurvivors, &qres );

```

QmpDataDestSetSuppCons

SETS THE OUTPUT SUPPRESSION CONSOLIDATED RECORDS FLAG.

Syntax

```
void QmpDataDestSetSuppCons( MpHnd in_hDataDest, QBOOL  
    in_bFlag, QRESULT* out_pResult );
```

in_hDataDest
Handle to data destination object. *Input*.

in_bFlag
Set to *QTRUE* to configure data destination to output consolidated records belonging to a suppression list, else *QFALSE*. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Use this function to configure the specified data destination to output consolidated records belonging to a suppression list.

Example

See example on [page 261](#).

QmpDataDestSetSuppMasters

SETS THE OUTPUT SUPPRESSION MASTER DUPLICATES FLAG.

Syntax

```
void QmpDataDestSetSuppMasters( MpHnd in_hDataDest, QBOOL
    in_bFlag, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_bFlag
    Set to QTRUE to configure data destination to output master duplicate
    records belonging to a suppression list, else QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output master duplicate records belonging to a suppression list.

See Also

[QmpDataDestGetSuppMasters](#).

Example

```
/* test setting the output flags */

/* sets the output suppression uniques flag */
QmpDataDestSetSuppUniques( hDataDestUniques, QFALSE, &qres );

/* sets the output inclusion master duplicates flag */
QmpDataDestSetInclMasters( hDataDestUniques, QFALSE, &qres );

/* sets the output suppression master duplicates flag */
QmpDataDestSetSuppMasters( hDataDestUniques, QFALSE, &qres );

/* sets output inclusion subordinate duplicates flag */
QmpDataDestSetInclSubords( hDataDestUniques, QFALSE, &qres );

/* sets output suppression subordinate duplicates flag */
QmpDataDestSetSuppSubords( hDataDestUniques, QFALSE, &qres );
```

QmpDataDestSetSuppSubords

SETS THE OUTPUT SUPPRESSION SUBORDINATE DUPLICATES FLAG.

Syntax

```
void QmpDataDestSetSuppSubords( MpHnd in_hDataDest, QBOOL
    in_bFlag, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_bFlag
    Set to QTRUE to configure data destination to output subordinate duplicate
    records belonging to a suppression data list, else QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output subordinate duplicate records belonging to a suppression data list.

See Also

[QmpDataDestGetSuppSubords](#).

Example

```
/* test setting the output flags */

/* sets the output suppression uniques flag */
QmpDataDestSetSuppUniques( hDataDestUniques, QFALSE, &qres );

/* sets the output inclusion master duplicates flag */
QmpDataDestSetInclMasters( hDataDestUniques, QFALSE, &qres );

/* sets the output suppression master duplicates flag */
QmpDataDestSetSuppMasters( hDataDestUniques, QFALSE, &qres );

/* sets output inclusion subordinate duplicates flag */
QmpDataDestSetInclSubords( hDataDestUniques, QFALSE, &qres );

/* sets output suppression subordinate duplicates flag */
QmpDataDestSetSuppSubords( hDataDestUniques, QFALSE, &qres );
```

QmpDataDestSetSuppUniques

SETS THE DATA DESTINATION OUTPUT SUPPRESSION UNIQUES FLAG.

Syntax

```
void QmpDataDestSetSuppUniques( MpHnd in_hDataDest, QBOOL
    in_bFlag, QRESULT* out_pResult
in_hDataDest
    Handle to data destination object. Input.
in_bFlag
    Set to QTRUE to configure data destination to output unique records
    belonging to a suppression list, else QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to configure the specified data destination to output unique records belonging to a suppression list.

See Also

[QmpDataDestGetSuppUniques](#)

Example

```
/* test setting the output flags */

/* sets the output suppression uniques flag */
QmpDataDestSetSuppUniques( hDataDestUniques, QFALSE, &qres );

/* sets the output inclusion master duplicates flag */
QmpDataDestSetInclMasters( hDataDestUniques, QFALSE, &qres );

/* sets the output suppression master duplicates flag */
QmpDataDestSetSuppMasters( hDataDestUniques, QFALSE, &qres );

/* sets output inclusion subordinate duplicates flag */
QmpDataDestSetInclSubords( hDataDestUniques, QFALSE, &qres );

/* sets output suppression subordinate duplicates flag */
QmpDataDestSetSuppSubords( hDataDestUniques, QFALSE, &qres );
```

QmpDataDestUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpDataDestUnregEveryNthRecFunc ( MpHnd in_hDataDest,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hDataDest
    Handle to data destination object. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function unregisters the specified event handler from the data destination, so that the event handler will no longer be notified of every Nth record processed.

The data destination can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
QmpDataDestUnregEveryNthRecFunc ( hDataDest,
                                     pDataDestEventHandler, pResult );
```


Function Class: QmpDataInp*

DATA INPUT PHASE FUNCTIONS.

Assigning Household IDs

The data input phase functions allow you to assign household ID values to a record field as the records are read into the DITR. Centrus Merge/Purge supports two kinds of household ID values: serial and user-defined. For serial IDs, the application specifies a starting number. The job assigns that number to the first input record, and the number is incremented for all subsequent records. For user-defined household IDs, the application builds an string expression (also called a picture) describing how the values should be constructed, using field values from the prototype record. The field to be assigned household IDs must be included in the prototype record.

Assigning household IDs is just the first step in the householding process. Once all of the records in the DITR are assigned a household ID, you would want to match records based upon address, creating dupe groups whose members all have the same address. You could then use data consolidation to write the master's household ID value to all the dupes in a group. A dupe group would then essentially be a household, with each dupe representing a person who lives at that address.

Quick Reference

Function	Description	Page
QmpDataInpAddDataSrc	Adds a non-preprocessed data source to the data input phase.*	281
QmpDataInpAddSerialId	Adds serial household ID values to input records during the data input phase.	282
QmpDataInpAddUserId	Adds user-defined household ID values to input records during the data input phase.	283
QmpDataInpClear	Clears a data input.	285
QmpDataInpCreate	Creates a data input object.	286
QmpDataInpGetDataSrcAt	Gets a data source from the data input phase.	288
QmpDataInpGetDataSrcCnt	Gets the number of data sources attached to the data input phase.	290
QmpDataInpGetDataSrcNameID	Gets the data source ID, given a data source name.	291
QmpDataInpGetIndxKeyCnt	Gets index key count.	292
QmpDataInpGetNthRecInterval	Gets data input Nth record event interval.	293
QmpDataInpGetSerialIdFldName	Gets the serial household ID field name.	294
QmpDataInpGetSerialIdStart	Gets the serial household id start value.	295

Function	Description	Page
QmpDataInpGetSerialIdTrim	Returns the trim property associated with a serial household id field.	296
QmpDataInpGetUserIdFldName	Gets the user-defined household ID field name.	297
QmpDataInpGetUserIdPicture	Gets the field expression picture used to construct a user-defined household ID.	298
QmpDataInpGetUserIdTrim	Returns the trim property associated with a user-defined household id field.	299
QmpDataInpIsValid	Tests whether a data input phase is valid.	300
QmpDataInpRegEveryNthRecFunc	Registers the data input phase to be notified of every Nth record processed.	301
QmpDataInpRemDataSrc	Removes specified data source from the data input.	302
QmpDataInpRemIndxKey	Removes specified index key from the data input.	303
QmpDataInpRemSerialId	Removes the values in serial household ID fields.	304
QmpDataInpRemUserId	Removes the values from user-defined household ID fields.	305
QmpDataInpSetNthReclInterval	Sets data input Nth record event interval.	306
QmpDataInpUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	307
QmpDataInpUseDataLstSvc	Specifies the data list service object to be used by data input.	308
QmpDataInpUseIndxKey	Specifies the index key object to be used by the data input.	309

QmpDataInpAddDataSrc

ADDS A NON-PREPROCESSED DATA SOURCE TO THE DATA INPUT PHASE.*

Syntax

```
void QmpDataInpAddDataSrc( MpHnd in_hDataInp, MpHnd
                           in_hDataSrc, QRESULT* out_pResult );
in_hDataInp
    Handle to data input. Input.
in_hDataSrc
    Handle to the data source. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

*The *in_hDataSrc* handle *must* be of type QMS_DATSRC_TYPE_FUNCTOR or QMS_DATSRC_TYPE_TABLE.

QmpDataInpAddDataSrc adds a non-preprocessed data source to the data input phase.

The data input phase can process one or more independent sources of records. Users can create any number of data sources of type QMS_DATSRC_TYPE_FUNCTOR or QMS_DATSRC_TYPE_TABLE, and add them to the data input phase. Once the phase is started using the **QmpPhaseStart** function, the data input phase will gather all the records from all the data sources, make sure that they have been assigned data list membership, and add them all to the DITR.

Example

See example on [page 286](#).

QmpDataInpAddSerialId

ADDS SERIAL HOUSEHOLD ID VALUES TO INPUT RECORDS DURING THE DATA INPUT PHASE.

Syntax

```
void QmpDataInpAddSerialId (MpHnd in_hDataInp, const char*
    in_szFldName, unsigned long in_ulStart, QMS_FIELD_TRIM
    in_Trim, QRESULT* out_pResult );
```

in_hDataInp
Handle to data input phase. *Input*.

in_szFldName
Field you want to add household IDs to. *Input*.

in_ulStart
Starting number for household IDs. *Input*.

in_Trim
This parameter is unused. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

`QmpDataInpAddSerialId` directs that household IDs are added to a specified field during the data input phase. The first ID is the value specified in *in_ulStart*, and subsequent values are incremented serially. The field name specified by *in_szFldName* must exist in the prototype record.

See Also

[QmpDataInpRemSerialId](#).

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );

/* The in_Trim parameter is just a placeholder */
/* --the value isn't used */
QmpDataInpAddSerialId( hDataInp, "SerialId", 1,
    QMS_FIELD_TRIM_BOTH, &qres );

/* now look at serial household id */
szIdBuf = QmpDataInpGetSerialIdFldName(hDataInp, &qres);
ulIdStart = QmpDataInpGetSerialIdStart(hDataInp, &qres);
FieldTrim1 = QmpDataInpGetSerialIdTrim(hDataInp, &qres);

/* now remove the household id fields ( still in */
/* prototype record */
QmpDataInpRemSerialId( hDataInp, "SerialId", &qres );
```

QmpDataInpAddUserId

ADDS USER-DEFINED HOUSEHOLD ID VALUES TO INPUT RECORDS DURING THE DATA INPUT PHASE.

Syntax

```
void QmpDataInpAddUserId (MpHnd in_hDataInp, const char*
in_szFldName, const char* in_szPicture, QMS_FIELD_TRIM
in_Trim, QRESULT* out_pResult );
```

in_hDataInp
Handle to data input phase. *Input*.

in_szFldName
Name of field to add user household IDs to. *Input*.

in_szPicture
Expression specifying how user household ID values are constructed.
Input.

in_Trim
Specifies how string fields used in the construction of household ID values should be trimmed. *Input*.
Valid enums are:

QMS_FIELD_TRIM_NONE
QMS_FIELD_TRIM_LEFT
QMS_FIELD_TRIM_RIGHT
QMS_FIELD_TRIM_BOTH

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDataInpAddUserId directs that user-defined household IDs be added to the specified field during the data input phase. The *in_szPicture* parameter specifies how the household ID values are constructed.

The field specified by *in_szFldName* must exist in the prototype record. The other fields specified in the *in_szPicture* expression must also exist in the prototype record.

The *in_szPicture* string is an expression describing how the user-defined household ID values should be constructed, using fields from the prototype record. The use of the **substr(name, start, count)** format is accepted. Multiple fields are separated by '+' characters.

For example, if the Address field is 35 characters wide, and the Zipcode field is 9 characters wide, then the picture "Address+Zipcode" is equivalent to "substr(Address,1,35) + substr(Zipcode,1,9)".

It is probably most sensible to call **QmpDataInpAddUserId** *after* the record prototype fields have been defined.

See Also

QmpDataInpRemUserId.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );
QmpDataInpAddUserId( hDataInp, "UserId",
    "Address+ZipCode", QMS_FIELD_TRIM_BOTH, &qres );
```

QmpDataInpClear

CLEAR A DATA INPUT.

Syntax

```
void QmpDataInpClear ( MpHnd in_hDataInp, QRESULT* out_pResult
    );
```

in_hDataInp
Handle to data input. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the specified data input object back to its initial state.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

See example on [page 286](#).

QmpDataInpCreate

Creates a data input object.

Syntax

```
MpHnd QmpDataInpCreate ( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns the handle to the data input object if successful, or `NULL` if there is an error.

Notes

This function creates a data input object. This object organizes the activity of gathering input data from one or more data source objects (see the `QmpDataSrc*` functions).

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
QmpDeclHnd( hDataInp );
long lCount;
QRESULT qres;

/* declare a handle to the data input phase and */
/* allocate the object */
hDataInp = QmpDataInpCreate(&qres);

/* Give the data input phase a previously allocated */
/* and initialized data list service */
QmpDataInpUseDataLstSvc( hDataInp,
hDataListService, &qres );

/* set the record prototype in the data input phase */
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );

/* give index keys to the data input phase */
QmpDataInpUseIdxKey( hDataInp, hIndexKey1, & qres );
QmpDataInpUseIdxKey( hDataInp, hIndexKey2, & qres );
QmpDataInpUseIdxKey( hDataInp, hIndexKey3, & qres );
lCount = QmpDataInpGetIdxKeyCnt( hDataInp, &qres );
if (lCount != 3 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of
index keys" );

/* add a data source (previously allocated and */
/* initialized) to the data input phase */
QmpDataInpAddDataSrc(hDataInp, hDataSrc, &qres);
lCount = QmpDataInpGetDataSrcCnt( hDataInp, &qres );
if (lCount != 1 )
```

```

QmpUtilChkResultCode( QRESULT_SEVERE_FAIL, "wrong number of
data sources" );

/* give a DITR to the data input phase */
QmpDataInpUseDITR( hDataInp, hDITR, &qres );

/* register a callback for events from the data */
/* input phase */
QmpDataInpRegEveryNthRecFunc( hDataInp,
                               pEveryNthRecordClient,
                               &qres );
QmpDataInpSetNthRecInterval( hDataInp, 500, &qres );

/* get the Nth record interval */
long lNthRecInterval = QmpDataInpGetNthRecInterval( hDataInp,
                                                    &qres );

/* Test whether the data input phase is valid */
QBOOL bIsDataInpValid = QmpDataInpIsValid(hDataInp, &qres );

/* start the data input phase */
QmpPhaseStart( hDataInp, &qres );

/* Remove the index keys from the data input phase */
QmpDataInpRemIdxKey( hDataInp, hIndexKey1, &qres );
QmpDataInpRemIdxKey( hDataInp, hIndexKey2, &qres );
QmpDataInpRemIdxKey( hDataInp, hIndexKey3, &qres );

/* clear and destroy the data input phase */
QmpDataInpClear( hDataInp, &qres );

/* remove a data source from the data input phase */
long lDataSrcID = QmpDataSrcGetID( hDataSrc, &qres );
QmpDataInpRemDataSrc(hDataInp, lDataSrcID, &qres);

/* unregister a callback for events from the data */
/* input phase */
QmpDataInpUnregEveryNthRecFunc( hDataInp,
                                 pEveryNthRecordClient,
                                 &qres );

/* destroy the data input phase */
QmpDataInpDestroy(hDataInp, &qres);

```

QmpDataInpGetDataSrcAt

GETS A DATA SOURCE FROM THE DATA INPUT PHASE.

Syntax

```
MpHnd QmpDataInpGetDataSrcAt ( MpHnd in_hDataInp, long
                                in_lIndex, QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
in_lIndex
    Index into data source collection. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns handle to the data source if successful, or NULL if there is an error.

Notes

The data input phase maintains a collection of data source objects, each one representing a different physical source of records. The application adds each data source to the data input phase separately.

QmpDataInpGetDataSrcAt returns a handle to the data source when given the position of the data source in the data source list.

Example

```
/* declare */
QmpDeclHnd( hRec );
QmpDeclHnd( hDataInp );
QmpDeclHnd( hDataSrc );
QmpDeclHnd( hDataSrc2 );

/* create */
hDataInp = QmpDataInpCreate( &qres );
hRec = QmpRecCreate( &qres );
hDataSrc = QmpDataSrcCreate(QMS_DATSRC_TYPE_FUNCTOR, "Data
                           Source", 2, &qres );

/* use */
QmpRecAdd( hRec, "Address", &qres );
QmpDataSrcSetRecProto( hDataSrc, hRec, &qres );
QmpDataSrcSetFillFunc( hDataSrc, my_fill_func, &qres );
QmpDataInpAddDataSrc( hDataInp, hDataSrc, &qres );

/* test */
hDataSrc2 = QmpDataInpGetDataSrcAt( hDataInp, 0, &qres );
if ( hDataSrc != hDataSrc2 ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data sources do
                           not match" );
}
```

```
/* destroy */
QmpRecDestroy(hRec, &qres);
QmpDataSrcDestroy(hDataSrc, &qres);
QmpDataInpDestroy(hDataInp, &qres);
```

QmpDataInpGetDataSrcCnt

GETS THE NUMBER OF DATA SOURCES ATTACHED TO THE DATA INPUT PHASE.

Syntax

```
long QmpDataInpGetDataSrcCnt ( MpHnd in_hDataInp, QRESULT*  
                               out_pResult );  
  
in_hDataInp  
    Handle to data input phase. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns the data source count if successful, or -1 if there is an error.

Notes

This function returns the number of physical sources of data (data sources) that have been given to the data input phase.

Example

See example on [page 286](#).

QmpDataInpGetDataSrcNameID

GETS THE DATA SOURCE ID, GIVEN A DATA SOURCE NAME.

Syntax

```
long QmpDataInpGetDataSrcNameID ( MpHnd in_hDataInp, const
                                char* in_szDataSrcName, QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
in_szDataSrcName
    Data source name. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the data source ID if successful, or -1 if there is an error.

Notes

This function returns the ID of a data source, when given the source's name. Data lists require that source IDs for data sources be added to them; this convenience function enables assigning a source ID when only the name is known.

Example

```
long lDataSrcId;
MpHnd hDataInp;
QRESULT qresult;

lDataSrcId = QmpDataInpGetDataSrcNameID ( hDataInp, "source
                                         alpha", &qresult );
```

QmpDataInpGetIdxKeyCnt

GETS INDEX KEY COUNT.

Syntax

```
long QmpDataInpGetIdxKeyCnt ( MpHnd in_hDataInp, QRESULT*  
    out_pResult );  
  
in_hDataInp  
    Handle to data input. Input.  
out_pResult  
    Result code. Output.
```

Return Value

Returns index key count if successful, or -1 if there is an error.

Notes

This function gets the number of index keys currently in the specified data input object.

See Also

[QmpDataInpUseIdxKey](#).

Example

See example on [page 286](#).

QmpDataInpGetNthRecInterval

GETS DATA INPUT NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpDataInpGetNthRecInterval ( MpHnd in_hDataInp,
                                QRESULT* out_pResult );
in_hDataInp
    Handle to data input. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns current record count interval notification setting for the data input object if successful, or -1 if there is an error.

Notes

This function gets the data input phase Nth record event interval.

The data input phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the data input phase processes 100 records, it will send out a notification to *all* registered clients.

See Also

[QmpDataInpRegEveryNthRecFunc](#).

Example

See example on [page 286](#).

QmpDataInpGetSerialIdFldName

GETS THE SERIAL HOUSEHOLD ID FIELD NAME.

Syntax

```
const char* QmpDataInpGetSerialIdFldName ( MpHnd in_hDataInp,
                                         QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the name of the field that holds the serial household ID.

Notes

See Also

QmpDataInpGetSerialIdStart.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );

QmpDataInpAddSerialId( hDataInp, "SerialId", 1,
                        QMS_FIELD_TRIM_BOTH, &qres );

/* now look at serial household id */
szIdBuf = QmpDataInpGetSerialIdFldName(hDataInp, &qres);
ulIdStart = QmpDataInpGetSerialIdStart(hDataInp, &qres);
FieldTrim1 = QmpDataInpGetSerialIdTrim(hDataInp, &qres);

/* now remove the household id fields ( still in */
/* prototype record ) */
QmpDataInpRemSerialId( hDataInp, "SerialId", &qres );
```

QmpDataInpGetSerialIdStart

GETS THE SERIAL HOUSEHOLD ID START VALUE.

Syntax

```
unsigned long QmpDataInpGetSerialIdStart ( MpHnd in_hDataInp,
                                         QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the starting number associated with a serial household ID field.

Notes

See Also

`QmpDataInpGetSerialIdFldName`, `QmpDataInpGetSerialIdTrim`.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );

QmpDataInpAddSerialId( hDataInp, "SerialId", 1,
                        QMS_FIELD_TRIM_BOTH, &qres );

/* now look at serial household id */
szIdBuf = QmpDataInpGetSerialIdFldName(hDataInp, &qres);
ulIdStart = QmpDataInpGetSerialIdStart(hDataInp, &qres);
FieldTrim1 = QmpDataInpGetSerialIdTrim(hDataInp, &qres);

/* now remove the household id fields ( still in */
/* prototype record ) */
QmpDataInpRemSerialId( hDataInp, "SerialId", &qres );
```

QmpDataInpGetSerialIdTrim

RETURNS THE TRIM PROPERTY ASSOCIATED WITH A SERIAL HOUSEHOLD ID FIELD.

Syntax

```
QMS_FIELD_TRIM QmpDataInpGetSerialIdTrim ( MpHnd in_hDataInp,
    QRESULT* out_pResult );
```

in_hDataInp

Handle to data input phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

The possible return values are:

QMS_FIELD_TRIM_NONE

QMS_FIELD_TRIM_LEFT

QMS_FIELD_TRIM_RIGHT

QMS_FIELD_TRIM_BOTH

Notes

This property is currently unused.

See Also

QmpDataInpGetSerialIdFldName, **QmpDataInpGetSerialStart**.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );

QmpDataInpAddSerialId( hDataInp, "SerialId", 1,
    QMS_FIELD_TRIM_BOTH, &qres );

/* now look at serial household id */
szIdBuf = QmpDataInpGetSerialIdFldName(hDataInp, &qres);
ulIdStart = QmpDataInpGetSerialIdStart(hDataInp, &qres);
FieldTrim1 = QmpDataInpGetSerialIdTrim(hDataInp, &qres);

/* now remove the household id fields ( still in */
/* prototype record ) */
QmpDataInpRemSerialId( hDataInp, "SerialId", &qres );
```

QmpDataInpGetUserIdFldName

GETS THE USER-DEFINED HOUSEHOLD ID FIELD NAME.

Syntax

```
const char* QmpDataInpGetUserIdFldName ( MpHnd in_hDataInp,
                                         QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the name of the field that holds the user-defined household ID.

Notes

See Also

`QmpDataInpGetUserIdTrim`, `QmpDataInpGetUserIdPicture`.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );
QmpDataInpAddUserId( hDataInp, "UserId",
                     "Address+ZipCode", QMS_FIELD_TRIM_BOTH, &qres );

/* now look at user household id */
szIdBuf = QmpDataInpGetUserIdFldName( hDataInp, &qres );
szIdBuf = QmpDataInpGetUserIdPicture( hDataInp, &qres );
FieldTrim1 = QmpDataInpGetUserIdTrim( hDataInp, &qres );

/* now remove the household id fields ( still */
/* in prototype record ) */
QmpDataInpRemUserId( hDataInp, "UserId", &qres );
```

QmpDataInpGetUserIdPicture

GETS THE FIELD EXPRESSION PICTURE USED TO CONSTRUCT A USER-DEFINED HOUSEHOLD ID.

Syntax

```
const char* QmpDataInpGetUserIdPicture ( MpHnd in_hDataInp,
                                         QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the expression string (picture) used to generate a user-defined household ID.

Notes

See Also

[QmpDataInpGetUserIdTrim](#), [QmpDataInpGetUserIdFldName](#).

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );
QmpDataInpAddUserId( hDataInp, "UserId",
                      "Address+ZipCode", QMS_FIELD_TRIM_BOTH, &qres );

/* now look at user household id */
szIdBuf = QmpDataInpGetUserIdFldName( hDataInp, &qres );
szIdBuf = QmpDataInpGetUserIdPicture( hDataInp, &qres );
FieldTrim1 = QmpDataInpGetUserIdTrim( hDataInp, &qres );

/* now remove the household id fields */
/* ( still in prototype record ) */
QmpDataInpRemUserId( hDataInp, "UserId", &qres );
```

QmpDataInpGetUserIdTrim

RETURNS THE TRIM PROPERTY ASSOCIATED WITH A USER-DEFINED HOUSEHOLD ID FIELD.

Syntax

```
QMS_FIELD_TRIM QmpDataInpGetUserIdTrim ( MpHnd in_hDataInp,
                                         QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
out_pResult
    Result code. Output.
```

Return Value

The possible return values are:

QMS_FIELD_TRIM_NONE
QMS_FIELD_TRIM_LEFT
QMS_FIELD_TRIM_RIGHT
QMS_FIELD_TRIM_BOTH

Notes

The trim property is used to trim string field values used to construct the household ID value.

See Also

QmpDataDestSetInclUniques, **QmpDataInpGetUserIdFldName**.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );
QmpDataInpAddUserId( hDataInp, "UserId",
                     "Address+ZipCode", QMS_FIELD_TRIM_BOTH, &qres );

/* now look at user household id */
szIdBuf = QmpDataInpGetUserIdFldName( hDataInp, &qres );
szIdBuf = QmpDataInpGetUserIdPicture( hDataInp, &qres );
FieldTrim1 = QmpDataInpGetUserIdTrim( hDataInp, &qres );

/* now remove the household id fields */
/* ( still in prototype record ) */
QmpDataInpRemUserId( hDataInp, "UserId", &qres );
```

QmpDataInpIsValid

TESTS WHETHER A DATA INPUT PHASE IS VALID.

Syntax

```
QBOOL QmpDataInpIsValid ( MpHnd in_hDataInp, QRESULT*  
                           out_pResult );
```

in_hDataInp

Handle to data input. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if data input phase is valid, else QFALSE.

Notes

QmpDataInpIsValid tests whether a data input phase is valid or not. A valid data input phase is one that is ready to start.

Example

See example on [page 286](#).

QmpDataInpRegEveryNthRecFunc

REGISTER THE DATA INPUT PHASE TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpDataInpRegEveryNthRecFunc ( MpHnd in_hDataInp,
                                    QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hDataInp
    Handle to data input. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The data input phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

See example on [page 286](#).

QmpDataInpRemDataSrc

REMOVES SPECIFIED DATA SOURCE FROM THE DATA INPUT.

Syntax

```
MpHnd QmpDataInpRemDataSrc ( MpHnd in_hDataInp, long  
    in_lDataSourceID, QRESULT* out_pResult );
```

in_hDataInp

Handle to data input. *Input*.

in_lDataSourceID

Data source ID of the data source to be removed. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to removed data source if successful, or `NULL` if there is an error.

Notes

This function removes the specified data source from the data input. The data source itself is not deleted, but the reference to it in the data input phase is removed. The application passes the ID of the data source as a parameter.

See Also

[QmpDataSrcGetID](#)

Example

See example on [page 286](#).

QmpDataInpRemIdxKey

REMOVES SPECIFIED INDEX KEY FROM THE DATA INPUT.

Syntax

```
MpHnd QmpDataInpRemIdxKey ( MpHnd in_hDataInp, MpHnd  
    in_hIdxKey, QRESULT* out_pResult );
```

in_hDataInp
Handle to data input. *Input*.

in_hIdxKey
Handle to the index key. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns handle to removed index key if successful, or NULL if there is an error.

Notes

`QmpDataInpRemIdxKey` removes a specified index key from the collection of index keys in the data input phase. The index key is *not* deleted—only its membership in the collection is removed. This function removes index keys that were originally added with `QmpDataInpUseIdxKey`.

Example

See example on [page 286](#).

QmpDataInpRemSerialId

REMOVES THE VALUES IN SERIAL HOUSEHOLD ID FIELDS.

Syntax

```
void QmpDataInpRemSerialId( MpHnd in_ hDataInp, const char*
    in_szFldName, QRESULT* out_pResult );

in_hDataInp
    Handle to data input phase. Input.
in_szFldName
    Name of serial household ID field. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpDataInpRemSerialId removes the value in a serial household ID field.

See Also

QmpDataInpAddSerialId.

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );
QmpDataInpAddSerialId( hDataInp, "SerialId", 1,
    QMS_FIELD_TRIM_BOTH, &qres );

/* now look at serial household id */
szIdBuf = QmpDataInpGetSerialIdFldName(hDataInp, &qres);
ulIdStart = QmpDataInpGetSerialIdStart(hDataInp, &qres);
FieldTrim1 = QmpDataInpGetSerialIdTrim(hDataInp, &qres);

/* now remove the household id fields ( still in */
/* prototype record ) */
QmpDataInpRemSerialId( hDataInp, "SerialId", &qres );
```

QmpDataInpRemUserId

REMOVES THE VALUES FROM USER-DEFINED HOUSEHOLD ID FIELDS.

Syntax

```
void QmpDataInpRemUserId( MpHnd in_ hDataInp, const char*
    in_szFldName, QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
in_szFldName
    Name of user-defined household ID field to remove values from. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpDataInpRemUserId removes the values from the specified user-defined household ID field in all records.

See Also

[QmpDataInpAddUserId](#).

Example

```
QmpPhaseSetRecProto( hDataInp, hProtoRec, &qres );
QmpDataInpAddUserId( hDataInp, "UserId",
    "Address+ZipCode", QMS_FIELD_TRIM_BOTH, &qres );

/* now look at user household id */
szIdBuf = QmpDataInpGetUserIdFldName( hDataInp, &qres );
szIdBuf = QmpDataInpGetUserIdPicture( hDataInp, &qres );
FieldTrim1 = QmpDataInpGetUserIdTrim( hDataInp, &qres );

/* now remove the household id fields ( still in */
/* prototype record ) */
QmpDataInpRemUserId( hDataInp, "UserId", &qres );
```

QmpDataInpSetNthRecInterval

SETS DATA INPUT NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpDataInpSetNthRecInterval (MpHnd in_hDataInp, long  
                                in_lInterval, QRESULT* out_pResult );
```

in_hDataInp
Handle to data input. *Input*.

in_lInterval
Data input Nth record event interval. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the data input phase Nth record event interval.

The data input phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the data input phase processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

[QmpDataInpRegEveryNthRecFunc](#), [QmpDataInpSetNthRecInterval](#).

Example

See example on [page 286](#).

QmpDataInpUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpDataInpUnregEveryNthRecFunc (MpHnd in_hDataInp,
                                      QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hDataInp
    Handle to data input. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function unregisters the specified event handler from the data input phase, so that the event handler will no longer be notified of every Nth record processed.

The data input phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

See example on [page 286](#).

QmpDataInpUseDataLstSvc

SPECIFIES THE DATA LIST SERVICE OBJECT TO BE USED BY DATA INPUT.

Syntax

```
void QmpDataInpUseDataLstSvc( MpHnd in_hDataInp, MpHnd  
                             in_hDataLstSvc, QRESULT* out_pResult );  
  
in_hDataInp  
Handle to data input. Input.  
  
in_hDataLstSvc  
Handle to data list service. Input.  
  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

`QmpDataInpUseDataLstSvc` specifies the data list service object to be used by data input. The data input phase uses the data list service (and its contained data lists) during the processing of the input records. The data input phase delegates to the data list service the responsibility of figuring out (for each record read in) exactly which data list the record belongs to. The data list service tells the data input which data list the record belongs to, and data input sets a field in the DITR which contains the data list ID of the record.

See Also

See the data list and data list service functions for more information on how data lists work.

Example

See example on [page 286](#).

QmpDataInpUseIdxKey

SPECIFIES THE INDEX KEY OBJECT TO BE USED BY THE DATA INPUT.

Syntax

```
void QmpDataInpUseIdxKey ( MpHnd in_hDataInp, MpHnd
                           in_hIdxKey, QRESULT* out_pResult );
in_hDataInp
    Handle to data input. Input.
in_hIdxKey
    Handle to index key. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpDataInpUseIdxKey specifies the index key object to be used by the data input phase. If any of the index key definitions use the Soundex transformation, an additional Soundex field will be appended to the DITR. This field will contain the Soundex value of another field. Later in the job, the additional Soundex field will be used in constructing ordered record tables.

If the application needs to find out how many index keys are being used by the data input phase, it can call **QmpDataInpGetIdxKeyCnt**. If the application needs to remove some index keys from the data input phase, it can call **QmpDataInpRemIdxKey**.

Example

See example on [page 286](#).

Function Class: QmpDataLst*

DATA LIST FUNCTIONS.

Creating a Data List

Minimally, an application must create a data list service and a single default data list, which specifies what to do with unassigned records. This minimal configuration of data lists in the data list service will guarantee that the data list service is able to assign all records to at least one list.

The application must create a default data list as the first data list created. It will be inserted into the data list service in a position reserved specifically for it.

Defining the Rules for Accepting a Record into a Data List

A source data list will accept records from any data source whose source ID is registered with the data list. Call `QmpDataLstSrcAddSrcID` to register (add) source IDs with the data list.

Fieldval data lists use a string expression for determining which records are accepted into the list. The expression is a combination of a field name from the prototype record, a comparator (see “[QMS_FLDEVAL_OPER](#)” on page 88), and some type of value. Call one of the `QmpDataLstFldSetExpr*` functions to set this string expression.

Default data lists by definition accept any records that are not assigned to a source or fieldval data list. There are no acceptance rules for default data lists.

Quick Reference

Function	Description	Page
QmpDataLstDefCreate	Creates a default data list.	313
QmpDataLstDefGetAction	Gets default data list action to be performed.	314
QmpDataLstDefSetAction	Sets default data list action to be performed.	315
QmpDataLstDestroy	Destroys a data list. Can be called for any data list type.	316
QmpDataLstFldCreate	Creates a field evaluation data list.	317
QmpDataLstFldFillExpr	Fills expression string from a fieldval data list.	318
QmpDataLstFldSetExprDate	Sets expression in this data list to a date object value.	319
QmpDataLstFldSetExprDateStruct	Sets expression in this data list to a QDATESTRUCT value.	320
QmpDataLstFldSetExprFloat	Sets float-type expression in fieldval data list.	321

Function	Description	Page
QmpDataLstFldSetExprLong	Sets long-type expression in fieldval data list.	322
QmpDataLstFldSetExprString	Sets string-type expression in fieldval data list.	323
QmpDataLstFldSetExprVar	Sets expression in this data list to a variant object value.	324
QmpDataLstFldSetExprVarStruct	Sets expression in this data list to a QVARSTRUCT value.	329
QmpDataLstGetCnt	Gets number of members of this data list. Can be called for any data list type.	330
QmpDataLstGetExprNumChar	Returns the field evaluation substring character count.	331
QmpDataLstGetExprStart	Returns the field evaluation substring start position.	332
QmpDataLstGetID	Gets data list ID. Can be called for any data list type.	333
QmpDataLstGetIntraLstMat	Gets Intra-list match property. Can be called for any data list type.	334
QmpDataLstGetOutputPref	Gets Output Preference. Can be called for any data list type.	335
QmpDataLstGetPriority	Gets Priority. Can be called for any data list type.	336
QmpDataLstIsMember	Tests whether a record is a member of the specified data list.	337
QmpDataLstIsValid	Tests whether the data list object is valid. Can be called for any data list type.	338
QmpDataLstSetExprSubStr	Sets expression substring for data list.	339
QmpDataLstSetIntraLstMat	Sets Intra-list match property. Can be called for any data list type.	341
QmpDataLstSetOutputPref	Sets Output Preference. Can be called for any data list type.	342
QmpDataLstSetPriority	Sets Priority. Can be called for any data list type.	343
QmpDataLstSrcAddSrcID	Adds a Source ID to a source data list.	344
QmpDataLstSrcCreate	Creates a source data list.	345
QmpDataLstSrcFillSrcIDs	Fills Source IDs in a source data list.	346
QmpDataLstSrcGetSrcIDCnt	Gets number of Source IDs in a source data list.	347

QmpDataLstDefCreate

CREATES A DEFAULT DATA LIST.

Syntax

```
MpHnd QmpDataLstDefCreate( const char* in_pszName, QRESULT*  
                           out_pResult );  
  
in_pszName  
    Name of data list. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to data list if successful, or `NULL` if there is an error.

Notes

This function creates a default data list (a list of type `QMS_DATLST_TYPE_DEFAULT`).

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

See example on [page 324](#).

QmpDataLstDefGetAction

GETS DEFAULT DATA LIST ACTION TO BE PERFORMED.

Syntax

```
QMS_DATLST_ACTION QmpDataLstDefGetAction( MpHnd in_hDataLst,  
                                         QRESULT* out_pResult );
```

in_hDataLst

Handle to data list. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns recommended list action if successful. Possible values are:

QMS_DATLST_ACTION_YES	Record is a member of specified data list.
QMS_DATLST_ACTION_IGNORE_RECORD	Ignore the record.
QMS_DATLST_ACTION_STOP_EXECUTE	Stop execution.

QMS_DATLST_ACTION_NONE is returned if an error occurs.

Notes

This function gets the list action to be performed. This function operates only on data lists of type QMS_DATLST_TYPE_DEFAULT.

Example

See example on [page 324](#).

QmpDataLstDefSetAction

SETS DEFAULT DATA LIST ACTION TO BE PERFORMED.

Syntax

```
void QmpDataLstDefSetAction( MpHnd in_hDataLst,
                            QMS_DATLST_ACTION in_Action, QRESULT* out_pResult );
```

in_hDataLst

Handle to data list. *Input*.

in_Action

Action to perform. *Input*.

Possible actions are:

QMS_DATLST_ACTION_YES	Record is a member of specified data list.
-----------------------	--

QMS_DATLST_ACTION_IGNORE_RECORD	Ignore the record.
---------------------------------	--------------------

QMS_DATLST_ACTION_STOP_EXECUTE	Stop execution.
--------------------------------	-----------------

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Sets default data list action to be performed. Only for data lists of type QMS_DATLST_TYPE_DEFAULT.

The application can tell the default data list what action to perform in case all other data lists fail to claim the record, and the default data list has a chance to examine the record. The options are:

- QMS_DATLST_ACTION_YES

This means that the default data list will accept the record

- QMS_DATLST_ACTION_IGNORE_RECORD

The client of the data list will ignore the record, and it will not use it.

- QMS_DATLST_ACTION_STOP_EXECUTION

The client of the data list will stop execution of its logic, and the program will go on to the next step.

Example

See example on [page 324](#).

QmpDataLstDestroy

DESTROYS A DATA LIST. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
void QmpDataLstDestroy( MpHnd in_hDataLst, QRESULT*  
                        out_pResult );  
  
in_hDataLst  
    Handle to data list. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

Destroys a data list.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

Example

See example on [page 353](#).

QmpDataLstFldCreate

CREATES A FIELD EVALUATION DATA LIST.

Syntax

```
MpHnd QmpDataLstFldCreate( const char* in_pszName, QRESULT*  
    out_pResult );
```

in_pszName
Name of data list. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns handle to data list if successful, or `NULL` if there is an error.

Notes

This function creates a field evaluation data list (a list of type `QMS_DATLST_TYPE_FIELDVAL`).

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

See example on [page 324](#).

QmpDataLstFldFillExpr

FILLS EXPRESSION STRING FROM A FIELDVAL DATA LIST.

Syntax

```
void QmpDataLstFldFillExpr( MpHnd in_hDataLst, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

io_szBuffer
Expression buffer. *Input, Output*.

in_lSize
Client-allocated size of buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function fills an expression string from a data list of type
QMS_DATLST_TYPE_FIELDVAL.

Example: “LastName = Smith”

Example

See example on [page 324](#).

QmpDataLstFldSetExprDate

SETS EXPRESSION IN THIS DATA LIST TO A DATE OBJECT VALUE.

Syntax

```
void QmpDataLstFldSetExprDate( MpHnd in_hDataLst, const char*
                               in_szFieldName, QMS_FLDEVAL_OPER in_Operator, MpHnd
                               in_hDate, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

in_szFieldName
Record field name. *Input*.

in_Operator
Operator. *Input*.

in_hDate
Handle to date object containing date value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data lists of type
QMS_DATLST_TYPE_FIELDVAL.

The application must set an expression for every data list of type
QMS_DATLST_TYPE_FIELDVAL. The expression is a combination of a field
name from the prototype record, a comparator (see “[QMS_FLDEVAL_OPER](#)”
[on page 88](#)), and a date object. When this data list examines a record for
possible membership, it will evaluate the actual fields of the record against
this expression. If the value of the expression is QTRUE, the record is accepted
as a member of the data list.

Example

See example on [page 324](#).

QmpDataLstFldSetExprDateStruct

SETS EXPRESSION IN THIS DATA LIST TO A QDATESTRUCT VALUE.

Syntax

```
void QmpDataLstFldSetExprDateStruct( MpHnd in_hDataLst, const
    char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator,
    QDATESTRUCT* in_pDateStruct, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

in_szFieldName
Record field name. *Input*.

in_Operator
Operator. *Input*.

in_pDateStruct
Pointer to the QDATESTRUCT structure containing a date value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data lists of type
QMS_DATLST_TYPE_FIELDVAL.

The application must set an expression for every data list of type
QMS_DATLST_TYPE_FIELDVAL. The expression is a combination of a field
name from the prototype record, a comparator (see “[QMS_FLDEVAL_OPER](#)”
[on page 88](#)), and a QDATESTRUCT structure. When this data list examines a
record for possible membership, it will evaluate the actual fields of the record
against this expression. If the value of the expression is QTRUE, the record is
accepted as a member of the data list.

Example

See example on [page 324](#).

QmpDataLstFldSetExprFloat

SETS FLOAT-TYPE EXPRESSION IN FIELDVAL DATA LIST.

Syntax

```
void QmpDataLstFldSetExprFloat(MpHnd in_hDataLst, const char*
                                in_szFieldName, QMS_FLDEVAL_OPER in_Operator, float
                                in_fValue, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_fValue
Float constant to compare against. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Sets float-type expression in data list of type QMS_DATLST_TYPE_FIELDVAL.

The application must set an expression for every data list of type QMS_DATLST_TYPE_FIELDVAL. If the expression is on a floating point field, the application should set a floating point expression. The expression is a combination of a field name from the prototype records, a comparator (see “[QMS_FLDEVAL_OPER](#)” on page 88), and a floating point constant. When this data list examines a record for possible membership, it will evaluate the actual fields of the record against this expression. If the value of the expression is QTRUE, the record is accepted as a member of the data list.

Example: “Height < 6.15”

Example

See example on [page 324](#).

QmpDataLstFldSetExprLong

SETS LONG-TYPE EXPRESSION IN FIELDVAL DATA LIST.

Syntax

```
void QmpDataLstFldSetExprLong( MpHnd in_hDataLst, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, long
    in_lValue, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_lValue
Long constant to compare against. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Sets long-type expression in data list of type QMS_DATLST_TYPE_FIELDVAL.

The application must set an expression for every data list of type QMS_DATLST_TYPE_FIELDVAL. If the expression is on a long field, the application should set a long expression. The expression is a combination of a field name from the prototype records, a comparator (see “[QMS_FLDEVAL_OPER](#)” on page 88), and a long constant. When this data list examines a record for possible membership, it will evaluate the actual fields of the record against this expression. If the value of the expression is QTRUE, the record is accepted as a member of the data list.

Example: “Age > 50”

Example

See example on [page 324](#).

QmpDataLstFldSetExprString

SETS STRING-TYPE EXPRESSION IN FIELDVAL DATA LIST.

Syntax

```
void QmpDataLstFldSetExprString( MpHnd in_hDataLst, const
                                char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator, const
                                char* in_szValue, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_szValue
String constant to compare against. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Sets string-type expression in data list of type QMS_DATLST_TYPE_FIELDVAL. The application must set an expression for every data list of type QMS_DATLST_TYPE_FIELDVAL. If the expression is on a string field, the application should set a string expression. The expression is a combination of a field name from the prototype records, a comparator (see “[QMS_FLDEVAL_OPER](#) on page 88”), and a string constant. When this data list examines a record for possible membership, it will evaluate the actual fields of the record against this expression. If the value of the expression is QTRUE, the record is accepted as a member of the data list.

Example: “LastName = Smith”

Example

See example on [page 324](#).

QmpDataLstFldSetExprVar

SETS EXPRESSION IN THIS DATA LIST TO A VARIANT OBJECT VALUE.

Syntax

```
void QmpDataLstFldSetExprVar( MpHnd in_hDataLst, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, MpHnd
    in_hVar, QRESULT* out_pResult );

in_hDataLst
    Handle to data list. Input.
in_szFieldName
    Record field name. Input.
in_Operator
    Operator. Input.
in_hVar
    Handle of variant containing constant value. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function is valid only for data lists of type
QMS_DATLST_TYPE_FIELDVAL.

The application must set an expression for every data list of type
QMS_DATLST_TYPE_FIELDVAL. The expression is a combination of a field
name from the prototype record, a comparator (see “[QMS_FLDEVAL_OPER](#)
on page 88), and a variant. When this data list examines a record for possible
membership, it will evaluate the actual fields of the record against this
expression. If the value of the expression is QTRUE, the record is accepted as a
member of the data list.

Example

```
long lVal;          /* counter, index into lists, etc. */
long lSrcIDArray[2]; /* array of data source ids */
char szExprBuf[100]; /* buffer for strings */
QMS_DATLST_ACTION action; /* data list action */
QVARSTRUCT varStruct;
QDATESTRUCT dateStruct;

/* declare data list service and data lists */
QmpDeclHnd( hDataListSvc );
QmpDeclHnd( hDataListDefault );
QmpDeclHnd( hDataListSrc );
QmpDeclHnd( hDataListFldVal );
QmpDeclHnd( hRec );
QmpDeclHnd( hDate );
QmpDeclHnd( hVar );
QmpDeclHnd( hFound );
```

```

/* create data list service and data lists */
hRec          = QmpRecCreate( &qres );
hDataListSvc  = QmpDataLstSvcCreate( &qres );
hDataListDefault = QmpDataLstDefCreate( "Default", &qres );
hDataListSrc   = QmpDataLstSrcCreate( "Source", &qres );
hDataListFldVal = QmpDataLstFldCreate( "FldVal", &qres );

/* Set/Get properties of the data list service */
QmpRecAdd( hRec, "Fname", &qres );/* string field */
QmpRecAdd( hRec, "Lname", &qres );/* string field */
QmpRecAdd( hRec, "Addr", &qres );/* string field */
QmpRecAddByTypePicture( hRec, "Date", QMS_VARIANT_DATE, 0, "mmm dd, ccyy",
    &qres );/* date field */
QmpRecAddByType( hRec, "Zip", QMS_VARIANT_LONG, 5, 0, &qres );/* long field */
QmpRecAddByType( hRec, "Height", QMS_VARIANT_FLOAT, 4, 2, &qres );
/* float, like 6.12 */

QmpDataLstSvcSetRecProto( hDataListSvc, hRec, &qres );
hFound = QmpDataLstSvcGetRecProto( hDataListSvc, &qres );
lVal = QmpRecGetFldCnt( hFound, &qres );
if ( lVal != 6 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Prototype record is not
        correct" );

/* Add the data lists to the data list service */
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListDefault, &qres );
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListSrc, &qres );
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListFldVal, &qres );

/* Set/Get some properties of the default-type data list. */
QmpDataLstDefSetAction( hDataListDefault, QMS_DATLST_ACTION_IGNORE_RECORD,
    &qres );
action = QmpDataLstDefGetAction( hDataListDefault, &qres );
if ( action != QMS_DATLST_ACTION_IGNORE_RECORD )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong default action" );

/* Set/Get some properties of the source-type data list. */
QmpDataLstSrcAddSrcID( hDataListSrc, 1, &qres );
QmpDataLstSrcAddSrcID( hDataListSrc, 2, &qres );
lVal = QmpDataLstSrcFillSrcIDs( hDataListSrc, 2, lSrcIDArray, &qres );
if ( lVal != 2 && lSrcIDArray[0] != 1 && lSrcIDArray[0] != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data sources in
        data list" );
lVal = QmpDataLstSrcGetSrcIDCnt( hDataListSrc, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data sources in
        data list" );

/* Set/Get some properties of the FldVal-type data list. */
QmpDataLstFldSetExprLong( hDataListFldVal, "Zip", QMS_FLDEVAL_OPER_EQUAL,
    80301, &qres );
QmpDataLstFldSetExprFloat( hDataListFldVal, "Height",
    QMS_FLDEVAL_OPER_GREATER, 6.0, &qres );
QmpDataLstFldSetExprString( hDataListFldVal, "Lname", QMS_FLDEVAL_OPER_LESS,
    "Smith", &qres );
QmpDataLstFldFillExpr( hDataListFldVal, szExprBuf, sizeof(szExprBuf), &qres
);
if ( QMS_STRICTCMP( szExprBuf, "Lname < Smith" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* Set/Get some properties of the FldVal-type data list using variants. */
hVar = QmpVarCreate( &qres );
QmpVarSetString( hVar, "John", &qres );
QmpDataLstFldSetExprVar( hDataListFldVal, "Fname", QMS_FLDEVAL_OPER_LESS,
    hVar, &qres );

```

QmpDataLstFldSetExprVar

```
QmpDataLstFldFillExpr( hDataListFldVal, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Fname < John" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
QmpVarFillVarStruct( hVar, &varStruct, &qres );
QmpDataLstFldSetExprVarStruct( hDataListFldVal, "Fname",
    QMS_FLDEVAL_OPER_EQUAL, &varStruct, &qres );
QmpDataLstFldFillExpr( hDataListFldVal, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Fname = John" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* Set/Get some properties of the FldVal-type data list using dates. */
hDate = QmpDateCreate( &qres );
QmpDateSetPicture( hDate, "jan 01, 2000", "mmm dd, ccyy", &qres );
QmpDateFillDateStruct( hDate, &dateStruct, &qres );
QmpDataLstFldSetExprDate( hDataListFldVal, "Date", QMS_FLDEVAL_OPER_GREATER,
    hDate, &qres );
QmpDataLstFldFillExpr( hDataListFldVal, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Date > 20000101" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
QmpDataLstFldSetExprDateStruct( hDataListFldVal, "Date",
    QMS_FLDEVAL_OPER_EQUAL, &dateStruct, &qres );
QmpDataLstFldFillExpr( hDataListFldVal, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Date = 20000101" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* See if a data list can claim a record (this is non-standard usage. */
/* Normally, you would allow the data list service to do this ). */
QmpRecSetCharPtrFldByName( hRec, "Fname", "James", &qres );
QmpRecSetCharPtrFldByName( hRec, "Lname", "Jones", &qres );
QmpRecSetCharPtrFldByName( hRec, "Addr", "123 Main", &qres );
QmpRecSetLongFldByName( hRec, "Zip", 80301, &qres );
QmpRecSetFloatFldByName( hRec, "Height", 6.0, &qres );
QmpRecSetDateFldByName( hRec, "Date", &dateStruct, &qres );
action = QmpDataLstIsMember( hDataListFldVal, hRec, -1, &qres );
if ( action != QMS_DATLST_ACTION_YES )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list failed to claim
        record" );
action = QmpDataLstIsMember( hDataListSrc, hRec, 3, &qres );
if ( action != QMS_DATLST_ACTION_NO )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list incorrectly claimed
        record" );
action = QmpDataLstIsMember( hDataListDefault, hRec, -1, &qres );
if ( action != QMS_DATLST_ACTION_IGNORE_RECORD )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list should have ignored
        record" );

/* Get number of members of a Data List. */
lVal = QmpDataLstGetCnt( hDataListFldVal, &qres );
if ( lVal != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of data
        list" );
lVal = QmpDataLstGetCnt( hDataListDefault, &qres );
if ( lVal != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of data
        list" );

/* Pass this record through the data list service */
/* (the way data lists SHOULD be used to evaluate records) */
hFound = QmpDataLstSvcDetMem( hDataListSvc, hRec, 3, &action, &qres );
if ( hFound != hDataListFldVal )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong data list claimed this
        record" );
hFound = QmpDataLstSvcDetMem( hDataListSvc, hRec, 1, &action, &qres );
if ( hFound != hDataListSrc )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong data list claimed this
        record" );
lVal = QmpDataLstGetCnt( hDataListFldVal, &qres );
if ( lVal != 2 )
```

```

        QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of data
        list" );
lVal = QmpDataLstGetCnt( hDataListSrc, &qres );
if ( lVal != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of data
        list" );

/* Get/Set priority, output preference, and intra-list match properties */
/* for a data list */
QmpDataLstSetOutputPref( hDataListFldVal, QMS_DATLST_OUTPUT_PREF_INCLUDE, &qres
);
if ( QmpDataLstGetOutputPref( hDataListFldVal, &qres ) !=
    QMS_DATLST_OUTPUT_PREF_INCLUDE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Output preference is wrong" );
QmpDataLstSetPriority( hDataListFldVal, 10, &qres );
if ( QmpDataLstGetPriority( hDataListFldVal, &qres ) != 10 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list priority is wrong" );
QmpDataLstSetIntraLstMat( hDataListFldVal, QFALSE, &qres );
if ( QmpDataLstGetIntraLstMat( hDataListFldVal, &qres ) != QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list intra-list match is
        wrong" );

/* Get number of data lists in data list service */
lVal = QmpDataLstSvcGetDataLstCnt( hDataListSvc, &qres );
if ( lVal != 3 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data lists" );

/* find out if data list service is valid */
if ( QmpDataLstSvcIsValid( hDataListSvc, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid data list service" );

/* find index of a data list inside the collection of data lists in the */
/* data list service. */
lVal = QmpDataLstSvcGetDataLstID( hDataListSvc, hDataListFldVal, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Index of data list is wrong" );

/* find index of a data list inside the collection of data lists in the */
/* data list service. */
lVal = QmpDataLstSvcGetDataLstNameID( hDataListSvc, "FldVal", &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Index of data list is wrong" );

/* find a data list in the data list service. */
hFound = NULL;
hFound = QmpDataLstSvcFindDataLst( hDataListSvc, hDataListFldVal, &qres );
if ( hFound != hDataListFldVal )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Failed to find data list" );

/* Get a data list by passing its data list id (the index of its location */
/* in the collection) */
hFound = QmpDataLstSvcGetDataLst( hDataListSvc, 2, &qres );
if ( hFound != hDataListFldVal )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Failed to find data list" );

/* Get data list ids for all of the data lists */
lVal = QmpDataLstGetID( hDataListDefault, &qres );
if ( lVal != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong default data list id" );
lVal = QmpDataLstGetID( hDataListSrc, &qres );
if ( lVal != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong source data list id" );
lVal = QmpDataLstGetID( hDataListFldVal, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong FldVal data list id" );

/* Determine validity of the data lists */
if ( QmpDataLstIsValid( hDataListDefault, &qres ) == QFALSE )

```

QmpDataLstFldSetExprVar

```
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid default data list" );
if ( QmpDataLstIsValid( hDataListSrc, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid source data list" );
if ( QmpDataLstIsValid( hDataListFldVal, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid FldVal data list" );

/* Dump contents of the Data List and Data List Service. */
remove( "dump.log" );
QmpDumpOpen( hDataListDefault, "dump.log", &qres );
QmpDumpDump( hDataListDefault, QTRUE, &qres );
QmpDumpClose( hDataListDefault, &qres );
QmpDumpOpen( hDataListSvc, "dump.log", &qres );
QmpDumpDump( hDataListSvc, QTRUE, &qres );
QmpDumpClose( hDataListSvc, &qres );

/* Clear the data list service */
QmpDataLstSvcClear( hDataListSvc, &qres );
lVal = QmpDataLstSvcGetDataLstCnt( hDataListSvc, &qres );
if ( lVal != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data lists" );

/* Destroy data list service and data lists */
QmpRecDestroy( hRec, &qres );
QmpVarDestroy( hVar, &qres );
QmpDateDestroy( hDate, &qres );
QmpDataLstDestroy( hDataListDefault, &qres );
QmpDataLstDestroy( hDataListSrc, &qres );
QmpDataLstDestroy( hDataListFldVal, &qres );
QmpDataLstSvcDestroy( hDataListSvc, &qres );
```

QmpDataLstFldSetExprVarStruct

SETS EXPRESSION IN THIS DATA LIST TO A QVARSTRUCT VALUE.

Syntax

```
void QmpDataLstFldSetExprVarStruct( MpHnd in_hDataLst, const
                                    char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator,
                                    QVARSTRUCT* in_pVarStruct, QRESULT* out_pResult );
in_hDataLst
Handle to data list. Input.
in_szFieldName
Record field name. Input.
in_Operator
Operator. Input.
in_pVarStruct
Pointer to the QVARSTRUCT structure containing a constant value. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

This function is valid only for data lists of type
QMS_DATLST_TYPE_FIELDVAL.

The application must set an expression for every data list of type
QMS_DATLST_TYPE_FIELDVAL. The expression is a combination of a field
name from the prototype record, a comparator (see “[QMS_FLDEVAL_OPER](#)
[on page 88](#)”), and a QVARSTRUCT structure. When this data list examines a
record for possible membership, it will evaluate the actual fields of the record
against this expression. If the value of the expression is QTRUE, the record is
accepted as a member of the data list.

Example

See example on [page 324](#).

QmpDataLstGetCnt

GETS NUMBER OF MEMBERS OF THIS DATA LIST. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
long QmpDataLstGetCnt( MpHnd in_hDataLst, QRESULT* out_pResult
) ;
```

in_hDataLst
Handle to data list. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns number of records belonging to the data list if successful, or -1 if there is an error.

Notes

Gets number of members of this data list. Can be called for any data list type. After the data input phase has run, all of the input records will have been assigned to data lists. Each data list keeps track of the number of members it has claimed. This function retrieves that number.

Example

See example on [page 353](#).

QmpDataLstGetExprNumChar

RETURNS THE FIELD EVALUATION SUBSTRING CHARACTER COUNT.

Syntax

```
int QmpDataLstGetExprNumChar ( MpHnd in_hDataLst, QRESULT*
                               out_pResult );
in_hDataLst
    Handle to data list. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns number of substring characters.

Notes

QmpDataLstGetExprNumChar is valid only for data lists of type QMS_DATLST_TYPE_FIELDVAL.

QmpDataLstGetExprNumChar returns the length of the field substring defined by **QmpDataLstSetExprSubStr**.

See Also

QmpDataLstSetExprSubStr.

Example

```
/* return the field evaluation substring start position */

/* return the object's starting position */
iFldExprStart = QmpDataLstGetExprStart( hDataLstTbl, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataLstGetExprNumChar( hDataLstTbl, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataLstSetExprSubStr( hDataLstTbl, 2, 4, &qres );

/* return the object's starting position */
iFldExprStart = QmpDataLstGetExprStart( hDataLstTbl, &qres );
if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataLstGetExprNumChar( hDataLstTbl, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataLstGetExprStart

RETURNS THE FIELD EVALUATION SUBSTRING START POSITION.

Syntax

```
int QmpDataLstGetExprStart ( MpHnd in_ hDataLst, QRESULT*
    out_pResult );
in_hDataLst
    Handle to data list. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the substring starting position.

Notes

QmpDataLstGetExprStart is valid only for data lists of type QMS_DATLST_TYPE_FIELDVAL.

QmpDataLstGetExprStart returns the starting position of the field substring defined by **QmpDataLstSetExprSubStr**.

See Also

QmpDataLstSetExprSubStr

Example

```
/* return the field evaluation substring start position */

/* return the object's starting position */
iFldExprStart = QmpDataLstGetExprStart( hDataLstTbl, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataLstGetExprNumChar( hDataLstTbl, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataLstSetExprSubStr( hDataLstTbl, 2, 4, &qres );

/* return the object's starting position */
iFldExprStart = QmpDataLstGetExprStart( hDataLstTbl, &qres );
if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataLstGetExprNumChar( hDataLstTbl, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataLstGetID

GETS DATA LIST ID. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
long QmpDataLstGetID( MpHnd in_hDataLst, QRESULT* out_pResult
);
```

in_hDataLst
Handle to data list. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the index of the specified data list in its containing data list service if successful, or -1 if there is an error.

Notes

Gets data list ID. Can be called for any data list type. The ID of the data list is identical to the index of the data list as it exists in the data list service. The default data list must *always* be the first data list which is added to the data list service. This means that it will always be in position 0, and have an index of 0.

Additional data lists of type FieldVal and Source can be added to the data list service, and their data list IDs (i.e., their indexes in the collection of data lists in the data list service) will be 1 or greater.

Example

See example on [page 353](#).

QmpDataLstGetIntraLstMat

GETS INTRA-LIST MATCH PROPERTY. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
QBOOL QmpDataLstGetIntraLstMat( MpHnd in_hDataLst, QRESULT*  
                                out_pResult );  
  
in_hDataLst  
        Handle to data list. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns QTRUE if intra-list matches are enabled, else QFALSE.

Notes

`QmpDataLstGetIntraLstMat` gets a data list's intra-list match property.

See Also

`QmpDataLstSetIntraLstMat`

Example

See example on [page 353](#).

QmpDataLstGetOutputPref

GETS OUTPUT PREFERENCE. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
QMS_DATLST_OUTPUT_PREF QmpDataLstGetOutputPref( MpHnd
    in_hDataLst, QRESULT* out_pResult );
in_hDataLst
    Handle to data list. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns data list output preference. Possible values are:

QMS_DATLST_OUTPUT_PREF_INCLUDE	This is an “inclusion” list.
QMS_DATLST_OUTPUT_PREF_SUPPRESS	This is a “suppression” list.

QMS_DATLST_OUTPUT_INCLUDE is returned if an error occurs.

Notes

QmpDataLstGetOutputPref gets the output preference. This function can be called for any data list type.

See Also

QmpDataDestSetOutputTypesWithBitmask,
QmpDataDestSetInclUniques, **QmpDataDestSetSuppUniques**,
QmpDataDestSetInclMasters, **QmpDataDestSetSuppMasters**,
QmpDataDestSetInclSubords, **QmpDataDestSetSuppSubords**,
QmpDataDestGetInclUniques, **QmpDataDestGetSuppUniques**,
QmpDataDestGetInclMasters

Example

See example on [page 353](#).

QmpDataLstGetPriority

GETS PRIORITY. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
long QmpDataLstGetPriority( MpHnd in_hDataLst, QRESULT*  
    out_pResult );  
  
in_hDataLst  
    Handle to data list. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns data list priority if successful, or -1 if there is an error.

Notes

`QmpDataLstGetPriority` gets the priority of the data list. The priority is set by calling `QmpDataLstSetPriority`.

Example

See example on [page 353](#).

QmpDataLstIsMember

TESTS WHETHER A RECORD IS A MEMBER OF THE SPECIFIED DATA LIST.

Syntax

```
QMS_DATLST_ACTION QmpDataLstIsMember( MpHnd in_hDataLst,
                                         MpHnd in_hRec, QRESULT* out_pResult );
```

in_hDataLst

Handle to data list. *Input*.

in_hRec

Handle to record to be evaluated. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns data list status of specified record if successful. Can return a variety of responses:

QMS_DATLST_ACTION_YES	Record is a member of specified data list.
QMS_DATLST_ACTION_NO	Record is NOT a member of specified data list.
QMS_DATLST_ACTION_RETRY	Start asking from the beginning (probably because the application tweaked the data list collection).
QMS_DATLST_ACTION_IGNORE_RECORD	Ignore the record. Used only by the "default" style data list.
QMS_DATLST_ACTION_STOP_EXECUTE	Stop execution. Used only by the "default" style data list.

QMS_DATLST_ACTION_NONE is returned if an error occurs.

Notes

Tests whether a record is a member of the data list. Normally, only the data list service would call this function ([QmpDataLstSvcDetMem](#)), but individual applications can call it if they need to. This function is passed a record with values in it, and a data source ID that the record comes from. The data list examines the record using the type of criteria it has been set up to use (field evaluation, data source origin, or default action), and either accepts or does not accept the record.

Example

See example on [page 353](#).

QmpDataLstIsValid

TESTS WHETHER THE DATA LIST OBJECT IS VALID. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
QBOOL QmpDataLstIsValidMpHnd in_hDataLst, QRESULT* out_pResult
) ;

in_hDataLst
Handle to data list. Input.

out_pResult
Result code. Output.
```

Return Value

Returns QTRUE if data list object is valid, else QFALSE.

Notes

Tests whether the data list object is valid. Can be called for any data list type. Each type of data list has a different validity check. All data lists must have a valid output preference and name. The default data list must have a valid action that it will perform if it encounters a record. The source type of data list makes sure that at least one source id has been registered with it. The FieldVal data list makes sure that a valid expression has been given to it.

Example

See example on [page 353](#).

QmpDataLstSetExprSubStr

SETS EXPRESSION SUBSTRING FOR DATA LIST.

Syntax

```
void QmpDataLstSetExprSubStr ( MpHnd in_hDataLst, int
                               in_iCharStartPos, int in_iNumChars, QRESULT* out_pResult );
in_hDataLst
    Handle to data list. Input.
in_iCharStartPos
    Start position. Input.
in_iNumChars
    Number of chars to consider. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpDataLstSetExprSubStr is valid only for data lists of type **QMS_DATSRC_TYPE_FUNCTOR** and **QMS_DATSRC_TYPE_TABLE**.

QmpDataLstSetExprSubStr adds a substring component to the expression created by one of the **QmpDataLstFldSetExpr*** functions. Specifically, it modifies the expression to evaluate a substring of the field. For example, in the expression “FirstName = ‘Fred’”, this function makes it possible to evaluate a substring of the FirstName field.

This function is valid for any of the data list expression types (date, float, variant, etc.). If the field is not of type string, the value is converted to a string. Be aware of how the field type in question represents its values internally, and how that will affect its conversion to a string.

See Also

QmpDataLstGetExprStart, **QmpDataLstGetExprNumChar**.

Example

```
/* return the field evaluation substring start position */

/* return the object's starting position */
iFldExprStart = QmpDataLstGetExprStart( hDataLst, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataLstGetExprNumChar( hDataLst, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataLstSetExprSubStr( hDataLst, 2, 4, &qres );
```

QmpDataLstSetExprSubStr

```
/* return the object's starting position */
iFldExprStart = QmpDataLstGetExprStart( hDataLst, &qres );
if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataLstGetExprNumChar( hDataLst, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataLstSetIntraLstMat

SETS INTRA-LIST MATCH PROPERTY. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
void QmpDataLstSetIntraLstMat( MpHnd in_hDataLst, QBOOL
    in_bIntraListMatch, QRESULT* out_pResult );
in_hDataLst
    Handle to data list. Input.
in_bIntraListMatch
    Intra-list match property. Set this to QTRUE to allow comparisons between
    data lists, QFALSE to forbid intra-list comparisons. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

If your application is using a data source that requires no internal record comparisons, you should create a data list of type “source” to contain these records, then set the *in_bIntraListMatch* argument in **QmpDataLstSetIntraLstMat** to forbid intra-list comparisons.

See Also

[QmpDataLstGetIntraLstMat](#)

Example

See example on [page 353](#).

QmpDataLstSetOutputPref

SETS OUTPUT PREFERENCE. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
void QmpDataLstSetOutputPref( MpHnd in_hDataLst,
    QMS_DATLST_OUTPUT_PREF in_OutputPref, QRESULT* out_pResult
);
```

in_hDataLst

Handle to data list. *Input*.

in_OutputPref

Output preference. *Input*.

QMS_DATLST_OUTPUT_PREF_INCLUDE	This is an “inclusion” list.
--------------------------------	------------------------------

QMS_DATLST_OUTPUT_PREF_SUPPRESS	This is a “suppression” list.
---------------------------------	-------------------------------

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function is used to set whether records from the specified data list are to be included or suppressed in output reports and tables.

This feature is useful in the processing of the DMA Pander file. The application can set the Pander file data list to be a suppression list, which assigns it the highest priority in terms of duplicate ranking. Any records from other data lists to records in this file are assigned a lower priority, and are eliminated during Duplicate Ranking. Since the “survivor” is a record from the suppression list, it is not output to the final report or table, thus eliminating records that match the Pander file.

Example

```
/* Set output preference for data list, where: */
/*      QMS_DATLST_OUTPUT_PREF_INCLUDE = 0,      */
/*      (this is an “inclusion” list)           */
/*      QMS_DATLST_OUTPUT_PREF_SUPPRESS = 1       */
/*      (this is a “suppression” list)          */
QmpDataLstSetOutputPref( hDataListSource
    QMS_DATLST_OUTPUT_PREF_SUPPRESS, pResult );
```

QmpDataLstSetPriority

SETS PRIORITY. CAN BE CALLED FOR ANY DATA LIST TYPE.

Syntax

```
void QmpDataLstSetPriority( MpHnd in_hDataLst, long  
                           in_lPriority, QRESULT* out_pResult );  
  
in_hDataLst  
    Handle to data list. Input.  
  
in_lPriority  
    Priority. May be any integer greater than 0 and less than 1000. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

This function sets the priority level of the specified data list.

Example

See example on [page 353](#).

QmpDataLstSrcAddSrcID

ADDS A SOURCE ID TO A SOURCE DATA LIST.

Syntax

```
void QmpDataLstSrcAddSrcID( MpHnd in_hDataLst, long  
                           in_lSourceID, QRESULT* out_pResult );
```

in_hDataLst
Handle to data list. *Input*.

in_lSourceID
Source ID. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function adds a source ID to a source data list.

One or more source IDs can be added to this type of data list. If, during data input, a record from one of the registered source IDs is encountered by this data list, the data list will claim that record as a member.

Example

See example on [page 324](#).

QmpDataLstSrcCreate

CREATES A SOURCE DATA LIST.

Syntax

```
MpHnd QmpDataLstSrcCreate( const char* in_pszName, QRESULT*  
    out_pResult );
```

in_pszName
Name of data list. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns handle to data list if successful, or `NULL` if there is an error.

Notes

This function creates a source data list (a list of type `QMS_DATLST_TYPE_SOURCE`).

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

See example on [page 324](#).

QmpDataLstSrcFillSrcIDs

FILLS SOURCE IDs IN A SOURCE DATA LIST.

Syntax

```
long QmpDataLstSrcFillSrcIDs( MpHnd in_hDataLst, long  
    in_lSize, long* io_lSourceIDArray, QRESULT* out_pResult );  
  
in_hDataLst  
    Handle to data list. Input.  
  
in_lSize  
    Size of output array. Input.  
  
io_lSourceIDArray  
    Client allocated array of source IDs. Input, Output.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns number of source IDs put into output array if successful, or -1 if there is an error.

Notes

This function fills source IDs in a source data list.

The application supplies an array to this function dimensioned (at least) as big as the number of source IDs that have been registered with the data list. This function populates that array with the source IDs that have been given to it.

Example

See example on [page 324](#).

QmpDataLstSrcGetSrcIDCnt

GETS NUMBER OF SOURCE IDs IN A SOURCE DATA LIST.

Syntax

```
long QmpDataLstSrcGetSrcIDCnt( MpHnd in_hDataLst, QRESULT*  
                               out_pResult );  
  
in_hDataLst  
    Handle to data list. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns the number of source IDs if successful, or -1 if there is an error.

Notes

This function returns the number of source IDs that have been added to a data list of type QMS_DATLST_TYPE_SOURCE .

Example

See example on [page 324](#).

Function Class: QmpDataLstSvc*

DATA LIST SERVICE FUNCTIONS.

The minimal configuration for the data list service contains a default data list. Without this, the data list service is not usable (the Centrus Merge/Purge library will generate an error if the data list service is used without a default data list). After adding a default data list, the application may add as many non-default data lists (source and fieldval types) as needed.

Note: All the records in a dupe group will not necessarily belong to a single data list. In fact, in a job that is working with multi-buyer data, it is extremely likely that records in a dupe group will each belong to a different data list. However, it is the case that a single record will always belong to a single data list. An application could unintentionally define the data list expressions such that a record would fail to fall into any non-default data list. Centrus Merge/Purge will assign all such records to the default data list.

Quick Reference

Function	Description	Page
QmpDataLstSvcAddDataLst	Adds a data list to the collection of data lists.	351
QmpDataLstSvcClear	Clears a data list service.	352
QmpDataLstSvcCreate	Creates a data list service.	353
QmpDataLstSvcDestroy	Destroys a data list service.	357
QmpDataLstSvcDetMem	Determines which data list a record belongs to.	358
QmpDataLstSvcFindDataLst	Finds a data list by passing in a reference to a data list.	360
QmpDataLstSvcGetDataLst	Gets a data list by passing in the data list ID.	361
QmpDataLstSvcGetDataLstCnt	Gets number of data lists in the data list service.	362
QmpDataLstSvcGetDataLstID	Gets a data list ID by providing the data list handle.	363
QmpDataLstSvcGetDataLstNameID	Gets a data list ID by providing the data list name.	364
QmpDataLstSvcGetInterLstMatByld	Gets the inter-list match property in the data list service by id.	365
QmpDataLstSvcGetInterLstMatByName	Gets the inter-list match property in the data list service by name.	366
QmpDataLstSvcGetRecProto	Gets the prototype record from the data list service.	367
QmpDataLstSvclsValid	Tests whether a data list service is valid.	368
QmpDataLstSvcSetInterLstMatByld	Sets the inter-list match property in the data list service by id.	369

Function	Description	Page
QmpDataLstSvcSetInterLstMatByName	Sets the inter-list match property in the data list service by name.	371
QmpDataLstSvcSetRecProto	Sets the prototype record in the data list service.	367

QmpDataLstSvcAddDataLst

ADDS A DATA LIST TO THE COLLECTION OF DATA LISTS.

Syntax

```
void QmpDataLstSvcAddDataLst( MpHnd in_hDataLstSvc, MpHnd
    in_hDataLst, QRESULT* out_pResult );
in_hDataLstSvc
    Handle to data list service.Input.
in_hDataLst
    Handle to the data list. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Adds a data list to the collection of data lists. At a minimum, the application must add a default data list to the data list service. It can optionally add other data lists (of type Source and FieldVal). However, it must first add the default data list. One and only one default data list can be added to the data list Service. Multiple data lists of type Source and FieldVal can be added.

See Also

[QmpDataLstSvcFindDataLst](#), [QmpDataLstSvcGetDataLst](#),
[QmpDataLstSvcGetDataLstID](#), [QmpDataLstSvcGetDataLstCnt](#)

Example

See example on [page 353](#).

QmpDataLstSvcClear

CLEARSA DATA LIST SERVICE.

Syntax

```
void QmpDataLstSvcClear( MpHnd in_hDataLstSvc, QRESULT*  
                        out_pResult );  
  
in_hDataLstSvc  
    Handle to data list service. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

This function removes references to all data list objects from the specified data list service.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

See example on [page 353](#).

QmpDataLstSvcCreate

Creates a data list service.

Syntax

```
MpHnd QmpDataLstSvcCreate( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns handle to the data list service if successful, or `NULL` if there is an error.

Notes

The data list service performs data list-related functions for clients of data lists, such as the data input phase and the record matching phase. The data input phase uses this service to do initial data list assignment of new records. The preprocessed data source object uses this service to do data list assignment of reference data records at the beginning of an application using reference tables.

The data list service is an aggregate of one or more data list objects. A data list object contains the specification of the criteria for assigning a record membership to a single data list. An application typically uses several data lists in a single data list service. The data list service manages and coordinates the activities of the collection of data lists contained in it.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
{
long lVal;                      /* counter, index into lists, etc.*/
long lSrcIDArray[2];            /* array of data source ids      */
char szExprBuf[100];           /* buffer for strings          */
QMS_DATLST_ACTION action;     /* data list action             */

/* declare data list service and data lists */
QmpDeclHnd( hDataListSvc );
QmpDeclHnd( hDataListDefault );
QmpDeclHnd( hDataListSrc );
QmpDeclHnd( hDataListFldVal );
QmpDeclHnd( hRec );
QmpDeclHnd( hFound );

/* create data list service and data lists */
hRec        = QmpRecCreate( &qres );
hDataListSvc = QmpDataLstSvcCreate( &qres );
hDataListDefault = QmpDataLstCreate( QMS_DATLST_TYPE_DEFAULT, "Default", &qres );
hDataListSrc   = QmpDataLstCreate( QMS_DATLST_TYPE_SOURCE,   "Source", &qres );
hDataListFldVal = QmpDataLstCreate( QMS_DATLST_TYPE_FIELDVAL, "FieldVal",
&qres );

/* Set/Get properties of the data list service */
```

QmpDataLstSvcCreate

```
QmpRecAdd( hRec, "Fname", &qres );
/* string field */
QmpRecAdd( hRec, "Lname", &qres );
/* string field */
QmpRecAdd( hRec, "Addr", &qres );
/* string field */
QmpRecAddByType( hRec, "Zip", QMS_VARIANT_LONG, 5, 0, &qres );
/* long field */
QmpRecAddByType( hRec, "Height", QMS_VARIANT_FLOAT, 4, 2, &qres );
/* float, like 6.12 */
QmpDataLstSvcSetRecProto( hDataListSvc, hRec, &qres );
hFound = QmpDataLstSvcGetRecProto( hDataListSvc, &qres );
lVal = QmpRecGetFldCnt( hFound, &qres );
if ( lVal != 5 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Prototype record is not
correct" );

/* Add the data lists to the data list service */
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListDefault, &qres );
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListSrc, &qres );
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListFldVal, &qres );

/* Set/Get some properties of the default-type data list. */
QmpDataLstSetAction( hDataListDefault, QMS_DATLST_ACTION_IGNORE_RECORD, &qres
);
action = QmpDataLstGetAction( hDataListDefault, &qres );
if ( action != QMS_DATLST_ACTION_IGNORE_RECORD )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong default action" );

/* Set/Get some properties of the source-type data list. */
QmpDataLstAddSrcID( hDataListSrc, 1, &qres );
QmpDataLstAddSrcID( hDataListSrc, 2, &qres );
lVal = QmpDataLstFillSrcIDs( hDataListSrc, 2, lSrcIDArray, &qres );
if ( lVal != 2 && lSrcIDArray[0] != 1 && lSrcIDArray[0] != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data
sources in data list" );
lVal = QmpDataLstGetSrcIDCnt( hDataListSrc, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data
sources in data list" );

/* Set/Get some properties of the fieldval-type data list. */
QmpDataLstSetExprLong( hDataListFldVal, "Zip", QMS_FLDEVAL_OPER_EQUAL, 80301,
&qres );
QmpDataLstSetExprFloat( hDataListFldVal, "Height", QMS_FLDEVAL_OPER_GREATER,
6.0, &qres );
QmpDataLstSetExprString( hDataListFldVal, "Lname", QMS_FLDEVAL_OPER_LESS,
"Smith", &qres );
QmpDataLstFillExpr( hDataListFldVal, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Lname < Smith" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* See if a data list can claim a record (this is non-standard usage. */
/* Normally, you would allow the data list service to do this ). */
QmpRecSetCharPtrFldByName( hRec, "Fname", "James", &qres );
QmpRecSetCharPtrFldByName( hRec, "Lname", "Jones", &qres );
QmpRecSetCharPtrFldByName( hRec, "Addr", "123 Main", &qres );
QmpRecSetLongFldByName( hRec, "Zip", 80301, &qres );
QmpRecSetFloatFldByName( hRec, "Height", 6.0, &qres );
action = QmpDataLstIsMember( hDataListFldVal, hRec, 3, &qres );
if ( action != QMS_DATLST_ACTION_YES )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list failed to claim
record" );
action = QmpDataLstIsMember( hDataListSrc, hRec, 3, &qres );
if ( action != QMS_DATLST_ACTION_NO )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list incorrectly
claimed record" );
action = QmpDataLstIsMember( hDataListDefault, hRec, 3, &qres );
```

```

if ( action != QMS_DATLST_ACTION_IGNORE_RECORD )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list should have
ignored record" );

/* Get number of members of a Data List. */
lVal = QmpDataLstGetCnt( hDataListFldVal, &qres );
if ( lVal != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of
data list" );
lVal = QmpDataLstGetCnt( hDataListDefault, &qres );
if ( lVal != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members
of data list" );

/* Pass this record through the data list service */
/* (the way data lists SHOULD be used to evaluate records) */
hFound = QmpDataLstSvcDetMem( hDataListSvc, hRec, 3, &action, &qres );
if ( hFound != hDataListFldVal )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong data list claimed
this record" );
hFound = QmpDataLstSvcDetMem( hDataListSvc, hRec, 1, &action, &qres );
if ( hFound != hDataListSrc )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong data list claimed
this record" );
lVal = QmpDataLstGetCnt( hDataListFldVal, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of
data list" );
lVal = QmpDataLstGetCnt( hDataListSrc, &qres );
if ( lVal != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "wrong number of members of
data list" );

/* Get/Set priority, output preference, and intra-list match properties */
/* for a data list */
QmpDataLstSetOutputPref( hDataListFldVal, QMS_DATLST_OUTPUT_PREF_INCLUDE, &qres
);
if ( QmpDataLstGetOutputPref( hDataListFldVal, &qres ) !=
QMS_DATLST_OUTPUT_PREF_INCLUDE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Output preference is
wrong" );
QmpDataLstSetPriority( hDataListFldVal, 10, &qres );
if ( QmpDataLstGetPriority( hDataListFldVal, &qres ) != 10 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list priority is
wrong" );
QmpDataLstSetIntraLstMat( hDataListFldVal, QFALSE, &qres );
if ( QmpDataLstGetIntraLstMat( hDataListFldVal, &qres ) != QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data list intra-list match
is wrong" );

/* Get number of data lists in data list service */
lVal = QmpDataLstSvcGetDataLstCnt( hDataListSvc, &qres );
if ( lVal != 3 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data
lists" );

/* find out if data list service is valid */
if ( QmpDataLstSvcIsValid( hDataListSvc, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid data list service"
);

/* find index of a data list inside the collection of data lists in */
/* the data list service. */
lVal = QmpDataLstSvcGetIdxOf( hDataListSvc, hDataListFldVal, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Index of data list is
wrong" );

```

QmpDataLstSvcCreate

```
/* find a data list in the data list service. */
hFound = NULL;
hFound = QmpDataLstSvcFindDataLst( hDataListSvc, hDataListFldVal, &qres );
if ( hFound != hDataListFldVal )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Failed to find data
list" );

/* Get a data list by passing its data list id (the index of its */
/* location in the collection) */
hFound = QmpDataLstSvcGetDataLst( hDataListSvc, 2, &qres );
if ( hFound != hDataListFldVal )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Failed to find data
list" );

/* Get data list ids for all of the data lists */
lVal = QmpDataLstGetID( hDataListDefault, &qres );
if ( lVal != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong default data list=
id" );
lVal = QmpDataLstGetID( hDataListSrc, &qres );
if ( lVal != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong source data list
id" );
lVal = QmpDataLstGetID( hDataListFldVal, &qres );
if ( lVal != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong fieldval data list
id" );

/* Determine validity of the data lists */
if ( QmpDataLstIsValid( hDataListDefault, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid default data
list" );
if ( QmpDataLstIsValid( hDataListSrc, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid source data
list" );
if ( QmpDataLstIsValid( hDataListFldVal, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Invalid fieldval data
list" );

/* Dump contents of the Data List and Data List Service. */
remove( "dump.log" );
QmpDataLstDump( hDataListDefault, QTRUE, "dump.log", &qres );
QmpDataLstSvcDump( hDataListSvc, QTRUE, "dump.log", &qres );

/* Clear the data list service */
QmpDataLstSvcClear( hDataListSvc, &qres );
lVal = QmpDataLstSvcGetDataLstCnt( hDataListSvc, &qres );
if ( lVal != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of data
lists" );

/* Destroy data list service and data lists */
QmpRecDestroy( hRec, &qres );
QmpDataLstDestroy( hDataListDefault, &qres );
QmpDataLstDestroy( hDataListSrc, &qres );
QmpDataLstDestroy( hDataListFldVal, &qres );
QmpDataLstSvcDestroy( hDataListSvc, &qres );
}
```

QmpDataLstSvcDestroy

DESTROYS A DATA LIST SERVICE.

Syntax

```
void QmpDataLstSvcDestroy( MpHnd in_hDataLstSvc, QRESULT*
                           out_pResult );
in_hDataLstSvc
    Handle to data list service. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Destroys a data list service.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

Example

See example on [page 353](#).

QmpDataLstSvcDetMem

DETERMINES WHICH DATA LIST A RECORD BELONGS TO.

Syntax

```
MpHnd QmpDataLstSvcDetMem( MpHnd in_hDataLstSvc, MpHnd  

in_hRec, QMS_DATLST_ACTION* out_Action, QRESULT*  

out_pResult );
```

in_hDataLstSvc

Handle to data list service. *Input*.

in_hRec

Handle to the prototype record. *Input*.

out_Action

Record info or recommended action. *Output*

Possible values are:

QMS_DATLST_ACTION_YES	Record is a member of specified data list.
QMS_DATLST_ACTION_NO	Record is NOT a member of specified data list.
QMS_DATLST_ACTION_RETRY	Start asking from the beginning (data list collection has changed).
QMS_DATLST_ACTION_IGNORE_RECORD	Ignore the record. Used only by the “default” style data list.
QMS_DATLST_ACTION_STOP_EXECUTE	Stop execution. Used only by the “default” style data list.

out_pResult

Result code. *Output*.

Return Value

If successful, returns handle to the data list that specified record belongs to. If there is an error, NULL is returned.

Notes

Determines which data list a record belongs to. This function is called by the data input phase for each record read in from the data sources. It is also called by **QmpDataSrcPrepare** (for preprocessed data sources) to update the data list membership of preprocessed data source records. The data list service asks each of its data lists in order to examine the record and either claim the record as a member, or allow the next data list to have a “chance” at it. Note that the data list at position 0 (the default data list) is the last data list which is allowed to examine the record.

For example, if 5 data lists have been added to the data list service, the order in which they would be allowed to examine the records would be: 1, 2, 3, 4, 0. The default data list always occupies position 0. If only a default data list has been added to the data list service, then it will be the only data list to examine the record for membership.

Example

See example on [page 353](#).

QmpDataLstSvcFindDataLst

FINDS A DATA LIST BY PASSING IN A REFERENCE TO A DATA LIST.

Syntax

```
MpHnd QmpDataLstSvcFindDataLst( MpHnd in_hDataLstSvc, MpHnd  
          in_hDataLst, QRESULT* out_pResult );  
  
in_hDataLstSvc  
    Handle to data list service. Input.  
  
in_hDataLst  
    Handle to the data list. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to a data list if successful, or NULL if there is an error.

Notes

Finds a data list by passing in a reference to a data list. This function allows an application to confirm that a given data list exists in the data list service. The application passes the data list handle to this function. This function finds the data list in its collection and returns that same handle, or NULL if it cannot find it.

Example

See example on [page 353](#).

QmpDataLstSvcGetDataLst

GETS A DATA LIST BY PASSING IN THE DATA LIST ID.

Syntax

```
MpHnd QmpDataLstSvcGetDataLst( MpHnd in_hDataLstSvc, long  
                                in_DataLstID, QRESULT* out_pResult );  
  
in_hDataLstSvc  
        Handle to data list service. Input.  
  
in_DataLstID  
        data list ID. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns handle to a data list if successful, or `NULL` if there is an error.

Notes

Gets a data list by passing in the data list ID. The application can pass the data list ID (the index of the data list in the data list service). The data list service looks up the data list in this position, and returns its handle to the application.

Example

See example on [page 353](#).

QmpDataLstSvcGetDataLstCnt

GETS NUMBER OF DATA LISTS IN THE DATA LIST SERVICE.

Syntax

```
long QmpDataLstSvcGetDataLstCnt( MpHnd in_hDataLstSvc ,
                                 QRESULT* out_pResult );
in_hDataLstSvc
    Handle to data list service. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns number of data lists in the data list service if successful, or -1 if there is an error.

Notes

Gets number of data lists in the data list service. For example, if the application has added one default data list, two source data lists, and three FieldVal data lists, the number returned by this function would be 6.

Example

See example on [page 353](#).

QmpDataLstSvcGetDataLstID

GETS A DATA LIST ID BY PROVIDING THE DATA LIST HANDLE.

Syntax

```
long QmpDataLstSvcGetDataLstID( MpHnd in_hDataLstSvc, MpHnd  
                               in_hDataLst, QRESULT* out_pResult );
```

in_hDataLstSvc
Handle to data list service. *Input*.

in_hDataLst
Data list handle. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the index of the data list in the data list service successful, or -1 if there is an error.

Notes

This function gets the position (index) of a data list in the data list service collection, given a handle to the data list. This is the same as getting the data list ID.

Example

See example on [page 324](#).

QmpDataLstSvcGetDataLstNameID

GETS A DATA LIST ID BY PROVIDING THE DATA LIST NAME.

Syntax

```
long QmpDataLstSvcGetDataLstNameID( MpHnd in_hDataLstSvc,
                                     const char* in_szDataLstName, QRESULT* out_pResult );
in_hDataLstSvc
    Handle to data list service. Input.
in_szDataLstName
    Data list name. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the index of the data list in the data list service successful, or -1 if there is an error.

Notes

This function gets the position (index) of a data list in the data list service collection, given a name to the data list. This is the same as getting the data list ID.

Example

See example on [page 324](#).

QmpDataLstSvcGetInterLstMatById

GETS THE INTER-LIST MATCH PROPERTY IN THE DATA LIST SERVICE BY ID.

Syntax

```
QBOOL QmpDataLstSvcGetInterLstMatById( MpHnd in_hDataLstSvc,
    long in_lDataListId1, long in_lDataListId2, QRESULT*  
    out_pResult );
```

in_hDataLstSvc
Handle to data list service. *Input*.

in_lDataListId1
First data list ID. *Input*.

in_lDataListId2
Second data list ID. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the specified lists allow matches between them, else QFALSE.

If both data list IDs are the same, **QmpDataLstSvcGetInterLstMatById** returns the intra-list match property of that data list (are matches between records in that data list allowed).

See Also

QmpDataLstSvcGetInterLstMatByName.

Example

```
QBOOL bInterListMatch = QFALSE;
int i, j;

/* test existing inter-list property, should be QTRUE */
for ( i = 0; i < lNumLists; i++ ) {
    for ( j = 0; j < lNumLists; j++ ) {
        bInterListMatch = QmpDataLstSvcGetInterLstMatById(
            hDataListSvc, i, j, &qres );
        if ( bInterListMatch != QTRUE )
            QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong
                Inter-list match property" );
    }
}
/* then set each to false */
for ( i = 0; i < lNumLists; i++ ) {
    for ( j = 0; j < lNumLists; j++ ) {
        QmpDataLstSvcSetInterLstMatById( hDataListSvc, i, j,
            QFALSE, &qres );
    }
}
```

QmpDataLstSvcGetInterLstMatByName

GETS THE INTER-LIST MATCH PROPERTY IN THE DATA LIST SERVICE BY NAME.

Syntax

```
QBOOL QmpDataLstSvcGetInterLstMatByName( MpHnd
    in_hDataLstSvc, const char* in_szName1, const char*
    in_szName2, QRESULT* out_pResult );

in_hDataLstSvc
    Handle to data list service. Input.

in_szName1
    First data list name. Input.

in_szName2
    Second data list name. Input.

out_pResult
    Result code. Output.
```

Return Value

Returns QTRUE if the specified lists allow matches between them, else QFALSE.

If both data list names are the same,

QmpDataLstSvcGetInterLstMatByName returns the intra-list match property of that data list (are matches between records in that data list allowed).

See Also

[QmpDataLstSvcGetInterLstMatById](#).

Example

```
QBOOL bInterListMatch = QFALSE;
int i, j;

/* test existing inter-list property, should be QTRUE */
for ( i = 0; i < lNumLists; i++ ) {
    for ( j = 0; j < lNumLists; j++ ) {
        bInterListMatch = QmpDataLstSvcGetInterLstMatByName(
            hDataListSvc, "list1", "list2", &qres );
        if ( bInterListMatch != QTRUE )
            QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong
                Inter-list match property" );
    }
}
/* then set each to false */
for ( i = 0; i < lNumLists; i++ ) {
    for ( j = 0; j < lNumLists; j++ ) {
        QmpDataLstSvcSetInterLstMatById( hDataListSvc, i, j,
            QFALSE, &qres );
    }
}
```

QmpDataLstSvcGetRecProto

GETS THE PROTOTYPE RECORD FROM THE DATA LIST SERVICE.

Syntax

```
MpHnd QmpDataLstSvcGetRecProto( MpHnd in_hDataLstSvc,  
                                QRESULT* out_pResult );  
  
in_hDataLstSvc  
    Handle to data list service. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to record if successful, or `NULL` if there is an error.

Notes

The data list service uses the record prototype to help it work with any data lists of type `FieldVal`. This is the same record prototype as is given to the various Centrus Merge/Purge phases.

Example

See example on [page 353](#).

QmpDataLstSvclsValid

TESTS WHETHER A DATA LIST SERVICE IS VALID.

Syntax

```
QBOOL QmpDataLstSvclsValid ( MpHnd in_hDataLstSvc, QRESULT*  
    out_pResult );
```

in_hDataLstSvc

Handle to data list service. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if list service is valid, else QFALSE.

Notes

Tests whether a data list service is valid. A valid data list service is one that has a default data list, and whose other data lists are also valid. Essentially, this function invokes `QmpDataLstIsValid` for each contained data list.

Example

See example on [page 353](#).

QmpDataLstSvcSetInterLstMatById

SETS THE INTER-LIST MATCH PROPERTY IN THE DATA LIST SERVICE BY ID.

Syntax

```
void QmpDataLstSvcSetInterLstMatById ( MpHnd in_hDataLstSvc,
                                         long in_lDataListId1, long in_lDataListId2, QBOOL in_bMatch,
                                         QRESULT* out_pResult );
```

in_hDataLstSvc
Handle to data list service. *Input*.

in_lDataListId1
First data list ID. *Input*.

in_lDataListId2
Second data list ID. *Input*.

in_bMatch
QTRUE enables matching, QFALSE suppress it. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDataLstSvcSetInterLstMatById determines whether records from two data lists will be matched against each other.

If both data list IDs are set to the same value, this function sets the intra-list match property (will records from the same data list be matched against each other).

See Also

QmpDataLstSvcSetInterLstMatByName.

Example

```
QBOOL bInterListMatch = QFALSE;
int i, j;

/* test existing inter-list property, should be QTRUE */
for ( i = 0; i < lNumLists; i++ ) {
    for ( j = 0; j < lNumLists; j++ ) {
        bInterListMatch = QmpDataLstSvcGetInterLstMatById(
            hDataLstSvc, i, j, &qres );
        if ( bInterListMatch != QTRUE )
            QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong
                Inter-list match property" );
    }
}
/* then set each to false */
for ( i = 0; i < lNumLists; i++ ) {
```

```
QmpDataLstSvcSetInterLstMatById
```

```
    for ( j = 0; j < lNumLists; j++ ) {
        QmpDataLstSvcSetInterLstMatById( hDataListSvc, i, j,
            QFALSE, &qres );
    }
}
```

QmpDataLstSvcSetInterLstMatByName

SETS THE INTER-LIST MATCH PROPERTY IN THE DATA LIST SERVICE BY NAME.

Syntax

```
void QmpDataLstSvcSetInterLstMatByName ( MpHnd
                                         in_hDataLstSvc, const char* in_szName1, const char*
                                         in_szName2, QBOOL in_bMatch, QRESULT* out_pResult );

in_hDataLstSvc
    Handle to data list service. Input.
in_szName1
    First data list name. Input.
in_szName2
    Second data list name. Input.
in_bMatch
    QTRUE enables matching, QFALSE suppress matching. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpDataLstSvcSetInterLstMatByName determines whether records from two data lists will be matched against each other.

If both data list names are set to the same value, this function sets the intra-list match property (will records from the same data list be matched against each other).

See Also

QmpDataLstSvcSetInterLstMatById.

Example

```
QBOOL bInterListMatch = QFALSE;
int i, j;

/* test existing inter-list property, should be QTRUE */
for ( i = 0; i < lNumLists; i++ ) {
    for ( j = 0; j < lNumLists; j++ ) {
        bInterListMatch = QmpDataLstSvcGetInterLstMatById(
            hDataListSvc, i, j, &qres );
        if ( bInterListMatch != QTRUE )
            QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong
                Inter-list match property" );
    }
}
/* then set each to false */
for ( i = 0; i < lNumLists; i++ ) {
```

```
QmpDataLstSvcSetInterLstMatByName
```

```
    for ( j = 0; j < lNumLists; j++ ) {
        QmpDataLstSvcSetInterLstMatByName( hDataListSvc,
            "list1", "list2", QFALSE, &qres );
    }
}
```

QmpDataLstSvcSetRecProto

SETS THE PROTOTYPE RECORD IN THE DATA LIST SERVICE.

Syntax

```
void QmpDataLstSvcSetRecProto( MpHnd in_hDataLstSvc, MpHnd  
                               in_hRec, QRESULT* out_pResult );
```

in_hDataLstSvc
Handle to data list service. *Input*.

in_hRec
Handle to the prototype record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The data list service uses the record prototype to help it work with any data lists of type FieldVal. This is the same record prototype as is given to the various Centrus Merge/Purge phases.

Example

See example on [page 353](#).

Function Class: QmpDataSrc*

DATA SOURCE FUNCTIONS.

Quick Reference

Function		Page
QmpDataSrcClear	Clears a data source. Can be called for any data source type.	377
QmpDataSrcDestroy	Destroys a data source. Can be called for any data source type.	378
QmpDataSrcFillExpr	Fills Expression string from the data source.*	379
QmpDataSrcFillStatSamp	Fills statistical sampling properties.*	380
QmpDataSrcFuncCreate	Creates a functor-type data source.	381
QmpDataSrcGetExprNumChar	Returns the field evaluation substring character count.	382
QmpDataSrcGetExprStart	Returns the field evaluation substring start position.	383
QmpDataSrcGetFillFunc	Gets fill functor in specified data source.*	384
QmpDataSrcGetID	Gets data source ID. Can be called for any data source type.	385
QmpDataSrcGetRecProto	Gets the prototype record from the specified data source.*	386
QmpDataSrcIsNextRecUsed	Determines if the next record in the data source will pass input sampling (and be used).	387
QmpDataSrcIsValid	Determine whether data source object is valid. Can be called for any data source type.	388
QmpDataSrcPreCreate	Creates a preprocessed-type data source.	389
QmpDataSrcPrepare	Updates the data list membership field of the preprocessed table. Only for data sources of type QMS_DATSRC_TYPE_PREPRO.	390
QmpDataSrcSetExprDate	Sets expression in this data source to a date object value.	391
QmpDataSrcSetExprDateStruct	Sets expression in this data source to a QDATESTRUCT value.	392
QmpDataSrcSetExprFloat	Sets float-type expression in specified data source.*	393
QmpDataSrcSetExprLong	Sets long-type expression in data source.*	394
QmpDataSrcSetExprString	Sets string-type expression in specified data source.*	395
QmpDataSrcSetExprSubStr	Sets expression substring for a data source.	396
QmpDataSrcSetExprVar	Sets expression in this data source to a variant object value.	398

Function		Page
QmpDataSrcSetExprVarStruct	Sets expression in this data source to a QVARSTRUCT value.	399
QmpDataSrcSetFillFunc	Sets fill functor in specified data source.*	400
QmpDataSrcSetRecProto	Sets the data fields via an application-owned record.*	401
QmpDataSrcSetStatSamp	Sets statistical sampling properties.*	402
QmpDataSrcTblCreate	Creates a table-type data source.	403
QmpDataSrcUseFldMap	Gives a field map to the data source.	405
QmpDataSrcUseTbl	Specifies the input table object to be used by the data source.	406

QmpDataSrcClear

CLEAR A DATA SOURCE. CAN BE CALLED FOR ANY DATA SOURCE TYPE.

Syntax

```
void QmpDataSrcClear( MpHnd in_hDataSrc, QRESULT* out_pResult  
);
```

in_hDataSrc
Handle to data source. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets a data source back to its initial (blank) state.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpDataSrcClear(hDataSrc, pResult );
```

QmpDataSrcDestroy

DESTROYS A DATA SOURCE. CAN BE CALLED FOR ANY DATA SOURCE TYPE.

Syntax

```
void QmpDataSrcDestroy( MpHnd in_hDataSrc, QRESULT*  
                        out_pResult );  
  
in_hDataSrc  
    Handle to data source. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

Destroys a data source. Can be called for any data source type.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

Example

```
QmpDataSrcDestroy ( (MpHnd)*(MpHnd*)ele->data, &Result );
```

QmpDataSrcFillExpr

FILLS EXPRESSION STRING FROM THE DATA SOURCE.*

Syntax

```
void QmpDataSrcFillExpr( MpHnd in_hDataSrc, char* io_szBuffer,
                         long in_lSize, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

io_szBuffer
Expression buffer. *Input*.

in_lSize
Client allocated size of buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function fills the *io_szBuffer* parameter with the expression currently set for this data source.

Example

```
QmpDataSrcFillExpr(hDataSrc, szBuffer, lSize, pResult );
```

QmpDataSrcFillStatSamp

FILLS STATISTICAL SAMPLING PROPERTIES.*

Syntax

```
void QmpDataSrcFillStatSamp( MpHnd in_hDataSrc,
    QMS_STAT_SAMPLING* out_StatSampling, QRESULT* out_pResult
);
```

in_hDataSrc

Handle to data source. *Input*.

out_StatSampling

Statistical sampling structure. *Output*.

`QMS_STAT_SAMPLING` is a structure the data source object uses. It defines the pattern of records to return to the data input phase. In **every member**, a value of 0 means “this part not specified”.

<code>lMaxRecords</code>	Maximum number of records to input from a data source.
--------------------------	--

<code>lFirstRecordNum</code>	Record number of first record to return from a data source.
------------------------------	---

<code>lLastRecordNum</code>	Record number of last record to return from a data source.
-----------------------------	--

<code>lInterval</code>	Sampling interval (return every “nth” record).
------------------------	--

<code>lGroupSize</code>	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record).
-------------------------	---

out_pResult

Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type `QMS_DATSRC_TYPE_FUNCTOR` and `QMS_DATSRC_TYPE_TABLE`.

`QmpDataSrcFillStatSamp` fills a statistical sampling structure with an object’s statistical sampling properties.

`QMS_STAT_SAMPLING` is a structure that a client can fill and pass to the data source object that uses it. It defines the pattern of records to return to the data input phase. In every member, a value of 0 means “this part not specified”.

Example

```
QmpDataSrcFillStatSamp(hDataSrc, pStatSamplingStruct, pResult
);
```

QmpDataSrcFuncCreate

CREATES A FUNCTOR-TYPE DATA SOURCE.

Syntax

```
MpHnd QmpDataSrcFuncCreate( const char* in_pszName, long  
    in_lDataSourceID, QRESULT* out_pResult );
```

in_pszName

Name of data source. *Input*.

in_lDataSourceID

Data source ID. This ID must be unique for each data source. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to data source if successful, or NULL if there is an error.

Notes

This function creates a functor-type data source.

If the creation function fails, an error code is returned in the *out_pResult* parameter.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

See example on [page 403](#).

QmpDataSrcGetExprNumChar

RETURNS THE FIELD EVALUATION SUBSTRING CHARACTER COUNT.

Syntax

```
int QmpDataSrcGetExprNumChar ( MpHnd in_hDataSrc, QRESULT*
                               out_pResult );
in_hDataSrc
    Handle to the data source. Input.
out_pResult
    Result code. Output.
```

Return Value

Number of characters in the substring character count.

Notes

This function is valid only for data sources of type
QMS_DATSRC_TYPE_FUNCTOR or QMS_DATSRC_TYPE_TABLE.

QmpDataSrcGetExprNumChar returns the length of the field substring
defined by **QmpDataSrcSetExprSubStr**.

See Also

QmpDataSrcSetExprSubStr.

Example

```
/* return the field evaluation substring start position */
/* return the object's starting position */
iFldExprStart = QmpDataSrcGetExprStart( hDataSrcTbl, &qres );

if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataSrcGetExprNumChar( hDataSrcTbl, &qres );

if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataSrcSetExprSubStr( hDataSrcTbl, 2, 4, &qres );

/* return the object's starting position */
iFldExprStart = QmpDataSrcGetExprStart( hDataSrcTbl, &qres );

if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataSrcGetExprNumChar( hDataSrcTbl, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataSrcGetExprStart

RETURNS THE FIELD EVALUATION SUBSTRING START POSITION.

Syntax

```
int QmpDataSrcGetExprStart ( MpHnd in_hDataSrc, QRESULT*
    out_pResult );
in_hDataSrc
    Handle to data source. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the starting position in the field evaluation substring.

Notes

QmpDataSrcGetExprStart is valid only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

QmpDataSrcGetExprStart returns the starting position of the field substring defined by **QmpDataSrcSetExprSubStr**.

See Also

QmpDataSrcSetExprSubStr.

Example

```
/* return the field evaluation substring start position */

/* return the object's starting position */
iFldExprStart = QmpDataSrcGetExprStart( hDataSrcTbl, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataSrcGetExprNumChar( hDataSrcTbl, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataSrcSetExprSubStr( hDataSrcTbl, 2, 4, &qres );

/* return the object's starting position */
iFldExprStart = QmpDataSrcGetExprStart( hDataSrcTbl, &qres );
if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataSrcGetExprNumChar( hDataSrcTbl, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataSrcGetFillFunc

GETS FILL FUNCTOR IN SPECIFIED DATA SOURCE.*

Syntax

```
QMS_FILL_REC_FUNC QmpDataSrcGetFillFunc( MpHnd in_hDataSrc,  
                                         QRESULT* out_pResult );  
  
in_hDataSrc  
        Handle to data source. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns the fill function pointer if successful, or NULL if there is an error.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR.

A data source is a class that fully encapsulates a physical source of application records. One data source object will exist for every physical source of records. data sources will interact with data lists so that individual records in the data sources can be broken out or assigned to different “logical” data lists. The Centrus Merge/Purge library will always access records through their corresponding data source object. The application can give data sources to data input, record matcher, and other phases that use them.

Example

```
pFillFunction = QmpDataSrcGetFillFunc(hDataSrc, pResult );
```

QmpDataSrcGetID

GETS DATA SOURCE ID. CAN BE CALLED FOR ANY DATA SOURCE TYPE.

Syntax

```
long QmpDataSrcGetID( MpHnd in_hDataSrc, QRESULT* out_pResult  
);
```

in_hDataSrc
Handle to data source. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns data source ID if successful, or -1 if there is an error.

Notes

This function gets the data source ID.

Example

```
lDataSrcID = QmpDataSrcGetID (hDataSrc, pResult );
```

QmpDataSrcGetRecProto

GETS THE PROTOTYPE RECORD FROM THE SPECIFIED DATA SOURCE.*

Syntax

```
MpHnd QmpDataSrcGetRecProto( MpHnd in_hDataSrc, QRESULT*  
    out_pResult );
```

in_hDataSrc

Handle to data source. *Input*.

out_pResult

Result code. *Output*.

Return Value

Return handle to record if successful, or NULL if there is an error.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and
QMS_DATSRC_TYPE_TABLE.

Example

```
hProtoRec = QmpDataSrcGetRecProto(hDataSrc, pResult );
```

QmpDataSrcIsNextRecUsed

DETERMINES IF THE NEXT RECORD IN THE DATA SOURCE WILL PASS INPUT SAMPLING (AND BE USED).

Syntax

```
QBOOL QmpDataSrcIsNextRecUsed( MpHnd in_hDataSrc, QRESULT*  
    out_pResult );  
  
in_hDataSrc  
    Handle to data source. Input.  
out_pResult  
    Result code. Output.
```

Return Value

Returns QTRUE if the next record in the data source will pass input sampling, else QFALSE.

Notes

This function is valid only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

The functor may call this function to avoid reading in records that will be skipped due to sampling.

Example

```
QBOOL my_fill_func(MpHnd io_hRec, MpHnd in_hCaller) {  
    QRESULT qres; /* return code */  
    static QBOOL bFirst = QTRUE;  
    QBOOL bWillBeUsed = QTRUE;  
    static QmpDeclHnd( hTblInput );  
    char pszTempPathBuffer[QMS_MAX_PATH] = { "" };  
    if ( bFirst ) {  
        /* For profiling use the smaller dataset... 1000.dbf */  
        /* To really challenge the phases...42500.dbf */  
        /* For normal usage and creating the working set...4400.dbf */  
        printf( "Open CodeBase table:\n" );  
        sprintf( pszTempPathBuffer, "%s%s%s", pszDataPath, pszDelim, pszTableName  
    );  
    hTblInput = QmpTblCbCreate( "", pszTempPathBuffer, &qres );  
    QmpTblOpen( hTblInput, &qres );  
    QmpTblMoveFirst( hTblInput, &qres );  
    bFirst = QFALSE;  
} else {  
    QmpTblMoveBy( hTblInput, 1, &qres );  
}  
if ( QmpTblAtEOF( hTblInput, &qres ) ) {  
    QmpTblClose( hTblInput, &qres );  
    QmpTblDestroy( hTblInput, &qres );  
    return QFALSE;  
}  
if ( QmpDataSrcIsNextRecUsed( in_hCaller, &qres ) )  
    QmpTblFillRec( hTblInput, io_hRec, &qres );  
return(QTRUE);  
}
```

QmpDataSrcIsValid

DETERMINE WHETHER DATA SOURCE OBJECT IS VALID. CAN BE CALLED FOR ANY DATA SOURCE TYPE.

Syntax

```
QBOOL QmpDataSrcIsValid ( MpHnd in_hDataSrc, QRESULT*  
    out_pResult );
```

in_hDataSrc

Handle to data source. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if data source object is valid, else QFALSE.

Notes

This function determines whether a data source object is valid.

Example

```
bIsDataSrcValid = QmpDataSrcIsValid (hDataSrc, pResult);
```

QmpDataSrcPreCreate

CREATES A PREPROCESSED-TYPE DATA SOURCE.

Syntax

```
MpHnd QmpDataSrcPreCreate( const char* in_pszName, long  
                           in_lDataSourceID, QRESULT* out_pResult );
```

in_pszName

Name of data source. *Input*.

in_lDataSourceID

Data source ID. This ID must be unique for each data source. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to data source if successful, or NULL if there is an error.

Notes

This function creates a preprocessed-type data source.

If the creation function fails, an error code is returned in the *out_pResult* parameter.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

See example on [page 403](#).

QmpDataSrcPrepare

UPDATES THE DATA LIST MEMBERSHIP FIELD OF THE PREPROCESSED TABLE. ONLY FOR DATA SOURCES OF TYPE QMS_DATSRC_TYPE_PREPRO.

Syntax

```
void QmpDataSrcPrepare ( MpHnd in_hDataSrc, MpHnd
    in_hDataLstSvc, QBOOL in_bZeroDupeGroups, QBOOL
    in_bUpdateSourceId, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_hDataLstSvc
Handle to data list service. *Input*.

in_bZeroDupeGroups
If QTRUE, zero out dupe groups. *Input*.

in_bUpdateSourceId
If QTRUE, update source ID field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function's primary task is to update the QQDATLST augmented record fields in the table of a preprocessed data source. "Update" means assign data list membership to a record if it is unassigned, or assign a valid data list membership if the current list value is invalid.

If *in_bUpdateSourceId* = QTRUE, **QmpDataSrcPrepare** will ensure that QQSRCID field values are the same as the preprocessed data source ID.

Example

```
QmpDataSrcPrepare ( hPreProDataSrc, hDataListService, QTRUE,
    QTRUE, &qres );
```

QmpDataSrcSetExprDate

SETS EXPRESSION IN THIS DATA SOURCE TO A DATE OBJECT VALUE.

Syntax

```
void QmpDataSrcSetExprDate( MpHnd in_hDataSrc, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, MpHnd
    in_hDate, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_hDate
Handle to date object containing a date value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data sources of type
QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a date expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

See example on [page 403](#).

QmpDataSrcSetExprDateStruct

SETS EXPRESSION IN THIS DATA SOURCE TO A QDATESTRUCT VALUE.

Syntax

```
void QmpDataSrcSetExprDateStruct( MpHnd in_hDataSrc, const
    char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator,
    QDATESTRUCT* in_pDateStruct, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_pDateStruct
Pointer to a QDATESTRUCT structure containing a date value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data sources of type
QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a QDATESTRUCT expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

See example on [page 403](#).

QmpDataSrcSetExprFloat

SETS FLOAT-TYPE EXPRESSION IN SPECIFIED DATA SOURCE.*

Syntax

```
void QmpDataSrcSetExprFloat( MpHnd in_hDataSrc, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, float
    in_fValue, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_fValue
Float constant to compare against. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a float expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

```
QmpDataSrcSetExprFloat(hDataSrc, szFieldName, in_Operator,
    fValue, pResult );
```

QmpDataSrcSetExprLong

SETS LONG-TYPE EXPRESSION IN DATA SOURCE.*

Syntax

```
void QmpDataSrcSetExprLong( MpHnd in_hDataSrc, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, long
    in_lValue, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_lValue
Long constant to compare against. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a long expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

```
QmpDataSrcSetExprLong(hDataSrc, szFieldName, in_Operator,
    lValue, pResult );
```

QmpDataSrcSetExprString

SETS STRING-TYPE EXPRESSION IN SPECIFIED DATA SOURCE.*

Syntax

```
void QmpDataSrcSetExprString( MpHnd in_hDataSrc, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, const char*
    in_szValue, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_szValue
String constant to compare against. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a string expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

```
QmpDataSrcSetExprString(hDataSrc, szFieldName, in_Operator,
    szValue, pResult );
```

QmpDataSrcSetExprSubStr

SETS EXPRESSION SUBSTRING FOR A DATA SOURCE.

Syntax

```
void QmpDataSrcSetExprSubStr ( MpHnd in_ hDataSrc, int
                               in_iCharStartPos, int in_iNumChars, QRESULT* out_pResult );
in_hDataSrc
Handle to data source. Input.
in_iCharStartPos
Start position. Input.
in_iNumChars
Number of chars to consider. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

`QmpDataSrcSetExprSubStr` is valid only for data sources of type `QMS_DATSRC_TYPE_FUNCTOR` and `QMS_DATSRC_TYPE_TABLE`.

`QmpDataSrcSetExprSubStr` adds a substring component to the expression created by one of the `QmpDataSrcSetExpr*` functions. Specifically, it modifies the expression to evaluate a substring of the field. For example, in the expression “`FirstName = ‘Fred’`”, this function makes it possible to evaluate a substring of the `FirstName` field.

This function is valid for any of the data source expression types (date, float, variant, etc.). If the field is not of type string, the value is converted to a string. Be aware of how the field type in question represents its values internally, and how that will affect its conversion to a string.

See Also

`QmpDataSrcGetExprStart`, `QmpDataSrcGetExprNumChar`.

Example

```
/* return the field evaluation substring start position */

/* return the object's starting position */
iFldExprStart = QmpDataSrcGetExprStart( hDataSrcTbl, &qres );
if ( iFldExprStart != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataSrcGetExprNumChar( hDataSrcTbl, &qres );
if ( iFldExprNumChars != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

QmpDataSrcSetExprSubStr( hDataSrcTbl, 2, 4, &qres );
```

```
/* return the object's starting position */
iFldExprStart = QmpDataSrcGetExprStart( hDataSrcTbl, &qres );
if ( iFldExprStart != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* return the object's number of characters */
iFldExprNumChars = QmpDataSrcGetExprNumChar( hDataSrcTbl, &qres );
if ( iFldExprNumChars != 4 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
```

QmpDataSrcSetExprVar

SETS EXPRESSION IN THIS DATA SOURCE TO A VARIANT OBJECT VALUE.

Syntax

```
void QmpDataSrcSetExprVar( MpHnd in_hDataSrc, const char*
    in_szFieldName, QMS_FLDEVAL_OPER in_Operator, MpHnd
    in_hVar, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_hVar
Handle of variant containing constant value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data sources of type
QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a variant expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

See example on [page 403](#).

QmpDataSrcSetExprVarStruct

SETS EXPRESSION IN THIS DATA SOURCE TO A QVARSTRUCT VALUE.

Syntax

```
void QmpDataSrcSetExprVarStruct( MpHnd in_hDataSrc, const
                                char* in_szFieldName, QMS_FLDEVAL_OPER in_Operator,
                                QVARSTRUCT* in_pVarStruct, QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_szFieldName
Field name. *Input*.

in_Operator
Operator. *Input*.

in_pVarStruct
Pointer to QVARSTRUCT structure containing constant value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets a QVARSTRUCT expression in the data source. The expression is used to filter input records to the data input phase. If the expression has been specified, and if it evaluates to QTRUE, the record is accepted and sent on to the client.

Example

See example on [page 403](#).

QmpDataSrcSetFillFunc

SETS FILL FUNCTOR IN SPECIFIED DATA SOURCE.*

Syntax

```
void QmpDataSrcSetFillFunc( MpHnd in_hDataSrc,
                           QMS_FILL_REC_FUNC in_Func, QRESULT* out_pResult );
in_hDataSrc
Handle to data source. Input.
in_Func
Pointer to record fill function. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR.

This function passes the address of an application function that will supply records to the data source.

The record fill function definition in the application should be as follows:

```
QBOOL FillRecFunc (MpHnd io_hRec, MpHnd in_hCaller )
```

The parameter *io_hRec* is a handle to the record to be filled and *in_hCaller* is the handle to the data source calling the function. The function return type is QBOOL, with the value set to QTRUE if the record was filled successfully and the application has more records to supply to this data source, or to QFALSE if the application has no more records for the data source. (The parameter *io_hRec* is ignored by the data source if QFALSE is returned).

Refer to the source code for the sample application *cmpsimpl* for an example of a fill functor.

Example

```
/* configure the data source */
QmpDataSrcSetFillFunc ( hDataSourceFunctor, pFillFunction,
                        pResult );
```

QmpDataSrcSetRecProto

SETS THE DATA FIELDS VIA AN APPLICATION-OWNED RECORD.*

Syntax

```
void QmpDataSrcSetRecProto( MpHnd in_hDataSrc, MpHnd in_hRec,
                           QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_hRec
Handle to the prototype record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_FUNCTOR and QMS_DATSRC_TYPE_TABLE.

This function sets the data fields used in the data source through the application-owned record prototype.

Example

```
QmpDataSrcSetRecProto (hDataSrc, hProtoRec, &qres);
```

QmpDataSrcSetStatSamp

SETS STATISTICAL SAMPLING PROPERTIES.*

Syntax

```
void QmpDataSrcSetStatSamp( MpHnd in_hDataSrc,
                           QMS_STAT_SAMPLING* in_StatSampling, QRESULT* out_pResult );
```

in_hDataSrc

Handle to data source. *Input*.

in_StatSampling

Statistical sampling structure. *Input*.

`QMS_STAT_SAMPLING` is a structure that a client can fill and pass to the data source object that uses it. It defines the pattern of records to return to the data input phase. In **every member**, a value of 0 means “this part not specified”.

<code>lMaxRecords</code>	Maximum number of records to input from a data source.
--------------------------	--

<code>lFirstRecordNum</code>	Record number of first record to return from a data source.
------------------------------	---

<code>lLastRecordNum</code>	Record number of last record to return from a data source.
-----------------------------	--

<code>lInterval</code>	Sampling interval (return every “nth” record). This must be greater than <code>lGroupSize</code> .
------------------------	--

<code>lGroupSize</code>	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record).
-------------------------	---

out_pResult

Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type `QMS_DATSRC_TYPE_FUNCTOR` and `QMS_DATSRC_TYPE_TABLE`.

`QMS_STAT_SAMPLING` is a structure that a client can fill and pass to the data source object that uses it. It defines the pattern of records to return to the data input phase. In every member, a value of 0 means “this part not specified”.

If you want to use either the interval or group size sampling properties, both properties must be set to a value greater than zero. It does not make sense to set a group size greater than an interval; doing so will cause an error.

Example

```
QmpDataSrcSetStatSamp(hDataSrc, pStatSamplingStruct, pResult
) ;
```

QmpDataSrcTblCreate

CREATES A TABLE-TYPE DATA SOURCE.

Syntax

```
MpHnd QmpDataSrcTblCreate( const char* in_pszName, long
                           in_lDataSourceID, QRESULT* out_pResult );
                           in_pszName
                           Name of data source. Input.
                           in_lDataSourceID
                           Data source ID. This ID must be unique for each data source. Input.
                           out_pResult
                           Result code. Output.
```

Return Value

Returns handle to data source if successful, or NULL if there is an error.

Notes

This function creates a table-type data source.

If the creation function fails, an error code is returned in the *out_pResult* parameter.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

```
/* declare */
QmpDeclHnd( hLog );
QmpDeclHnd( hRec );
QmpDeclHnd( hDataInp );
QmpDeclHnd( hDataSrcFunc );
QmpDeclHnd( hDataSrcTbl );
QmpDeclHnd( hDataSrcPre );
QmpDeclHnd( hTestDataSrc );
QmpDeclHnd( hTblInput );
QmpDeclHnd( hTblPrepro );
QmpDeclHnd( hVar );
QmpDeclHnd( hDate );

/* variables */
char szExprBuf[QMS_MAX_NAME];           /* buffer for strings */
QVARSTRUCT varStruct;
QDATESTRUCT dateStruct;

/* setup miscellaneous stuff. */
hLog = QmpGlbGetLog( &qres );
QmpLogSetAppErrorHandler( hLog, AppErrorHandler, &qres );
hDataInp = QmpDataInpCreate( &qres );
hRec = QmpRecCreate( &qres );
QmpRecAdd( hRec, "Address", &qres );
QmpRecAdd( hRec, "FName", &qres );
QmpRecAddByTypePicture( hRec, "Date", QMS_VARIANT_DATE, 0, "mm/dd/yy", &qres );

/* create and setup functor data source */
hDataSrcFunc = QmpDataSrcFuncCreate( "Functor Data Source", 1, &qres );
```

QmpDataSrcTblCreate

```
QmpDataSrcSetRecProto( hDataSrcFunc, hRec, &qres );
QmpDataSrcSetFillFunc( hDataSrcFunc, my_fill_func, &qres );

/* create and setup table data source */
hTblInput = QmpTblCbCreate( "", "InputTbl.dbf", &qres );
hDataSrcTbl = QmpDataSrcTblCreate( "Table Data Source", 2, &qres );
QmpDataSrcSetRecProto( hDataSrcTbl, hRec, &qres );
QmpDataSrcUseTbl( hDataSrcTbl, hTblInput, &qres );

/* create and setup preprocessed data source */
hTblPrepro = QmpTblCbCreate( "", "prepro.dbf", &qres );
hDataSrcPre = QmpDataSrcPreCreate( "Preprocessed Data Source", 3, &qres );
QmpDataSrcUseTbl( hDataSrcPre, hTblPrepro, &qres );

/* add functor and table data source to data input. */
QmpDataInpAddDataSrc( hDataInp, hDataSrcFunc, &qres );
QmpDataInpAddDataSrc( hDataInp, hDataSrcTbl, &qres );
hTestDataSrc = QmpDataInpGetDataSrcAt( hDataInp, 0, &qres );
if ( hDataSrcFunc != hTestDataSrc ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data sources do not match" );
}
hTestDataSrc = QmpDataInpGetDataSrcAt( hDataInp, 1, &qres );
if ( hDataSrcTbl != hTestDataSrc ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Data sources do not match" );
}

/* Set/Get the filter of a data source using variants. */
hVar = QmpVarCreate( &qres );
QmpVarSetString( hVar, "John", &qres );
QmpDataSrcSetExprVar( hDataSrcTbl, "Fname", QMS_FLDEVAL_OPER_LESS, hVar,
    &qres );
QmpDataSrcFillExpr( hDataSrcTbl, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Fname < John" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
QmpVarFillVarStruct( hVar, &varStruct, &qres );
QmpDataSrcSetExprVarStruct( hDataSrcTbl, "Fname", QMS_FLDEVAL_OPER_EQUAL,
    &varStruct, &qres );
QmpDataSrcFillExpr( hDataSrcTbl, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Fname = John" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* Set/Get some properties of the data source using dates. */
hDate = QmpDateCreate( &qres );
QmpDateSetPicture( hDate, "jan 01, 2000", "mmm dd, ccyy", &qres );
QmpDateFillDateStruct( hDate, &dateStruct, &qres );
QmpDataSrcSetExprDate( hDataSrcTbl, "Date", QMS_FLDEVAL_OPER_GREATER, hDate,
    &qres );
QmpDataSrcFillExpr( hDataSrcTbl, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Date > 20000101" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );
QmpDataSrcSetExprDateStruct( hDataSrcTbl, "Date", QMS_FLDEVAL_OPER_EQUAL,
    &dateStruct, &qres );
QmpDataSrcFillExpr( hDataSrcTbl, szExprBuf, sizeof(szExprBuf), &qres );
if ( QMS_STRICTCMP( szExprBuf, "Date = 20000101" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Expression is wrong" );

/* destroy */
QmpRecDestroy(hRec, &qres);
QmpDataSrcDestroy(hDataSrcFunc, &qres);
QmpDataSrcDestroy(hDataSrcTbl, &qres);
QmpDataSrcDestroy(hDataSrcPre, &qres);
QmpPhaseDestroy(hDataInp, &qres);
QmpTblDestroy(hTblInput, &qres);
QmpTblDestroy(hTblPrepro, &qres);
QmpVarDestroy( hVar, &qres );
QmpDateDestroy( hDate, &qres );
```

QmpDataSrcUseFldMap

GIVES A FIELD MAP TO THE DATA SOURCE.

Syntax

```
void QmpDataSrcUseFldMap( MpHnd in_hDataSrc, MpHnd in_hFldMap,
                           QRESULT* out_pResult );
```

in_hDataSrc
Handle to data source. *Input*.

in_hFldMap
Handle to field map. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function gives a field map to a data source. This is necessary so that sources of records are identified and made available for the field mapping process. Call `QmpDataSrcUseFldMap` separately for each data source you want to use as a field mapping record source.

Example

See example on [page 507](#).

QmpDataSrcUseTbl

SPECIFIES THE INPUT TABLE OBJECT TO BE USED BY THE DATA SOURCE.

Syntax

```
void QmpDataSrcUseTbl( MpHnd in_hDataSrc, MpHnd in_hTbl,  
                      QRESULT* out_pResult );  
  
in_hDataSrc  
    Handle to data source. Input.  
  
in_hTbl  
    Handle to table to use for input. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_TABLE or QMS_DATSRC_TYPE_PREPRO.

This function should be called *once*.

Example

```
QmpDataSrcUseTbl(hDataSrc, hTbl, pResult );
```

Function Class: QmpDate*

DATE OBJECT FUNCTIONS.

The Centrus Merge/Purge library is “date-aware”. Date fields can be added to tables, and the `QmpRec*` and `QmpVar*` functions recognize “date” as a data type. The QVARSTRUCT structure contains date information, and the data structure QDATESTRUCT contains dates in a very simple C structure.

To aid in advanced manipulation of dates, the Centrus Merge/Purge library contains the `QmpDate*` function class. The functions in this class allow you to create and destroy date objects, as well as set the date in a variety of ways (using Julian dates and a variety of normal “human readable” dates). The date object is also useful for formatting dates for reports.

Quick Reference

Function	Description	Page
QmpDateCreate	Creates a date object.	408
QmpDateDestroy	Destroys a date object.	409
QmpDateFillDataPicture	Returns a string containing the data storage picture.	410
QmpDateFillDateStruct	Fills a QDATESTRUCT structure with the date inside a date object.	411
QmpDateFillDefPrintPicture	Returns a string containing the default print picture.	412
QmpDateFillFormat	Formats the date inside a date object to match a “picture”.	413
QmpDateGetDateStr	Returns a date in the raw “ccyymmdd” format.	414
QmpDateGetDay	Returns the day of the month (1 - 31) as an integer.	415
QmpDateGetFormat	Formats an object’s date to match a picture, such as “mmmmmmmm dd, ccyy”.	416
QmpDateGetMonth	Returns the month as an integer (January is 1, December is 12).	417
QmpDateGetMonthStr	Returns an object’s month value as a character string (“January”).	418
QmpDateGetYear	Returns an object’s year value as an integer.	419
QmpDateSetDateStruct	Sets the date inside a date object with a QDATESTRUCT structure.	420
QmpDateSetJulian	Sets the date inside a date object with a Julian date value.	421
QmpDateSetPicture	Sets the date in a date object using a string representation and a picture.	422
QmpDateSetRaw	Sets the date inside a date object using a string of the format “ccyymmdd”.	423

QmpDateCreate

CREATES A DATE OBJECT.

Syntax

```
MpHnd QmpDateCreate( QRESULT* out_pResult );  
  
out_pResult  
Result code. Output.
```

Return Value

Returns a handle to the date object if successful, or `NULL` if there is an error.

Notes

If the creation function fails, an error code is returned in the `out_pResult` parameter. Application programs should always test the result code for success.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

`QmpDateDestroy`

Example

```
MpHnd hdate;  
QRESULT presult;  
  
hdate = QmpDateCreate( &presult );
```

QmpDateDestroy

DESTROYS A DATE OBJECT.

Syntax

```
void QmpDateDestroy( MpHnd in_hDate, QRESULT* out_pResult );
```

in_hDate
Handle to date object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

See Also

QmpDateCreate

Example

```
MpHnd hdate;  
QRESULT presult;  
  
QmpDateDestroy (hdate);
```

QmpDateFillDataPicture

RETURNS A STRING CONTAINING THE DATA STORAGE PICTURE.

Syntax

```
void QmpDateFillDataPicture( char* io_szBuffer, long
    in_lSize, QRESULT* out_pResult );
```

io_szBuffer
Output buffer. *Input, output.*

in_lSize
Size of client-allocated buffer. *Input.*

out_pResult
Result code. *Output.*

Return Value

None.

Notes

This function generates a string for the client that describes how the date object stores dates internally (the “picture”). An example of a date picture is “mmmmmmmm dd, ccyy”. The client application must allocate space for the *io_szBuffer* buffer.

No handle is needed, since this information is static to the date object.

See Also

[QmpDateFillDefPrintPicture](#)

Example

```
QRESULT presult;
long lBufferSize = 25;
char szBuffer[lBufferSize];

QmpDateFillDataPicture( szBuffer, lBufferSize, &presult );
```

QmpDateFillDateStruct

FILLS A QDATESTRUCT STRUCTURE WITH THE DATE INSIDE A DATE OBJECT.

Syntax

```
void QmpDateFillDateStruct( MpHnd in_hDate, QDATESTRUCT*
    out_pDateStruct, QRESULT* out_pResult );
```

in_hDate

Handle to the date object. *Input*.

out_pDateStruct

Pointer to the QDATESTRUCT structure. *Output*.

The structure contains one member:

char ccyyymmdd[9];	Date plus one byte for null character.
--------------------	--

out_pResult

Result code. *Output*.

Return Value

None.

Notes

See Also

[QmpDateSetDateStruct](#)

Example

```
MpHnd hdate;
QRESULT presult;
QDATESTRUCT datestruct;

QmpDateFillDateStruct(hdate, &datestruct, &presult);
```

QmpDateFillDefPrintPicture

RETURNS A STRING CONTAINING THE DEFAULT PRINT PICTURE.

Syntax

```
void QmpDateFillDefPrintPicture( char* io_szBuffer, long
                               in_lSize, QRESULT* out_pResult );
  io_szBuffer
    Output buffer. Input, output.
  in_lSize
    Size of client-allocated buffer. Input.
  out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function generates a string for the client that describes how dates are printed in reports unless otherwise specified (the “print picture”). An example of a print picture is “mmmmmmmm dd, ccyy”. The client application must allocate space for the *io_szBuffer* buffer.

No handle is needed, since this information is static to the date object.

See Also

[QmpDateFillDataPicture](#)

Example

```
QRESULT presult;
long lBufferSize = 25;
char szBuffer[lBufferSize];

QmpDateFillDefPrintPicture( szBuffer, lBufferSize, &presult
                           );
```

QmpDateFillFormat

FORMATS THE DATE INSIDE A DATE OBJECT TO MATCH A “PICTURE”.

Syntax

```
void QmpDateFillFormat( MpHnd in_hDate, const char*
    in_szPicture, char* io_szBuffer, long in_lSize, QRESULT*
    out_pResult );
```

in_hDate
Handle to the date object. *Input*.

in_szPicture
The picture. For example, “mmmmmmmm dd, ccyy”. *Input*.

io_szBuffer
The formatted data buffer. This string buffer must be allocated by the client. *Input, output*.

in_lSize
Size of client-allocated buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function formats the date inside a date object to match that of a picture string. The formatted date string is placed inside the application-allocated *io_szBuffer* buffer.

See Also

[QmpDateGetFormat](#)

Example

```
MpHnd hdate;
QRESULT presult;
char szpicture[9];
char szbuffer[9];

strcpy (szpicture, "mmddyy");
QmpDateFillFormat(hdate, szpicture, szbuffer,
    sizeof(szbuffer), &presult);
```

QmpDateGetDateStr

RETURNS A DATE IN THE RAW “CCYYMMDD” FORMAT.

Syntax

```
const char* QmpDateGetDateStr( MpHnd in_hDate, QRESULT*  
    out_pResult );
```

in_hDate
Handle to the date object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns a pointer to a raw date string if successful, or `NULL` if there is an error.
The string is a copy of the date object’s internal representation.

Notes

This function returns an object’s date value. The date string is formatted in the object’s internal “ccyyymmdd” format.

See Also

`QmpDateGetDay`, `QmpDateGetMonth`, `QmpDateGetMonthStr`,
`QmpDateGetYear`, `QmpDateFillDateStruct`.

Example

```
MpHnd hdate;  
QRESULT presult;  
const char *pszdate;  
  
pszdate = QmpDateGetDateStr(hdate, &presult);
```

QmpDateGetDay

RETURNS THE DAY OF THE MONTH (1 - 31) AS AN INTEGER.

Syntax

```
int QmpDateGetDay( MpHnd in_hDate, QRESULT* out_pResult );
```

in_hDate
Handle to the date object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the day of the month (1-31) if successful, or -1 if there is an error.

See Also

`QmpDateGetDateStr`, `QmpDateGetMonth`, `QmpDateGetMonthStr`,
`QmpDateGetYear`, `QmpDateFillDateStruct`.

Example

```
MpHnd hdate;
QRESULT presult;
int day;

day = QmpDateGetDay(hdate, &presult);
```

QmpDateGetFormat

FORMATS AN OBJECT'S DATE TO MATCH A PICTURE, SUCH AS “MMMMMM DD, CCYY”.

Syntax

```
const char* QmpDateGetFormat( MpHnd in_hDate, const char*
    in_szPicture, QRESULT* out_pResult );
```

in_hDate

Handle to the date object. *Input*.

in_szPicture

The picture. For example, “mmmmmm dd, ccyy”. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns a pointer to the formatted date string if successful, or `NULL` if there is an error.

Notes

This function formats an object's date to match that of a “picture” string, such as “mmmmmm dd, ccyy”. A pointer to the formatted date string is returned.

See Also

[QmpDateFillFormat](#)

Example

```
MpHnd hdate;
QRESULT presult;
const char *pszformatdate;
char szpicture[9];

strcpy(szpicture, "mmddyy");
pszformatdate = QmpDateGetFormat(hdate, szpicture, &presult);
```

QmpDateGetMonth

RETURNS THE MONTH AS AN INTEGER (JANUARY IS 1, DECEMBER IS 12).

Syntax

```
int QmpDateGetMonth( MpHnd in_hDate, QRESULT* out_pResult );
```

in_hDate
Handle to the date object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the month as an integer (1-12) if successful, or -1 if there is an error.

See Also

`QmpDateGetDateStr`, `QmpDateGetDay`, `QmpDateGetMonthStr`,
`QmpDateGetYear`, `QmpDateFillDateStruct`.

Example

```
MpHnd hdate;
QRESULT presult;
int month;

month = QmpDateGetMonth(hdate, &presult);
```

QmpDateGetMonthStr

RETURNS AN OBJECT'S MONTH VALUE AS A CHARACTER STRING ("JANUARY").

Syntax

```
const char* QmpDateGetMonthStr( MpHnd in_hDate, QRESULT*  
                               out_pResult );
```

in_hDate
Handle to the date object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns a pointer to a string if successful, or `NULL` if there is an error.

Notes

This function returns the month that corresponds to an object's date value.

See Also

`QmpDateGetDay`, `QmpDateGetMonth`, `QmpDateGetYear`,
`QmpDateFillDateStruct`.

Example

```
MpHnd hdate;  
QRESULT presult;  
const char *pszMonth;  
  
pszMonth = QmpDateGetMonthStr(hdate, &presult);
```

QmpDateGetYear

RETURNS AN OBJECT'S YEAR VALUE AS AN INTEGER.

Syntax

```
int QmpDateGetYear( MpHnd in_hDate, QRESULT* out_pResult );
```

in_hDate
Handle to the date object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns an object's year value as an integer if successful, or -1 if there is an error.

Notes

This function returns the year that corresponds to an object's date value. The full year is returned; for example, "1998".

See Also

[QmpDateGetDateStr](#), [QmpDateGetDay](#), [QmpDateGetMonthStr](#),
[QmpDateGetMonth](#), [QmpDateFillDateStruct](#).

Example

```
MpHnd hdate;
QRESULT presult;
int year;

year = QmpDateGetYear(hdate, &presult);
```

QmpDateSetDateStruct

SETS THE DATE INSIDE A DATE OBJECT WITH A QDATESTRUCT STRUCTURE.

Syntax

```
void QmpDateSetDateStruct( MpHnd in_hDate, QDATESTRUCT*
    in_pDateStruct, QRESULT* out_pResult );
```

in_hDate

Handle to the date object. *Input*.

in_pDateStruct

Pointer to the QDATESTRUCT structure. *Input*.

The structure contains one member:

char <i>ccyyymmdd</i> [9];	Date plus one byte for null character.
------------------------------	--

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets a date object's internal date using a QDATESTRUCT structure.

See Also

[QmpDateFillDateStruct](#)

Example

```
MpHnd hdate;
QRESULT presult;
QDATESTRUCT datestruct;

strcpy(datestruct .ccyyymmdd, "19980901");
QmpDateSetDateStruct(hdate, &datestruct, &presult);
```

QmpDateSetJulian

SETS THE DATE INSIDE A DATE OBJECT WITH A JULIAN DATE VALUE.

Syntax

```
void QmpDateSetJulian( MpHnd in_hDate, long in_lJulian,
                      QRESULT* out_pResult );
in_hDate
    Handle to the date object. Input.
in_lJulian
    The Julian date value. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

A Julian day is defined as the number of days since Jan. 1, 4713 B.C..

See Also

[QmpDateSetRaw](#), [QmpDateSetDateStruct](#), [QmpDateSetPicture](#).

Example

```
MpHnd hdate;
QRESULT presult;
long ljulian;

ljulian = 2451057;
QmpDateSetJulian(hdate, ljulian, &presult);
```

QmpDateSetPicture

SETS THE DATE IN A DATE OBJECT USING A STRING REPRESENTATION AND A PICTURE.

Syntax

```
void QmpDateSetPicture( MpHnd in_hDate, const char* in_szDate,
    const char* in_szPicture, QRESULT* out_pResult );
```

in_hDate

Handle to the date object. *Input*.

in_szDate

The formatted date string. For example, “january 13, 1995”. *Input*.

in_szPicture

The picture. For example, “mmmmmmmm dd, ccyy”. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets the date inside a date object. The *in_szDate* parameter is a string representation of the date. The *in_szPicture* parameter is a string that describes how the date object should interpret the date string.

See Also

[QmpDateFillDataPicture](#)

Example

```
MpHnd hdate;
QRESULT presult;
char szpicture[9];
char szdate[9];

strcpy(szdate, "09011998");
strcpy(szpicture, "mmddccyy");
QmpDateSetPicture(hdate, szdate, szpicture, &presult);
```

QmpDateSetRaw

SETS THE DATE INSIDE A DATE OBJECT USING A STRING OF THE FORMAT “CCYYMMDD”.

Syntax

```
void QmpDateSetRaw( MpHnd in_hDate, const char* in_szDate,
                     QRESULT* out_pResult );
in_hDate
    Handle to the date object. Input.
in_szDate
    The string containing the raw date. It must follow the format “ccyyymmdd”.
    Input.
out_pResult
    Result code. Output.
```

Return Value

None.

See Also

[QmpDateSetJulian](#), [QmpDateSetDateStruct](#), [QmpDateSetPicture](#).

Example

```
MpHnd hdate;
QRESULT presult;
char szraw[9];

strcpy(szraw, "19980901");
QmpDateSetRaw(hdate, szraw, &presult);
```


Function Class: QmpDump*

FUNCTIONS USED TO DUMP AN OBJECT'S CONTENTS TO A FILE.

The **QmpDump*** functions allow objects to dump their contents to an output file. A handle to any of the following objects can be passed to the **QmpDump*** functions:

- QmpTbl
- QmpMatRes
- QmpRec
- QmpDupGrpRec
- QmpDupsGrp
- QmpDataLstSvc
- QmpDataLst
- QmpDataSrc

To dump an object's contents to a file:

1. Call **QmpDumpOpen** to open the dump file.
2. Call **QmpDumpDump** with the handle of the object to be dumped.
3. Close the dump file with **QmpDumpClose**.

This process allows the dump file to be opened just once before all dumping of data occurs, and then closed when finished.

Quick Reference

Function	Description	Page
QmpDumpOpen	Opens a dump file to accept input.	426
QmpDumpDump	dumps an object's contents to a file.	427
QmpDumpClose	Closes a dump file.	428

QmpDumpOpen

OPENS A DUMP FILE TO ACCEPT INPUT.

Syntax

```
void QmpDumpOpen( MpHnd in_hDump, const char* in_szFilename,
                  QRESULT* out_pResult );
```

in_hDump
Handle to the dump object. *Input*.

in_szFilename
The file name to dump to. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function opens a dump file in preparation for calling [QmpDumpDump](#).

See Also

[QmpDumpDump](#), [QmpDumpClose](#).

Example

```
QmpDeclHnd( hDupGrps );      /* the dupe groups object */  
  
hDupGrps = QmpDupGrpsCreate( hDupGrpsTbl, &qres );  
  
/* Assume dupe groups phase ran to completion... */  
/* Dump the entire dupe group. */  
QmpDumpOpen( hDupGrps, "dump.log", &qres );  
QmpDumpDump( hDupGrps, QTRUE, &qres );  
QmpDumpClose( hDupGrps, &qres );
```

QmpDumpDump

DUMPS AN OBJECT'S CONTENTS TO A FILE.

Syntax

```
void QmpDumpDump( MpHnd in_hDump, const QBOOL in_bPrintHeader,
                     QRESULT* out_pResult );
```

in_hDump
Handle to the dump object. *Input*.

in_bPrintHeader
Print header or not. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

See Also

[QmpDumpOpen](#), [QmpDumpClose](#).

Example

See example on [page 426](#).

QmpDumpClose

CLOSES A DUMP FILE.

Syntax

```
void QmpDumpClose( MpHnd in_hDump, QRESULT* out_pResult );
```

in_hDump
Handle to the dump object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function closes a dump file after it has been opened with `QmpDumpOpen` and written to by `QmpDumpDump`.

See Also

`QmpDumpOpen`, `QmpDumpDump`.

Example

See example on [page 426](#).

Function Class: QmpDupGrpRec*

DUPE GROUPS RECORD FUNCTIONS.

The dupe groups record functions are provided as a convenience for the dupe groups table. A dupe groups record is simply a Centrus Merge/Purge record whose fields are already defined to match those in the dupe groups table.

Quick Reference

Function	Description	Page
QmpDupGrpRecCreate	Creates a dupe group record.	430
QmpDupGrpRecDestroy	Destroys a dupe group record.	432
QmpDupGrpRecGetDatLstID	Gets data list ID in dupe group record.	433
QmpDupGrpRecGetDupGrpID	Gets dupe group id in dupe group record.	434
QmpDupGrpRecGetMasSubScore	Gets master/subordinate score in dupe group record.	435
QmpDupGrpRecGetMasSubStat	Gets master/subordinate status in dupe group record.	436
QmpDupGrpRecGetPrimKey	Gets primary key in dupe group record.	437
QmpDupGrpRecGetSrcID	Gets source id in dupe group record.	438

QmpDupGrpRecCreate

Creates a dupe group record.

Syntax

```
MpHnd QmpDupGrpRecCreate( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns handle to dupe group record if successful, or `NULL` if there is an error.

Notes

If the creation function fails, an error code is returned in the `out_pResult` parameter. Application programs should always test the result code for success.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
/* Destroy old dupe groups table */
if ( QmpTblExists( hDuplicatesTable, &qres ) ) {
    QmpTblDestroyRep( hDuplicatesTable, &qres );
}

/* Set and get the maximum number of allowable members in a dupe group */
QmpDupGrpsSetMaxDupGrpMem( hDuplicatesGroups, 12, &qres );
lMaxDuplicateGroupMembers = QmpDupGrpsGetMaxDupGrpMem( hDuplicatesGroups, &qres );
if ( lMaxDuplicateGroupMembers != 12 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Max number of dupe group
members is not correct" );

/* Give the dupe groups phase several tables to work with and set other */
/* properties */
QmpDupGrpsUseMatRes( hDuplicatesGroups, hMatRes, &qres );
QmpPhaseUseDITR( hDuplicatesGroups, hDITR, &qres );
QmpDupGrpsUseRecMat( hDuplicatesGroups, hRecordMatcher, &qres );
QmpDupGrpsUseDataLstSvc( hDuplicatesGroups, hDataListService, &qres );
QmpDupGrpsRegEveryNthRecFunc( hDuplicatesGroups, pEveryNthRecordClient, &qres );
QmpDupGrpsSetNthRecInterval( hDuplicatesGroups, 100, &qres );
lInterval = QmpDupGrpsGetNthRecInterval( hDuplicatesGroups, &qres );
if ( lInterval != 100 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "event interval is incorrect" );
    QmpDupGrpsSetThreshold( hDuplicatesGroups, 91, &qres );
if ( QmpDupGrpsGetThreshold( hDuplicatesGroups, &qres ) != 91 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "dupe groups threshold is
incorrect" );

/* Start the dupe groups phase */
printf( "Starting Dupe Group generation:\n" );
QmpPhaseStart( hDuplicatesGroups, &qres );

/* Get the array of dupe group ids. Obtain number of dupe group masters */
/* and subordinates */
lNumDuplicatesGroups = QmpDupGrpsGetDupGrpCnt( hDuplicatesGroups, &qres );
if ( lNumDuplicatesGroups <= 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "No dupe groups found" );
```

```

pDupGrpIDArray = (long*)malloc( lNumDupeGroups * sizeof( long ) );
lNumDupeGroups2 = QmpDupGrpsFillIDs( hDupeGroups, lNumDupeGroups,
    pDupGrpIDArray, &qres );
if ( lNumDupeGroups != lNumDupeGroups2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "inconsistent number of dupe
        groups" );
lNumMasters = QmpDupGrpsGetMasterCnt( hDupeGroups, &qres );
if ( lNumMasters != lNumDupeGroups )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "inconsistent number of dupe
        groups" );
lNumSubords = QmpDupGrpsGetSubordCnt( hDupeGroups, &qres );

/* Process each of the dupe groups in the array of dupe group ids */
for ( i = 0; i < lNumDupeGroups; i++ ) {

    /* Get the number of members in a dupe group */
    lNumMembers = QmpDupGrpsGetMemberCnt( hDupeGroups, pDupGrpIDArray[i], &qres
    );
    pMemberArray = (long*)malloc( lNumMembers * sizeof( long ) );

    /* Fill an array with dupe group member record numbers (the record */
    /* numbers of the dupe group records in the dupe group table) */
    lNumMembers2 = QmpDupGrpsFillMembers( hDupeGroups, pDupGrpIDArray[i],
        lNumMembers, pMemberArray, &qres );
    if ( lNumMembers != lNumMembers2 )
        QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "inconsistent number of dupe
            group members" );

    /* Process each member in the dupe group */
    hDupeGroupRec = QmpDupGrpRecCreate( &qres );
    for ( j = 0; j < lNumMembers; j++ ) {

        /* Get the master dupe from this dupe group */
        lMasterRec = QmpDupGrpsGetMasterDupe( hDupeGroups, lNumMembers,
            pMemberArray, &qres );

        /* Fill a record with the dupe group table values */
        QmpDupGrpsFillRec( hDupeGroups, lMasterRec, hDupeGroupRec, &qres );

        /* Get the fields out of a dupe group record */
        lKey = QmpDupGrpRecGetPrimKey( hDupeGroupRec, &qres );
        lSrcID = QmpDupGrpRecGetSrcID( hDupeGroupRec, &qres );
        lDatLstID = QmpDupGrpRecGetDatLstID( hDupeGroupRec, &qres );
        lDupGrpID = QmpDupGrpRecGetDupGrpID( hDupeGroupRec, &qres );
        cMasSubStat = QmpDupGrpRecGetMasSubStat( hDupeGroupRec, &qres );
        iMasSubScore = QmpDupGrpRecGetMasSubScore( hDupeGroupRec, &qres );
    }
    free ( pMemberArray );
    pMemberArray = NULL;
}

/* Filter out the low ranking members of a dupe group. */
hNewDuplicatesTable = QmpTblCbCreate( "", "newDuplicatesTable", &qres );
hNewDupsGroups = QmpDupGrpsCreate( hNewDuplicatesTable, &qres );
QmpDupGrpsRemLowMembers( hDupeGroups, 95, hNewDupsGroups, &qres );

/* Dump the entire dupe group, and a single dupe group record */
remove ( "dump.log" );
QmpDumpOpen( hDupeGroups, "dump.log", &qres );
QmpDumpDump( hDupeGroups, QTRUE, &qres );
QmpDumpClose( hDupeGroups, &qres );
QmpDumpOpen( hDupeGroupRec, "dump.log", &qres );
QmpDumpDump( hDupeGroupRec, QTRUE, &qres );
QmpDumpClose( hDupeGroupRec, &qres );

/* free resources and arrays */
free ( pDupGrpIDArray );
pDupGrpIDArray = NULL;

```

QmpDupGrpRecDestroy

DESTROYS A DUPE GROUP RECORD.

Syntax

```
void QmpDupGrpRecDestroy( MpHnd in_hDupGrpRec, QRESULT*
                           out_pResult );
in_hDupGrpRec
    Handle to dupe group record. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Destroys a dupe group record.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

Example

```
MpHnd hDupGrpRec;
QRESULT presult;

QmpDupGrpRecDestroy( hDupGrpRec, &presult );
```

QmpDupGrpRecGetDatLstID

GETS DATA LIST ID IN DUPE GROUP RECORD.

Syntax

```
long QmpDupGrpRecGetDatLstID( MpHnd in_hDupGrpRec, QRESULT*  
                               out_pResult );
```

in_hDupGrpRec

Handle to dupe group record. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns data list ID if successful, or -1 if there is an error.

Notes

QmpDupGrpRecGetDatLstID returns the ID of the data list that the dupe group record belongs to.

Example

See example on [page 430](#).

QmpDupGrpRecGetDupGrpID

GETS DUPE GROUP ID IN DUPE GROUP RECORD.

Syntax

```
long QmpDupGrpRecGetDupGrpID( MpHnd in_hDupGrpRec, QRESULT*  
                               out_pResult );
```

in_hDupGrpRec

Handle to dupe group record. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns dupe group ID if successful, or -1 if there is an error.

Notes

`QmpDupGrpRecGetDupGrpID` returns the ID of the dupe group that the record belongs to.

Example

See example on [page 430](#).

QmpDupGrpRecGetMasSubScore

GETS MASTER/SUBORDINATE SCORE IN DUPE GROUP RECORD.

Syntax

```
int QmpDupGrpRecGetMasSubScore( MpHnd in_hDupGrpRec, QRESULT*  
                                out_pResult );
```

in_hDupGrpRec

Handle to dupe group record. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns master/subordinate score (0 to 100) if successful, or -1 if there is an error.

Notes

Every subordinate duplicate in a dupe group is compared against the master duplicate. This match score is stored in every dupe group record, and **QmpDupGrpRecGetMasSubScore** returns that score.

Note that the master record in the current duplicate group will always return a score of 100. This is because the master record was compared to itself.

Example

See example on [page 430](#).

QmpDupGrpRecGetMasSubStat

GETS MASTER/SUBORDINATE STATUS IN DUPE GROUP RECORD.

Syntax

```
char QmpDupGrpRecGetMasSubStat( MpHnd in_hDupGrpRec, QRESULT*  
                                out_pResult );  
  
in_hDupGrpRec  
        Handle to dupe group record. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns master/subordinate status ('M', 'S', or '?' only) if successful, or 0 if there is an error.

Notes

Each dupe group has a master duplicate record and at least one subordinate duplicate record. The dupe groups phase determines whether each record in a dupe group is a master or subordinate. **QmpDupGrpRecGetMasSubStat** identifies a dupe group record as a master or subordinate.

If the status hasn't yet been determined, a '?' character is returned.

Example

See example on [page 430](#).

QmpDupGrpRecGetPrimKey

GETS PRIMARY KEY IN DUPE GROUP RECORD.

Syntax

```
long QmpDupGrpRecGetPrimKey( MpHnd in_hDupGrpRec, QRESULT*
    out_pResult );
in_hDupGrpRec
    Handle to dupe group record. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns primary key if successful, or -1 if there is an error.

Notes

The primary key is a number assigned internally by the Centrus Merge/Purge library that is unique to each record in each data source. If a job has one data source, the primary key uniquely identifies a record. If a job has multiple data sources, the primary key and source ID values uniquely identify a record.

It is important to note that the primary key cannot be assigned by the application. If the application requires its own primary key, the application should create an additional field in the prototype record to store its own “custom” primary key. For example, you may wish to use an account number from a record as the primary key.

Example

See example on [page 430](#).

QmpDupGrpRecGetSrcID

GETS SOURCE ID IN DUPE GROUP RECORD.

Syntax

```
long QmpDupGrpRecGetSrcID( MpHnd in_hDupGrpRec, QRESULT*  
                           out_pResult );
```

in_hDupGrpRec
Handle to dupe group record. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns source ID if successful, or -1 if there is an error.

Notes

The source ID identifies the data source the record came from. Each source ID is unique to each data source. The source ID and primary key values, when used together, uniquely identify a record.

Example

See example on [page 430](#).

Function Class: QmpDupGrps*

DUPE GROUPS PHASE FUNCTIONS.

The dupe groups class of functions manipulates duplicate group information. Using these functions, you may retrieve specific information about duplicate groups that have been formed through the matching process. For example, the dupe groups functions allow you to retrieve:

- the count of total duplicates found
- the count of master duplicates
- the count of subordinate duplicates
- information about a specific dupe group and/or its members

Rules for Choosing the Master Duplicate

One of the dupe groups phase's responsibilities is to choose the master dupe in a dupe group. The application decides which rule should be used to select the master dupe. The available rules are:

Master Duplicate Rule (Priority)	Explanation
Data list	Use a dupe's data list assignment to determine the master dupe.
Fldval	Use an expression of the form <i>FieldVal Operator Constant</i> to determine the master dupe. The dupe that supplies the value that makes this expression true becomes the master dupe.
Largest	The dupe that supplies the largest value of a certain field becomes the master dupe.
Random	The master dupe is determined randomly.
Smallest	The dupe that supplies the smallest value of a certain field becomes the master dupe.

Tips for Using the Dupe Groups Phase Effectively

The best application of this functionality is to run the record matcher at a low threshold, then run the dupe groups phase at various thresholds equal to or greater than the record matcher threshold. Note that running the dupe groups phase at a threshold lower than the record matcher threshold doesn't make any sense: it gives the same results as not setting the dupe groups threshold at all.

Tweaking the dupe groups threshold is useful when generating a near miss report. This report lists the dupe groups generated by a job, as well as the records that almost made it into those dupe groups. If you are dissatisfied with the results of a near miss report, you can change the dupe groups threshold (which will increase or decrease the number of duplicates), rerun the dupe groups phase, and create a new near miss report. You do not need to rerun the data input or record matching phases.

Quick Reference

Function	Description	Page
QmpDupGrpsAddPreProcDataSrc	Specifies the preprocessed data source to be used by the dupe group phase.*	443
QmpDupGrpsClear	Sets the object back to the initial state.	444
QmpDupGrpsCreate	Creates a dupe groups object.	445
QmpDupGrpsFillExpression	Gets a string version of the expression used in fldval ranking priority.	446
QmpDupGrpsFillIDs	Fills the client-supplied array with the dupe group ids.	447
QmpDupGrpsFillMembers	Returns the record numbers of members of a dupe group.	448
QmpDupGrpsFillRec	Fills a client-supplied dupe group record from a record number in the dupe groups object.	449
QmpDupGrpsGetDupGrpCnt	Gets dupe group count (i.e., number of dupe groups).	450
QmpDupGrpsGetExpressionNumChars	Gets the character length of a substring used in fldval ranking priority.	451
QmpDupGrpsGetExpressionStartPos	Returns the starting position of a substring used in fldval ranking priority.	452
QmpDupGrpsGetLargestDupGrp	Gets the id of the largest dupe group.	453
QmpDupGrpsGetMasterCnt	Gets number of master duplicates.	454
QmpDupGrpsGetMasterDupe	Gets the master dupe for a dupe group. The dupe group is specified by passing in the dupe group record numbers.	455
QmpDupGrpsGetMaxDupGrpMem	Gets the maximum number of allowable dupe group members.	456
QmpDupGrpsGetMemberCnt	Gets the number of members in a dupe group.	457
QmpDupGrpsGetNthReclInterval	Gets duplicate groups Nth record event interval.	458
QmpDupGrpsGetRankingPriority	Gets the dupe groups ranking priority.	459
QmpDupGrpsGetRankingPriorityFieldName	Gets the dupe groups ranking priority field name.	460

Function	Description	Page
QmpDupGrpsGetRankingPriority FieldNameVB	Gets the dupe groups ranking priority field name (for Visual Basic applications).	461
QmpDupGrpsGetSortFieldName	Gets the dupe groups sort field name.	462
QmpDupGrpsGetSortFieldNameVB	Gets the dupe groups sort field name (for Visual Basic applications).	463
QmpDupGrpsGetSortOrder	Gets the dupe groups sort order.	464
QmpDupGrpsGetSubordCnt	Gets number of subordinate duplicates in all dupe groups.	465
QmpDupGrpsGetThreshold	Gets the dupe groups phase threshold.	466
QmpDupGrpsGetTreatmentType	Gets treatment type of field used for dupe group master selection. Valid for smallest and largest ranking priorities.	467
QmpDupGrpsRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	468
QmpDupGrpsRemLowMembers	Filters out the low ranking members of all dupe groups.	469
QmpDupGrpsRemPreProcDataSrc	Remove a preprocessed data source from the duplicate groups.	470
QmpDupGrpsSetExpressionNumChars	Sets the character length of a substring used in fldval ranking priority.	471
QmpDupGrpsSetExpressionStartPos	Sets the starting position of a substring used in fldval ranking priority.	472
QmpDupGrpsSetFldValPriority	Sets fldval priority for dupe group master selection.	473
QmpDupGrpsSetMaxDupGrpMem	Sets the maximum number of allowable dupe group members.	474
QmpDupGrpsSetNthRecInterval	Sets duplicate groups Nth record event interval.	475
QmpDupGrpsSetSimplePriority	Sets datalist and random priorities for dupe group master selection.	476
QmpDupGrpsSetSortOrder	Sets the sort order for all dupes in a dupe group.	477
QmpDupGrpsSetThreshold	Sets the dupe groups phase threshold.	478
QmpDupGrpsSetTreatmentPriority	Sets smallest and largest priorities for dupe group master selection.	479
QmpDupGrpsTblClose	Closes the table representation used by the dupe groups.	480
QmpDupGrpsTempDGTtblClose	Closes the tempDGT.	481
QmpDupGrpsUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	482
QmpDupGrpsUseDataLstSvc	Specifies the data list service object to be used by the dupe groups object.	483

Function	Description	Page
QmpDupGrpsUseMatRes	Specifies the match result object to be used by the dupe groups object.	484
QmpDupGrpsUseRecMat	Specifies the record matcher to be used by the dupe groups object.	485
QmpDupGrpsUseTbl	Specify the table to be used by the dupe groups object.	486
QmpDupGrpsUseTempDGTbl	Gives a table to the dupe groups object to be used as the tempDGT.	487

QmpDupGrpsAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE DUPE GROUP PHASE.*

Syntax

```
void QmpDupGrpsAddPreProcDataSrc ( MpHnd in_hDupGrps, MpHnd
    in_hDataSource, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups phase. Input.
in_hDataSource
    Handle to preprocessed data source to be used. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the dupe groups phase.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

Example

```
QmpDupGrpsAddPreProcDataSrc ( hDupGrps, hPreProcDataSource,
    pResult );
```

QmpDupGrpsClear

SETS THE OBJECT BACK TO THE INITIAL STATE.

Syntax

```
void QmpDupGrpsClear( MpHnd in_hDupGrps, QRESULT* out_pResult  
);
```

in_hDupGrps

Handle to dupe groups. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets the dupe groups object back to its initial state. After calling **QmpDupGrpsClear**, you can reset any property of the dupe groups phase, including the dupe groups threshold, and restart the phase.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
MpHnd hDupGrps;  
QRESULT presult;  
  
QmpDupGrpsClear( hDupGrps, &presult );
```

QmpDupGrpsCreate

CREATES A DUPE GROUPS OBJECT.

Syntax

```
MpHnd QmpDupGrpsCreate( MpHnd in_hTable, QRESULT* out_pResult  
);
```

in_hTable

Handle to client-supplied table to use as dupe groups table. This *must* be a CodeBase table. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to dupe groups if successful, or NULL if there is an error.

Notes

The dupe groups table object supplied *must* be a CodeBase table.

The dupe groups table should not be destroyed until the uniques report phase is finished executing.

If the creation function fails, an error code is returned in the *out_pResult* parameter. Application programs should always test the result code for success.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

See example on [page 430](#).

QmpDupGrpsFillExpression

GETS A STRING VERSION OF THE EXPRESSION USED IN FLDVAL RANKING PRIORITY.

Syntax

```
void QmpDupGrpsFillExpression( MpHnd in_hDupGrps, char*
                               io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hDupGrps
Handle to dupe groups phase. *Input*.

io_szBuffer
Client-allocated buffer to hold the expression string. *Input, output*.

in_lSize
Size of client-allocated buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for QMS_RANKING_PRIORITY_FLDVAL master ranking.

The QMS_RANKING_PRIORITY_FLDVAL master ranking scheme uses an expression of the form *FieldValue Operator Constant*.

QmpDupGrpsFillExpression returns a string version of this expression.

Example

```
QmpDupGrpsFillExpression( in_hDupGrps, &io_szBuffer,
                           in_lSize, &out_pResult );
```

QmpDupGrpsFillIDs

FILLS THE CLIENT-SUPPLIED ARRAY WITH THE DUPE GROUP IDS.

Syntax

```
long QmpDupGrpsFillIDs( MpHnd in_hDupGrps, long in_lSize,
                        long* out_lDupeGroupIDArray, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups. Input.
in_lSize
    Size of output array. Input.
out_lDupeGroupIDArray
    Client allocated array of dupe group IDs. Output.
out_pResult
    Result code. Output.
```

Return Value

Returns number of IDs put into output array if successful, or -1 if there is an error.

See Also

[QmpDupGrpsGetDupGrpCnt](#)

Notes

A dupe group ID is assigned by the Centrus Merge/Purge library to uniquely identify each duplicate group found. The client application must allocate an array for the dupe group IDs.

Example

See example on [page 430](#).

QmpDupGrpsFillMembers

RETURNS THE RECORD NUMBERS OF MEMBERS OF A DUPE GROUP.

Syntax

```
long QmpDupGrpsFillMembers( MpHnd in_hDupGrps, long
    in_lDupeGroupID, long in_lSize, long* out_lMemberArray,
    QRESULT* out_pResult );
```

in_hDupGrps

Handle to dupe groups. *Input*.

in_lDupeGroupID

Dupe group ID. *Input*.

in_lSize

Size of output array. *Input*.

out_lMemberArray

Client allocated array of dupe group record numbers. *Output*.

out_pResult

Result code. *Output*.

Return Value

Returns number of record numbers put into output array if successful, or -1 if there is an error.

Notes

QmpDupGrpsFillMembers fills a client-supplied array with the record numbers of members of a particular dupe group. The record numbers are from the dupe groups table and can be used in other functions for retrieving dupe group record-specific information (e.g. **QmpDupGrpRecGetPrimKey** returns the primary key from a specific dupe group member/record).

See Also

[QmpDupGrpsGetDupGrpCnt](#), [QmpDupGrpsFillIDs](#)

Example

See example on [page 430](#).

QmpDupGrpsFillRec

FILLS A CLIENT-SUPPLIED DUPE GROUP RECORD FROM A RECORD NUMBER IN THE DUPE GROUPS OBJECT.

Syntax

```
void QmpDupGrpsFillRec( MpHnd in_hDupGrps, long in_lRecNum,
                         MpHnd io_hDupGrpRec, QRESULT* out_pResult );
```

in_hDupGrps
Handle to dupe groups. *Input*.

in_lRecNum
Record number of member in dupe groups table. *Input*.

io_hDupGrpRec
Handle to client-owned dupe group record to fill. *Input, Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDupGrpsFillRec copies information from a specified record in the dupe groups table to a client-supplied dupe group record.

See Also

[QmpDupGrpsGetDupGrpCnt](#), [QmpDupGrpsFillIDs](#)

Example

See example on [page 430](#).

QmpDupGrpsGetDupGrpCnt

GETS DUPE GROUP COUNT (I.E., NUMBER OF DUPE GROUPS).

Syntax

```
long QmpDupGrpsGetDupGrpCnt( MpHnd in_hDupGrps, QRESULT*  
    out_pResult );
```

in_hDupGrps
Handle to dupe groups. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns dupe group count if successful, or -1 if there is an error.

Notes

QmpDupGrpsGetDupGrpCnt returns the number of dupe groups in the dupe groups object.

Example

See example on [page 430](#).

QmpDupGrpsGetExpressionNumChars

GETS THE CHARACTER LENGTH OF A SUBSTRING USED IN FLDVAL RANKING PRIORITY.

Syntax

```
int QmpDupGrpsGetExpressionNumChars( MpHnd in_hDupGrps,
                                     QRESULT* out_pResult );
in_hDupGrps
Handle to the dupe groups phase. Input.
out_pResult
Result code. Output.
```

Return Value

Returns the character length of the field's substring.

Notes

This function is valid only for QMS_RANKING_PRIORITY_FLDVAL master ranking.

`QmpDupGrpsSetExpressionNumChars` sets the character length of the substring of the field string used in the fldval master ranking scheme.
`QmpDupGrpsGetExpressionNumChars` gets the character length of that substring.

Example

```
iSubstringLength = QmpDupGrpsGetExpressionNumChars(
    in_hDupGrps, &out_pResult );
```

QmpDupGrpsGetExpressionStartPos

RETURNS THE STARTING POSITION OF A SUBSTRING USED IN FLDVAL RANKING PRIORITY.

Syntax

```
int QmpDupGrpsGetExpressionStartPos( MpHnd in_hDupGrps,
                                     QRESULT* out_pResult );
in_hDupGrps
    Handle to the dupe groups phase. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the starting position of the field's substring.

Notes

This function is valid only for QMS_RANKING_PRIORITY_FLDVAL master ranking.

`QmpDupGrpsSetExpressionStartPos` sets the starting position of the substring of the field string used in the fldval master ranking scheme.
`QmpDupGrpsGetExpressionStartPos` gets the starting position of that substring.

Example

```
iStartPos = QmpDupGrpsGetExpressionStartPos( in_hDupGrps,
                                             &out_pResult );
```

QmpDupGrpsGetLargestDupGrp

GETS THE ID OF THE LARGEST DUPE GROUP.

Syntax

```
long QmpDupGrpsGetLargestDupGrp( MpHnd in_hDupGrps, QRESULT*  
    out_pResult );
```

in_hDupGrps

Handle to dupe groups object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the dupe group ID of the largest dupe group if successful, or -1 if there is an error.

Notes

Use **QmpDupGrpsGetMemberCnt** to determine the number of members in the largest dupe group.

Example

```
long lLargestDupGrpID;  
MpHnd hDupGrps;  
QRESULT presult;  
  
lLargestDupGrpID = QmpDupGrpsGetLargestDupGrp( hDupGrps,  
    &presult );
```

QmpDupGrpsGetMasterCnt

GETS NUMBER OF MASTER DUPLICATES.

Syntax

```
long QmpDupGrpsGetMasterCnt( MpHnd in_hDupGrps, QRESULT*  
    out_pResult );
```

in_hDupGrps

Handle to dupe groups object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns number of master duplicates in all dupe groups if successful, or -1 if there is an error.

Notes

QmpDupGrpsGetMasterCnt returns the number of master duplicates in all the dupe groups.

Example

See example on [page 430](#).

QmpDupGrpsGetMasterDupe

GETS THE MASTER DUPE FOR A DUPE GROUP. THE DUPE GROUP IS SPECIFIED BY PASSING IN THE DUPE GROUP RECORD NUMBERS.

Syntax

```
long QmpDupGrpsGetMasterDupe( MpHnd in_hDupGrps, long  
    in_lSize, long* in_lMemberArray, QRESULT* out_pResult );  
  
in_hDupGrps  
    Handle to dupe groups. Input.  
  
in_lSize  
    Size of input array. Input.  
  
in_lMemberArray  
    Client allocated array of dupe group record numbers. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns the record number of the master dupe if successful, or -1 if there is an error.

Notes

QmpDupGrpsGetMasterDupe returns the master dupe record number for a dupe group. The dupe group is specified by passing in the record numbers of all the members of the dupe group through *in_lMemberArray*.

Example

See example on [page 430](#).

QmpDupGrpsGetMaxDupGrpMem

GETS THE MAXIMUM NUMBER OF ALLOWABLE DUPE GROUP MEMBERS.

```
int QmpDupGrpsGetMaxDupGrpMem ( MpHnd in_hDupGrps, QRESULT*  
                                out_pResult );  
  
in_hDupGrps  
    Handle to dupe groups. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns the maximum number of allowable dupe group members if successful, or -1 if there is an error.

Notes

QmpDupGrpsGetMaxDupGrpMem returns the maximum number of allowed dupe group members. If the maximum number of dupe group members is exceeded in a given dupe group, a warning is generated. You can choose to ignore this warning or take other actions. If you choose to ignore this warning, the dupe group is generated normally.

This function was implemented to provide a sanity check when performing matching. It is possible to set up matching criteria that might produce more duplicate records than expected. This function is intended to warn the application programmer of such possibilities and to take appropriate action.

Example

See example on [page 430](#).

QmpDupGrpsGetMemberCnt

GETS THE NUMBER OF MEMBERS IN A DUPE GROUP.

Syntax

```
long QmpDupGrpsGetMemberCnt( MpHnd in_hDupGrps, long  
    in_lDupeGroupID, QRESULT* out_pResult );  
  
in_hDupGrps  
    Handle to dupe groups object. Input.  
  
in_lDupeGroupID  
    Dupe group ID. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns number of members in a dupe group if successful, or -1 if there is an error.

Notes

`QmpDupGrpsGetMemberCnt` returns the number of members in a specific dupe group.

See Also

`QmpDupGrpsGetDupGrpCnt`, `QmpDupGrpsFillIDs`

Example

See example on [page 430](#).

QmpDupGrpsGetNthRecInterval

GETS DUPLICATE GROUPS NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpDupGrpsGetNthRecInterval ( MpHnd in_hDupGrps,
                                    QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns dupe groups Nth record event interval if successful, or -1 if there is an error.

Notes

This function gets the dupe groups Nth record event interval.

The dupe groups can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the dupe groups processes 100 records, it will send out a notification to *all* registered clients.

See Also

`QmpDupGrpsRegEveryNthRecFunc`, `QmpDupGrpsSetNthRecInterval`,
`QmpDupGrpsUnregEveryNthRecFunc`

Example

See example on [page 430](#).

QmpDupGrpsGetRankingPriority

GETS THE DUPE GROUPS RANKING PRIORITY.

Syntax

```
QMS_RANKING_PRIORITY QmpDupGrpsGetRankingPriority( MpHnd
    in_hDupGrps, QRESULT* out_pResult );
```

in_hDupGrps

Handle to the dupe groups object. *Input*.

out_pResult

Result code. *Output*.

Return Value

This function returns the rule for determining the master dupe in a dupe group. Valid return values are:

`QMS_RANKING_PRIORITY_DATALIST`

`QMS_RANKING_PRIORITY_FLDVAL`

`QMS_RANKING_PRIORITY_LARGEST`

`QMS_RANKING_PRIORITY_SMALLEST`

`QMS_RANKING_PRIORITY_RANDOM`

Notes

Example

```
priority = QmpDupGrpsGetRankingPriority( in_hDupGrps,
    &out_pResult );
```

QmpDupGrpsGetRankingPriorityFieldName

GETS THE DUPE GROUPS RANKING PRIORITY FIELD NAME.

Syntax

```
const char* QmpDupGrpsGetRankingPriorityFieldName( MpHnd  
    in_hDupGrps, QRESULT* out_pResult );  
  
in_hDupGrps  
    Handle to the dupe groups object. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns the name of the ranking field used to determine the master duplicate.

Notes

Example

```
szFieldName = QmpDupGrpsGetRankingPriorityFieldName(  
    in_hDupGrps, &out_pResult );
```

QmpDupGrpsGetRankingPriorityFieldNameVB

GETS THE DUPE GROUPS RANKING PRIORITY FIELD NAME (FOR VISUAL BASIC APPLICATIONS).

Syntax

```
void QmpDupGrpsGetRankingPriorityFieldNameVB( MpHnd
    in_hDupGrps, char* io_szBuffer, long in_lSize, QRESULT*
    out_pResult );
```

in_hDupGrps
Handle to the dupe groups object. *Input*.

io_szBuffer
Buffer used to hold the field name. *Input, output*.

in_lSize
Size of the field name buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function gets the name of the ranking field used to determine the master duplicate. It is designed to be used by Visual Basic applications.

Example

```
QmpDupGrpsGetRankingPriorityFieldNameVB( in_hDupGrps,
    &io_szBuffer, in_lSize, &out_pResult );
```

QmpDupGrpsGetSortFieldName

GETS THE DUPE GROUPS SORT FIELD NAME.

Syntax

```
const char* QmpDupGrpsGetSortFieldName( MpHnd in_hDupGrps,
                                         QRESULT* out_pResult );
in_hDupGrps
    Handle to the dupe groups phase. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the name of the field used to sort the dupes in a dupe group.

Example

```
szFieldName = QmpDupGrpsGetSortFieldName( in_hDupGrps,
                                         &out_pResult );
```

QmpDupGrpsGetSortFieldNameVB

GETS THE DUPE GROUPS SORT FIELD NAME (FOR VISUAL BASIC APPLICATIONS).

Syntax

```
void QmpDupGrpsGetSortFieldNameVB( MpHnd in_hDupGrps, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
in_hDupGrps
    Handle to the dupe groups phase. Input.
io_szBuffer
    Buffer to hold the field name. Input, output.
in_lSize
    Size of the buffer to hold the field name. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function gets the name of the field used to sort the dupes in a dupe group. It is designed to be used with Visual Basic applications.

Example

```
QmpDupGrpsGetSortFieldNameVB( in_hDupGrps, &io_szBuffer,
    in_lSize, &out_pResult );
```

QmpDupGrpsGetSortOrder

GETS THE DUPE GROUPS SORT ORDER.

Syntax

```
QMS_SORT_ORDER QmpDupGrpsGetSortOrder( MpHnd in_hDupGrps,  
                                QRESULT* out_pResult );
```

in_hDupGrps
Handle to the dupe groups phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the value that determines how all dupes in a dupe group will be sorted. The valid return values are:

`QMS_SORT_ORDER_NONE`
`QMS_SORT_ORDER_ASCENDING`
`QMS_SORT_ORDER_DESCENDING`

See Also

[QmpDupGrpsSetSortOrder](#).

Example

```
sortOrder = QmpDupGrpsGetSortOrder( in_hDupGrps, &out_pResult  
);
```

QmpDupGrpsGetSubordCnt

GETS NUMBER OF SUBORDINATE DUPLICATES IN ALL DUPE GROUPS.

Syntax

```
long QmpDupGrpsGetSubordCnt( MpHnd in_hDupGrps, QRESULT*  
    out_pResult );
```

in_hDupGrps
Handle to dupe groups. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns number of subordinate duplicates in all dupe groups if successful, or -1 if there is an error.

Notes

QmpDupGrpsGetSubordCnt returns the number of subordinate duplicates in all the dupe groups.

Example

See example on [page 430](#).

QmpDupGrpsGetThreshold

GETS THE DUPE GROUPS PHASE THRESHOLD.

Syntax

```
int QmpDupGrpsGetThreshold( MpHnd in_hDupGrps, QRESULT*
    out_pResult );
in_hDupGrps
    Handle to dupe groups. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the dupe groups phase threshold if successful, or -1 if there is an error.

Notes

The dupe groups threshold determines which records in the match result table are accepted into the dupe groups table during the dupe groups “build” subphase. If the threshold is set, the score of a match result table record must be equal to or greater than the threshold to be accepted into the dupe groups table. If the application does not set the dupe groups threshold, all records from the match result table are accepted into the dupe groups table.

The best application of this new functionality is to run the record matcher at a low threshold, then run the dupe groups phase at various thresholds equal to or greater than the record matcher threshold. Note that running the dupe groups phase at a threshold lower than the record matcher threshold doesn’t make any sense: it gives the same results as not setting the dupe groups threshold at all.

Tweaking the dupe groups threshold is useful when generating a near miss report. This report lists the dupe groups generated by a job, as well as the records that almost made it into those dupe groups. If you are dissatisfied with the results of a near miss report, you can change the dupe groups threshold (which will increase or decrease the number of duplicates), rerun the dupe groups phase, and create a new near miss report. You do not need to rerun the data input or record matching phases.

See Also

[QmpDupGrpsSetThreshold](#).

Example

See example on [page 430](#).

QmpDupGrpsGetTreatmentType

GETS TREATMENT TYPE OF FIELD USED FOR DUPE GROUP MASTER SELECTION. VALID FOR SMALLEST AND LARGEST RANKING PRIORITIES.

Syntax

```
QMS_CONSOL_TREAT QmpDupGrpsGetTreatmentType( MpHnd
    in_hDupGrps, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups object. Input.
out_pResult
    Result code. Output.
```

Return Value

This function returns the field type for the field used to determine the master dupe. The valid return values are:

QMS_CONSOL_TREAT_UNDEFINED
QMS_CONSOL_TREAT_STRING
QMS_CONSOL_TREAT_NUMBER
QMS_CONSOL_TREAT_DATE

Notes

This function is valid only for the QMS_RANKING_PRIORITY_SMALLEST and QMS_RANKING_PRIORITY_LARGEST priorities.

Example

```
treatType = QmpDupGrpsGetTreatmentType( in_hDupGrps,
    &out_pResult );
```

QmpDupGrpsRegEveryNthRecFunc

Registers the event handler to be notified of every Nth record processed.

Syntax

```
void QmpDupGrpsRegEveryNthRecFunc ( MpHnd in_hDupGrps,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hDupGrps
Handle to dupe groups. *Input*.

in_Func
Pointer to event handler. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The dupe groups phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

See example on [page 430](#).

QmpDupGrpsRemLowMembers

FILTERS OUT THE LOW RANKING MEMBERS OF ALL DUPE GROUPS.

Syntax

```
void QmpDupGrpsRemLowMembers( MpHnd in_hDupGrps, int
    in_iThreshold, MpHnd io_hNewDupGrps, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups object. Input.
in_iThreshold
    Dupe group members with match scores below this threshold are removed.
    Input.
io_hNewDupGrps
    Handle to new dupe groups object. The application creates one of these,
    and passes it to the Centrus Merge/Purge library. The library then fills it
    with dupe group members scoring at or above the threshold. Input/Output.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

It is possible for an application program to generate a dupe group with members whose scores (as compared to the master dupe) are below the record matcher's "match" threshold. This is because of the two-stage process by which matches are identified and scored. During the first stage, the record matcher compares sets of records and identifies pairs of potential matches based on their match scores. The dupe groups phase then examines the results of the record matcher and determines the actual members of the dupe groups. It also builds the dupe groups table, figures out the master and subordinates for each dupe group, and ensures that each master is compared and scored against its subordinates. The result of this process is a dupe groups object in which every master is compared with its subordinates, and a match score generated for each comparison.

At this point, the application could call `QmpDupGrpsRemLowMembers` to eliminate subordinates in a dupe group that do not match well with the master.

Note that this procedure can potentially create a dupe group with a single member, or eliminate entire dupe groups altogether.

Example

See example on [page 430](#).

QmpDupGrpsRemPreProcDataSrc

REMOVE A PREPROCESSED DATA SOURCE FROM THE DUPLICATE GROUPS.

Syntax

```
MpHnd QmpDupGrpsRemPreProcDataSrc ( MpHnd in_hDupGrps, long
                                         in_lDataSourceID, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups. Input.
in_lDataSourceID
    Data source ID of the data source to remove. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns handle to removed data source if successful, or `NULL` if there is an error.

Notes

Allows an application to remove a preprocessed data source from a dupe groups object. The dupe groups object currently can accept only one data source of type `QMS_DATSRC_TYPE_PREPRO`. If you add a preprocessed data source, you must remove it before adding another.

See Also

[QmpDupGrpsAddPreProcDataSrc](#)

Example

```
long lDataSourceID;
MpHnd hDupGrps;
MpHnd hRemPreproDatSrc;
QRESULT presult;

hRemPreproDatSrc = QmpDupGrpsRemPreProcDataSrc ( hDupGrps,
                                                lDataSourceID, &presult );
```

QmpDupGrpsSetExpressionNumChars

SETS THE CHARACTER LENGTH OF A SUBSTRING USED IN FLDVAL RANKING PRIORITY.

Syntax

```
void QmpDupGrpsSetExpressionNumChars( MpHnd in_hDupGrps, int  
    in_iNumChars, QRESULT* out_pResult );
```

in_hDupGrps
Handle to the dupe groups phase. *Input*.

in_iNumChars
Number of characters in field's substring. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function is valid only for QMS_RANKING_PRIORITY_FLDVAL master ranking.

This function is used with **QmpDupGrpsSetExpressionStartPos** to create a substring of the field string used in the fldval master ranking scheme. **QmpDupGrpsSetExpressionStartPos** sets the starting position of the substring; **QmpDupGrpsSetExpressionNumChars** sets the length of the substring.

Example

```
QmpDupGrpsSetExpressionNumChars( in_hDupGrps, in_iNumChars,  
    &out_pResult );
```

QmpDupGrpsSetExpressionStartPos

SETS THE STARTING POSITION OF A SUBSTRING USED IN FLDVAL RANKING PRIORITY.

Syntax

```
void QmpDupGrpsSetExpressionStartPos( MpHnd in_hDupGrps, int  
                                     in_iStart, QRESULT* out_pResult );  
  
in_hDupGrps  
    Handle to the dupe groups phase. Input.  
in_iStart  
    The starting position of the field's substring. Input.  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

This function is valid only for QMS_RANKING_PRIORITY_FLDVAL master ranking.

This function is used with `QmpDupGrpsSetExpressionNumChars` to create a substring of the field string used in the fldval master ranking scheme.

`QmpDupGrpsSetExpressionStartPos` sets the starting position of the substring; `QmpDupGrpsSetExpressionNumChars` sets the length of the substring.

Example

```
QmpDupGrpsSetExpressionStartPos( in_hDupGrps, in_iStart,  
                                &out_pResult );
```

QmpDupGrpsSetFldValPriority

SETS FLDVAL PRIORITY FOR DUPE GROUP MASTER SELECTION.

Syntax

```
void QmpDupGrpsSetFldValPriority( MpHnd in_hDupGrps,
    QMS_RANKING_PRIORITY in_Priority, const char*
    in_pszFldName, QMS_FLDEVAL_OPER in_Operator, MpHnd in_hVar,
    QRESULT* out_pResult );
```

in_hDupGrps
Handle to the dupe groups phase. *Input*.

in_Priority
Priority to use. The only valid value is QMS_RANKING_PRIORITY_FLDVAL.
Input.

in_pszFldName
Field to use in ranking. *Input*.

in_Operator
Operator used to construct the comparison expression. *Input*.

in_hVar
Handle to variant containing the constant value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the fldval rule for determining the master dupe in a dupe group.

Example

```
QmpDupGrpsSetFldValPriority ( in_hDupGrps,
    QMS_RANKING_PRIORITY_FLDVAL, "FieldA",
    QMS_FLDEVAL_OPER_EQUAL, in_hVar, &out_pResult );
```

QmpDupGrpsSetMaxDupGrpMem

SETS THE MAXIMUM NUMBER OF ALLOWABLE DUPE GROUP MEMBERS.

Syntax

```
void QmpDupGrpsSetMaxDupGrpMem ( MpHnd in_hDupGrps, int  
                                in_iMaxNum, QRESULT* out_pResult );  
  
in_hDupGrps  
    Handle to dupe groups. Input.  
  
in_iMaxNum  
    Maximum number of dupe group members. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

QmpDupGrpsSetMaxDupGrpMem sets the maximum number of allowed dupe group members. If the number of maximum dupe group members is exceeded in a given dupe group, a warning is generated. You can choose to ignore this warning, or take other actions. If you choose to ignore this warning, the dupe group is generated normally.

This function was implemented to provide a sanity check when performing matching. It is possible to set up matching criteria that might produce more duplicate records than expected. The function is intended to warn the application programmer of such possibilities and to take appropriate action.

See Also

[QmpDupGrpsGetMaxDupGrpMem](#)

Example

See example on [page 430](#).

QmpDupGrpsSetNthRecInterval

SETS DUPLICATE GROUPS NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpDupGrpsSetNthRecInterval ( MpHnd in_hDupGrps, long
                                    in_lInterval , QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups. Input.
in_lInterval
    Dupe groups Nth record event interval. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the dupe groups Nth record event interval.

The dupe groups can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the dupe groups processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

[QmpDupGrpsGetNthRecInterval](#), [QmpDupGrpsRegEveryNthRecFunc](#),
[QmpDupGrpsUnregEveryNthRecFunc](#)

Example

See example on [page 430](#).

QmpDupGrpsSetSimplePriority

SETS DATALIST AND RANDOM PRIORITIES FOR DUPE GROUP MASTER SELECTION.

Syntax

```
void QmpDupGrpsSetSimplePriority ( MpHnd in_hDupGrps,
                                   QMS_RANKING_PRIORITY in_Priority, QRESULT* out_pResult );
```

in_hDupGrps
Handle to the dupe groups phase. *Input*.

in_Priority
Priority to use. The only valid values are
QMS_RANKING_PRIORITY_DATALIST and
QMS_RANKING_PRIORITY_RANDOM. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the datalist and random rules for determining the master dupe in a dupe group.

Example

```
QmpDupGrpsSetSimplePriority ( hDupGrps,
                               QMS_RANKING_PRIORITY_DATALIST, &out_pResult );
```

QmpDupGrpsSetSortOrder

SETS THE SORT ORDER FOR ALL DUPES IN A DUPE GROUP.

Syntax

```
void QmpDupGrpsSetSortOrder( MpHnd in_hDupGrps, const char*
    in_pszFldName, QMS_SORT_ORDER in_Order, QRESULT*
    out_pResult );
```

in_hDupGrps

Handle to the dupe groups phase. *Input*.

in_pszFldName

Field to sort on. *Input*.

in_Order

Sort order. *Input*.

The valid enums are:

QMS_SORT_ORDER_NONE

QMS_SORT_ORDER_ASCENDING

QMS_SORT_ORDER_DESCENDING

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function determines how all dupes in a dupe group will be sorted. A master duplicate is chosen after all the dupes have been sorted.

Example

```
QmpDupGrpsSetSortOrder( in_hDupGrps, "FieldX",
    QMS_SORT_ORDER_DESCENDING, &out_pResult );
```

QmpDupGrpsSetThreshold

SETS THE DUPE GROUPS PHASE THRESHOLD.

Syntax

```
void QmpDupGrpsSetThreshold( MpHnd in_hDupGrps, int
                             in_iThreshold, QRESULT* out_pResult );
in_hDupGrps
Handle to dupe groups. Input.
in_iThreshold
Dupe groups threshold. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

The dupe groups threshold determines which records in the match result table are accepted into the dupe groups table during the dupe groups “build” subphase. If the threshold is set, the score of a match result table record must be equal to or greater than the threshold to be accepted into the dupe groups table. If the application does not set the dupe groups threshold, all records from the match result table are accepted into the dupe groups table.

The best application of this new functionality is to run the record matcher at a low threshold, then run the dupe groups phase at various thresholds equal to or greater than the record matcher threshold. Note that running the dupe groups phase at a threshold lower than the record matcher threshold doesn’t make any sense: it gives the same results as not setting the dupe groups threshold at all.

Tweaking the dupe groups threshold is useful when generating a near miss report. This report lists the dupe groups generated by a job, as well as the records that almost made it into those dupe groups. If you are dissatisfied with the results of a near miss report, you can change the dupe groups threshold (which will increase or decrease the number of duplicates), rerun the dupe groups phase, and create a new near miss report. You do not need to rerun the data input or record matching phases.

See Also

[QmpDupGrpsGetThreshold](#).

Example

See example on [page 430](#).

QmpDupGrpsSetTreatmentPriority

SETS SMALLEST AND LARGEST PRIORITIES FOR DUPE GROUP MASTER SELECTION.

Syntax

```
void QmpDupGrpsSetTreatmentPriority( MpHnd in_hDupGrps,
    QMS_RANKING_PRIORITY in_Priority, const char*
    in_pszFldName, QMS_CONSOL_TREAT in_TreatmentType, QRESULT*
    out_pResult );
```

in_hDupGrps

Handle to the dupe groups phase. *Input*.

in_Priority

Priority to use. The only valid values are
QMS_RANKING_PRIORITY_SMALLEST and
QMS_RANKING_PRIORITY_LARGEST. *Input*.

in_pszFldName

Field to use in ranking. *Input*.

in_TreatmentType

Type of field being tested. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets the smallest and largest rules for determining the master dupe in a dupe group.

Example

```
QmpDupGrpsSetTreatmentPriority( in_hDupGrps,
    QMS_RANKING_PRIORITY_SMALLEST, "FieldX",
    QMS_CONSOL_TREAT_STRING, &out_pResult );
```

QmpDupGrpsTblClose

CLOSES THE TABLE REPRESENTATION USED BY THE DUPE GROUPS.

Syntax

```
void QmpDupGrpsTblClose ( MpHnd in_hDupGrps, QRESULT*  
                           out_pResult );  
  
in_hDupGrps  
    Handle to dupe groups. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

If `QmpDupGrpsClear` is called, `QmpDupGrpsTblClose` must be called in order to destroy the file.

Example

```
QmpDupGrpsTblClose ( in_hDupGrps, &out_pResult );
```

QmpDupGrpsTempDGTblClose

CLOSES THE TEMPDGT.

Syntax

```
void QmpDupGrpsTempDGTblClose ( MpHnd in_hDupGrps, QRESULT*  
    out_pResult );
```

in_hDupGrps
Handle to dupe groups. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Call **QmpDupGrpsTempDGTblClose** to close the table used as the tempDGT (so that another application may use it, for example).

See Also

QmpDupGrpsUseTempDGTbl, **QmpDupGrpsClear**.

Example

```
QmpDeclHnd( hDupeGroups );  
QRESULT qres;  
  
QmpDupGrpsTempDGTableClose( hDupeGroups, &qres );
```

QmpDupGrpsUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpDupGrpsUnregEveryNthRecFunc ( MpHnd in_hDupGrps,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function unregisters the specified event handler from the dupe groups phase, so that the event handler will no longer be notified of every Nth record processed.

The dupe groups phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
MpHnd hDupGrps;
QMS_EVERY_NTHREC_FUNC pEveryNthRecordClient;
QRESULT presult;

QmpDupGrpsUnregEveryNthRecFunc ( hDupGrps,
                                         pEveryNthRecordClient, &presult );
```

QmpDupGrpsUseDataLstSvc

SPECIFIES THE DATA LIST SERVICE OBJECT TO BE USED BY THE DUPE GROUPS OBJECT.

Syntax

```
void QmpDupGrpsUseDataLstSvc( MpHnd in_hDupGrps, MpHnd  
                               in_hDataLstSvc, QRESULT* out_pResult );
```

in_hDupGrps
Handle to dupe groups object. *Input*.

in_hDataLstSvc
Handle to data list service object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDupGrpsUseDataLstSvc allows the application to attach a data list service object to the dupe groups phase.

The library will produce an error if a data list service object has not been given to the dupe groups phase once it has been started.

See Also

QmpDataLstSvcCreate

Example

See example on [page 430](#).

QmpDupGrpsUseMatRes

SPECIFIES THE MATCH RESULT OBJECT TO BE USED BY THE DUPE GROUPS OBJECT.

Syntax

```
void QmpDupGrpsUseMatRes( MpHnd in_hDupGrps, MpHnd in_hMatRes,
                           QRESULT* out_pResult );
```

in_hDupGrps
Handle to dupe groups object. *Input*.

MpHnd in_hMatRes
Handle to match result object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpDupGrpsUseMatRes allows the application to attach a match result object to the dupe groups phase.

The library will produce an error if a match result object has not been given to the dupe groups phase once it has been started.

See Also

QmpMatResCreate

Example

See example on [page 430](#).

QmpDupGrpsUseRecMat

SPECIFIES THE RECORD MATCHER TO BE USED BY THE DUPE GROUPS OBJECT.

Syntax

```
void QmpDupGrpsUseRecMat ( MpHnd in_hDupGrps, MpHnd  
    in_hRecMat, QRESULT* out_pResult );
```

in_hDupGrps

Handle to dupe groups object. *Input*.

in_hRecMat

Handle to record matcher object. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpDupGrpsUseRecMat allows the application to attach a record matcher object to the dupe groups phase.

The dupe groups phase utilizes the record matcher to make comparisons between master and subordinate duplicates within a dupe group.

The library will produce an error if a data list service object has not been given to the dupe groups phase once it has been started.

See Also

[QmpRecMatCreate](#)

Example

See example on [page 430](#).

QmpDupGrpsUseTbl

SPECIFY THE TABLE TO BE USED BY THE DUPE GROUPS OBJECT.

Syntax

```
void QmpDupGrpsUseTbl ( MpHnd in_hDupGrps, MpHnd in_hTbl,
    QRESULT* out_pResult );
```

in_hDupGrps
Handle to dupe groups object. *Input*.

in_hTbl
Handle to the dupe groups table object to use. This *must* be a CodeBase table. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The dupe groups phase uses a table to store the internal representation of the dupe groups data. A table is assigned to the phase in its creation function **QmpDupGrpsCreate**.

If you wish to assign another table to the dupe groups phase you must clear the current dupe groups object by calling **QmpDupGrpsClear**, and then call **QmpDupGrpsUseTbl**.

The dupe groups table *must* be a CodeBase table.

The library will produce an error if a table has not been given to the dupe groups phase once it has been started.

The dupe groups table should not be destroyed until the uniques report phase is finished executing.

Example

```
MpHnd hDupGrps;
MpHnd hTable;
QRESULT presult;

QmpDupGrpsUseTbl ( hDupGrps, hTable, &presult );
```

QmpDupGrpsUseTempDGTbl

GIVES A TABLE TO THE DUPE GROUPS OBJECT TO BE USED AS THE TEMPDGT.

Syntax

```
void QmpDupGrpsUseTempDGTbl ( MpHnd in_hDupGrps, MpHnd
                               in_hTbl, QRESULT* out_pResult );
in_hDupGrps
    Handle to dupe groups object. Input.
in_hTbl
    Handle to table to use for tempDGT. This must be a CodeBase table. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The tempDGT *must* be a CodeBase table.

The dupe groups phase requires a tempDGT; `QmpDupGrpsUseTempDGTbl` is used to hand a table to the dupe groups object to be used as a tempDGT. If you want to use another table as the tempDGT, clear the dupe groups object by calling `QmpDupGrpsClear`, and call `QmpDupGrpsUseTempDGTbl` with another table.

To close the tempDGT (so that another application may use it, for example), call `QmpDupGrpsTempDGTblClose`.

See Also

`QmpDupGrpsTempDGTblClose`, `QmpDupGrpsClear`.

Example

```
QmpDeclHnd( hDupeGroups );
QmpDeclHnd( hDuplicatesTempTable );
QRESULT qres;

hDuplicatesTempTable = QmpTblCbCreate( "", "dupestemptable", &qres
                                      );
QmpDupGrpsUseTempDGTbl( hDupeGroups, hDuplicatesTempTable, &qres
                        );
```

`QmpDupGrpsUseTempDGTbl`

Function Class: QmpDupsRpt*

DUPLICATES REPORT FUNCTIONS.

Quick Reference

Function	Description	Page
QmpDupsRptAddPreProcDataSrc	Specifies the preprocessed data source to be used by the duplicates report phase.*	490
QmpDupsRptCreate	Creates a duplicates report.	491
QmpDupsRptGetNthRecInterval	Gets duplicates report Nth record event interval.	492
QmpDupsRptRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	493
QmpDupsRptSetDupeGrpId	Specifies whether to print dupe group column IDs in the report.	494
QmpDupsRptSetDupeGrpScore	Specifies whether to print dupe group column scores in the report.	495
QmpDupsRptSetGrouping	Specifies desired type of grouping for duplicate records.	496
QmpDupsRptSetListId	Specifies whether to print data list ID columns in the report.	497
QmpDupsRptSetNthRecInterval	Sets duplicates report Nth record event interval.	498
QmpDupsRptSetPrintKeys	Specifies whether to print record primary key columns in the report.	499
QmpDupsRptSetPrintMatRes	Specifies whether to print record match result columns in the report.	500
QmpDupsRptSetPrintSrc	Specifies whether to print a record source ID column in the report.	501
QmpDupsRptUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	502
QmpDupsRptUseCons	Gives the data consolidation phase to a duplicates report.	503
QmpDupsRptUseDupGrp	Specifies the dupe groups object to use for the duplicates report.	504

QmpDupsRptAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE DUPLICATES REPORT PHASE.*

Syntax

```
void QmpDupsRptAddPreProcDataSrc ( MpHnd in_hDupsRpt, MpHnd
    in_hDataSrc, QRESULT* out_pResult

in_hDupsRpt
    Handle to duplicates report object. Input.
in_hDataSrc
    Handle to preprocessed data source to be used. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the duplicates report phase.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

Example

```
printf( "Starting Duplicate Report phase:\n" );
QmpDupsRptAddPreProcDataSrc( hDuplicatesReport,
    hPreProDataSrc, &qres );
QmpPhaseStart( hDuplicatesReport, &qres );
```

QmpDupsRptCreate

CREATES A DUPLICATES REPORT.

Syntax

```
MpHnd QmpDupsRptCreate ( QRESULT* out_pResult );  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to duplicates report if successful, or `NULL` if there is an error.

Notes

If the creation function fails, an error code is returned in the *out_pResult* parameter. Application programs should always test the result code for success.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
/* Creates a duplicates report and checks for success */  
MpHnd hDupsRpt = QmpDupsRptCreate ( hJob, &qres );  
if (QmpUtilFailed( qres ))  
    ...error handling
```

QmpDupsRptGetNthRecInterval

GETS DUPLICATES REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpDupsRptGetNthRecInterval ( MpHnd in_hDupsRpt,
                                    QRESULT* out_pResult );
in_hDupsRpt
    Handle to duplicates report. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns duplicates report Nth record event interval if successful, or -1 if there is an error.

Notes

This function gets the duplicates report phase Nth record event interval.

The duplicates report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the duplicates report phase processes 100 records, it will send out a notification to *all* registered clients.

See Also

[QmpDupsRptRegEveryNthRecFunc](#), [QmpDupsRptSetNthRecInterval](#),
[QmpDupsRptUnregEveryNthRecFunc](#)

Example

```
lDupsRepNthRecInterval = QmpDupsRptGetNthRecInterval
                           (hDupsRpt, pResult );
```

QmpDupsRptRegEveryNthRecFunc

REGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpDupsRptRegEveryNthRecFunc ( MpHnd in_hDupsRpt,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hDupsRpt
    Handle to duplicates report. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code.
```

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The duplicates report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
if ( bEveryNth ) {
    QmpDupsRptRegEveryNthRecFunc( hDuplicatesReport,
                                    pEveryNthRecordClient, &qres );
    QmpDupsRptSetNthRecInterval( hDuplicatesReport, 200, &qres
                                );
}
```

QmpDupsRptSetDupeGrpId

SPECIFIES WHETHER TO PRINT DUPE GROUP COLUMN IDs IN THE REPORT.

Syntax

```
void QmpDupsRptSetDupeGrpId( MpHnd in_hDupsRpt, QBOOL  
    in_bDupeGrpId, QRESULT* out_pResult );
```

in_hDupsRpt

Handle to duplicates report object. *Input*.

in_bDupeGrpId

If QTRUE, print dupe group IDs. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function specifies whether to print dupe group column IDs in the report.

Example

```
QmpDupsRptSetDupeGrpId(hDupsRpt, QTRUE, pResult );
```

QmpDupsRptSetDupeGrpScore

SPECIFIES WHETHER TO PRINT DUPE GROUP COLUMN SCORES IN THE REPORT.

Syntax

```
void QmpDupsRptSetDupeGrpScore( MpHnd in_hDupsRpt, QBOOL  
                                in_bDupeGrpScore, QRESULT* out_pResult );
```

in_hDupsRpt

Handle to duplicates report object. *Input*.

in_bDupeGrpScore

If QTRUE, print dupe group scores. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The dupe group score is the match score between a master duplicate and subordinate duplicate. All the subordinates in a dupe group are scored against the group master.

Example

```
QmpDupsRptSetDupeGrpScore(hDupsRpt, QTRUE, pResult );
```

QmpDupsRptSetGrouping

SPECIFIES DESIRED TYPE OF GROUPING FOR DUPLICATE RECORDS.

Syntax

```
void QmpDupsRptSetGrouping ( MpHnd in_hDupsRpt,
    QMS_MATCH_GROUPING in_Grouping, QRESULT* out_pResult );
```

in_hDupsRpt

Handle to duplicates report object. *Input*.

in_Grouping

Grouping type. *Input*.

Valid enums are:

QMS_GROUP_MATCH_BY_DUP_GROUP	Displays groups of matching records.
------------------------------	--------------------------------------

QMS_GROUP_MATCH_BY_MASTER_RANKING	Displays groups of matching records with the top-ranked “master duplicate” listed first, followed by the subordinate duplicates.
-----------------------------------	--

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function specifies the way matched records are displayed in the duplicates report.

Example

```
/* Specify how to group the duplicate records */
QmpDupsRptSetGrouping( hDupsRpt,
    QMS_GROUP_MATCH_BY_DUP_GROUP, &qres );
```

QmpDupsRptSetListId

SPECIFIES WHETHER TO PRINT DATA LIST ID COLUMNS IN THE REPORT.

Syntax

```
void QmpDupsRptSetListId( MpHnd in_hDupsRpt, QBOOL in_bListId,  
                           QRESULT* out_pResult );
```

in_hDupsRpt
Handle to duplicates report object. *Input*.

in_bListId
If QTRUE, print data list IDs. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

A record's data list ID field records which data list the record belongs to.

Example

```
QmpDupsRptSetListId(hDupsRpt, QTRUE, pResult );
```

QmpDupsRptSetNthRecInterval

SETS DUPLICATES REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpDupsRptSetNthRecInterval ( MpHnd in_hDupsRpt, long
                                in_lInterval, QRESULT* out_pResult );
in_hDupsRpt
    Handle to duplicates report. Input.
in_lInterval
    Duplicates report Nth record event interval. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the duplicate reports phase Nth record event interval.

The duplicate reports phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the duplicate reports phase processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

[QmpDupsRptGetNthRecInterval](#), [QmpDupsRptRegEveryNthRecFunc](#),
[QmpDupsRptUnregEveryNthRecFunc](#)

Example

```
if ( bEveryNth ) {
    QmpDupsRptRegEveryNthRecFunc( hDuplicatesReport,
                                   pEveryNthRecordClient, &qres );
    QmpDupsRptSetNthRecInterval( hDuplicatesReport, 200, &qres
                                );
}
```

QmpDupsRptSetPrintKeys

SPECIFIES WHETHER TO PRINT RECORD PRIMARY KEY COLUMNS IN THE REPORT.

Syntax

```
void QmpDupsRptSetPrintKeys (MpHnd in_hDupsRpt, QBOOL  

    in_bPrintKeys, QRESULT* out_pResult );
```

in_hDupsRpt
 Handle to duplicates report object. *Input*.

in_bPrintKeys
 If QTRUE, print record primary keys. *Input*.

out_pResult
 Result code. *Output*.

Return Value

None.

Notes

The primary key is a number (generally a record number) that is unique to a data source. It is usually the counter value when the records are read in by the data input phase. If a job has one data source, the primary key uniquely identifies a record. If a job has multiple data sources, the primary key and source ID values uniquely identify a record.

See Also

[QmpDupsRptSetPrintSrc](#), [QmpDupsRptSetPrintMatRes](#).

Example

```
/* Specify whether to print the primary keys, source */  

/* IDs, and match results */  

QmpDupsRptSetPrintKeys ( hDupsRpt, QTRUE, &qres );  

QmpDupsRptSetPrintSrc ( hDupsRpt, QTRUE, &qres );  

QmpDupsRptSetPrintMatRes ( hDupsRpt, QTRUE, &qres );
```

QmpDupsRptSetPrintMatRes

SPECIFIES WHETHER TO PRINT RECORD MATCH RESULT COLUMNS IN THE REPORT.

Syntax

```
void QmpDupsRptSetPrintMatRes (MpHnd in_hDupsRpt, QBOOL  
    in_bPrintMatchResult, QRESULT* out_pResult );
```

in_hDupsRpt

Handle to duplicates report object. *Input*.

in_bPrintMatchResult

If QTRUE, print record match results. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

See Also

[QmpDupsRptSetPrintSrc](#), [QmpDupsRptSetPrintKeys](#).

Example

```
/* Specify whether to print the primary keys, source */  
/* IDs, and match results */  
QmpDupsRptSetPrintKeys ( hDupsRpt, QTRUE, &qres );  
QmpDupsRptSetPrintSrc ( hDupsRpt, QTRUE, &qres );  
QmpDupsRptSetPrintMatRes ( hDupsRpt, QTRUE, &qres );
```

QmpDupsRptSetPrintSrc

SPECIFIES WHETHER TO PRINT A RECORD SOURCE ID COLUMN IN THE REPORT.

Syntax

```
void QmpDupsRptSetPrintSrc ( MpHnd in_hDupsRpt, QBOOL
                           in_bPrintSources, QRESULT* out_pResult );
in_hDupsRpt
Handle to duplicates report object.
in_bPrintSources
If QTRUE, print record source IDs.
out_pResult
Result code.
```

Return Value

None.

Notes

The record source ID field identifies which data source the record came from.

See Also

[QmpDupsRptSetPrintKeys](#), [QmpDupsRptSetPrintMatRes](#).

Example

```
/* Specify whether to print the primary keys, source */
/* IDs, and match results */
QmpDupsRptSetPrintKeys ( hDupsRpt, QTRUE, &qres );
QmpDupsRptSetPrintSrc ( hDupsRpt, QTRUE, &qres );
QmpDupsRptSetPrintMatRes( hDupsRpt, QTRUE, &qres );
```

QmpDupsRptUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpDupsRptUnregEveryNthRecFunc ( MpHnd in_hDupsRpt,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hDupsRpt
Handle to duplicates report. *Input*.

in_Func
Pointer to event handler. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function unregisters the specified event handler from the duplicates report, so that the event handler will no longer be notified of every Nth record processed.

The duplicates report can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
QmpDupsRptUnregEveryNthRecFunc (hDupsRpt,
                                  pDupsRptEventHandler, pResult );
```

QmpDupsRptUseCons

GIVES THE DATA CONSOLIDATION PHASE TO A DUPLICATES REPORT.

Syntax

```
void QmpDupsRptUseCons( MpHnd in_hDupsRpt, MpHnd in_hCons,
                         QRESULT* out_pResult );
```

in_hDupsRpt
Handle to duplicates report. *Input*.

in_hCons
Handle to the data consolidation phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The duplicates report requires the data consolidation phase to make sure that any updated duplicates are properly registered.

Example

```
QmpDupsRptUseCons ( hDupsRpt, hCons, &qres );
```

QmpDupsRptUseDupGrp

SPECIFIES THE DUPE GROUPS OBJECT TO USE FOR THE DUPLICATES REPORT.

Syntax

```
void QmpDupsRptUseDupGrp( MpHnd in_hDupsRpt, MpHnd  
    in_hdUpGrps, QRESULT* out_pResult );
```

in_hDupsRpt

Handle to duplicates report object. *Input*.

in_hdUpGrps

Handle to dupe groups object. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The dupe groups phase is used to determine master/subordinate status and dupe group membership.

Example

```
QmpDupsRptUseDupGrp ( hDupsRpt, DupeGroups, pResult );
```

Function Class: QmpFldMap*

FUNCTIONS FOR MAPPING RECORD FIELDS.

Field mapping takes fields from one record and maps them to another. A common use of field mapping in Centrus Merge/Purge is mapping fields from data source tables to the fields in the prototype record.

For example, imagine you have three tables. Table one has the fields LNAME, MIDNAME and ADDR. Table two has the fields LASTNAME, MIDDLENAME and ADDRESS1. Table three has the fields LAST, MIDDLE and ADDRESS. You wish to input these table records into Centrus Merge/Purge and match them. To do this using field mapping, create a prototype record containing the fields LAST, MIDDLE and ADDRESS. Map table one field LNAME to field LAST, field MIDNAME to MIDDLE, and field ADDR to ADDRESS. Map table two field LASTNAME to LAST, field MIDDLENAME to MIDDLE, and field ADDRESS1 to ADDRESS. Finally, map table 3 field LAST to LAST, field MIDDLE to MIDDLE, and field ADDRESS to ADDRESS.

Another use of field mapping is to use a subset of record fields in a merge/purge job. For example, if you have a reference table with many fields, and you wish to use only a few of them in your merge/purge job, you can map the few fields to associated fields in the prototype record.

Quick Reference

Function	Description	Page
QmpFldMapClear	Clears a field map.	506
QmpFldMapCreate	Creates a field map.	507
QmpFldMapDestroy	Destroys a field map.	509
QmpFldMapIsValid	Determines if a field map is valid.	510
QmpFldMapMapFldToFld	Maps the field of a record to the field of another record.	511
QmpFldMapUseRecs	Gives the pair of mapped records to the field map.	512

QmpFldMapClear

CLEARs A FIELD MAP.

Syntax

```
void QmpFldMapClear( MpHnd in_hFldMap, QRESULT* out_pResult );
```

in_hFldMap
Handle to field map object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function clears a field map definition.

Example

```
MpHnd hFldMap;  
QRESULT qres;  
  
QmpFldMapClear( hFldMap, &qres );
```

QmpFldMapCreate

CREATES A FIELD MAP.

Syntax

```
MpHnd QmpFldMapCreate( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns a handle to the field map if successful, or `NULL` if there is an error.

Notes

Once a field map is created, it must be further defined. Use `QmpFldMapUseRecs` to define the pair of records being mapped, and `QmpFldMapMapFldToFld` for each field you want to map between records.

If the creation function fails, an error code is returned in the `out_pResult` parameter. Application programs should always test the result code for success.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

`QmpFldMapUseRecs`, `QmpFldMapMapFldToFld`.

Example

```
QmpDeclHnd( hTable );
QmpDeclHnd( hTableOut );
QmpDeclHnd( hMyRec );
QmpDeclHnd( hProtoRec );
QmpDeclHnd( hDataInp );
QmpDeclHnd( hDataSrc );
QmpDeclHnd( hFldMap );
QmpDeclHnd( hVar );
const char* szFieldName;
const char* szValue;
hFldMap = QmpFldMapCreate( &qres );

/* set up the table */
hTable = QmpTblFTxtCreate( "", "..\\testwrap\\1000.sdf", &qres );
QmpTblOpen( hTable, &qres );
szFieldName = QmpTblGetFldName( hTable, 0, &qres );
printf( "Fieldname is: %s\n", szFieldName );
szFieldName = QmpTblGetFldName( hTable, 1, &qres );
printf( "Fieldname is: %s\n", szFieldName );
szFieldName = QmpTblGetFldName( hTable, 2, &qres );
printf( "Fieldname is: %s\n", szFieldName );
QmpTblMoveFirst( hTable, &qres );
hProtoRec = QmpRecCreate( &qres );
QmpTblGetFlds( hTable, hProtoRec, &qres );
QmpTblFillRec( hTable, hProtoRec, &qres );
hVar = QmpRecGetFldByHnd( hProtoRec, 0, &qres );
szValue = QmpVarGetString( hVar, &qres );
```

QmpFldMapCreate

```
hVar = QmpRecGetFldByHnd( hProtoRec, 1, &qres );
szValue = QmpVarGetString( hVar, &qres );
hVar = QmpRecGetFldByHnd( hProtoRec, 2, &qres );
szValue = QmpVarGetString( hVar, &qres );
hDataInp = QmpDataInpCreate( &qres );
hDataSrc = QmpDataSrcTblCreate( "Data Source", 2, &qres );
hMyRec = QmpRecCreate( &qres );

/* use */
QmpRecAddByType( hMyRec, "First", QMS_VARIANT_STRING, 15, 0, &qres );
QmpRecAddByType( hMyRec, "Last", QMS_VARIANT_STRING, 15, 0, &qres );
QmpRecAddByType( hMyRec, "Addr", QMS_VARIANT_STRING, 35, 0, &qres );
remove( "..\\testwrap\\test.sdf" );
remove( "..\\testwrap\\test.fmt" );
hTableOut = QmpTblFTxtCreate( "", "..\\testwrap\\test.sdf", &qres );
QmpTblDefineFlds( hTableOut, hMyRec, &qres );
QmpTblCreateRep( hTableOut, &qres );

/* setup the field map */
QmpFldMapUseRecs( hFldMap, hProtoRec, hMyRec, &qres );
QmpFldMapMapFldToFld( hFldMap, "FirstName", "First", &qres );
QmpFldMapMapFldToFld( hFldMap, "LastName", "Last", &qres );
QmpFldMapMapFldToFld( hFldMap, "Address", "Addr", &qres );
QmpDataSrcUseTbl( hDataSrc, hTable, &qres );
QmpDataSrcUseFldMap( hDataSrc, hFldMap, &qres );
QmpDataSrcSetRecProto( hDataSrc, hProtoRec, &qres );
QmpDataInpAddDataSrc( hDataInp, hDataSrc, &qres );
QmpTblFillMappedRec( hTable, hFldMap, hMyRec, &qres );
hVar = QmpRecGetFldByHnd( hMyRec, 0, &qres );
szValue = QmpVarGetString( hVar, &qres );
hVar = QmpRecGetFldByHnd( hMyRec, 1, &qres );
szValue = QmpVarGetString( hVar, &qres );
hVar = QmpRecGetFldByHnd( hMyRec, 2, &qres );
szValue = QmpVarGetString( hVar, &qres );
QmpTblSetMappedRec( hTableOut, hFldMap, hMyRec, &qres );

/* and the next record */
QmpTblMoveBy( hTable, 1, &qres );
QmpTblFillMappedRec( hTable, hFldMap, hMyRec, &qres );
hVar = QmpRecGetFldByHnd( hMyRec, 0, &qres );
szValue = QmpVarGetString( hVar, &qres );
hVar = QmpRecGetFldByHnd( hMyRec, 1, &qres );
szValue = QmpVarGetString( hVar, &qres );
hVar = QmpRecGetFldByHnd( hMyRec, 2, &qres );
szValue = QmpVarGetString( hVar, &qres );
QmpTblSetMappedRec( hTableOut, hFldMap, hMyRec, &qres );
QmpTblClose( hTableOut, &qres );
QmpTblDestroy( hTableOut, &qres );

/* clean up */
QmpTblClose( hTable, &qres );
QmpTblDestroy( hTable, &qres );
QmpRecDestroy( hMyRec, &qres );
QmpRecDestroy( hProtoRec, &qres );
QmpDataSrcDestroy( hDataSrc, &qres );
QmpPhaseDestroy( hDataInp, &qres );
QmpFldMapDestroy( hFldMap, &qres );
```

QmpFldMapDestroy

DESTROYS A FIELD MAP.

Syntax

```
void QmpFldMapDestroy( MpHnd in_hFldMap, QRESULT* out_pResult
    );
```

in_hFldMap
Handle to field map. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

Example

See example on [page 507](#).

QmpFldMapIsValid

DETERMINES IF A FIELD MAP IS VALID.

Syntax

```
QBOOL QmpFldMapIsValid( MpHnd in_hFldMap, QRESULT* out_pResult  
);
```

in_hFldMap

Handle to field map. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if the field map is valid, else QFALSE.

Example

```
MpHnd hFldMap;  
QRESULT qres;  
QBOOL qIsFldMapValid;  
  
qIsFldMapValid = QmpFldMapIsValid( hFldMap, &qres );
```

QmpFldMapMapFldToFld

MAPS THE FIELD OF A RECORD TO THE FIELD OF ANOTHER RECORD.

Syntax

```
void QmpFldMapMapFldToFld( MpHnd in_hFldMap, const char*
    in_szInFldName, const char* in_szOutFldName, QRESULT*
    out_pResult );
```

in_hFldMap
Handle to field map. *Input*.

in_szInFldName
Field name of inner field (field being mapped to). *Input*.

in_szOutFldName
Field name of outer field (field being mapped from). *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Call **QmpFldMapMapFldToFld** for each pair of fields you want to map between records. For example, if you have a pair of records and want to map two sets of fields between them, you would call **QmpFldMapMapFldToFld** twice.

Example

See example on [page 507](#).

QmpFldMapUseRecs

GIVES THE PAIR OF MAPPED RECORDS TO THE FIELD MAP.

Syntax

```
void QmpFldMapUseRecs( MpHnd in_hFldMap, MpHnd in_hInnerRec,
```

```
    MpHnd in_hOuterRec, QRESULT* out_pResult );
```

in_hFldMap

Handle to field map. *Input*.

in_hInnerRec

Handle to mapped record (destination record). *Input*.

in_hOuterRec

Handle to mapping record (source record). *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Call **QmpFldMapUseRecs** to define the pair of records you want to map fields between.

Example

See example on [page 507](#).

Function Class: QmpGlb*

GLOBAL (MISCELLANEOUS) FUNCTIONS.

The **QmpGlb*** class is a “catch-all” function grouping.

Quick Reference

Function	Description	Page
QmpGlbFillVers	Gets the library version number.	514
QmpGlbGetLog	Gets system wide log object.	515
QmpGlbIsLicenseDemo	Tests whether license is demo version.	516
QmpGlbIsLicenseLoaded	Tests whether a license is valid (loaded).	517
QmpGlbSetLicense	Sets the license	518

QmpGlbFillVers

GETS THE LIBRARY VERSION NUMBER.

Syntax

```
void QmpGlbFillVers ( char* io_szBuffer, long in_lSize, int*
                      out_iMajor, int* out_iMinor, int* out_iPatch, QRESULT*
                      out_pResult );

io_szBuffer
    Buffer to put version number into. Input, Output.
in_lSize
    Size of buffer. Input.
out_iMajor
    Major version. Output.
out_iMinor
    Minor version. Output.
out_iPatch
    Patch number. Output.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function gets the library version number. The version number is a combination of three numbers: the major version, the minor version, and the patch version. The major version is incremented each major release. The minor version is incremented each minor release, and the patch version is incremented each time a patch release is made.

Example

```
char buffer[TEST_BUF_SIZE];
int iMajor, iMinor, iPatch;

QmpGlbFillVers( buffer, sizeof(buffer), &iMajor,
                  &iMinor, &iPatch, &qres );
printf( "%s\n", buffer );
printf("Major, minor, and patch version: %d, %d, %d\n", iMajor,
      iMinor, iPatch );
```

This prints out:

Major, minor, and patch version: 1, 0, 0

QmpGlbGetLog

GETS SYSTEM WIDE LOG OBJECT.

Syntax

```
MpHnd QmpGlbGetLog ( QRESULT* out_pResult );  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to system wide log object if successful, or `NULL` if there is an error.

Notes

Use this function to fetch the handle of the system-wide log object.

Example

```
/* Create a callback function for the */  
/* QMS_EVENT_EVERYNTHRECORD event */  
void EventHandler( unsigned long in_uID, char* in_szName, long  
                  in_lRecordCount )  
{  
    MpHnd hLog = QmpGlbGetLog( &qres );  
    QmpLogPutCoutData( hLog, "%s processed %d records\n",  
                  in_szName, in_lRecordCount );  
}
```

The sample code produces this output if the data input phase is the originator of the event:

```
Data Input 1 processed 500 records  
Data Input 2 processed 1000 records  
Data Input 3 processed 1500 records  
. . .
```

QmpGlbIsLicenseDemo

TESTS WHETHER LICENSE IS DEMO VERSION.

Syntax

```
QBOOL QmpGlbIsLicenseDemo ( QRESULT* out_pResult );  
out_pResult  
    Return code. Output.
```

Return Value

Returns QTRUE if demo license is loaded, else QFALSE.

Notes

The “demo” version of the Centrus Merge/Purge license enables limited functionality. This function tests whether a demo license is currently loaded.

See Also

[QmpGlbSetLicense](#), [QmpGlbIsLicenseLoaded](#).

Example

```
QBOOL m_bIsDemo = QmsGlbIsLicenseDemo( &qres );  
if ( m_lRecordCount > 1000 && m_bIsDemo ) {  
    QmpChkresultCode( QRESULT_WARN_DEMO_EXCEEDED, "Demo license  
        volume exceeded" );  
}
```

QmpGlbIsLicenseLoaded

TESTS WHETHER A LICENSE IS VALID (LOADED).

Syntax

```
QBOOL QmpGlbIsLicenseLoaded ( QRESULT* out_pResult );
out_pResult
    Return code. Output.
```

Return Value

Returns QTRUE if valid license is loaded, else QFALSE.

Notes

A valid license file and password are required to invoke Centrus Merge/Purge. This function checks the license status.

See Also

[QmpGlbSetLicense](#), [QmpGlbIsLicenseDemo](#).

Example

```
QBOOL m_bLoaded = QmsGlbIsLicenseLoaded( &qres );
if ( m_lRecordCount > 1000 && m_bIsDemo ) {
    QmpChkresultCode(QRESULT_SEVERE_INVALID_LICENSE, "License file
        has not been loaded" );
}
```

QmpGlbSetLicense

SETS THE LICENSE

Syntax

```
void QmpGlbSetLicense( unsigned long password, const char*
    filename, QRESULT* out_pResult );
```

password
The unsigned long integer used as a password. *Input*.

filename
The fully qualified name of the password file. *Input*.

out_pResult
Return code. *Output*.

Return Value

None.

Notes

This function identifies the license file and password used when invoking Centrus Merge/Purge. It must be called when the application is initialized.

See Also

[QmpGlbIsLicenseLoaded](#), [QmpGlbIsLicenseDemo](#).

Example

```
char sLicPathAndFile[256];
unsigned long ulPassword;
hInifil = QmpIniFilCreate(
    "s:\\\\cmp\\\\win32\\\\release\\\\bin\\\\cmp.ini", pResult );
QmpIniFilReadItemValStrVB(
    hInifil,
    sLicPathAndFile,
    sizeof( sLicPathAndFile ),
    "[SETUP]",
    "LicenseFile",
    "",
    pResult );
ulPassword = QmpIniFilReadULong(
    hInifil,
    "[SETUP]",
    "Password",
    0,
    pResult );
QmpGlbSetLicense( ulPassword, sLicPathAndFile, pResult );
```

Function Class: QmplIdxGen*

INDEX GENERATOR PHASE FUNCTIONS—OBSOLETE.

The index generation phase, along with the QmpIdxGen* functions, are obsolete. The functions in this class may still be called, but no longer do any work. They have been kept as part of the C interface for backwards compatibility. You may safely remove calls to all **QmpIndexGen*** functions from any existing Centrus Merge/Purge application.

QmpIndxGenClear

OBSOLETE

While it is legal to call this function, it no longer does any work or returns a value.

QmpIdxGenCreate

OBSOLETE

While it is legal to call this function, it no longer does any work or returns a value.

QmpIndxGenGetIndxKeyCnt

OBSOLETE

While it is legal to call this function, it no longer does any work or returns a value.

QmpIdxGenIsValid

OBSOLETE

While it is legal to call this function, it no longer does any work or returns a value.

QmpIndxGenRemIndxKey

OBSOLETE

While it is legal to call this function, it no longer does any work or returns a value.

QmpIdxGenUseIdxKey

OBSOLETE

While it is legal to call this function, it no longer does any work or returns a value.

Function Class: QmpIdxKey*

FUNCTIONS FOR MANIPULATING INDEX KEY DEFINITIONS.

Index keys are defined by specifying fields on which to order the input records. Either single field or compound field index keys may be defined. Either the field's entire value or a sub-string of the field's value may be specified. The soundex value of a field's contents may also be used in defining an index key.

Index Keys should be defined with characteristics of the input data taken into consideration. For example, if the input data contains a telephone number field that is considered to be of high quality, then the telephone field would be a good field to use in an index key. However, even though the telephone number field is of high quality, 100% of the telephone field values are not necessarily present or correct, so additional fields should be considered for ordering. If the application is concentrating on households, then last name and address are important fields and would also be good candidates to be used in defining the index keys. If the input data contains a lot of Asian names along with a lot of western names, then the first name field might also be important, since many Asian cultures swap the order of their first and last names.

If any of the fields being considered for use in defining index keys are likely to contain values which occur frequently, such as "Smith" in a last name field, then that field should be combined with one or more other fields to form a compound field index key. This approach will aid the record matching process, since it will help bring records close together which are similar to one another across other fields. A good strategy for defining index keys is to identify the various fields which would generate useful orderings and create several compound-field index keys using each of the fields combined in different orders.

For example, suppose the fields "LastName", "Address", and "ZipCode" are identified as good fields for index orderings. One index key could be defined as the combination LastName, Address, and ZipCode, while another index key could be defined as the combination Address, ZipCode, and LastName, and yet another index key as the combination of ZipCode, LastName, and Address. Index keys formed using substrings or soundex transformations of these fields can also result in very useful orderings.

It is important to recognize that the index fields used to define index keys do not necessarily have anything to do with the match fields used in the matching process. They often do, but there is no requirement for any relationship.

Quick Reference

Function	Description	Page
QmpIdxKeyAddCompByHnd	Adds a key component by field handle to the index key.	529
QmpIdxKeyAddCompByName	Adds a key component by name to the index key.	531
QmpIdxKeyClear	Clears an index key.	533
QmpIdxKeyCreate	Creates an index key.	534
QmpIdxKeyDestroy	Destroys an index key.	535
QmpIdxKeyFillExpr	Fills a pre-allocated string with the index key expression.	536
QmpIdxKeyGetIgnoreAlpha	Gets the index key “ignore letters” setting.	537
QmpIdxKeyGetIgnoreNum	Gets the index key “ignore numerals” setting.	538
QmpIdxKeyGetIgnorePunct	Gets “ignore punctuation” field index key property.	539
QmpIdxKeyGetIgnoreSpaces	Gets the “index key ignore white space” setting.	540
QmpIdxKeyGetTblIdxName	Gets the name of an index key definition.	541
QmpIdxKeyIsValid	Determine whether an index key is valid.	542
QmpIdxKeySetIgnoreAlpha	Sets an index key definition’s “ignore letters” setting.	543
QmpIdxKeySetIgnoreNum	Sets an index key definition’s “ignore numerals” setting.	544
QmpIdxKeySetIgnorePunct	Sets “ignore punctuation” field index key property.	545
QmpIdxKeySetIgnoreSpaces	Sets an index key definition’s “ignore white space” setting.	546
QmpIdxKeySetRecProto	Specifies the data record’s data field structure via an application-owned record.	547
QmpIdxKeySetTblIdxName	Sets the name of an index key definition.	548
QmpIdxKeyUsesSoundex	Inquires whether an index key uses Soundex.	549

QmpIdxKeyAddCompByHnd

ADDS A KEY COMPONENT BY FIELD HANDLE TO THE INDEX KEY.

Syntax

```
QBOOL QmpIdxKeyAddCompByHnd (MpHnd in_hIdxKey, int in_iFld,
    QMS_INDXKEY_XFORM_TYPE in_xform, int in_iCharStartPos, int
    in_iNumChars, QRESULT* out_pResult );
```

in_hIdxKey

Handle to index key. *Input*.

in_iFld

Field handle of index key component. *Input*.

in_xform

Transformation to apply to the field. *Input*.

Valid enums are:

QMS_INDXKEY_XFORM_IDENTITY	Field not transformed.
----------------------------	------------------------

QMS_INDXKEY_XFORM_SOUNDEX	Soundex equivalent of field used.
---------------------------	-----------------------------------

in_iCharStartPos

Starting position in first field (one-based, just like field criteria). *Input*.

in_iNumChars

Number of chars from starting position in first field. Must be 1 or above.
Input.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if key component is successfully added, else QFALSE.

Notes

The starting character of a field is considered to have an index of 1. The number of characters from the starting position includes the starting character. For example, in the following field:

J O N E S

- Character 1 is “J”
- The values *in_iCharStartPos* = 1 and *in_iNumChars* = 1 define the first character “J”
- The values *in_iCharStartPos* = 1 and *in_iNumChars* = 3 define the string “JON”
- The values *in_iCharStartPos* = 3 and *in_iNumChars* = 1000 define the string “NES” (An *in_iNumChars* value greater than the length of the field defines the portion of the field from the starting position to the end of the field.)

Example

```
/* Create index key on Address + Soundex(LastName) + */
/* ZipCode + Zip4 */
QmpIndxKeySetTblIndxName( hIndxKeyAddr, "Address", &qres );
QmpIndxKeyAddCompByHnd( hIndxKeyAddr, hAddress,
    QMS_INDEXKEY_XFORM_IDENTITY, 1, 0, &qres );
QmpIndxKeyAddCompByHnd( hIndxKeyAddr, hLastName,
    QMS_INDEXKEY_XFORM_SOUNDDEX, 1, 0, &qres );
QmpIndxKeyAddCompByHnd( hIndxKeyAddr, hZipCode,
    QMS_INDEXKEY_XFORM_IDENTITY, 1, 0, &qres );
QmpIndxKeyAddCompByHnd( hIndxKeyAddr, hZip4,
    QMS_INDEXKEY_XFORM_IDENTITY, 1, 0, &qres );
```

QmplIdxKeyAddCompByName

ADDS A KEY COMPONENT BY NAME TO THE INDEX KEY.

Syntax

```
QBOOL QmplIdxKeyAddCompByName ( MpHnd in_hIdxKey, const char*
    in_szFieldName, QMS_INDXKEY_XFORM_TYPE in_xform, int
    in_iCharStartPos, int in_iNumChars, QRESULT* out_pResult );

in_hIdxKey
    Handle to index key. Input.
in_szFieldName
    Field name of index key component. Input.
in_xform
    Transformation to apply to the field. Input.
    Valid enums are:
    

|                            |                                   |
|----------------------------|-----------------------------------|
| QMS_INDXKEY_XFORM_IDENTITY | Field not transformed.            |
| QMS_INDXKEY_XFORM_SOUNDEX  | Soundex equivalent of field used. |


in_iCharStartPos
    Starting position in first field (one-based, just like field criteria). Input.
in_iNumChars
    Number of chars from starting position in first field. Must be 1 or above.
    Input.
out_pResult
    Result code. Output.
```

Return Value

Returns QTRUE if key component is successfully added, else QFALSE.

Notes

The starting character of a field is considered to have an index of 1. The number of characters from the starting position includes the starting character. For example, in the following field:

J O N E S

- Character 1 is “J”
- The values *in_iCharStartPos* = 1 and *in_iNumChars* = 1 define the first character “J”
- The values *in_iCharStartPos* = 1 and *in_iNumChars* = 3 define the string “JON”
- The values *in_iCharStartPos* = 3 and *in_iNumChars* = 1000 define the string “NES” (An *in_iNumChars* value greater than the length of the field defines the portion of the field from the starting position to the end of the field.)

Example

```
/* Index Key setup. */
hIndexKey1 = QmpIdxKeyCreate( &qres );
hIndexKey2 = QmpIdxKeyCreate( &qres );
hIndexKey3 = QmpIdxKeyCreate( &qres );

QmpIdxKeySetRecProto( hIndexKey1, hProtoRec, &qres );
QmpIdxKeySetRecProto( hIndexKey2, hProtoRec, &qres );
QmpIdxKeySetRecProto( hIndexKey3, hProtoRec, &qres );

QmpIdxKeyAddCompByName( hIndexKey1, szAddress,
    QMS_INDEXKEY_XFORM_IDENTITY, 1, strlen(szAddress), &qres );
QmpIdxKeyAddCompByName( hIndexKey2, szLastName,
    QMS_INDEXKEY_XFORM_IDENTITY, 1, strlen(szLastName), &qres );
QmpIdxKeyAddCompByName( hIndexKey3, szZipCode,
    QMS_INDEXKEY_XFORM_IDENTITY, 1, strlen(szZipCode), &qres );
```

QmpIdxKeyClear

CLEAR AN INDEX KEY.

Syntax

```
void QmpIdxKeyClear ( MpHnd in_hIdxKey, QRESULT* out_pResult
) ;
```

in_hIdxKey
Name of index key. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the specified index key object back to its initial state.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpIdxKeyClear (hIdxKey, pResult ) ;
```

QmpIdxKeyCreate

CREATES AN INDEX KEY.

Syntax

```
MpHnd QmpIdxKeyCreate ( QRESULT* out_pResult );  
out_pResult  
Result code. Output.
```

Return Value

Returns the handle to the index key if successful, or `NULL` if there is an error.

Notes

This function creates an index key. If the creation function fails, an error code is returned in the `out_pResult` parameter.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

[QmpIdxKeyDestroy](#)

Example

```
/* Creates an index key and checks for success */  
MpHnd hIndexKey = QmpIdxKeyCreate ( &qres );  
if (QmpUtilFailed( qres ))  
    ...error handling
```

QmpIdxKeyDestroy

DESTROYS AN INDEX KEY.

Syntax

```
void QmpIdxKeyDestroy ( MpHnd in_hIdxKey, QRESULT*  
    out_pResult );
```

in_hIdxKey
Handle to index key. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function destroys an index key.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

See Also

QmpIdxKeyCreate

Example

```
QmpIdxKeyDestroy (hIdxKey, &Result);
```

QmpIdxKeyFillExpr

FILLS A PRE-ALLOCATED STRING WITH THE INDEX KEY EXPRESSION.

Syntax

```
void QmpIdxKeyFillExpr ( MpHnd in_hIdxKey, char*  
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hIdxKey
Handle to index key. *Input*.

io_szBuffer
Expression buffer. *Input, Output*.

in_lSize
Caller-allocated size of buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Generates a string for the client containing the index key expression in a dBase-like format. The caller owns the returned string into which the expression is placed. Space for the contents of the string expression *io_szBuffer* must be allocated prior to the call.

Example

```
QmpIdxKeyFillExpr (hIdxKey, szBuffer, lSize, pResult );
```

QmpIdxKeyGetIgnoreAlpha

GETS THE INDEX KEY “IGNORE LETTERS” SETTING.

Syntax

```
QBOOL QmpIdxKeyGetIgnoreAlpha ( MpHnd in_hIdxKey, QRESULT*  
    out_pResult );
```

in_hIdxKey

Handle to index key definition. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns `QTRUE` if the index key definition ignores letters, `QFALSE` (the default) if the index key definition doesn't ignore letters or if there is an error.

Notes

Use this function to get the current “ignore letters” setting for an index key definition. If the setting is `QTRUE`, letters in fields will be ignored when building index keys with them.

For example, if the “ignore letters” setting is `QTRUE`, “200 Main”, “200 First”, and “200 Second” would all appear in an index key as “200”.

See Also

`QmpIdxKeySetIgnoreAlpha`.

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;  
  
bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );  
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );  
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );  
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );  
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );  
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );  
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );  
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeyGetIgnoreNum

GETS THE INDEX KEY “IGNORE NUMERALS” SETTING.

Syntax

```
QBOOL QmpIdxKeyGetIgnoreNum ( MpHnd in_hIdxKey, QRESULT*  
    out_pResult );
```

in_hIdxKey
Handle to index key definition. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the index key definition ignores numerals, QFALSE (the default) if it doesn't or if there is an error.

Notes

Use this function to get the current “ignore numerals” setting for an index key definition. If the setting is QTRUE, numerals in fields will be ignored when building index keys them.

For example, if the “ignore numerals” setting is QTRUE, “200 Main”, “300 Main”, and “400 Main” would all index as “Main”.

See Also

[QmpIdxKeySetIgnoreNum](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;

bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeyGetIgnorePunct

GETS “IGNORE PUNCTUATION” FIELD INDEX KEY PROPERTY.

Syntax

```
QBOOL QmpIdxKeyGetIgnorePunct ( MpHnd in_hIdxKey, QRESULT*  
    out_pResult );
```

in_hIdxKey

Handle to index key definition. *Input*.

out_pResult

Result code. *Output*.

Return Value

QTRUE if field punctuation is ignored, QFALSE if field punctuation is considered when building an index.

Notes

The ignore punctuation property directs the Centrus Merge/Purge library to either ignore or respect punctuation in the fields being indexed. If set to QTRUE, all punctuation characters are removed from the string before the field is used to build an index key. A punctuation character is any printing character which is not a space, number, or alphabetic character between ‘a’ and ‘z’ or ‘A’ and ‘Z’. This property is useful when indexing string fields containing abbreviations or titles such as “Dr.”, “St.”, “MD.”, or “P.O.”.

See Also

[QmpIdxKeySetIgnorePunct](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;

bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeyGetIgnoreSpaces

GETS THE “INDEX KEY IGNORE WHITE SPACE” SETTING.

Syntax

```
QBOOL QmpIdxKeyGetIgnoreSpaces ( MpHnd in_hIdxKey, QRESULT*  
                                out_pResult );
```

in_hIdxKey

Handle to index key. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if white spaces are ignored in fields used to build indexes, QFALSE (the default) if white spaces aren’t ignored or if there is an error.

Notes

Use this function to get the current “ignore white space” setting for an index key definition. If the setting is QTRUE, white space in fields will be ignored when building index keys.

For example, if the “ignore white space” setting is QTRUE, “MergePurge”, “Merge Purge”, and “ Merge Purge ” would all index as “MergePurge”.

See Also

[QmpIdxKeySetIgnoreSpaces](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;  
  
bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );  
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );  
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );  
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );  
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );  
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );  
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );  
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeyGetTblIdxName

GETS THE NAME OF AN INDEX KEY DEFINITION.

Syntax

```
const char* QmpIdxKeyGetTblIdxName ( MpHnd in_hIdxKey,
                                     QRESULT* out_pResult );
in_hIdxKey
    Handle to index key. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns pointer to index key name if successful, or `NULL` if there is an error.

Notes

QmpIdxKeyGetTblIdxName returns a pointer to a string containing the name of an index key definition. This name will be used to identify the index key in reports and output tables.

See Also

[QmpIdxKeySetTblIdxName](#)

Example

```
pIndexName = QmpIdxKeyGetTblIdxName (hIdxKey, pResult);
```

QmpIdxKeyIsValid

DETERMINE WHETHER AN INDEX KEY IS VALID.

Syntax

```
QBOOL QmpIdxKeyIsValid ( MpHnd in_hIdxKey, QRESULT*  
    out_pResult );
```

in_hIdxKey
Handle to index key. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the index key is valid, otherwise QFALSE.

Notes

An index key is valid if it has been properly allocated and initialized.

Example

```
bIsIndexKeyValid = QmpIdxKeyIsValid (hIdxKey, pResult );
```

QmpIdxKeySetIgnoreAlpha

SETS AN INDEX KEY DEFINITION'S "IGNORE LETTERS" SETTING.

Syntax

```
void QmpIdxKeySetIgnoreAlpha ( MpHnd in_hIdxKey, QBOOL
    in_bIgnoreAlpha, QRESULT* out_pResult );
in_hIdxKey
    Handle to index key definition. Input.
in_bIgnoreAlpha
    "Ignore letters" setting. QTRUE is on, QFALSE (the default) is off. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to set the "ignore letters" setting for an index key definition. If the setting is set to QTRUE, letters in fields will be ignored when building index keys with them.

For example, if the "ignore letters" setting is QTRUE, "200 Main", "200 First", and "200 Second" would all index as "200".

See Also

[QmpIdxKeyGetIgnoreAlpha](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;

bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeySetIgnoreNum

SETS AN INDEX KEY DEFINITION'S "IGNORE NUMERALS" SETTING.

Syntax

```
void QmpIdxKeySetIgnoreNum ( MpHnd in_hIdxKey, QBOOL
                             in_bIgnoreNum, QRESULT* out_pResult );

in_hIdxKey
    Handle to index key definition. Input.

in_bIgnoreNum
    "Ignore numerals" setting. QTRUE is on, QFALSE (default) is off. Input.

out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Use this function to set the current "ignore numerals" setting for an index key definition. If the setting is QTRUE, numerals in fields will be ignored when building index keys them.

For example, if the "ignore numerals" setting is QTRUE, "200 Main", "300 Main", and "400 Main" would all index as "Main".

See Also

[QmpIdxKeyGetIgnoreNum](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;

bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeySetIgnorePunct

SETS “IGNORE PUNCTUATION” FIELD INDEX KEY PROPERTY.

Syntax

```
void QmpIdxKeySetIgnorePunct ( MpHnd in_hIdxKey, QBOOL
    in_bIgnorePunct, QRESULT* out_pResult );
in_hIdxKey
    Handle to index key definition. Input.
in_bIgnorePunct
    Ignore punctuation property flag. QTRUE directs field punctuation to be
    ignored, QFALSE directs field punctuation to be considered in an index.
    Default is QFALSE. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The ignore punctuation property directs the Centrus Merge/Purge library to either ignore or respect punctuation in the fields being indexed. If set to QTRUE, all punctuation characters are removed from the string before the field is used to build an index key. A punctuation character is any printing character which is not a space, number, or alphabetic character between ‘a’ and ‘z’ or ‘A’ and ‘Z’. This property is useful when indexing string fields containing abbreviations or titles such as “Dr.”, “St.”, “MD.”, or “P.O.”.

See Also

[QmpIdxKeyGetIgnorePunct](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;

bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeySetIgnoreSpaces

SETS AN INDEX KEY DEFINITION'S "IGNORE WHITE SPACE" SETTING.

Syntax

```
void QmpIdxKeySetIgnoreSpaces ( MpHnd in_hIdxKey, QBOOL
                               in_bIgnoreSpaces, QRESULT* out_pResult );
in_hIdxKey
Handle to index key definition. Input.
in_bIgnoreSpaces
"Ignore white space"; QTRUE is on, QFALSE (the default) is off. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

Use this function to set the "ignore white space" setting for an index key definition. If the setting is QTRUE, white space in fields will be ignored when building index keys with them.

For example, if the "ignore white space" setting is QTRUE, "MergePurge", "Merge Purge", and " Merge Purge " would all appear in an index key as "MergePurge".

See Also

[QmpIdxKeyGetIgnoreSpaces](#).

Example

```
QBOOL bWhiteSpace, bPunct, bAlpha, bNumeral, bTest;

bWhiteSpace = QmpIdxKeyGetIgnoreSpaces( hIndexKey1, &qres );
bPunct = QmpIdxKeyGetIgnorePunct( hIndexKey1, &qres );
bAlpha = QmpIdxKeyGetIgnoreAlpha( hIndexKey1, &qres );
bNumeral = QmpIdxKeyGetIgnoreNum( hIndexKey1, &qres );
QmpIdxKeySetIgnoreSpaces( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnorePunct( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreAlpha( hIndexKey1, QTRUE, &qres );
QmpIdxKeySetIgnoreNum( hIndexKey1, QTRUE, &qres );
```

QmpIdxKeySetRecProto

SPECIFIES THE DATA RECORD'S DATA FIELD STRUCTURE VIA AN APPLICATION-OWNED RECORD.

Syntax

```
void QmpIdxKeySetRecProto ( MpHnd in_hIdxKey, MpHnd in_hRec,
                           QRESULT* out_pResult );
in_hIdxKey
    Handle to index key. Input.
in_hRec
    Handle to the prototype record. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function specifies to the index key the structure of the data records. The specified structure must be the same as that specified for all phases that use data records. Typically, the same application-owned record is used for this purpose throughout the application.

See Also

[QmpPhaseSetRecProto](#), [QmpDataDestSetRecProto](#),
[QmpDataLstSvcSetRecProto](#), [QmpDataSrcSetRecProto](#).

Example

```
QmpIdxKeySetRecProto ( hIndexKeyAddr, hSourceRecord, pResult
                        );
```

QmpIdxKeySetTblIdxName

SETS THE NAME OF AN INDEX KEY DEFINITION.

Syntax

```
void QmpIdxKeySetTblIdxName (MpHnd in_hIdxKey, const char*
    in_pszName, QRESULT* out_pResult );
in_hIdxKey
    Handle to index key. Input.
in_pszName
    Name of index key. This name must be 10 or fewer characters in length.
    Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

After creating an index key definition with `QmpIdxKeyCreate`, call `QmpIdxKeySetTblIdxName` to set the definition's name. This name will be used to identify the index in reports, output tables, and in the underlying database. This function has a different purpose than `QmpObjSetName`, which establishes the name of an object in your application, not the name to be used in the underlying database.

See Also

`QmpIdxKeyGetTblIdxName`, `QmpObjSetName`.

Example

```
QmpIdxKeySetTblIdxName ( hIndexKeyAddr, "AddrLine1",
    pResult );
```

QmpIdxKeyUsesSoundex

INQUIRES WHETHER AN INDEX KEY USES SOUNDEX.

Syntax

```
QBOOL QmpIdxKeyUsesSoundex ( MpHnd in_hIdxKey, QRESULT*  
    out_pResult );
```

in_hIdxKey
Handle to index key. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if index key uses soundex, else QFALSE.

Notes

Index keys may be configured to use a soundex-transformed version of the specified field. This function inquires whether an index key uses the field string as-is or a soundex-transformation of the string.

See Also

[QmpIdxKeyAddCompByHnd](#), [QmpIdxKeyAddCompByName](#).

Example

```
QBOOL bUsesSoundex;  
  
bUsesSoundex = QmpIdxKeyUsesSoundex (hIdxKey, pResult);
```


Function Class: QmpIniFil*

FUNCTIONS FOR MANIPULATING THE .INI FILE.

Centrus Merge/Purge applications can use a .ini file in which program variables are stored. For example, Centrus Merge/Purge applications require a license file and password. These variables may be stored in a .ini file. The use of .ini files is purely a convenience, and is not required.

Quick Reference

Function	Description	Page
QmpIniFilCreate	Creates an INI File object.	552
QmpIniFilDestroy	Destroys an INI File object.	553
QmpIniFilReadItemValStrVB	Reads a string from the INI file (Visual Basic version).	554
QmpIniFilReadULong	Reads an unsigned long from the INI file (Visual Basic version).	556

QmpIniFilCreate

CREATES AN INI FILE OBJECT.

Syntax

```
MpHnd QmpIniFilCreate( const char* in_szFileName, QRESULT*  
    out_pResult );
```

in_szFileName

Fully qualified name of the INI file. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to an INI file if successful, or `NULL` if there is an error.

Notes

This function creates an INI file in which program variables can be stored. For example, Centrus Merge/Purge applications require a license file and password. The demo programs shown in this manual get these variables from an INI file. However, if you want to write a program which obtains the license file and password some other way, you can do so. The use of INI files is purely a convenience, and is not required.

A Centrus Merge/Purge INI file might typically take the following form:

```
[SETUP]  
LicenseFile=C:\mergepurge\cmp.lic  
Password=12345678
```

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

`QmpIniFilDestroy`, `QmpIniFilReadULong`,
`QmpIniFilReadItemValStrVB`.

Example

```
hIniFil = QmpIniFilCreate (  
    "s:\\cmp\\win32\\release\\bin\\cmp.ini", pResult );
```

QmpIniFilDestroy

DESTROYS ANINI FILE OBJECT.

Syntax

```
void QmpIniFilDestroy( MpHnd in_hIniFil, QRESULT* out_pResult  
);  
  
in_szFileName  
Handle toINI file. Input.  
  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

Destroys anINI File object.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

Example

```
/* Destruction of anINI file object */  
QmpIniFilDestroy ( hIniFil, &qres );
```

QmpIniFilReadItemValStrVB

READS A STRING FROM THE INI FILE (VISUAL BASIC VERSION).

Syntax

```
void QmpIniFilReadItemValStrVB( MpHnd in_hIniFil, char*
    io_szBuffer, long in_lSize, const char* in_szSection, const
    char* in_szItem, const char* in_szDefaultValue, QRESULT*
    out_pResult );

in_szFileName
    Handle to INI file. Input.

io_szBuffer
    Buffer to put string into. Input, Output.

in_lSize
    Size of buffer. Input.

in_szSection
    Section of INI file in which item is located. Input.

in_szItem
    Item to read. Input.

in_szDefaultValue
    Default value to assign if item is missing. Input.

out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Centrus Merge/Purge applications can use an INI file in which program variables are stored. For example, Centrus Merge/Purge applications require a license file and password. These variables may be stored in an INI file. The use of INI files is purely a convenience, and is not required.

A Centrus Merge/Purge INI file might typically take the following form:

```
[SETUP]
LicenseFile=C:\mergepurge\cmp.lic
Password=12345678
```

Example

```
/* This example shows how to use a string from an */
/* INI file during the license file setup */

char pszLicPathAndFile[QMS_MAX_PATH];
unsigned long ulPassword;

hIniFil = QmpIniFilCreate(
    "s:\\cmp\\win32\\debug\\bin\\cmp.ini", &qres );
```

```
QmpIniFilReadItemValStrVB( hIniFil, pszLicPathAndFile,
    sizeof( pszLicPathAndFile ), "[SETUP]", "LicenseFile", "",
    &qres );
ulPassword = QmpIniFilReadULong( hIniFil, "[SETUP]",
    "Password", 0, &qres );
QmpGlbSetLicense( ulPassword, pszLicPathAndFile, &qres );
```

QmpIniFilReadULong

READS AN UNSIGNED LONG FROM THE INI FILE (VISUAL BASIC VERSION).

Syntax

```
unsigned long QmpIniFilReadULong( MpHnd in_hIniFil, const
                                char* in_szSection, const char* in_szItem, unsigned long
                                in_uDefaultValue, QRESULT* out_pResult );

in_szFileName
    Handle to INI file. Input.

in_szSection
    Section of INI file in which item is located. Input.

in_szItem
    Item to read. Input.

in_uDefaultValue
    Default value to assign if item is missing. Input.

out_pResult
    Result code. Output.
```

Return Value

Returns the unsigned long value of an INI file variable if successful, or 0 if there is an error.

Notes

Centrus Merge/Purge applications can use an INI file in which program variables are stored. For example, Centrus Merge/Purge applications require a license file and password. These variables may be stored in an INI file. The use of INI files is purely a convenience, and is not required.

A Centrus Merge/Purge INI file might typically take the following form:

```
[ SETUP ]
LicenseFile=C:\mergepurge\cmp.lic
Password=12345678
```

Example

```
/* This example shows how to use a unsigned long */
/* from an INI file during the license file setup */

char pszLicPathAndFile[QMS_MAX_PATH];
unsigned long ulPassword;

hIniFil = QmpIniFilCreate(
    "s:\\cmp\\win32\\debug\\bin\\cmp.ini", &qres );
QmpIniFilReadItemValStrVB( hIniFil, pszLicPathAndFile, sizeof(
    pszLicPathAndFile ), "[SETUP]", "LicenseFile", "", &qres );
ulPassword = QmpIniFilReadULong( hIniFil, "[SETUP]",
    "Password", 0, &qres );
QmpGlbSetLicense( ulPassword, pszLicPathAndFile, &qres );
```

Function Class: QmpJobRpt*

JOB SUMMARY REPORT FUNCTIONS.

Quick Reference

Function	Description	Page
QmpJobRptAddPhase	Add a phase to the job summary report.	558
QmpJobRptAddPreProcDataSrc	Specifies the preprocessed data source to be used by the job summary report phase.*	559
QmpJobRptCreate	Creates a job summary report.	560
QmpJobRptFindPhase	Find a phase in the job summary report.	561
QmpJobRptGetPhaseCnt	Get number of phases in the job summary report.	562
QmpJobRptRemPhase	Remove a phase from the job summary report.	563
QmpJobRptSetRecProto	Provides a prototype record to the job summary report.	564
QmpJobRptUseDataLstSvc	Specifies the data list service object to be used by the job report phase.	566

QmpJobRptAddPhase

ADD A PHASE TO THE JOB SUMMARY REPORT.

Syntax

```
void QmpJobRptAddPhase (MpHnd in_hJobRpt, MpHnd in_hPhase,
    QRESULT* out_pResult );
in_hJobRpt
    Handle to job summary report. Input.
in_hPhase
    Handle to the phase to add. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The job summary report needs access to each of the phases that it will report about. This is accomplished by attaching each phase of the job to the job summary report with **QmpJobRptAddPhase**. Call **QmpJobRptAddPhase** separately for each phase to be added. The order that the phases are added is the order the phase information appears in the report.

It is not necessary to add the job summary report phase with **QmpJobRptAddPhase**. The job summary report will always print summary statistics about its own phase.

Obviously, all phases that are attached to the job summary report phase must remain intact until the report is completed.

Example

```
/* Add all phases to this report. */
QmpJobRptAddPhase( hSummaryReport, hDataInp, &qres );
QmpJobRptAddPhase( hSummaryReport, hIndexGenerator, &qres );
QmpJobRptAddPhase( hSummaryReport, hRecordMatcher, &qres );
QmpJobRptAddPhase( hSummaryReport, hMatchCorrelator, &qres );
QmpJobRptAddPhase( hSummaryReport, hDupeGroups, &qres );
QmpJobRptAddPhase( hSummaryReport, hCons, &qres );
QmpJobRptAddPhase( hSummaryReport, hTableGen, &qres );
QmpJobRptAddPhase( hSummaryReport, hUniquesReport, &qres );
QmpJobRptAddPhase( hSummaryReport, hDuplicatesReport, &qres );
QmpJobRptAddPhase( hSummaryReport, hListByListReport, &qres );
```

QmpJobRptAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE JOB SUMMARY REPORT PHASE.*

Syntax

```
void QmpJobRptAddPreProcDataSrc (MpHnd in_hJobRpt, MpHnd  
                                in_hDataSrc, QRESULT* out_pResult );  
  
in_hJobRpt  
Handle to job summary report. Input.  
  
in_hDataSrc  
Handle to the preprocessed data source. Input.  
  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the job summary report phase. It is necessary to add a preprocessed data source in order for the preprocessed data source records to be added to the report.

Example

See example on [page 564](#).

QmpJobRptCreate

CREATES A JOB SUMMARY REPORT.

Syntax

```
MpHnd QmpJobRptCreate ( QRESULT* out_pResult );  
out_pResult  
Result code. Output.
```

Return Value

Returns a handle to the job summary report if successful, or `NULL` if there is an error.

Notes

If the creation function fails, an error code is returned in the `out_pResult` parameter.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
/* Creates a job summary report object and checks for */  
/* success */  
MpHnd hJobRpt = QmpJobRptCreate ( hJob, &qres );  
if (QmpUtilFailed( qres ))  
    ...error handling
```

QmpJobRptFindPhase

FIND A PHASE IN THE JOB SUMMARY REPORT.

Syntax

```
MpHnd QmpJobRptFindPhase ( MpHnd in_hJobRpt, MpHnd in_hPhase,  
    QRESULT* out_pResult );
```

in_hJobRpt

Handle to job summary report. *Input*.

in_hPhase

Handle to the phase to find. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns a handle to the phase found in the job summary report object if successful, or NULL if there is an error.

Notes

This function finds a phase in the job summary report.

Example

```
hFoundPhase = QmpJobRptFindPhase ( hJobRpt, hPhase, pResult );
```

QmpJobRptGetPhaseCnt

GET NUMBER OF PHASES IN THE JOB SUMMARY REPORT.

Syntax

```
long QmpJobRptGetPhaseCnt (MpHnd in_hJobRpt, QRESULT*  
    out_pResult );
```

in_hJobRpt
Handle to job summary report. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the phase count of the job summary report if successful, or -1 if there is an error.

Notes

`QmpJobRptGetPhaseCnt` returns the number of phases added to the job summary report object using `QmpJobRptAddPhase`.

Example

```
lPhaseCount = QmpJobRptGetPhaseCnt (hJobRpt, pResult );
```

QmpJobRptRemPhase

REMOVE A PHASE FROM THE JOB SUMMARY REPORT.

Syntax

```
MpHnd QmpJobRptRemPhase (MpHnd in_hJobRpt, MpHnd in_hPhase,
                           QRESULT* out_pResult );
```

in_hJobRpt

Handle to job summary report. *Input*.

in_hPhase

Handle to the phase to remove. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to phase removed from the job summary report if successful, or NULL if there is an error.

Notes

The job summary report needs access to each of the phases that it will report about. This is accomplished by attaching each phase of the job to the job summary report with **QmpJobRptAddPhase**. Call **QmpJobRptRemPhase** separately for each phase to be removed.

Example

```
hRemovedPhase = QmpJobRptRemPhase ( hSummaryReport, hDataInp,
                                      &qres );
```

QmpJobRptSetRecProto

PROVIDES A PROTOTYPE RECORD TO THE JOB SUMMARY REPORT.

Syntax

```
void QmpJobRptSetRecProto ( MpHnd in_hJobRpt, MpHnd in_hRec,
                           QRESULT* out_pResult );
```

in_hJobRpt
Handle to job summary report object. *Input*.

in_hRec
Handle to the prototype record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Near the top of the job summary report is a list of the fields used for matching records. These fields are otherwise known as a prototype record. The prototype record used for this report is handed to the report phase using this function.

Example

```
QmpDeclHnd( hSummaryReport );
hSummaryReport = QmpJobRptCreate( &qres );

/* Print a job summary report */
QmpRptSetLinesPerPage( hSummaryReport, -1, &qres );
sprintf( szStringPtr, "Rpt-Summary-%d.log", iThreshHold );
sprintf( pszTempPathBuffer, "%s%s%s", pszOutputPathBuffer, pszDelim,
        szStringPtr );
remove(pszTempPathBuffer);
QmpRptSetFilename( hSummaryReport, pszTempPathBuffer, &qres );
sprintf( szTempPtr, ": Summary Report for a Threshold of %d", iThreshHold );
strcat( szStringPtr, szTempPtr );
QmpRptSetTitle( hSummaryReport, szStringPtr, &qres );
QmpPhaseUseDITR( hSummaryReport, hDITR, &qres );
QmpJobRptUseDataLstSvc( hSummaryReport, hDataListService, &qres );
QmpJobRptSetRecProto( hSummaryReport, hProtoRec, &qres );
if ( bUsePrePro ) {
    QmpJobRptAddPreProcDataSrc( hSummaryReport, hPreProDataSrc, &qres );
}

/* Add all phases to this report. */
QmpJobRptAddPhase( hSummaryReport, hDataInp, &qres );
QmpJobRptAddPhase( hSummaryReport, hIndexGenerator, &qres );
QmpJobRptAddPhase( hSummaryReport, hRecordMatcher, &qres );
QmpJobRptAddPhase( hSummaryReport, hDupeGroups, &qres );
QmpJobRptAddPhase( hSummaryReport, hTableGen, &qres );
QmpJobRptAddPhase( hSummaryReport, hDuplicatesReport, &qres );
QmpJobRptAddPhase( hSummaryReport, hNearMissReport, &qres );
QmpJobRptAddPhase( hSummaryReport, hUniquesReport, &qres );
QmpJobRptAddPhase( hSummaryReport, hListByListReport, &qres );
```

```
/* run the summary report */
printf( "Starting Summary Report phase:\n" );
QmpPhaseStart( hSummaryReport, &qres );
QmpPhaseDestroy( hSummaryReport, &qres );
```

QmpJobRptUseDataLstSvc

SPECIFIES THE DATA LIST SERVICE OBJECT TO BE USED BY THE JOB REPORT PHASE.

Syntax

```
void QmpJobRptUseDataLstSvc ( MpHnd in_hJobRpt, MpHnd  
                             in_hDataLstSvc, QRESULT* out_pResult );
```

in_hJobRpt
Handle to job summary report object. *Input*.

in_hDataLstSvc
Handle to data list service object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The data list service is needed to determine the number of data lists to be reported in the data input summary section of the report.

Example

See example on [page 564](#).

Function Class: QmpListByListRpt*

LIST-BY-LIST REPORT FUNCTIONS.

Quick Reference

Function	Description	Page
QmpListByListRptAddPreProcDataSrc	Specifies the preprocessed data source to be used by the list-by-list report phase.*	568
QmpListByListRptCreate	Creates a list-by-list report.	569
QmpListByListRptGetNthRecInterval	Gets List-by-List report Nth record event interval.	570
QmpListByListRptRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	571
QmpListByListRptSetNthRecInterval	Sets list-by-list report Nth record event interval.	572
QmpListByListRptUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	573
QmpListByListRptUseDataLstSvc	Specifies the data list service object to be used.	574
QmpListByListRptUseDupGrp	Attaches a dupe groups object to the list-by-list report.	575
QmpListByListRptUseMatRes	Specifies the match result object to use in the report.	576

QmpListByListRptAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE LIST-BY-LIST REPORT PHASE.*

Syntax

```
void QmpListByListRptAddPreProcDataSrc (MpHnd
    in_hListByListRpt, MpHnd in_hDataSrc, QRESULT* out_pResult
);

in_hListByListRpt
    Handle to list-by-list report. Input.

in_hDataSrc
    Handle to the preprocessed data source. Input.

out_pResult
    Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the list-by-list report phase. It is necessary to add a preprocessed data source in order for the preprocessed data source records to be added to the report.

Example

```
QRESULT qres;                                /* return code */
QmpDeclHnd( hListByListReport );
QmpDeclHnd( hPreProDataSrc ); /* PreProcessed Data source */

hListByListReport = QmpListByListRptCreate( &qres );
hPreProDataSrc = QmpDataSrcCreate( QMS_DATSRC_TYPE_PREPRO,
    "prepro.{dbf,cdx}", 2, &qres );

QmpListByListRptAddPreProcDataSrc( hListByListReport,
    hPreProDataSrc, &qres );
```

QmpListByListRptCreate

CREATES A LIST-BY-LIST REPORT.

Syntax

```
MpHnd QmpListByListRptCreate(QRESULT* out_pResult);  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to ListByList report if successful, or `NULL` if there is an error.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
hListByListReport = QmpListByListRptCreate( pResult );
```

QmpListByListRptGetNthRecInterval

GETS LIST-BY-LIST REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpListByListRptGetNthRecInterval ( MpHnd in_hListRpt,
                                         QRESULT* out_pResult );
in_hListRpt
Handle to list-by-list report. Input.
out_pResult
Result code. Output.
```

Return Value

Returns the list-by-list report Nth record event interval if successful, or -1 if there is an error.

Notes

This function gets the list-by-list report phase Nth record event interval.

The list-by-list report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an occasional notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100 records, then each time the list-by-list report phase processes 100 records, it will send out a notification to all registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

`QmpListByListRptRegEveryNthRecFunc`,
`QmpListByListRptSetNthRecInterval`,
`QmpListByListRptUnregEveryNthRecFunc`.

Example

```
lDupsRepNthRecInterval = QmpListByListRptGetNthRecInterval
(hListRpt, pResult );
```

QmpListByListRptRegEveryNthRecFunc

REGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpListByListRptRegEveryNthRecFunc ( MpHnd in_hListRpt,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hListRpt

Handle to list-by-list report. *Input*.

in_Func

Pointer to event handler. *Input*.

out_pResult

Result code.

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The list-by-list report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
if ( bEveryNth ) {
    QmpListByListRptRegEveryNthRecFunc( hList-by-ListReport,
                                         pEveryNthRecordClient, &qres );
    QmpListByListRptSetNthRecInterval( hList-by-ListReport, 200,
                                       &qres );
}
```

QmpListByListRptSetNthRecInterval

SETS LIST-BY-LIST REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpListByListRptSetNthRecInterval ( MpHnd in_hListRpt,
                                         long in_lInterval, QRESULT* out_pResult );
in_hListRpt
Handle to list-by-list report. Input.
in_lInterval
List-by-list report Nth record event interval. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

This function sets the list-by-list report phase Nth record event interval.

The list-by-list report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an occasional notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100 record, then each time the list-by-list report phase processes 100 records, it will send out a notification to all registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

QmpListByListRptGetNthRecInterval,
QmpListByListRptRegEveryNthRecFunc,
QmpListByListRptUnregEveryNthRecFunc.

Example

```
if ( bEveryNth ) {
    QmpListByListRptRegEveryNthRecFunc( hList-by-ListReport,
                                         pEveryNthRecordClient, &qres );
    QmpListByListRptSetNthRecInterval( hList-by-ListReport, 200,
                                       &qres );
}
```

QmpListByListRptUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpListByListRptUnregEveryNthRecFunc ( MpHnd
    in_hListRpt, QMS_EVERY_NTHREC_FUNC in_Func, QRESULT*
    out_pResult );
```

in_hListRpt
Handle to list-by-list report. *Input.*

in_Func
Pointer to event handler. *Input.*

out_pResult
Result code. *Output.*

Return Value

None.

Notes

This function unregisters the specified event handler from the list-by-list report, so that the event handler will no longer be notified of every Nth record processed.

The list-by-list report can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
QmpListByListRptUnregEveryNthRecFunc (hListRpt,
    pListRptEventHandler, pResult );
```

QmpListByListRptUseDataLstSvc

SPECIFIES THE DATA LIST SERVICE OBJECT TO BE USED.

Syntax

```
void QmpListByListRptUseDataLstSvc ( MpHnd in_hListByListRpt,
                                    MpHnd in_hDataLstSvc, QRESULT* out_pResult );
in_hListByListRpt
    Handle to ListByList report object. Input.
in_hDataLstSvc
    Handle to data list service object. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The data list service is needed by the list-by-list report to determine the number of data lists to compare.

Example

```
QmpPhaseSetRecProto( hListByListReport, hSourceRecord, pResult
                     );
QmpListByListRptUseDupGrp( hListByListReport, hDupeGroups,
                           pResult );
QmpListByListRptUseDataLstSvc( hListByListReport,
                                 hDataListService, pResult );
```

QmpListByListRptUseDupGrp

ATTACHES A DUPE GROUPS OBJECT TO THE LIST-BY-LIST REPORT.

Syntax

```
void QmpListByListRptUseDupGrp( MpHnd in_hListByListRpt,
                                MpHnd in_hDupGrps, QRESULT* out_pResult );
in_hListByListRpt
    Handle to list-by-list report object. Input.
in_hDupGrps
    Handle to a dupe groups object. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Calling **QmpListByListRptUseDupGrp** enables the list-by-list report to include the following information:

- total record count
- master duplicate count
- subordinate duplicate count
- unique record count

Here is an example of how this information is presented in the report:

```
Summary: List By List Report
Number of Records: 42500
Number of Masters: 13001
Number of Subordinates: 13042
Number of Uniques: 16457
```

QmpListByListRptUseDupGrp gives the list-by-list report master dupe and subordinate dupe count information. With this information in hand, the list-by-list report can then calculate and report the unique record count.

Including this count information in the list-by-list report by calling **QmpListByListRptUseDupGrp** is optional. Both functions should be called together, not separately.

Example

```
QmpListByListRptUseDITR(hListByListRpt, hTbl, pResult );
QmpListByListRptUseDupGrp(hListByListRpt, hDupGrps, pResult
);
```

QmpListByListRptUseMatRes

SPECIFIES THE MATCH RESULT OBJECT TO USE IN THE REPORT.

Syntax

```
void QmpListByListRptUseMatRes( MpHnd in_hListByListRpt,
                               MpHnd in_hMatRes, QRESULT* out_pResult );
in_hListByListRpt
Handle to list-by-list report object. Input.
in_hMatRes
Handle to match result object. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

The match result object contains the record match information generated during the record matcher phase.

Example

```
QmpRptSetTitle( hListByListReport, szStringPtr, &qres );
QmpRptSetSubtitle( hListByListReport, "Generated for Make-A-
Wish Foundation", &qres );
QmpListByListRptUseDITR( hListByListReport, hDITR, &qres );
QmpListByListRptUseMatRes( hListByListReport, hMatRes, &qres
);
printf( "Starting List-by-List Report phase:\n" );
QmpPhaseStart( hListByListReport, &qres );
```

Function Class: QmpLog*

FUNCTIONS USED TO MANIPULATE ERROR REPORTING AND EXECUTION TRACING.

Quick Reference

Function	Description	Page
QmpLogAddErrFilename	Adds a file name destination for error log information to the log object.	578
QmpLogAddErrFilepointer	Adds a file pointer for error log information to the log object (can use stdio or stderr, also).	579
QmpLogAddTraceFilename	Adds a file name destination for trace log information to the log object.	580
QmpLogAddTraceFilepointer	Adds a file pointer for trace log information to the log object (can use stdio or stderr, also).	581
QmpLogClear	Clears a log object of all output destinations.	582
QmpLogGetAppErrorHandler	Get the application-supplied error handler.	583
QmpLogGetErrFatalLvl	Gets fatality level in the error facility.	584
QmpLogGetErrRptLvl	Gets report level in the error facility	585
QmpLogGetIndentChar	Gets indentation character.	586
QmpLogGetTraceActive	Gets active status of trace facility.	587
QmpLogGetTraceIndentIncr	Gets indent increment of trace facility.	588
QmpLogGetTraceRptLvl	Gets report level of trace facility.	589
QmpLogLookUpErrMsg	Looks up the string associated with an error number.	590
QmpLogLookUpErrMsgVB	Looks up the string associated with an error number (Visual Basic version).	591
QmpLogRemErrFilename	Removes an error file name from the log object.	592
QmpLogRemTraceFilename	Removes a trace file name from the log object.	593
QmpLogSetAppErrorHandler	Set the application-supplied error handler.	594
QmpLogSetErrFatalLvl	Sets fatality level in the error facility.	595
QmpLogSetErrRptLvl	Sets report level in the error facility.	596
QmpLogSetIndentChar	Sets indentation character.	597
QmpLogSetTraceActive	Sets active status of trace facility.	598
QmpLogSetTraceIndentIncr	Sets Indent increment of trace facility.	599
QmpLogSetTraceRptLvl	Sets report level of trace facility.	600

QmpLogAddErrFilename

ADDS A FILE NAME DESTINATION FOR ERROR LOG INFORMATION TO THE LOG OBJECT.

Syntax

```
void QmpLogAddErrFilename( MpHnd in_hLog, const char*
    in_szFilename, QBOOL in_bOverwrite, QRESULT* out_pResult );
in_hLog
    Handle to system-wide log object. Input.
in_szFilename
    File name to add. Input.
in_bOverwrite
    Whether to overwrite existing file. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpLogAddErrFilename enables the application to specify a file destination for error log information.

Example

```
/* register errors and trace with stdout */
hLog = QmpGlbGetLog( &qres );
QmpLogSetTraceActive( hLog, QTRUE, &qres );
QmpLogAddErrFilepointer( hLog, stdout, &qres );
QmpLogSetErrRptLvl( hLog, QMS_ERROR_WARNING, &qres );
QmpLogAddTraceFilepointer( hLog, stdout, &qres );
sprintf( pszErrLogPathAndFile, "%s\\PhaseErr.log",
    pszCurrentWorkingFolder );
QmpLogAddErrFilename( hLog, pszErrLogPathAndFile, QTRUE, &qres
);
QmpLogSetAppErrorHandler( hLog, AppErrorHandler, &qres );
```

QmpLogAddErrFilepointer

ADDS A FILE POINTER FOR ERROR LOG INFORMATION TO THE LOG OBJECT (CAN USE STDIO OR STDERR, ALSO).

Syntax

```
void QmpLogAddErrFilepointer ( MpHnd in_hLog, FILE* in_Fp,
                           QRESULT* out_pResult );
in_hLog
Handle to system wide log object. Input.
in_Fp
File pointer. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

QmpLogAddErrFilepointer enables the application to specify a pointer to a file destination for error log information.

Example

```
/* register errors and trace with stdout */
hLog = QmpGlbGetLog( &qres );
QmpLogSetTraceActive( hLog, QTRUE, &qres );
QmpLogAddErrFilepointer( hLog, stdout, &qres );
QmpLogSetErrRptLvl( hLog, QMS_ERROR_WARNING, &qres );
QmpLogAddTraceFilepointer( hLog, stdout, &qres );
sprintf( pszErrLogPathAndFile, "%s\\PhaseErr.log",
         pszCurrentWorkingFolder );
QmpLogAddErrFilename( hLog, pszErrLogPathAndFile, QTRUE, &qres );
QmpLogSetAppErrorHandler( hLog, AppErrorHandler, &qres );
```

QmpLogAddTraceFilename

ADDS A FILE NAME DESTINATION FOR TRACE LOG INFORMATION TO THE LOG OBJECT.

Syntax

```
void QmpLogAddTraceFilename ( MpHnd in_hLog, const char*
                             in_szFilename, QBOOL in_bOverwrite, QRESULT* out_pResult );
in_hLog
    Handle to system-wide log object. Input.
in_szFilename
    File name to add. Input.
in_bOverwrite
    Whether to overwrite existing file. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpLogAddTraceFilename enables the application to specify a file destination for trace log information.

Example

```
void
SetupJobTraceLogging( void )
{
    /* remove old dump log file so we don't append to it. */
    remove( "Results.log" );

    /* Add a trace file and ostream */
    QmpLogAddTraceFilename( hLog, "Trace.log", QTRUE, pResult );
    QmpLogAddTraceFilepointer( hLog, stdout, pResult );

    /* Set to false and 0 for no tracing, or true and > 0 for */
    /* some tracing. */
    QmpLogSetTraceActive( hLog, QFALSE, pResult );
    QmpLogSetTraceRptLvl( hLog, 0, pResult );

}
```

QmpLogAddTraceFilepointer

ADDS A FILE POINTER FOR TRACE LOG INFORMATION TO THE LOG OBJECT (CAN USE STDIO OR STDERR, ALSO).

Syntax

```
void QmpLogAddTraceFilepointer ( MpHnd in_hLog, FILE* in_Fp,
    QRESULT* out_pResult );
in_hLog
    Handle to system wide log object. Input.
in_Fp
    File pointer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpLogAddTraceFilepointer enables the application to specify a pointer to a file destination for trace log information.

Example

```
void
SetupJobTraceLogging( void )
{
    /* remove old dump log file so we don't append to it. */
    remove( "Results.log" );

    /* Add a trace file and ostream */
    QmpLogAddTraceFilename( hLog, "Trace.log", QTRUE, pResult );
    QmpLogAddTraceFilepointer( hLog, stdout, pResult );

    /* Set to false and 0 for no tracing, or true and > 0 for */
    /* some tracing. */
    QmpLogSetTraceActive( hLog, QFALSE, pResult );
    QmpLogSetTraceRptLvl( hLog, 0, pResult );

}
```

QmpLogClear

CLEAR A LOG OBJECT OF ALL OUTPUT DESTINATIONS.

Syntax

```
void QmpLogClear(MpHnd in_hLog, QRESULT* out_pResult );
```

in_hLog
Handle to log object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function clears the log object. This means setting it back to a default and empty state, including removing all output destination references. The reset variables and the values they are set to include:

- Error fatality level set to QMS_ERROR_FATAL
- Error report level set to QMS_ERROR_WARNING
- Trace indent increment set to 3
- Trace report level set to 0
- Trace indent amount set to 0

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpLogClear(hLog, pResult );
```

QmpLogGetAppErrorHandler

GET THE APPLICATION-SUPPLIED ERROR HANDLER.

Syntax

```
QMS_ERR_HANDLER QmpLogGetAppErrorHandler ( MpHnd in_hLog,  
                                         QRESULT* out_pResult );  
  
in_hLog  
    Handle to system wide log object. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns pointer to error handler function if successful, or `NULL` if there is an error.

Notes

`QmpLogGetAppErrorHandler` returns a pointer to the error handler function that was registered with `QmpLogSetAppErrorHandler`.

See Also

`QmpLogSetAppErrorHandler`

Example

```
pAppErrorHandler = QmpLogGetAppErrorHandler (hLog, pResult );
```

QmpLogGetErrFatalLvl

GETS FATALITY LEVEL IN THE ERROR FACILITY.

Syntax

```
QMS_ERROR_LEVEL QmpLogGetErrFatalLvl ( MpHnd in_hLog,  
    QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

out_pResult

Result code. *Output*.

Return Value

If successful, valid return values are:

```
QMS_ERROR_INFORMATIVE  
QMS_ERROR_WARNING  
QMS_ERROR_SEVERE  
QMS_ERROR_FATAL  
QMS_ERROR_NONE
```

QMS_ERROR_NONE is returned if there is an error.

Notes

The fatality level is the error level at which application execution is halted.

Example

```
ErrorLevel = QmpLogGetErrFatalLvl ( hLog, pResult );
```

QmpLogGetErrRptLvl

GETS REPORT LEVEL IN THE ERROR FACILITY

Syntax

```
QMS_ERROR_LEVEL QmpLogGetErrRptLvl ( MpHnd in_hLog, QRESULT*  
    out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

out_pResult

Result code. *Output*.

Return Value

If successful, valid return values are:

```
QMS_ERROR_INFORMATIVE  
QMS_ERROR_WARNING  
QMS_ERROR_SEVERE  
QMS_ERROR_FATAL  
QMS_ERROR_NONE
```

QMS_ERROR_NONE is returned if there is an error.

Notes

The report level determines what kinds of errors are written to the logging facility. For example, if the report level was QMS_ERROR_SEVERE, only errors of class QMS_ERROR_SEVERE and QMS_ERROR_FATAL would be written to the log file.

Example

```
ErrorReportLevel = QmpLogGetErrRptLvl ( hLog, pResult );
```

QmpLogGetIndentChar

GETS INDENTATION CHARACTER.

Syntax

```
char QmpLogGetIndentChar ( MpHnd in_hLog, QRESULT* out_pResult
) ;
```

in_hLog

Handle to system wide log object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns a char if successful, or 0 if there is an error.

Notes

QmpLogGetIndentChar returns the indentation character used in trace logging files.

Example

```
char chIndentChar;
QRESULT* pResult;
QmpDeclHnd(hLog);

hLog = QmpGlbGetLog( pResult );
chIndentChar = QmpLogGetIndentChar (hLog, pResult );
```

QmpLogGetTraceActive

GETS ACTIVE STATUS OF TRACE FACILITY.

Syntax

```
QBOOL QmpLogGetTraceActive ( MpHnd in_hLog, QRESULT*  
    out_pResult );
```

in_hLog
Handle to system wide log object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the trace facility is active, otherwise QFALSE.

Notes

This function gets the active status of the trace facility.

Example

```
QBOOL bIsTraceFacilityActive = QFALSE;  
QRESULT* pResult;  
QmpDeclHnd(hLog);  
  
hLog = QmpGlbGetLog( pResult );  
bIsTraceFacilityActive = QmpLogGetTraceActive (hLog, pResult  
);
```

QmpLogGetTraceIndentIncr

GETS INDENT INCREMENT OF TRACE FACILITY.

Syntax

```
int QmpLogGetTraceIndentIncr ( MpHnd in_hLog, QRESULT*  
    out_pResult );
```

in_hLog
Handle to system wide log object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the indent increment of the trace facility if successful, or -1 if there is an error.

Notes

The indent increment determines how far text is indented in a trace log.

Example

```
int iTraceIndentIncr = 0;  
QRESULT* pResult;  
QmpDeclHnd(hLog);  
  
hLog = QmpGlbGetLog( pResult );  
iTraceIndentIncr = QmpLogGetTraceIndentIncr (hLog, pResult );
```

QmpLogGetTraceRptLvl

GETS REPORT LEVEL OF TRACE FACILITY.

Syntax

```
int QmpLogGetTraceRptLvl ( MpHnd in_hLog, QRESULT* out_pResult
    );
```

in_hLog

Handle to system wide log object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns trace report level if successful, or -1 if there is an error.

Example

```
int iTraceRptLvl = 0;
QRESULT* pResult;
QmpDeclHnd(hLog);

hLog = QmpGlbGetLog( pResult );
iTraceRptLvl = QmpLogGetTraceRptLvl (hLog, pResult );
```

QmpLogLookUpErrMsg

LOOKS UP THE STRING ASSOCIATED WITH AN ERROR NUMBER.

Syntax

```
const char* QmpLogLookUpErrMsg( MpHnd in_hLog, QRESULT  
                               in_ErrNum, QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

in_ErrNum

Error number. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns pointer to a message if successful, or `NULL` if there is an error.

Notes

Use this function to look up the explanatory message associated with an error number.

Example

```
fprintf(stderr, "%s", QmpLogLookUpErrMsg(LogObject, Result,  
                                         &RES));
```

QmpLogLookUpErrMsgVB

LOOKS UP THE STRING ASSOCIATED WITH AN ERROR NUMBER (VISUAL BASIC VERSION).

Syntax

```
void QmpLogLookUpErrMsgVB( MpHnd in_hLog, QRESULT in_ErrNum,
                           char* io_szBuffer, long in_lSize, QRESULT* out_pResult );
in_hLog
Handle to the system-wide log object. Input.
in_ErrNum
Error number. Input.
io_szBuffer
Buffer to put message into. Input, output.
in_lSize
Size of buffer. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Example

```
MpHnd hLog;
QRESULT ErrNum;
char szbuffer[1024];
QRESULT presult;

QmpLogLookUpErrMsgVB( hLog, ErrNum, szbuffer,
                      sizeof(szbuffer), &presult );
```

QmpLogRemErrFilename

REMOVES AN ERROR FILE NAME FROM THE LOG OBJECT.

Syntax

```
void QmpLogRemErrFilename ( MpHnd in_hLog, const char*  
                           in_szFilename, QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

in_szFilename

File name to remove. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpLogRemErrFilename allows the application to remove the error file name from the log object.

Example

```
QRESULT* pResult;  
QmpDeclHnd(hLog);  
  
hLog = QmpGlbGetLog( pResult );  
QmpLogRemErrFilename (hLog, "Qerror.txt", pResult );
```

QmpLogRemTraceFilename

REMOVES A TRACE FILE NAME FROM THE LOG OBJECT.

Syntax

```
void QmpLogRemTraceFilename ( MpHnd in_hLog, const char*
    in_szFilename, QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

in_szFilename

File name to remove. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpLogRemTraceFilename allows the application to remove the trace file name from the log object.

Example

```
QRESULT* pResult;
QmpDeclHnd(hLog);

hLog = QmpGlbGetLog( pResult );
QmpLogRemTraceFilename (hLog, "Qtrace.txt", pResult );
```

QmpLogSetAppErrorHandler

SET THE APPLICATION-SUPPLIED ERROR HANDLER.

Syntax

```
void QmpLogSetAppErrorHandler ( MpHnd in_hLog, QMS_ERR_HANDLER
                               in_AppErrorHandler, QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

in_AppErrorHandler

Pointer to application error handler function. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The error handler being set gets “first crack” at a Centrus Merge/Purge error. The error handler can take action in the following ways:

QMS_ERR_ACTION_HANDLED	Handle the error. Library should do no further error processing on it.
QMS_ERR_ACTION_CONTINUE	Error has not been handled. Continue processing the error.
QMS_ERR_ACTION_ABORT	Treat the error as if it were a fatal error. Stop processing.

See Also

[QmpLogGetAppErrorHandler](#)

Example

```
QmpLogSetAppErrorHandler ( hLog, pAppErrorHandler, pResult );
```

QmpLogSetErrFatalLvl

SETS FATALITY LEVEL IN THE ERROR FACILITY.

Syntax

```
void QmpLogSetErrFatalLvl ( MpHnd in_hLog, QMS_ERROR_LEVEL  
    in_FatalLvl, QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

in_FatalLvl

Error fatality level. *Input*.

Valid enums are:

QMS_ERROR_INFORMATIVE

QMS_ERROR_WARNING

QMS_ERROR_SEVERE

QMS_ERROR_FATAL

QMS_ERROR_NONE

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The fatality level is the error level at which errors halt execution of the application. For example, if the fatality level was set to QMS_ERROR_SEVERE, errors of type QMS_ERROR_SEVERE and QMS_ERROR_FATAL would halt the application.

Example

```
/* Change the settings for error and trace logging */

QmpLogSetErrFatalLvl ( hLog, QMS_ERROR_WARNING, pResult );
QmpLogSetErrRptLvl ( hLog, QMS_ERROR_WARNING, pResult );
```

QmpLogSetErrRptLvl

SETS REPORT LEVEL IN THE ERROR FACILITY.

Syntax

```
void QmpLogSetErrRptLvl ( MpHnd in_hLog, QMS_ERROR_LEVEL  
    in_ReportLvl, QRESULT* out_pResult );
```

in_hLog

Handle to system wide log object. *Input*.

in_ReportLvl

Error report level. *Input*.

Valid enums are:

QMS_ERROR_INFORMATIVE
QMS_ERROR_WARNING
QMS_ERROR_SEVERE
QMS_ERROR_FATAL
QMS_ERROR_NONE

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The report level is the error level at which errors are written to the error log file.

For example, if the report level is set to QMS_ERROR_WARNING, errors of types QMS_ERROR_WARNING, QMS_ERROR_SEVERE, and QMS_ERROR_FATAL will be handed over to the error log facility.

Example

```
/* Change the settings for error and trace logging */  
  
QmpLogSetErrFatalLvl ( hLog, QMS_ERROR_WARNING, pResult );  
QmpLogSetErrRptLvl ( hLog, QMS_ERROR_WARNING, pResult );
```

QmpLogSetIndentChar

SETS INDENTATION CHARACTER.

Syntax

```
void QmpLogSetIndentChar ( MpHnd in_hLog, char in_cIndentChar,
                          QRESULT* out_pResult );
in_hLog
    Handle to system wide log object. Input.
in_cIndentChar
    Indentation character. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpLogSetIndentChar sets the indentation character used in error and trace logging files.

Example

```
char chIndentChar = ' - ';
QRESULT* pResult;
QmpDeclHnd(hLog);

hLog = QmpGlbGetLog( pResult );
QmpLogSetIndentChar (hLog, chIndentChar, pResult );
```

QmpLogSetTraceActive

SETS ACTIVE STATUS OF TRACE FACILITY.

Syntax

```
void QmpLogSetTraceActive ( MpHnd in_hLog, QBOOL in_bActive,  
    QRESULT* out_pResult );
```

in_hLog
Handle to system wide log object. *Input*.

in_bActive
Trace active status. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

If *in_bActive* = QTRUE, the trace facility is activated. If *in_bActive* = QFALSE, the facility is deactivated.

Example

```
QRESULT* pResult;  
QmpDeclHnd(hLog);  
  
hLog = QmpGlbGetLog( pResult );  
QmpLogSetTraceActive (hLog, QTRUE, pResult );
```

QmpLogSetTraceIndentIncr

SETS INDENT INCREMENT OF TRACE FACILITY.

Syntax

```
void QmpLogSetTraceIndentIncr ( MpHnd in_hLog, int
    in_iIndentIncr, QRESULT* out_pResult );
```

in_hLog Handle to system wide log object. *Input*.

in_iIndentIncr Trace indent increment. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The indent increment determines how far text is indented in a trace log.

Example

```
QRESULT* pResult;
QmpDeclHnd(hLog);

hLog = QmpGlbGetLog( pResult );
QmpLogSetTraceIndentIncr (hLog, 5, pResult );
```

QmpLogSetTraceRptLvl

SETS REPORT LEVEL OF TRACE FACILITY.

Syntax

```
void QmpLogSetTraceRptLvl ( MpHnd in_hLog, int in_iReportLvl,
                            QRESULT* out_pResult );
```

in_hLog
Handle to system wide log object. *Input*.

in_iReportLvl
Trace report level. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Example

```
QRESULT* pResult;
QmpDeclHnd(hLog);

hLog = QmpGlbGetLog( pResult );
QmpLogSetTraceRptLvl (hLog, 3, pResult );
```

Function Class: QmpMatRes*

FUNCTIONS USED TO MANIPULATE THE MATCH RESULT OBJECT.

Quick Reference

Function	Description	Page
QmpMatResCreate	Creates a match result object.	602
QmpMatResDestroy	Destroys a match result object.	603

QmpMatResCreate

CREATES A MATCH RESULT OBJECT.

Syntax

```
MpHnd QmpMatResCreate (MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl

Handle to client-supplied table to use as internal representation. This must be a CodeBase table. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to match result if successful, or `NULL` if there is an error.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

The match result object makes a copy of the table that is given to it. It is this table copy that the match result object works with.

The application shouldn't attempt to open or manipulate the match result table while the record matcher or dupe groups phases are executing. These phases have exclusive access to the table, and any table manipulations the application attempts at these times will fail.

Remember to destroy the table used for the match result at the end of the program.

Example

```
QRESULT qres;           /* return code      */
QmpDeclHnd( hMatResTable ) /* Match Result Table */
QmpDeclHnd( hMatRes );   /* Match Result      */

/* create a match result table to be used by the match */
/* result */
hMatResTable = QmpTblCbCreate( "c:\\temp", "matres.dbf", &qres
);

/* create the match result object */
hMatRes = QmpMatResCreate( hMatResTable, &qres );

/* destroy the table object */
QmpTblDestroy( hMatResTable, &qres );

/* destroy match result */
QmpMatResDestroy( hMatRes, &qres );
```

QmpMatResDestroy

DESTROYS A MATCH RESULT OBJECT.

Syntax

```
void QmpMatResDestroy ( MpHnd in_hMatRes, QRESULT* out_pResult
    );
in_hMatRes
    Handle to match result. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function destroys the specified match result object.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

The application shouldn't attempt to open or manipulate the match result table while the record matcher or dupe groups phases are executing. These phases have exclusive access to the table, and any table manipulations the application attempts at these times will fail.

Example

```
QRESULT qres; /* return code */
QmpDeclHnd( hMatResTable ); /* Match Result Table */
QmpDeclHnd( hMatRes ); /* Match Result */

/* create match result table to be used by the match result */
hMatResTable = QmpTblCbCreate( "c:\\temp", "matres.dbf", &qres
);

/* create the match result object */
hMatRes = QmpMatResCreate( hMatResTable, &qres );

/* destroy the table object */
QmpTblDestroy( hMatResTable, &qres );

/* destroy match result */
QmpMatResDestroy( hMatRes, &qres );
```

`QmpMatResDestroy`

Function Class: QmpMissRpt*

NEAR MISS REPORT FUNCTIONS.

Using the Near Miss Report to Tune the Record Matching Threshold

Finding an acceptable matching threshold is a trial-and-error process. After match criteria have been defined, a record matching threshold value must be chosen. If this threshold is too low, low-quality matches will slip through. If the threshold is too high, too few matches of acceptable quality will be produced. The best way to find out what the threshold value should be is to compare subordinate duplicates with the master duplicate. By looking at enough of these matches, you will be able to decide upon a threshold that divides “good” matches from “bad” ones.

The near miss report simplifies this process by hiding the dupe groups that you already know contain only good matches. This is done by setting the report’s acceptance level. This value is a threshold, similar to the record matching threshold. The near miss report only displays dupe groups that contain at least one record whose duplicate/master match score is below the near miss acceptance level. This allows you to focus on dupe groups with questionable matches.

Using the near miss report in conjunction with the dupe groups threshold to select a matching threshold also saves time. Instead of re-executing the entire job after changing the record matching threshold, you may reset the dupe groups threshold and re-execute only the dupe groups and near miss report phases. This technique requires you to initially set the record matching threshold to a low value (to generate many matches for the match result table). When resetting the dupe groups threshold, you must increase the value, which will restrict which records from the match result table are used to build the dupe groups table. The end result is the same as if you had reset the record matcher threshold and re-executed the entire job.

Note: The near miss report can be better thought of as a “barely made it” report as it shows duplicate groups that were above the match threshold, but below the acceptance level (i.e., had at least one member with a low match score).

To use the near miss report to find a good matching threshold:

1. Set the job’s record matching threshold to a low value. This will cause the record matcher to generate many initial matches.
2. Set the acceptance level of the near miss report to a high value. This will cause the report to initially include many dupe groups. (Remember that only one subordinate needs to match the master duplicate below the acceptance level in order for the whole dupe group to be included in the report.)
3. Run the job and generate a near miss report.

4. Examine the matches in the report and try to find a subordinate/master match threshold that separates acceptable matches from unacceptable ones.
If there are too many duplicate groups with “good” matches that clutter the report, lower the acceptance level and regenerate the report. The regenerated report will contain fewer dupe groups with lower-quality matches between the subordinates and masters.
5. Once you have identified a near miss report acceptance level that produces reports with no “good” matches, set the dupe groups threshold to that value and rerun the dupe groups and near miss report phases.
This will cause only the records in the match result table with a match value equalling or exceeding the dupe groups threshold to be included in the dupe groups table. This method saves time because you only need to re-execute the dupe groups phase and all phases following it again, not the entire job.
6. Continue tweaking the acceptance level and dupe groups threshold values until you arrive at a record matching threshold/dupe groups threshold you are satisfied with.

Note: The default acceptance level value is 90.

Quick Reference

Function	Description	Page
QmpMissRptAddPreProcDataSrc	Specifies the preprocessed data source to be used by the near miss report phase.*	608
QmpMissRptCreate	Creates a near miss report.	609
QmpMissRptDestroy	Destroys a near miss report.	611
QmpMissRptGetAcceptLvl	Retrieves the acceptance level of the near miss report.	612
QmpMissRptGetNthRecInterval	Gets the near miss report Nth record event interval.	613
QmpMissRptRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	614
QmpMissRptSetAcceptLvl	Sets the acceptance level of the near miss report.	615
QmpMissRptSetDupeGrpId	Specifies whether to print dupe group column IDs in the report.	616
QmpMissRptSetDupeGrpScore	Specifies whether to print dupe group column scores in the report.	617
QmpMissRptSetListId	Specifies whether to print the data list ID column in the report.	618
QmpMissRptSetNthRecInterval	Sets near miss report Nth record event interval.	619

Function	Description	Page
QmpMissRptSetPrintKeys	Specifies whether to print the record primary key column in the report.	620
QmpMissRptSetPrintSrc	Specifies whether to print a record source ID column in the report.	621
QmpMissRptSetShowSubords	Specifies whether to print subordinate duplicates in the near miss report.	622
QmpMissRptUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	623
QmpMissRptUseCons	Gives the data consolidation phase to a near miss report.	624
QmpMissRptUseDupGrp	Specifies the dupe groups object to use for the near miss report.	625

QmpMissRptAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE NEAR MISS REPORT PHASE.*

Syntax

```
void QmpMissRptAddPreProcDataSrc ( MpHnd in_hMissRpt, MpHnd  
    in_hDataSrc, QRESULT* out_pResult );
```

in_hMissRpt
Handle to near miss report object. *Input*.

in_hDataSrc
Handle to preprocessed data source to be used. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the near miss report phase.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

Example

See example on [page 609](#).

QmpMissRptCreate

CREATES A NEAR MISS REPORT.

Syntax

```
MpHnd QmpMissRptCreate( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns handle to near miss report if successful, or `NULL` if there is an error.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

`QmpMissRptDestroy`

Example

```
QmpDeclHnd( hNearMissReport );
hNearMissReport = QmpMissRptCreate( &qres );
QmpRptSetLinesPerPage( hNearMissReport, -1, &qres );
QmpRptSetPageWidthInChars( hNearMissReport, 79, &qres );
QmpMissRptSetPrintKeys( hNearMissReport, QTRUE, &qres );
QmpMissRptSetPrintSrc( hNearMissReport, QTRUE, &qres );
QmpMissRptUseDupGrp( hNearMissReport, hDupeGroups, &qres );
iAcceptance = ( iThreshHold > 90 ) ? iThreshHold : iThreshHold + 10;
QmpMissRptSetAcceptLvl( hNearMissReport, iAcceptance, &qres );
QmpPhaseSetRecProto( hNearMissReport, hProtoRec, &qres );

/* Print a Near Miss report */
sprintf( szStringPtr, "Rpt-NearMiss-%d.log", iThreshHold );
sprintf( pszTempPathBuffer, "%s%s%s", pszOutputPathBuffer, pszDelim,
        szStringPtr );
remove(pszTempPathBuffer);
QmpRptSetFilename( hNearMissReport, pszTempPathBuffer, &qres );
sprintf( szTempPtr, ": Near Miss Report for a Threshold of %d", iThreshHold );
strcat( szStringPtr, szTempPtr );
QmpRptSetTitle( hNearMissReport, szStringPtr, &qres );
QmpRptSetSubtitle( hNearMissReport, "Generated for Make-A-Wish Foundation",
                    &qres );
QmpMissRptUseDITR( hNearMissReport, hDITR, &qres );
if ( bEveryNth ) {
    QmpMissRptRegEveryNthRecFunc( hNearMissReport, pEveryNthRecordClient,
                                    &qres );
    QmpMissRptSetNthRecInterval( hNearMissReport, 200, &qres );
}
if ( bUseSampling ) {
    OutputSampling.lMaxRecords = 1000;      /* maximum number of records */
                                            /* to output */
    OutputSampling.lFirstRecordNum = 1;      /* first record to output */
    OutputSampling.lLastRecordNum = 1000;     /* last record to output */
    OutputSampling.lInterval = 0;            /* Sampling interval for output */
    OutputSampling.lGroupSize = 0;           /* size of group beginning at each */
                                            /* output interval */
    OutputSampling.lMaxDupeGroups = 5;       /* max number of Dupe Groups to */
```

QmpMissRptCreate

```
        /* output. */
OutputSampling.lDupesFirst = 1;           /* first Dupe Group to output. */
OutputSampling.lDupesLast = 20;            /* last DupeGroup to output. */
OutputSampling.lDupesInterval = 2;          /* Dupe Group sampling interval. */
QmpRptSetStatSamp( hNearMissReport, &OutputSampling, &qres );
}

printf( "Starting Near Miss Report phase:\n" );
if ( bUsePrePro ) {
    QmpMissRptAddPreProcDataSrc( hNearMissReport, hPreProDataSrc, &qres );
}
QmpPhaseStart( hNearMissReport, &qres );
QmpPhaseDestroy( hNearMissReport, &qres );
```

QmpMissRptDestroy

DESTROYS A NEAR MISS REPORT.

Syntax

```
void QmpMissRptDestroy( MpHnd in_hMissRpt, QRESULT*  
    out_pResult );
```

in_hMissRpt

Handle to the near miss report. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

See Also

QmpMissRptCreate

Example

```
QmpMissRptDestroy( in_hMissRpt, out_pResult );
```

QmpMissRptGetAcceptLvl

RETRIEVES THE ACCEPTANCE LEVEL OF THE NEAR MISS REPORT.

Syntax

```
int QmpMissRptGetAcceptLvl( MpHnd in_hMissRpt, QRESULT*  
    out_pResult );
```

in_hMissRpt
Handle to the near miss report. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the acceptance level of the near miss report if successful, or -1 if there is an error.

Notes

The acceptance level determines which records (and their associated dupe groups) are displayed in the near miss report. A dupe group (including all of its duplicates) is listed in the report if the score of one of the duplicates in the group is below the acceptance level, but above the dupe groups threshold. (If the dupe groups threshold is not set, then the record matching threshold is used.)

See Also

[QmpMissRptSetAcceptLvl](#)

Example

See example on [page 609](#).

QmpMissRptGetNthRecInterval

GETS THE NEAR MISS REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpMissRptGetNthRecInterval( MpHnd in_hMissRpt, QRESULT*
```

```
    out_pResult );
```

in_hMissRpt

Handle to the near miss report. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the report's Nth record event interval if successful, or `-1` if there is an error.

Notes

This function gets the near miss report phase's Nth record event interval.

The near miss report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an "occasional" notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the near miss report phase processes 100 records, it will send out a notification to *all* registered clients.

See Also

`QmpMissRptRegEveryNthRecFunc`, `QmpMissRptSetNthRecInterval`,
`QmpMissRptUnregEveryNthRecFunc`.

Example

```
lNthRecInterval = QmpMissRptGetNthRecInterval( in_hMissRpt,
```

```
    out_pResult );
```

QmpMissRptRegEveryNthRecFunc

Registers the event handler to be notified of every Nth record processed.

Syntax

```
void QmpMissRptRegEveryNthRecFunc ( MpHnd in_hMissRpt,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hMissRpt
Handle to near miss report. *Input*.

in_Func
Pointer to event handler. *Input*.

out_pResult
Result code.

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The near miss report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

See Also

[QmpMissRptGetNthRecInterval](#), [QmpMissRptSetNthRecInterval](#),
[QmpMissRptUnregEveryNthRecFunc](#).

Example

See example on [page 609](#).

QmpMissRptSetAcceptLvl

SETS THE ACCEPTANCE LEVEL OF THE NEAR MISS REPORT.

Syntax

```
void QmpMissRptSetAcceptLvl( MpHnd in_hMissRpt, int  
                           in_iAcceptanceLevel, QRESULT* out_pResult );
```

in_hMissRpt

Handle to the near miss report. *Input*.

in_iAcceptanceLevel

Acceptance level between 1 and 100. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The acceptance level determines which records (and their associated dupe groups) are displayed in the near miss report. A dupe group (including all of its duplicates) is listed in the report if the score of one of the duplicates in the group is below the acceptance level.

See Also

[QmpMissRptGetAcceptLvl](#)

Example

See example on [page 609](#).

QmpMissRptSetDupeGrpId

SPECIFIES WHETHER TO PRINT DUPE GROUP COLUMN IDs IN THE REPORT.

Syntax

```
void QmpMissRptSetDupeGrpId( MpHnd in_hMissRpt, QBOOL  
    in_bDupeGrpId, QRESULT* out_pResult );
```

in_hMissRpt
Handle to near miss report object. *Input*.

in_bDupeGrpId
If QTRUE, print dupe group IDs. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function specifies whether to print dupe group column IDs in the report.
The column name is “DupeID” when included in the report.

Example

```
QmpMissRptSetDupeGrpId( in_hMissRpt, QTRUE, out_pResult );
```

QmpMissRptSetDupeGrpScore

SPECIFIES WHETHER TO PRINT DUPE GROUP COLUMN SCORES IN THE REPORT.

Syntax

```
void QmpMissRptSetDupeGrpScore( MpHnd in_hMissRpt, QBOOL  
    in_bDupeGrpScore, QRESULT* out_pResult );
```

in_hMissRpt

Handle to near miss report object. *Input*.

in_bDupeGrpScore

If QTRUE, print dupe group scores. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The dupe group score is the match score between a master duplicate and subordinate duplicate. All the subordinates in a dupe group are scored against the group master. The column name is “Score” when included in the report.

Example

```
QmpMissRptSetDupeGrpScore( in_hMissRpt, QTRUE, out_pResult  
);
```

QmpMissRptSetListId

SPECIFIES WHETHER TO PRINT THE DATA LIST ID COLUMN IN THE REPORT.

Syntax

```
void QmpMissRptSetListId( MpHnd in_hMissRpt, QBOOL in_bListId,
                           QRESULT* out_pResult );
```

in_hMissRpt
Handle to near miss report object. *Input*.

in_bListId
If QTRUE, print data list IDs. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

A record's data list ID field records which data list the record belongs to. The column name is "ListID" when included in the report.

Example

```
QmpMissRptSetListId( in_hMissRpt, QTRUE, out_pResult );
```

QmpMissRptSetNthRecInterval

SETS NEAR MISS REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpMissRptSetNthRecInterval ( MpHnd in_hMissRpt, long
                                   in_lInterval, QRESULT* out_pResult );
in_hMissRpt
Handle to near miss report. Input.
in_lInterval
Near miss report Nth record event interval. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

This function sets the near miss report phase's Nth record event interval.

The near miss reports phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an "occasional" notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the near miss reports phase processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

[QmpMissRptGetNthRecInterval](#), [QmpMissRptRegEveryNthRecFunc](#),
[QmpMissRptUnregEveryNthRecFunc](#).

Example

See example on [page 609](#).

QmpMissRptSetPrintKeys

SPECIFIES WHETHER TO PRINT THE RECORD PRIMARY KEY COLUMN IN THE REPORT.

Syntax

```
void QmpMissRptSetPrintKeys (MpHnd in_hMissRpt, QBOOL  
    in_bPrintKeys, QRESULT* out_pResult );
```

in_hMissRpt

Handle to near miss report object. *Input*.

in_bPrintKeys

If QTRUE, print record primary keys. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The primary key is a number (generally a record number) that is unique to a data source. It is usually the counter value when the records are read in by the data input phase. If a job has one data source, the primary key uniquely identifies a record. If a job has multiple data sources, the primary key and source ID values uniquely identify a record.

The column name is “Key” when included in the report.

See Also

[QmpMissRptSetPrintSrc](#)

Example

See example on [page 609](#).

QmpMissRptSetPrintSrc

SPECIFIES WHETHER TO PRINT A RECORD SOURCE ID COLUMN IN THE REPORT.

Syntax

```
void QmpMissRptSetPrintSrc ( MpHnd in_hMissRpt, QBOOL  
    in_bPrintSources, QRESULT* out_pResult );  
  
in_hMissRpt  
    Handle to near miss report object. Input.  
  
in_bPrintSources  
    If QTRUE, print record source IDs. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

The record source ID field identifies which data source the record came from.
The column name is “SrcID” when included in the report.

See Also

[QmpMissRptSetPrintKeys](#)

Example

See example on [page 609](#).

QmpMissRptSetShowSubords

SPECIFIES WHETHER TO PRINT SUBORDINATE DUPLICATES IN THE NEAR MISS REPORT.

Syntax

```
void QmpMissRptSetShowSubords ( MpHnd in_hMissRpt, QBOOL
                               in_bShowSubords, QRESULT* out_pResult );
in_hMissRpt
Handle to near miss report. Input.
in_bShowSubords
If QTRUE, print subordinate duplicates. The default is QFALSE (do not show
subordinate duplicates). Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

If the value of *in_bShowSubords* is set to QTRUE, all types of duplicates, including subordinate duplicates, are included in the report. If the value of *in_bShowSubords* is set to QFALSE, or never set at all, only master duplicates and subordinate duplicates scoring below the acceptance level are included.

For example, if *in_bShowSubords* is set to QTRUE, a dupe group in the report would look like this:

M/S	DupeID	Score	ListID	Key	SrcID	FIRSTNAME	LASTNAME
M	2	100	2	2	8	ANTHON	ALMARALES
S	2	100	1	2	1	ANTHON	ALMARALES
S	2	100	1	3	1	ANTHON	ALMARALES
S	2	82	1	4	1	ATHON	ALMRALES
->	2	67	1	5	1	ANTHO	ALPARALES

If *in_bShowSubords* is set to QFALSE, the same dupe group would be represented as such:

M/S	DupeID	Score	ListID	Key	SrcID	FIRSTNAME	LASTNAME
M	2	100	2	2	8	ANTHON	ALMARALES
->	2	67	1	5	1	ANTHO	ALPARALES

Example

```
QmpDeclHnd( hMissRpt );
QRESULT qres;

QmpMissRptSetShowSubords( hMissRpt, QTRUE, &qres );
```

QmpMissRptUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpMissRptUnregEveryNthRecFunc ( MpHnd in_hMissRpt,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hMissRpt
    Handle to near miss report. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function unregisters the specified event handler from the near miss report, so that the event handler will no longer be notified of every Nth record processed.

The near miss report can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

See Also

[QmpMissRptGetNthRecInterval](#), [QmpMissRptRegEveryNthRecFunc](#),
[QmpMissRptSetNthRecInterval](#).

Example

```
QmpMissRptUnregEveryNthRecFunc ( in_hMissRpt, in_Func,
                                         out_pResult );
```

QmpMissRptUseCons

GIVES THE DATA CONSOLIDATION PHASE TO A NEAR MISS REPORT.

Syntax

```
void QmpMissRptUseCons( MpHnd in_hMissRpt, MpHnd in_hCons,  
    QRESULT* out_pResult );
```

in_hMissRpt

Handle to a near miss report. *Input*.

in_hCons

Handle to data consolidation phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The near miss report requires the data consolidation phase to make sure that any updated duplicates are properly registered.

Example

```
QmpMissRptUseCons ( hMissRpt, hCons, &qres );
```

QmpMissRptUseDupGrp

SPECIFIES THE DUPE GROUPS OBJECT TO USE FOR THE NEAR MISS REPORT.

Syntax

```
void QmpMissRptUseDupGrp( MpHnd in_hMissRpt, MpHnd  
                           in_hDupGrps, QRESULT* out_pResult );
```

in_hMissRpt

Handle to near miss report object. *Input*.

in_hDupGrps

Handle to dupe groups object. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

The dupe groups object is used to determine a record's master/subordinate status and dupe group membership.

Example

See example on [page 609](#).

Function Class: QmpObj*

FUNCTIONS FOR MANIPULATING LIBRARY OBJECTS.

Most objects in the Centrus Merge/Purge C interface are descended from a fundamental Centrus Merge/Purge object called the “QmpObj”. The functions that support this object (getting the object name, setting the object name, and getting the object ID) will work with the handle of any other Centrus Merge/Purge object. In other words, *everything* in a Centrus Merge/Purge application is also a Centrus Merge/Purge object.

With any object handle, you can find out the name of the corresponding object, set the name of that object, or get the integer ID of that object. The convenient base object makes it very easy to pass handles in and out of all the Centrus Merge/Purge functions, to add objects to internal collections, and to treat Centrus Merge/Purge objects in a uniform and consistent manner.

Quick Reference

Function	Description	Page
QmpObjGetID	Gets the ID of an object.	628
QmpObjGetName	Gets the name of the specified object.	629
QmpObjGetNameVB	Gets the name of the specified object (Visual Basic version).	630
QmpObjSetName	Sets the name of the object.	631

QmpObjGetID

GETS THE ID OF AN OBJECT.

Syntax

```
unsigned long QmpObjGetID ( MpHnd in_hObj, QRESULT*  
    out_pResult );
```

in_hObj
Handle to object. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the (unique) ID number of an object if successful, or 0 if there is an error.

Notes

Every Centrus Merge/Purge entity that has a handle may be manipulated with the `QmpObj*` functions.

Each object created by the library is assigned an ID that is unique for the duration of program execution. The IDs are assigned sequentially, but since the library creates many objects behind the scenes, gaps in the ID sequence will occur from the perspective of the library user.

The IDs can be used to distinguish or identify an object. This is particularly useful for certain callbacks registered with the library which, when invoked by the library, receive the ID (or handle) of the object performing the callback (or on behalf of the object for which the callback is being made). If the application callback is written to behave differently based on the value of the object ID, this will allow the use of a single callback function to service callbacks from a number of different objects.

See Also

`QmpDataSrcSetFillFunc`
`QmpDataInpRegEveryNthRecFunc`
`QmpRecMatRegEveryNthRecFunc`
`QmpDupsRptRegEveryNthRecFunc`
`QmpUniqsRptRegEveryNthRecFunc`
`QmpDupGrpsRegEveryNthRecFunc`
`QmpDataDestRegEveryNthRecFunc`

Example

```
ulObjectID = QmpObjGetID (hObj, pResult);
```

QmpObjGetName

GETS THE NAME OF THE SPECIFIED OBJECT.

Syntax

```
const char* QmpObjGetName( MpHnd in_hObj, QRESULT* out_pResult
    );
```

in_hObj

Handle to object. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns pointer to object's name if successful, or `NULL` if there is an error.

Notes

Every Centrus Merge/Purge entity that has a handle may be manipulated with the `QmpObj*` functions. This function gets the name of the object.

Example

```
/* This example shows how a record matcher */
/* can be treated as a CMP object */

char nameBuffer[100];
const char* szName;
MpHnd hRecMat;

hRecMat = QmpRecMatCreate(QMS_CRITERIA_AVERAGE, 90, QFALSE,
    &qres);
QmpObjSetName(hRecMat, "The Matcher", &qres);
szName = QmpObjGetName(hRecMat, &qres);
QmpObjGetNameVB(hRecMat, nameBuffer, sizeof(nameBuffer),
    &qres);
```

QmpObjGetNameVB

GETS THE NAME OF THE SPECIFIED OBJECT (VISUAL BASIC VERSION).

Syntax

```
void QmpObjGetNameVB( MpHnd in_hObj, char* io_szBuffer, long
                      in_lSize, QRESULT* out_pResult );

in_hObj
    Handle to object. Input.
io_szBuffer
    Buffer to put name into. Input, Output.
in_lSize
    Size of buffer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Every Centrus Merge/Purge entity that has a handle may be manipulated with the `QmpObj*` functions. This function gets the name of the object.

Example

```
/* This example shows how a record matcher */
/* can be treated as a CMP object */

char nameBuffer[100];
const char* szName;
MpHnd hRecMat;

hRecMat = QmpRecMatCreate(QMS_CRITERIA_AVERAGE, 90, QFALSE,
                           &qres);
QmpObjSetName(hRecMat, "The Matcher", &qres);
szName = QmpObjGetName(hRecMat, &qres);
QmpObjGetNameVB(hRecMat, nameBuffer, sizeof(nameBuffer),
                 &qres);
```

QmpObjSetName

SETS THE NAME OF THE OBJECT.

Syntax

```
void QmpObjSetName( MpHnd in_hObj, const char* in_pszName,
                     QRESULT* out_pResult );
in_hObj
    Handle to object. Input.
in_pszName
    Pointer to name. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Every Centrus Merge/Purge entity that has a handle may be manipulated with the **QmpObj*** functions. This function sets the object's name, which is recognized by the application.

Example

```
/* This example shows how a record matcher */
/* can be treated as a CMP object */

char nameBuffer[100];
const char* szName;
MpHnd hRecMat;

hRecMat = QmpRecMatCreate(QMS_CRITERIA_AVERAGE, 90, QFALSE,
                           &qres);
QmpObjSetName(hRecMat, "The Matcher", &qres);
szName = QmpObjGetName(hRecMat, &qres);
QmpObjGetNameVB(hRecMat, nameBuffer, sizeof(nameBuffer),
                 &qres);
```


Function Class: QmpPassThru*

FUNCTIONS THAT PREVENT UNUSED FIELDS FROM BEING READ INTO THE DITR.

The data pass-through process basically consists of:

- Creating a pass-through prototype record, containing all of the pass-through fields
- Creating a field map object for each data source containing pass-through fields; the map associates data source field names with pass-through prototype record field names
- Attaching the objects needed by the pass-through process (data input phase, table generation phase, pass-through prototype record, field maps, etc.)

Quick Reference

Function	Description	Page
QmpPassThruSetRecord	Sets the prototype record for data pass-through.	634
QmpPassThruUseDataInp	Adds the data input phase for data pass-through.	635
QmpPassThruUseFldMap	Sets a field map for the data pass-through record.	636

QmpPassThruSetRecord

SETS THE PROTOTYPE RECORD FOR DATA PASS-THROUGH.

Syntax

```
void QmpPassThruSetRecord ( MpHnd in_hTblGen, MpHnd in_hRec,
                           QRESULT* out_pResult );
in_hTblGen
    Handle to table generation phase. Input.
in_hRec
    Handle to pass-through prototype record. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

When using data passthrough, the table generation phase needs to be given the fields defined as passthrough. The table generation phase will then use the fieldmap to map from the table data source to the passthrough record.

See Also

[QmpPassThruUseFldMap](#).

Example

```
/* test data passthrough */
hPassThruFldMap = QmpFldMapCreate( &qres );
QmpTblOpen( hTblInput, &qres );
hInputRec = QmpRecCreate( &qres );
QmpTblGetFlds( hTblInput, hInputRec, &qres );
QmpTblClose( hTblInput, &qres );
QmpFldMapUseRecs( hPassThruFldMap, hPassThruRec, hInputRec,
                   &qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szCity, "CITY", &qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szState, "STATE", &qres
);
QmpFldMapMapFldToFld( hPassThruFldMap, szRecId, "RECID", &qres
);

QmpPassThruSetRecord( hTableGen, hPassThruRec, &qres );
QmpPassThruUseDataInp( hTableGen, hDataInp, &qres );
QmpPassThruUseFldMap( hDataInp, hPassThruFldMap, 0, &qres );
```

QmpPassThruUseDataInp

ADDS THE DATA INPUT PHASE FOR DATA PASS-THROUGH.

Syntax

```
void QmpPassThruUseDataInp( MpHnd in_hTblGen, MpHnd
                            in_hDataInp, QRESULT* out_pResult );
in_hTblGen
    Handle to table generation phase. Input.
in_hDataInp
    Handle to data input phase. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

If you are using data passthrough, the table generation phase needs the actual table data sources to retrieve information from. By giving table generation the data input phase, the data sources are given in the order they are added to the DITR.

See Also

[QmpPassThruSetRecord](#), [QmpPassThruUseFldMap](#).

Example

```
/* test data passthrough */
hPassThruFldMap = QmpFldMapCreate( &qres );
QmpTblOpen( hTblInput, &qres );
hInputRec = QmpRecCreate( &qres );
QmpTblGetFlds( hTblInput, hInputRec, &qres );
QmpTblClose( hTblInput, &qres );
QmpFldMapUseRecs( hPassThruFldMap, hPassThruRec, hInputRec,
&qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szCity, "CITY", &qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szState, "STATE", &qres
);
QmpFldMapMapFldToFld( hPassThruFldMap, szRecId, "RECID", &qres
);

QmpPassThruSetRecord( hTableGen, hPassThruRec, &qres );
QmpPassThruUseDataInp( hTableGen, hDataInp, &qres );
QmpPassThruUseFldMap( hDataInp, hPassThruFldMap, 0, &qres );
```

QmpPassThruUseFldMap

SETS A FIELD MAP FOR THE DATA PASS-THROUGH RECORD.

Syntax

```
void QmpPassThruUseFldMap( MpHnd in_hDataInp, MpHnd
                           in_hFldMap, int in_iIndex, QRESULT* out_pResult );
in_hDataInp
    Handle to data input phase. Input.
in_hFldMap
    Handle to field map. Input.
in_iIndex
    Index value of a data source attached to the data input phase. This is the
    data source whose fields are being mapped. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Call **QmpPassThruUseFldMap** to set a data source and its associated field map for data pass-through.

See Also

QmpPassThruSetRecord.

Example

```
/* test data passthrough */
hPassThruFldMap = QmpFldMapCreate( &qres );
QmpTblOpen( hTblInput, &qres );
hInputRec = QmpRecCreate( &qres );
QmpTblGetFlds( hTblInput, hInputRec, &qres );
QmpTblClose( hTblInput, &qres );
QmpFldMapUseRecs( hPassThruFldMap, hPassThruRec, hInputRec,
                   &qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szCity, "CITY", &qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szState, "STATE", &qres );
QmpFldMapMapFldToFld( hPassThruFldMap, szRecId, "RECID", &qres );

QmpPassThruSetRecord( hTableGen, hPassThruRec, &qres );
QmpPassThruUseDataInp( hTableGen, hDataInp, &qres );
QmpPassThruUseFldMap( hDataInp, hPassThruFldMap, 0, &qres
) ;
```

Function Class: QmpPhase*

FUNCTIONS FOR MANIPULATING PHASES.

The following table lists the library phases (in their typical order in an application), as well as the classes of functions devoted to them:

Phase	Class of Functions Devoted to the Phase
Data input	QmpDataInp*
Record matching	QmpRecMat*
Dupe groups	QmpDupGrps*
Data consolidation	QmpCons*
Table generation	QmpTblGen*
Unique records report	QmpUniqsRpt*
Duplicate records report	QmpDupsRpt*
Consolidated records report	QmpConsRpt*
List-by-list report	QmpListByListRpt*
Near miss report	QmpMissRpt*
Job summary report	QmpJobRpt*

In the diagram “[Centrus Merge/Purge Object Hierarchy](#)” on page 65, notice that the data input phase, record matching phase, and all the other phase objects are derived from the QmpPhase object. A handle to any of these phase objects may be passed to any QmpPhase* function.

Quick Reference

Function	Description	Page
QmpPhaseDestroy	Destroys a phase object.	638
QmpPhaseDiskSpace	Retrieves an estimate of the disk space needed by a phase.	639
QmpPhaseGetState	Gets the current state of a phase.	643
QmpPhaseGetType	Returns the type of a phase.	645
QmpPhaseSetRecProto	Sets the fields structure for input data via an application-owned record.	646
QmpPhaseStart	Starts the phase.	647
QmpPhaseStop	Stops the phase.	648
QmpPhaseUseDITR	Specifies the Data Input Table Repository to be used by the phase.	649

QmpPhaseDestroy

DESTROYS A PHASE OBJECT.

Syntax

```
void QmpPhaseDestroy ( MpHnd in_hPhase, QRESULT* out_pResult  
);
```

in_hPhase

Handle to phase object. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function destroys the specified phase object. Some examples of phases that can be destroyed using this function include the record matching phase, data input phase, dupe groups phase, or any report phase.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

Example

See example on [page 649](#).

QmpPhaseDiskSpace

RETRIEVES AN ESTIMATE OF THE DISK SPACE NEEDED BY A PHASE.

Syntax

```
long QmpPhaseDiskSpace ( MpHnd in_hPhase, QMS_DISK_USAGE*
    io_pDiskUse, QRESULT* out_pResult );
in_hPhase
    Handle to phase object. Input.
io_pDiskUse
    Pointer to disk usage structure. See “QMS\_DISK\_USAGE” on page 82 for
    structure details. Input, output.
out_pResult
    Result code. Output.
```

Return Value

QmpPhaseDiskSpace returns the number of kilobytes of disk space a phase is predicted to use. The following table explains the return values for the various phases if successful. The structure members referred to in the table are from the [QMS_DISK_USAGE](#) structure. Of course, -1 is returned if there is an error.

Phase Given to Function	Return Value
Data Input	DITR size; same as structure member IDataInputUsage .
Record Matching	The predicted sizes for the sorted DITR + the predicted size for the sorted versions of the preprocessed data source (should they need to be built); same as structure member IMatchUsage .
Dupe Groups	The predicted size of the dupe groups table at 5% matching + the predicted size of the tempDGT at 5% matching. The same as the sum of structure members ITempDgt5Usage and IDupeTable5Usage .
Table Generation	The total number of bytes required if each table included every input record; same as structure member ITableGenUsage .
Duplicates Report	The total number of predicted bytes for the duplicate report, based upon total sampling; same as structure member IDupeRptUsage .
Near Miss Report	The total number of predicted bytes for the near miss report, based upon ½ total; same as structure member IMissRptUsage .
Consolidated Records Report	The total number of predicted bytes for the consolidated report based upon ½ total; same as structure member IConsRptUsage .
Uniques Report	The total number of predicted bytes for the uniques report, based upon total sampling; same as structure member IUniqRptUsage .
List-by-List Report	The total number of predicted bytes for the list-by-list report, based upon total sampling; same as structure member IListRptUsage .
Job Summary Report	The total number of predicted bytes for the Job Summary report, based upon total sampling; same as structure member IJobRptUsage .

Notes

QmpPhaseDiskSpace returns an estimate of the disk space that a phase will require during execution. It also sets the appropriate member variables of a structure passed in as a parameter.

QmpPhaseDiskSpace sets most of the member variables in the QMS_DISK_USAGE structure, but at least one is set aside for use by the application. The following table lists the member variables set by calling **QmpPhaseDiskSpace** with a particular phase handle:

Structure Member	Entity Responsible for Assigning a Value
ITotalUsage	Application
IDitrRecCount IDitrRecWidth ISoundexCount IDataInpUsage	QmpPhaseDiskSpace called with data input handle
IPreProCount IMatchUsage IMatchRes5Usage	QmpPhaseDiskSpace called with record matcher handle
IDupeTable5Usage ITempDgt5Usage LDupeConsUsage	QmpPhaseDiskSpace called with dupe groups handle
ITableGenUsage	QmpPhaseDiskSpace called with table generator handle
IDupeRptUsage	QmpPhaseDiskSpace called with dupes report handle
IMissRptUsage	QmpPhaseDiskSpace called with near miss report handle
IConsRptUsage	QmpPhaseDiskSpace called with consolidated records report handle
IUniqRptUsage	QmpPhaseDiskSpace called with uniques report handle
IListRptUsage	QmpPhaseDiskSpace called with list-by-list report handle
IJobRptUsage	QmpPhaseDiskSpace called with job summary report handle

To determine how much disk space a job will require, first determine what level of record matching you want to base the estimate upon. The disk space requirements of some job components, such as the match result table or dupe groups table, can't be precisely predicted before run time because the number of matches is still unknown. In these cases, Centrus Merge/Purge uses a 5% matching rate as the basis of its calculations. If you want to make an estimate based upon a different matching rate, use the library value as a starting point for your own calculations.

Secondly, remember that for a given phase, the return value of **QmpPhaseDiskSpace** and the values stored in the QMS_DISK_USAGE structure sometimes represent different things. For example, the return value of **QmpPhaseDiskSpace** when called with a record matcher handle is the size of the sorted DITR added to the size of the sorted versions of the preprocessed data source. This value does not include the size of the match result table, which corresponds to the structure member IMatchRes5Usage. When **QmpPhaseDiskSpace** is called with a dupe groups handle, the return value is

the sum of the dupe groups table and the tempDGT sizes. This value does not include the size of the consolidated records table, which corresponds to the structure member `lDupeConsUsage`. Depending upon the information you want reflected in your estimate, estimating a total disk space requirement from the data structure members, rather than from `QmpPhaseDiskSpace` return values, may be more accurate.

For example, imagine that you want to calculate the disk space required for a job. This job has all of the phases represented, and you feel strongly that it will have a 50% match rate. The total job requirement is calculated by taking the sum of all the individual phase requirements, which are represented in the following table:

Phase	Calculating Disk Space Requirement Using QMS_DISK_USAGE Structure
Data Input	<code>IDataInputUsage</code>
Record Matching	<code>IMatchUsage + (10 * IMatchRes5Usage)</code>
Dupe Groups	<code>(10 * IDupeTable5Usage) + (10 * ITempDgt5Usage) + IDupeConsUsage</code>
Table Generation	<code>ITableGenUsage</code>
Dupes Report	<code>IDupeRptUsage</code>
Near Miss Report	<code>IMissRptUsage</code>
Consolidated Records Report	<code>IConsRptUsage</code>
Uniques Report	<code>IUniqRptUsage</code>
List-by-list Report	<code>IListRptUsage</code>
Job Summary Report	<code>IJobRptUsage</code>

Notice that the structure elements that represent the match result table, dupe groups table, and tempDGT requirements are each multiplied by 10. By default, the library calculates the space requirement of these tables based on a 5% matching rate. Multiplying these values by 10 gives a space requirement based on a 50% matching rate.

When calling `QmpPhaseDiskSpace` more than once, pass in the phase handles in execution order. For example, if a job contained data input, record matching, and dupe groups phases, you call `QmpPhaseDiskSpace` three times: the first time with the data input handle, the second with the record matcher handle, and the third with the dupe groups handle.

It is the application's responsibility to determine if there is enough actual disk space to fulfill the demand.

Example

```
long lDiskSpaceRequired;
QmpDeclHnd( hDataInputPhase );
QMS_DISK_USAGE DiskUseStruct;
QRESULT qres;

lDiskSpaceRequired = QmpPhaseDiskSpace ( hDataInputPhase,
    &DiskUseStruct, &qres );
```

QmpPhaseGetState

GETS THE CURRENT STATE OF A PHASE.

Syntax

```
QMS_PHASE_STATE QmpPhaseGetState ( MpHnd in_hPhase, QRESULT*  
    out_pResult );
```

in_hPhase
Handle to phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns phase state if successful, or QMS_PHASE_STATE_NONE if there is an error.

Notes

All of the phases go through several different states during their lifetimes. The only time an application can invoke this function once a phase has been started and is running is in response to an event notification. The following list details the meaning of the various states:

QMS_PHASE_STATE_NONE	Each phase sets its state to QMS_PHASE_STATE_NONE when destroyed. QmpPhaseGetState should never return this result.
QMS_PHASE_STATE_IDLE	Each phase sets its state to QMS_PHASE_STATE_IDLE when initially created and when cleared.
QMS_PHASE_STATE_READY	A phase's state is set to QMS_PHASE_STATE_READY when it is ready to be started.
QMS_PHASE_STATE_DONE	A phase enters this state when it has completed its work after being started. This is the case whether the state completes its work normally or is cancelled.
QMS_PHASE_STATE_FAILED	A phase enters this state if it encounters an error condition while running that prevents it from completing normally.
QMS_PHASE_STATE_STARTING	For a short period of time after a phase is started, it enters this phase on the way to entering the QMS_PHASE_STATE_STARTED state.
QMS_PHASE_STATE_STARTED	A phase enters this state when it has successfully been started.

QMS_PHASE_STATE_STOPPING	For a short period of time after a phase has been cancelled, it enters this phase on the way to entering the QMS_PHASE_STATE_STOPPED state.
QMS_PHASE_STATE_STOPPED	A phase enters this state when it has successfully been stopped.

See Also

QmpPhaseStart, **QmpPhaseStop**

Example

```
QMS_PHASE_STATE state = QmpPhaseGetState ( hDataInput, &qRes );
```

QmpPhaseGetType

RETURNS THE TYPE OF A PHASE.

Syntax

```
QMS_PHASE_TYPE QmpPhaseGetType ( MpHnd in_hPhase, QRESULT*  
    out_pResult );
```

in_hPhase

Handle to phase. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns phase type if successful. Possible values are:

QMS_PHASE_TYPE_NONE
QMS_PHASE_TYPE_MATCH
QMS_PHASE_TYPE_REPORT
QMS_PHASE_TYPE_DATAINPUT
QMS_PHASE_TYPE_TABLEINDEX
QMS_PHASE_TYPE_DUPEGROUPS
QMS_PHASE_TYPE_TABLEGENERATOR
QMS_PHASE_TYPE_UNIQUE_REPORT
QMS_PHASE_TYPE_DUPS_REPORT
QMS_PHASE_TYPE_NEARMISS_REPORT
QMS_PHASE_TYPE_CONSOL_REPORT
QMS_PHASE_TYPE_LISTBYLIST_REPORT
QMS_PHASE_TYPE_CONSOLIDATION
QMS_PHASE_TYPE_SUMM_REPORT

If there is an error, QMS_PHASE_TYPE_NONE is returned.

Notes

This routine is typically used by code that is used for various kinds of phases, but needs to do something differently depending on the type of phase. The value QMS_PHASE_TYPE_NONE is briefly used while a phase is initially being constructed and when a phase is destroyed.

See Also

QmpPhaseGetState

Example

```
QMS_PHASE_TYPE PhaseType = QmpPhaseGetType (hPhase, pResult);
```

QmpPhaseSetRecProto

SETS THE FIELDS STRUCTURE FOR INPUT DATA VIA AN APPLICATION-OWNED RECORD.

Syntax

```
void QmpPhaseSetRecProto ( MpHnd in_hPhase, MpHnd in_hRec,
                           QRESULT* out_pResult );
```

in_hPhase
Handle to phase. *Input*.

in_hRec
Handle to the prototype record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpPhaseSetRecProto uses an application-owned record to set the data fields structure for a phase. All phases must use the same data fields structure. Typically, an application uses the same record in this call for all phases.

Example

```
QmpPhaseSetRecProto ( hListByListReport, hSourceRecord,
                        pResult );
```

QmpPhaseStart

STARTS THE PHASE.

Syntax

```
void QmpPhaseStart ( MpHnd in_hPhase, QRESULT* out_pResult );  
in_hPhase  
Handle to phase. Input.  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

Once a phase has been successfully set up, it is started using this function.

See Also

[QmpPhaseGetState](#), [QmpPhaseGetType](#)

Example

```
/* Start data input and record matcher */  
QmpPhaseStart ( hDataInput, pResult );  
QmpPhaseStart ( hMatcher, pResult );
```

QmpPhaseStop

STOPS THE PHASE.

Syntax

```
void QmpPhaseStop ( MpHnd in_hPhase, QRESULT* out_pResult );
```

in_hPhase
Handle to phase. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function stops a phase's execution. A phase cannot continue after being stopped. This function can only be invoked when the application has control. The only time the application gains control once a phase has started but not yet completed is in response to an event generated by the phase. The phase will terminate gracefully at the next opportunity.

See Also

[QmpPhaseStart](#)

Example

```
/* Stop data input and record matcher */  
QmpPhaseStop ( hDataInput, pResult );  
QmpPhaseStop ( hMatcher, pResult );
```

QmpPhaseUseDITR

SPECIFIES THE DATA INPUT TABLE REPOSITORY TO BE USED BY THE PHASE.

Syntax

```
void QmpPhaseUseDITR ( MpHnd in_hPhase, MpHnd in_hTbl,
                      QRESULT* out_pResult );
in_hPhase
    Handle to the phase. Input.
in_hTbl
    Handle to DITR. This table must be of type CodeBase. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This functions specifies the DITR object to be used by a phase.
A DITR table *must* be of type CodeBase.

Example

```
QmpDeclHnd( hDITR );
hDITR = QmpTblCbCreate( "", "ditr.dbf", &qres );
QmpIdxGenUseIdxKey( hIndexGenerator, hIndexKey1, & qres );
QmpIdxGenUseIdxKey( hIndexGenerator, hIndexKey2, & qres );
QmpIdxGenUseIdxKey( hIndexGenerator, hIndexKey3, & qres );
QmpPhaseUseDITR( hIndexGenerator, hDITR, & qres );
printf( "Starting Index generation phase:\n" );
QmpPhaseStart( hIndexGenerator, &qres );
...
QmpPhaseDestroy( hIndexGenerator, &qres );
QmpTblClose( hDITR, &qres );
QmpTblDestroy( hDITR, &qres );
```


Function Class: QmpRec*

FUNCTIONS FOR MANIPULATING RECORD OBJECTS.

Quick Reference

Function	Description	Page
QmpRecAdd	Adds a field to a record.	653
QmpRecAddByType	Adds a field of a specified type to a record.	654
QmpRecAddByTypePicture	Adds a field to a record and specifies its type and picture.	656
QmpRecAddWithWidth	Adds a field to a record (with width data).	657
QmpRecClear	Clears a record.	658
QmpRecClearData	Clears the contents of a record, leaving its field definitions alone.	659
QmpRecCopy	Copies one record's structure to another.	660
QmpRecCreate	Creates a record.	661
QmpRecDestroy	Destroys a record.	662
QmpRecGetDataPictureByHnd	Queries data format by handle.	663
QmpRecGetDataPictureByName	Queries data format by name.	664
QmpRecGetFldByHnd	Get the value of a field by passing in the index of the field.	665
QmpRecGetFldByName	Gets the value of a field by passing in the name of the field.	666
QmpRecGetFldCnt	Gets the current field count for the specified record.	667
QmpRecGetFldDecimalsByHnd	Returns the field decimals when given the field index.	668
QmpRecGetFldDecimalsByName	Returns field decimals when given the field name.	669
QmpRecGetFldHnd	Gets the handle of a field by passing in the name of the field.	670
QmpRecGetFldName	Gets the name of a field.	671
QmpRecGetFldNameVB	Gets the name of a field (Visual Basic version).	672
QmpRecGetFldTrimByHnd	Gets the field trimming property by passing in the handle of the field.	673
QmpRecGetFldTrimByName	Gets the field trimming property by passing in the name of the field.	674
QmpRecGetFldTypeByHnd	Query field type by index.	675
QmpRecGetFldTypeByName	Query field type by name.	676
QmpRecGetFldWidthByHnd	Get the field width using the index.	677

Function	Description	Page
QmpRecGetFldWidthByName	Query field width by name.	678
QmpRecIsValid	Tests whether a record is valid.	679
QmpRecSetCharPtrFldByHnd	Sets a character field.	680
QmpRecSetCharPtrFldByName	Sets a character field.	681
QmpRecSetDataPictureByHnd	Sets the preferred data format for the field.	682
QmpRecSetDataPictureByName	Sets the preferred data format for the field.	683
QmpRecSetDateFldByHnd	Sets a date field by handle.	684
QmpRecSetDateFldByName	Sets a date field by name.	686
QmpRecSetDoubleFldByHnd	Sets a double field by handle.	687
QmpRecSetDoubleFldByName	Sets a double field by name.	688
QmpRecSetFldByHnd	Sets the value of a field from a variant.	689
QmpRecSetFldByName	Sets the value of a field from a variant.	690
QmpRecSetFldTrimByHnd	set the value of the field trimming by passing in the handle of the field	691
QmpRecSetFldTrimByName	Sets the field trimming property by passing in the name of the field.	692
QmpRecSetFloatFldByHnd	Sets the value of a float field.	693
QmpRecSetFloatFldByName	Sets the value of a float field.	694
QmpRecSetLongFldByHnd	Sets the value of a long field.	695
QmpRecSetLongFldByName	Sets the value of a long field.	696
QmpRecValidFld	Tests whether a field name represents a valid field.	697
QmpRecValidHnd	Tests whether a field index represents a valid field.	698

QmpRecAdd

ADDS A FIELD TO A RECORD.

Syntax

```
int QmpRecAdd( MpHnd in_hRec, const char* in_szFldName,
                QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Name of field to add. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the index of the added field if successful, or -1 if there is an error.

Notes

This function adds a data field to a record's structure. Default values of 20 for the field width, QMS_VARIANT_STRING for the field type, and 0 for the field decimals are used.

See Also

`QmpRecAddByType`, `QmpRecAddWithWidth`

Example

```
iZipCodeIndex = QmpRecAdd (hSourceRecord, "ZipCode", pResult
);
```

QmpRecAddByType

ADDS A FIELD OF A SPECIFIED TYPE TO A RECORD.

Syntax

```
int QmpRecAddByType( MpHnd in_hRec, const char* in_szFldName,
                      QMS_VARIANT_TYPE in_FieldType, int in_iFldWidth, int
                      in_iDecimals, QRESULT* out_pResult );
```

in_hRec

Handle to record. *Input*.

in_szFldName

Name of field to be added. *Input*.

in_FieldType

Field type. *Input*.

Allowable field types are:

QMS_VARIANT_EMPTY	Indicates an uninitialized variant
-------------------	------------------------------------

QMS_VARIANT_BOOLEAN	
---------------------	--

QMS_VARIANT_CHAR	
------------------	--

QMS_VARIANT_DOUBLE	
--------------------	--

QMS_VARIANT_FLOAT	
-------------------	--

QMS_VARIANT_INT	
-----------------	--

QMS_VARIANT_LONG	
------------------	--

QMS_VARIANT_STRING	
--------------------	--

QMS_VARIANT ULONG	
-------------------	--

QMS_VARIANT USHORT	
--------------------	--

QMS_VARIANT_VOIDSTAR	Generic pointer to void; for structures, etc.
----------------------	---

in_iFldWidth

Field width. *Input*.

in_iDecimals

Field decimals. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the index of the added field if successful, or -1 if there is an error.

Notes

QmpRecAddByType adds a data field to a record's structure.

If the field holds floating point information (i.e., the type is QMS_VARIANT_DOUBLE or QMS_VARIANT_FLOAT), the field width includes the integer, decimal point, and decimal digits. For example, if the added field could hold the following floating point number:

3.14159

The field width would need to be 7 (1 integer digit, 1 decimal point, and 5 decimals).

See Also

`QmpRecAdd`, `QmpRecAddWithWidth`

Example

```
QmpRecAddByType(hSeekRecord, "QQPRIKEY", QMS_VARIANT_LONG,
                  width, 0, &Result);
if(!qmpSucceeded(Result)) {
    DisplayResultMsg(Result, "Unable to add a new field to" "seek
                      record");
}
```

QmpRecAddByTypePicture

ADDS A FIELD TO A RECORD AND SPECIFIES ITS TYPE AND PICTURE.

Syntax

```
int QmpRecAddByTypePicture( MpHnd in_hRec, const char*
    in_szFldName, QMS_VARIANT_TYPE in_FldType, int
    in_iFldWidth, const char* in_szPicture, QRESULT* out_pResult
);
```

in_hRec

Handle to record. *Input*.

in_szFldName

Name of field to be added. *Input*.

in_FldType

Field type. *Input*.

Allowable field types are:

QMS_VARIANT_EMPTY	Indicates an uninitialized variant
QMS_VARIANT_BOOLEAN	
QMS_VARIANT_CHAR	
QMS_VARIANT_DOUBLE	
QMS_VARIANT_FLOAT	
QMS_VARIANT_INT	
QMS_VARIANT_LONG	
QMS_VARIANT_STRING	
QMS_VARIANT_ULONG	
QMS_VARIANT USHORT	
QMS_VARIANT_VOIDSTAR	Generic pointer to void; for structures, etc.

in_iFldWidth

Field width. *Input*.

in_szPicture

Data picture. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the index of the added field if successful, or -1 if there is an error.

Notes

This function adds a field to a record and specifies its type and picture. It is currently used only for field type QMS_VARIANT_DATE.

Example

See example on [page 684](#).

QmpRecAddWithWidth

ADDS A FIELD TO A RECORD (WITH WIDTH DATA).

Syntax

```
int QmpRecAddWithWidth( MpHnd in_hRec, const char*
    in_szFldName, int in_iFldWidth, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Name of field to be added. Input.
in_iFldWidth
    Field width. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the index of the added field if successful, or -1 if there is an error.

Notes

QmpRecAddWithWidth adds a data field to a record's structure.

Default values of **QMS_VARIANT_STRING** for the field type and 0 for the field decimals are used.

See Also

QmpRecAdd, **QmpRecAddByType**

Example

```
iAddr = QmpRecAddWithWidth(hSourceRecord, "AddrLine1", 35,
    pResult );
iCity = QmpRecAddWithWidth(hSourceRecord, "City", 20, pResult
);
iFirstName = QmpRecAddWithWidth(hSourceRecord, "FirstName",
    10, pResult );
iLastName = QmpRecAddWithWidth(hSourceRecord, "LastName", 15,
    pResult );
iState = QmpRecAddWithWidth(hSourceRecord, "State", 2, pResult
);
iZipCode = QmpRecAddWithWidth(hSourceRecord, "ZipCode", 10,
    pResult );
```

QmpRecClear

CLEAR A RECORD.

Syntax

```
void QmpRecClear( MpHnd in_hRec, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets a record back to its initial (blank) state.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QRESULT qRes;  
  
QmpRecClear(hPrototype, &qRes);
```

QmpRecClearData

CLEARSES THE CONTENTS OF A RECORD, LEAVING ITS FIELD DEFINITIONS ALONE.

Syntax

```
void QmpRecClearData( MpHnd in_hRec, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function clears the fields of a record, but leaves the record's field definitions intact.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

See example on [page 684](#).

QmpRecCopy

COPIES ONE RECORD'S STRUCTURE TO ANOTHER.

Syntax

```
void QmpRecCopy (MpHnd in_hDestRec, MpHnd in_hSrcRec, QRESULT*  
    out_pResult );
```

in_hDestRec
Handle to destination record. *Input*.

in_hSrcRec
Handle to source record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpRecCopy clears the destination record and copies the data field structure from the source record to the destination record. No data field values are copied.

See Also

QmpRecClear, **QmpRecCreate**

Example

```
QmpDeclHnd( hDestRec );  
QmpDeclHnd( hSrcRec );  
  
hDestRec = QmpRecCreate( &qres );  
hSrcRec = QmpRecCreate( &qres );  
  
QmpRecAddByType( hSrcRec, "Fname", QMS_VARIANT_STRING, 20, 0,  
    &qres );  
QmpRecAddByType( hSrcRec, "Lname", QMS_VARIANT_STRING, 15, 0,  
    &qres );  
QmpRecAddByType( hSrcRec, "Age", QMS_VARIANT_LONG, 10, 0,  
    &qres );  
QmpRecAddByType( hSrcRec, "Height", QMS_VARIANT_FLOAT, 10, 3,  
    &qres );  
QmpRecCopy( hDestRec, hSrcRec, &qres );
```

QmpRecCreate

CREATES A RECORD.

Syntax

```
MpHnd QmpRecCreate( QRESULT* out_pResult );  
out_pResult  
    Result code. Output.
```

Return Value

Returns the handle of a new record if successful, or `NULL` if there is an error.

Notes

If the creation function fails, an error code is returned in the `out_pResult` parameter. Application programs should always test the result code for success.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

See Also

`QmpRecCopy`, `QmpRecDestroy`

Example

```
hSourceRecord = QmpRecCreate ( pResult );
```

QmpRecDestroy

DESTROYS A RECORD.

Syntax

```
void QmpRecDestroy ( MpHnd in_hRec, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function destroys the specified record.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

See Also

`QmpRecCreate`, `QmpRecClear`, `QmpRecCopy`

Example

```
QmpRecDestroy(hSourceRec, &qres);
```

QmpRecGetDataPictureByHnd

QUERIES DATA FORMAT BY HANDLE.

Syntax

```
const char* QmpRecGetDataPictureByHnd( MpHnd in_hRec, int  
                                      in_iFld, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the data format if successful, else NULL.

Notes

This function queries the data format by handle. At this time, it only applies to fields of type QMS_VARIANT_DATE.

Example

See example on [page 684](#).

QmpRecGetDataPictureByName

QUERIES DATA FORMAT BY NAME.

Syntax

```
const char* QmpRecGetDataPictureByName( MpHnd in_hRec, const  
                                      char* in_szFldName, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_szFldName
Field name. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the data format if successful, else NULL.

Notes

This function queries the data format by name. At this time, it only applies to fields of type QMS_VARIANT_DATE.

Example

See example on [page 684](#).

QmpRecGetFldByHnd

GET THE VALUE OF A FIELD BY PASSING IN THE INDEX OF THE FIELD.

Syntax

```
MpHnd QmpRecGetFldByHnd ( MpHnd in_hRec, int in_iFld, QRESULT*  
    out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns a handle to the variant containing the value if successful, or NULL if there is an error.

Notes

Given the index to a field, this function returns a handle to a variant containing the field's value.

See Also

[QmpRecGetFld](#), [QmpRecGetFldByName](#)

Example

```
/* Translate the intermediate record into the prototype */  
/* record */  
  
pVariant = QmpRecGetFldByHnd( himRecord, iImAddrLine1, pResult  
    );  
QmpRecSetFldByHnd( io_hRec, iAddr, pVariant, pResult );  
  
pVariant = QmpRecGetFldByHnd( himRecord, iImCity, pResult );  
QmpRecSetFldByHnd( io_hRec, iCity, pVariant, pResult );  
  
pVariant = QmpRecGetFldByHnd( himRecord, iImFirstName, pResult  
    );  
QmpRecSetFldByHnd( io_hRec, iFirstName, pVariant, pResult );
```

QmpRecGetFldByName

GETS THE VALUE OF A FIELD BY PASSING IN THE NAME OF THE FIELD.

Syntax

```
MpHnd QmpRecGetFldByName ( MpHnd in_hRec, const char*  
    in_szFldName, QRESULT* out_pResult );
```

in_hRec

Handle to record. *Input*.

in_szFldName

Field name. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns a handle to the variant containing the value if successful, or NULL if there is an error.

Notes

Given the name of a field, this function returns a handle to a variant containing the field's value.

See Also

[QmpRecGetFld](#), [QmpRecGetFldByHnd](#)

Example

```
hFieldValueVariant = QmpRecGetFldByName (hRec, "Lastname",  
    pResult );
```

QmpRecGetFldCnt

GETS THE CURRENT FIELD COUNT FOR THE SPECIFIED RECORD.

Syntax

```
long QmpRecGetFldCnt( MpHnd in_hRec, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns number of fields in specified record if successful, or -1 if there is an error.

Notes

This function gets the current field count for the specified record.

See Also

QmpRecAdd, **QmpRecAddByType**, **QmpRecAddWithWidth**,
QmpRecGetFld, **QmpRecCopy**.

Example

```
hProtoRec    = QmpRecCreate( &qres );
QmpTblOpen( hTblInput, &qres );
QmpTblMoveFirst( hTblInput, &qres );
hInputRec = QmpRecCreate( &qres );
QmpTblGetFlds( hTblInput, hInputRec, &qres );
iNumFields = QmpRecGetFldCnt( hInputRec, &qres );
```

QmpRecGetFldDecimalsByHnd

RETURNS THE FIELD DECIMALS WHEN GIVEN THE FIELD INDEX.

Syntax

```
int QmpRecGetFldDecimalsByHnd ( MpHnd in_hRec, int in_iFld,
                                QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_iFld
    Field handle. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the number of digits to the right of the decimal point if successful, or -1 if there is an error.

Notes

This function returns the number of decimal places allowed for the field represented by the passed-in field index.

See Also

[QmpRecAdd](#)
[QmpRecAddByType](#)
[QmpRecAddWithWidth](#)
[QmpRecGetFldDecimalsByName](#)
[QmpRecGetFldTypeByHand](#)
[QmpRecGetFldTypeByName](#)
[QmpRecGetFldWidthByHnd](#)
[QmpRecGetFldWidthByName](#)

Example

```
iDecimalCount = QmpRecGetFldTypeByHnd (hInputRec, 4, &qres);
```

QmpRecGetFldDecimalsByName

RETURNS FIELD DECIMALS WHEN GIVEN THE FIELD NAME.

Syntax

```
int QmpRecGetFldDecimalsByName ( MpHnd in_hRec, const char*
                                in_szFldName, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Field name. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the number of digits to the right of the decimal point if successful, or -1 if there is an error.

Notes

This function returns the number of decimal places allowed for the field represented by the passed-in field name.

See Also

- [QmpRecAdd](#)
- [QmpRecAddByType](#)
- [QmpRecAddWithWidth](#)
- [QmpRecGetFldDecimalsByHnd](#)
- [QmpRecGetFldTypeByHnd](#)
- [QmpRecGetFldTypeByName](#)
- [QmpRecGetFldWidthByHnd](#)
- [QmpRecGetFldWidthByName](#)

Example

```
iDecimalCount = QmpRecGetFldDecimalsByName (hInputRec,
                                             "Zipcode", &qres);
```

QmpRecGetFldHnd

GETS THE HANDLE OF A FIELD BY PASSING IN THE NAME OF THE FIELD.

Syntax

```
int QmpRecGetFldHnd ( MpHnd in_hRec, const char* in_szFldName,  
                      QRESULT* out_pResult );
```

in_hRec

Handle to record. *Input*.

in_szFldName

Name of field. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the index of the field if successful, or -1 if there is an error.

Notes

Internally, data fields for a record are stored in a 0-based array. This function returns the index of a particular field in that array.

Example

See example on [page 684](#).

QmpRecGetFldName

GETS THE NAME OF A FIELD.

Syntax

```
const char* QmpRecGetFldName ( MpHnd in_hRec, int in_iFld,
                           QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_iFld
    Field index. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the name of a field if successful, or `NULL` if there is an error.

Notes

Given the index for a field, this function returns the name of the field.

See Also

[QmpRecAdd](#)
[QmpRecAddByType](#)
[QmpRecAddWithWidth](#)
[QmpRecGetFldHnd](#)
[QmpRecGetFldNameVB](#)

Example

```
for ( i = 0; i < iNumFields; i++ ) {
    const char *szFieldname;
    int iFieldWidth;
    int iHandle;
    szFieldname = QmpRecGetFldName( hInputRec, i, &qres );
    iFieldWidth = QmpRecGetFldWidthByHnd( hInputRec, i, &qres );
    iHandle = QmpRecAddWithWidth( hProtoRec, (char
        *)szFieldname, iFieldWidth, &qres );
}
```

QmpRecGetFldNameVB

GETS THE NAME OF A FIELD (VISUAL BASIC VERSION).

Syntax

```
void QmpRecGetFldNameVB ( MpHnd in_hRec, int in_iFld, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

io_szBuffer
Buffer to put field name into. *Input, Output*.

in_lSize
Size of buffer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Given the index for a field, this function places the name of the field into a string provided by the caller.

The application must pre-allocate the string to hold the field name.

See Also

[QmpRecAdd](#), [QmpRecAddByType](#), [QmpRecAddWithWidth](#),
[QmpRecGetFldName](#)

Example

```
QmpRecGetFldNameVB (hRec, 3, szBuffer, lBufferSize, pResult );
```

QmpRecGetFldTrimByHnd

GETS THE FIELD TRIMMING PROPERTY BY PASSING IN THE HANDLE OF THE FIELD.

Syntax

```
QMS_FIELD_TRIM QmpRecGetFldTrimByHnd ( MpHnd in_hRec, int
                                         in_iFld, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
The handle of record's field. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns trim property of a field. The valid return values are:

QMS_FIELD_TRIM_NONE
QMS_FIELD_TRIM_LEFT
QMS_FIELD_TRIM_RIGHT
QMS_FIELD_TRIM_BOTH

Notes

Field trimming (the trimming of blank characters or extraneous zeros) is meant to be applied to fields being input from a data source, or output to a data destination.

See Also

[QmpRecGetFldTrimByName](#).

Example

```
QMS_FIELD_TRIM FieldTrim1, FieldTrim2;
int iAddressHdl;

/* test setting/getting field trimming property */
FieldTrim1 = QmpRecGetFldTrimByHnd( hRec, iAddressHdl, &qres
);
FieldTrim2 = QmpRecGetFldTrimByName( hRec, "Address", &qres );
if ( FieldTrim1 != FieldTrim2 ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL,
                          "field trim properties do not match" );
}
```

QmpRecGetFldTrimByName

GETS THE FIELD TRIMMING PROPERTY BY PASSING IN THE NAME OF THE FIELD.

Syntax

```
QMS_FIELD_TRIM QmpRecGetFldTrimByName ( MpHnd in_hRec, const
                                         char* in_szFldName, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Name of field. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns trim property of a field. The valid return values are:

QMS_FIELD_TRIM_NONE
QMS_FIELD_TRIM_LEFT
QMS_FIELD_TRIM_RIGHT
QMS_FIELD_TRIM_BOTH

Notes

Field trimming (the trimming of blank characters or extraneous zeros) is meant to be applied to fields being input from a data source, or output to a data destination.

See Also

[QmpRecGetFldTrimByHnd](#).

Example

```
QMS_FIELD_TRIM FieldTrim1, FieldTrim2;
int iAddressHdl;

/* test setting/getting field trimming property */
FieldTrim1 = QmpRecGetFldTrimByHnd( hRec, iAddressHdl, &qres );
FieldTrim2 = QmpRecGetFldTrimByName( hRec, "Address", &qres
);
if ( FieldTrim1 != FieldTrim2 ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL,
        "field trim properties do not match" );
}
```

QmpRecGetFldTypeByHnd

QUERY FIELD TYPE BY INDEX.

Syntax

```
QMS_VARIANT_TYPE QmpRecGetFldTypeByHnd ( MpHnd in_hRec, int
                                         in_iFld, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the field variant type if successful. QMS_VARIANT_TYPE stores the types of values a CQmsVariant object can contain. The possible return values are:

QMS_VARIANT_EMPTY	An uninitialized CQmsVariant
QMS_VARIANT_BOOLEAN	
QMS_VARIANT_CHAR	
QMS_VARIANT_DOUBLE	
QMS_VARIANT_FLOAT	
QMS_VARIANT_INT	
QMS_VARIANT_LONG	
QMS_VARIANT_STRING	
QMS_VARIANT_ULONG	
QMS_VARIANT_USHORT	
QMS_VARIANT_VOIDSTAR	Generic pointer to void (structures, etc.)

If there is an error, QMS_VARIANT_EMPTY is returned.

Notes

QmpRecGetFldTypeByHnd returns the type of a particular field in a record, when given the field index.

See Also

QmpRecAdd, **QmpRecAddByType**, **QmpRecAddWithWidth**,
QmpRecGetFldTypeByName

Example

```
variant_type = QmpRecGetFldTypeByHnd (hInputRec, 3, &qres);
```

QmpRecGetFldTypeByName

QUERY FIELD TYPE BY NAME.

Syntax

```
QMS_VARIANT_TYPE QmpRecGetFldTypeByName ( MpHnd in_hRec,
                                         const char* in_szFldName, QRESULT* out_pResult );
```

in_hRec

Handle to record. *Input*.

in_szFldName

Field name. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns the field variant type if successful. QMS_VARIANT_TYPE stores the types of values a CQmsVariant object can contain. The possible return values are:

QMS_VARIANT_EMPTY	An uninitialized CQmsVariant
-------------------	------------------------------

QMS_VARIANT_BOOLEAN	
---------------------	--

QMS_VARIANT_CHAR	
------------------	--

QMS_VARIANT_DOUBLE	
--------------------	--

QMS_VARIANT_FLOAT	
-------------------	--

QMS_VARIANT_INT	
-----------------	--

QMS_VARIANT_LONG	
------------------	--

QMS_VARIANT_STRING	
--------------------	--

QMS_VARIANT ULONG	
-------------------	--

QMS_VARIANT USHORT	
--------------------	--

QMS_VARIANT_VOIDSTAR	Generic pointer to void (structures, etc.)
----------------------	--

If there is an error, QMS_VARIANT_EMPTY is returned.

Notes

QmpRecGetFldTypeByName returns the type of a particular field in a record, when given the field name.

See Also

QmpRecAdd, **QmpRecAddByType**, **QmpRecAddWithWidth**,
QmpRecGetFldTypeByHnd

Example

```
variant_type = QmpRecGetFldTypeByName (hInputRec,
                                         "firstname", &qres);
```

QmpRecGetFldWidthByHnd

GET THE FIELD WIDTH USING THE INDEX.

Syntax

```
int QmpRecGetFldWidthByHnd ( MpHnd in_hRec, int in_iFld,
                            QRESULT* out_pResult ) ;

in_hRec
    Handle to record. Input.
in_iFld
    Field index. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the field width if successful, or -1 if there is an error.

Notes

QmpRecGetFldWidthByHnd returns the width of a particular field in a record when given the field index.

See Also

QmpRecAdd, **QmpRecAddByType**, **QmpRecAddWithWidth**,
QmpRecGetFldWidthByName

Example

```
iFirstNameWidth = QmpRecGetFldWidthByHnd( hInputRec, 0, &qres
                                         );
iLastNameWidth = QmpRecGetFldWidthByHnd( hInputRec, 1, &qres
                                         );
iAddressWidth = QmpRecGetFldWidthByHnd( hInputRec, 2, &qres
                                         );
```

QmpRecGetFldWidthByName

QUERY FIELD WIDTH BY NAME.

Syntax

```
int QmpRecGetFldWidthByName ( MpHnd in_hRec, const char*
                             in_szFldName, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Field name. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns the field width if successful, or -1 if there is an error.

Notes

QmpRecGetFldWidthByName returns the width of a particular field in a record, when given the name of the field.

See Also

QmpRecAdd, **QmpRecAddByType**, **QmpRecAddWithWidth**,
QmpRecGetFldWidthByHnd

Example

```
iCityWidth = QmpRecGetFldWidthByName( hInputRec, "CITY",
                                      &qres );
iStateWidth = QmpRecGetFldWidthByName( hInputRec, "STATE",
                                       &qres );
iZipCodeWidth = QmpRecGetFldWidthByName( hInputRec,
                                         "ZIPCODE", &qres );
```

QmpRecIsValid

TESTS WHETHER A RECORD IS VALID.

Syntax

```
QBOOL QmpRecIsValid ( MpHnd in_hRec, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the record is valid, else QFALSE.

Notes

This function tests whether a record is valid.

Example

```
bIsRecValid = QmpRecIsValid (hRec, pResult);
```

QmpRecSetCharPtrFldByHnd

SETS A CHARACTER FIELD.

Syntax

```
void QmpRecSetCharPtrFldByHnd ( MpHnd in_hRec, int in_iFld,
                               char* in_szValue, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

in_szValue
Value of field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

`QmpRecSetCharPtrFldByHnd` sets the value of the field specified by the field index to a string value.

See Also

[QmpRecSetCharPtrFldByName](#)
[QmpRecSetFldByHnd](#)
[QmpRecSetFldByName](#)
[QmpRecSetFloatFldByHnd](#)
[QmpRecSetFloatFldByName](#)
[QmpRecSetLongFldByHnd](#)
[QmpRecSetLongFldByName](#)

Example

See example on [page 684](#).

QmpRecSetCharPtrFldByName

SETS A CHARACTER FIELD.

Syntax

```
void QmpRecSetCharPtrFldByName ( MpHnd in_hRec, char*
                                in_szFldName, char* in_szValue, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Name of field. Input.
in_szValue
    Value of field. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpRecSetCharPtrFldByName sets the value of the field specified by name to a string value.

See Also

- [QmpRecSetCharPtrFldByHnd](#)
- [QmpRecSetFldByHnd](#)
- [QmpRecSetFldByName](#)
- [QmpRecSetFloatFldByHnd](#)
- [QmpRecSetFloatFldByName](#)
- [QmpRecSetLongFldByHnd](#)
- [QmpRecSetLongFldByName](#)

Example

See example on [page 684](#).

QmpRecSetDataPictureByHnd

SETS THE PREFERRED DATA FORMAT FOR THE FIELD.

Syntax

```
void QmpRecSetDataPictureByHnd( MpHnd in_hRec, int in_iFld,
                               const char* in_szPicture, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

in_szPicture
Data picture. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the preferred data format for the field. At this time, it only applies to fields of type QMS_VARIANT_DATE.

Example

See example on [page 684](#).

QmpRecSetDataPictureByName

SETS THE PREFERRED DATA FORMAT FOR THE FIELD.

Syntax

```
void QmpRecSetDataPictureByName( MpHnd in_hRec, char*
    in_szFldName, const char* in_szPicture, QRESULT* out_pResult
);
```

in_hRec
Handle to record. *Input*.

in_szFldName
Field name. *Input*.

in_szPicture
Data picture. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the preferred data format for the field. At this time, it only applies to fields of type QMS_VARIANT_DATE.

Example

See example on [page 684](#).

QmpRecSetDateFldByHnd

SETS A DATE FIELD BY HANDLE.

Syntax

```
void QmpRecSetDateFldByHnd ( MpHnd in_hRec, int in_iFld,
                            QDATESTRUCT* in_Date, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

in_Date
Pointer to the QDATESTRUCT structure, containing the new field value.
Input.
The structure contains one member:

char ccyyymmdd[9]; Date plus one byte for null character.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function uses a structure to set the date field of a record. The field is identified using an index value.

Example

```
int iFldHnd, iFldName, iFldHeight, iFldDate, iFldAge;
QDATESTRUCT dateStruct;
const char* szPicture;
QmpDeclHnd( hRec );
hRec = QmpRecCreate( &qres );

/* Add some fields of various types. */
iFldName = QmpRecAdd( hRec, "FirstName", &qres );
iFldHeight = QmpRecAddByType(hRec, "Height", QMS_VARIANT_DOUBLE, 10, 3, &qres );
iFldDate = QmpRecAddByTypePicture(hRec, "Date", QMS_VARIANT_DATE, 8, "mm-
    dd-yy", &qres );
iFldAge = QmpRecAddByType(hRec, "Age", QMS_VARIANT_LONG, 10, 0, &qres );

/* Get field handle */
iFldHnd = QmpRecGetFldHnd( hRec, "Date", &qres );
if (iFldHnd != iFldDate)
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "field handle is incorrect" );

/* Set some field values */
QmpRecSetCharPtrFldByHnd ( hRec, iFldName, "John", &qres );
QmpRecSetCharPtrFldByName( hRec, "FirstName", "David", &qres );
QmpRecSetDoubleFldByHnd ( hRec, iFldHeight, 6.1, &qres );
QmpRecSetDoubleFldByName ( hRec, "Height", 6.2, &qres );
QmpRecSetLongFldByHnd   ( hRec, iFldAge, 30, &qres );
```

```
QmpRecSetLongFldByName ( hRec, "Age", 31, &qres );
strcpy( dateStruct.ccyyymmdd, "19990101");
QmpRecSetDateFldByHnd ( hRec, iFldDate, &dateStruct, &qres );
strcpy( dateStruct.ccyyymmdd, "19990202");
QmpRecSetDateFldByName ( hRec, "Date", &dateStruct, &qres );

/* Set/Get data picture */
QmpRecSetDataPictureByHnd ( hRec, iFldDate, "mm/dd/ccyy", &qres );
QmpRecSetDataPictureByName( hRec, "Date", "mmm dd, ccyy", &qres );
szPicture = QmpRecGetDataPictureByHnd ( hRec, iFldDate, &qres );
szPicture = QmpRecGetDataPictureByName( hRec, "Date", &qres );
if (strcmp( szPicture, "mmm dd, ccyy" ) != 0 )
    QmpUtilChkResultCode( QRESULT_SEVERE_FAIL, "field data picture is
incorrect" );

/* clear the data from the record, but not the field definitions. */
QmpRecClearData( hRec, &qres );
QmpRecDestroy( hRec, &qres );
```

QmpRecSetDateFldByName

SETS A DATE FIELD BY NAME.

Syntax

```
void QmpRecSetDateFldByName ( MpHnd in_hRec, const char*
    in_szFldName, QDATESTRUCT* in_pDate, QRESULT* out_pResult
);
```

in_hRec

Handle to record. *Input*.

in_szFldName

Name of field. *Input*.

in_pDate

Pointer to the QDATESTRUCT structure, containing the new field value.

Input.

The structure contains one member:

```
char ccyyymmdd[ 9 ]; Date plus one byte for null character.
```

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function uses a structure to set the date field of a record. The field is identified by name.

Example

See example on [page 684](#).

QmpRecSetDoubleFldByHnd

SETS A DOUBLE FIELD BY HANDLE.

Syntax

```
void QmpRecSetDoubleFldByHnd ( MpHnd in_hRec, int in_iFld,  
                               double in_dVal, QRESULT* out_pResult );
```

in_hRec

Handle to record. *Input*.

in_iFld

Field index. *Input*.

in_dVal

Double value of field. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

This function sets the value of the field specified by the field index to a double value.

Example

See example on [page 684](#).

QmpRecSetDoubleFldByName

SETS A DOUBLE FIELD BY NAME.

Syntax

```
void QmpRecSetDoubleFldByName ( MpHnd in_hRec, const char*
                                in_szFldName, double in_dVal, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_szFldName
Field name. *Input*.

in_dVal
Double value of field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the value of the field specified by the field name to a double value.

Example

See example on [page 684](#).

QmpRecSetFldByHnd

SETS THE VALUE OF A FIELD FROM A VARIANT.

Syntax

```
void QmpRecSetFldByHnd ( MpHnd in_hRec, int in_iFld, MpHnd
                         in_hVar, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_iFld
    Field index. Input.
in_hVar
    Handle of variant containing field value. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the value of the field specified by the field index to the value contained by the variant represented by the specified variant handle.

See Also

[QmpRecSetFldByName](#)
[QmpRecSetCharPtrFldByHnd](#)
[QmpRecSetCharPtrFldByName](#)
[QmpRecSetFloatFldByHnd](#)
[QmpRecSetFloatFldByName](#)
[QmpRecSetLongFldByHnd](#)
[QmpRecSetLongFldByName](#)

Example

```
/* Translate the intermediate record into the prototype */
/* record */

pVariant = QmpRecGetFldByHnd( himRecord, iImAddrLine1, pResult
                           );
QmpRecSetFldByHnd( io_hRec, iAddr, pVariant, pResult );

pVariant = QmpRecGetFldByHnd( himRecord, iImCity, pResult );
QmpRecSetFldByHnd( io_hRec, iCity, pVariant, pResult );

pVariant = QmpRecGetFldByHnd( himRecord, iImFirstName, pResult
                           );
QmpRecSetFldByHnd( io_hRec, iFirstName, pVariant, pResult );
```

QmpRecSetFldByName

SETS THE VALUE OF A FIELD FROM A VARIANT.

Syntax

```
void QmpRecSetFldByName ( MpHnd in_hRec, const char*
    in_szFldName, MpHnd in_hVar, QRESULT* out_pResult );
in_hRec
    Handle to record. Input.
in_szFldName
    Field name. Input.
in_hVar
    Handle of variant containing the field value. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the value of the field specified by the field name to the value contained by the variant represented by the specified variant handle.

See Also

[QmpRecSetFldByHnd](#)
[QmpRecSetCharPtrFldByHnd](#)
[QmpRecSetCharPtrFldByName](#)
[QmpRecSetFloatFldByHnd](#)
[QmpRecSetFloatFldByName](#)
[QmpRecSetLongFldByHnd](#)
[QmpRecSetLongFldByName](#)

Example

```
QmpRecSetFldByName (hRec, "Sales", hSalesFieldVariant, pResult
);
```

QmpRecSetFldTrimByHnd

SETS THE VALUE OF THE FIELD TRIMMING BY PASSING IN THE HANDLE OF THE FIELD

Syntax

```
void QmpRecSetFldTrimByHnd ( MpHnd in_hRec, int in_iFld,
                           QMS_FIELD_TRIM in_qTrim, QRESULT* out_pResult );
```

in_hRec

Handle to record. *Input*.

in_iFld

Handle of field to apply trimming property to. *Input*.

in_qTrim

Trim value. `QMS_FIELD_TRIM_BOTH` is the default value. *Input*.

Valid enums are:

`QMS_FIELD_TRIM_NONE`

`QMS_FIELD_TRIM_LEFT`

`QMS_FIELD_TRIM_RIGHT`

`QMS_FIELD_TRIM_BOTH`

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Field trimming (the trimming of blank characters or extraneous zeros) is meant to be applied to fields being input from a data source, or output to a data destination.

See Also

[QmpRecSetFldTrimByName](#).

Example

```
QMS_FIELD_TRIM FieldTrim;
int iAddressHdl;

/* test setting/getting field trimming property */
QmpRecSetFldTrimByHnd( hRec, iAddressHdl,
                       QMS_FIELD_TRIM_LEFT, &qres );
FieldTrim = QmpRecGetFldTrimByHnd( hRec, iAddressHdl, &qres );
if ( FieldTrim != QMS_FIELD_TRIM_LEFT ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "field trim
properties do not match" );
}
```

QmpRecSetFldTrimByName

SETS THE FIELD TRIMMING PROPERTY BY PASSING IN THE NAME OF THE FIELD.

Syntax

```
void QmpRecSetFldTrimByName ( MpHnd in_hRec, const char*
    in_szFldName, QMS_FIELD_TRIM in_qTrim, QRESULT* out_pResult
) ;
```

in_hRec
Handle to record. *Input*.

in_szFldName
Name of field to apply trimming property to. *Input*.

in_qTrim
Trim value. QMS_FIELD_TRIM_BOTH is the default value. *Input*.
Valid enums are:

QMS_FIELD_TRIM_NONE
QMS_FIELD_TRIM_LEFT
QMS_FIELD_TRIM_RIGHT
QMS_FIELD_TRIM_BOTH

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Field trimming (the trimming of blank characters or extraneous zeros) is meant to be applied to fields being input from a data source, or output to a data destination.

See Also

[QmpRecSetFldTrimByHnd](#).

Example

```
QMS_FIELD_TRIM FieldTrim;

/* test setting/getting field trimming property */
QmpRecSetFldTrimByName( hRec, "Address", QMS_FIELD_TRIM_LEFT,
    &qres );
FieldTrim = QmpRecGetFldTrimByName( hRec, "Address", &qres );
if ( FieldTrim != QMS_FIELD_TRIM_LEFT ) {
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL,
        "field trim properties do not match" );
}
```

QmpRecSetFloatFldByHnd

SETS THE VALUE OF A FLOAT FIELD.

Syntax

```
void QmpRecSetFloatFldByHnd ( MpHnd in_hRec, int in_iFld,
                             float in_fVal, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

in_fVal
Float value of field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the value of the field specified by the field index to a float value.

See Also

[QmpRecSetFloatFldByName](#)
[QmpRecSetFldByHnd](#)
[QmpRecSetFldByName](#)
[QmpRecSetCharPtrFldByHnd](#)
[QmpRecSetCharPtrFldByName](#)
[QmpRecSetLongFldByHnd](#)
[QmpRecSetLongFldByName](#)

Example

```
QmpRecSetFloatFldByHnd ( io_hRec, iFloatField, fFieldValue,
                         pResult );
```

QmpRecSetFloatFldByName

SETS THE VALUE OF A FLOAT FIELD.

Syntax

```
void QmpRecSetFloatFldByName ( MpHnd in_hRec, const char*
    in_szFldName, float in_fVal, QRESULT* out_pResult );
in_hRec
Handle to record. Input.
in_szFldName
Field name. Input.
in_fVal
Float value of field. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

This function sets the value of the field specified by the field name to a float value.

See Also

[QmpRecSetFloatFldByHnd](#)
[QmpRecSetCharPtrFldByHnd](#)
[QmpRecSetCharPtrFldByName](#)
[QmpRecSetLongFldByHnd](#)
[QmpRecSetLongFldByName](#)
[QmpRecSetFldByName](#)
[QmpRecSetFldByHnd](#)

Example

```
QmpRecSetFloatFldByName ( io_hRec, "Sales", fSalesFieldValue,
    pResult );
```

QmpRecSetLongFldByHnd

SETS THE VALUE OF A LONG FIELD.

Syntax

```
void QmpRecSetLongFldByHnd ( MpHnd in_hRec, int in_iFld, long
                            in_lVal, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

in_lVal
Long value of field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the value of the field specified by the field index to a long value.

See Also

[QmpRecSetLongFldByName](#)
[QmpRecSetCharPtrFldByHnd](#)
[QmpRecSetCharPtrFldByName](#)
[QmpRecSetFloatFldByHnd](#)
[QmpRecSetFloatFldByName](#)
[QmpRecSetFldByName](#)
[QmpRecSetFldByHnd](#)

Example

See example on [page 684](#).

QmpRecSetLongFldByName

SETS THE VALUE OF A LONG FIELD.

Syntax

```
void QmpRecSetLongFldByName ( MpHnd in_hRec, char*  
    in_szFldName, long in_lVal, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_szFldName
Field name. *Input*.

in_lVal
Long value of field. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the value of a field specified by the field name to a long value.

See Also

[QmpRecSetLongFldByHnd](#)
[QmpRecSetCharPtrFldByHnd](#)
[QmpRecSetCharPtrFldByName](#)
[QmpRecSetFloatFldByHnd](#)
[QmpRecSetFloatFldByName](#)
[QmpRecSetFldByName](#)
[QmpRecSetFldByHnd](#)

Example

See example on [page 684](#).

QmpRecValidFld

TESTS WHETHER A FIELD NAME REPRESENTS A VALID FIELD.

Syntax

```
QBOOL QmpRecValidFld ( MpHnd in_hRec, const char*
    in_szFldName, QRESULT* out_pResult );
```

in_hRec
Handle to record. *Input*.

in_szFldName
Field name. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if field name is valid, otherwise QFALSE.

Notes

This function determines whether a specified field name matches any of the field names in the record. The match test is not case sensitive.

See Also

[QmpRecValidHnd](#)

Example

```
bIsFieldNameValid = QmpRecValidFld ( hRec, szFldName, pResult
    );
```

QmpRecValidHnd

TESTS WHETHER A FIELD INDEX REPRESENTS A VALID FIELD.

Syntax

```
QBOOL QmpRecValidHnd ( MpHnd in_hRec, int in_iFld, QRESULT*  
    out_pResult );
```

in_hRec
Handle to record. *Input*.

in_iFld
Field index. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if the field handle is valid, otherwise QFALSE.

Notes

This function determines whether a specified field index matches a valid field in the record.

See Also

[QmpRecValidFld](#)

Example

```
bIsFldHndValid = QmpRecValidHnd (in_hRec, iFldHnd, pResult );
```

Function Class: QmpRecMat*

RECORD MATCHING PHASE FUNCTIONS.

Notes about the Dynamic Sliding Window

The dynamic sliding window may increase the number of matches, but at the cost of slower job execution. If the application defines a poor index key definition (one with extremely low cardinality), job execution performance using the dynamic sliding window will be especially poor.

For example, index key definitions based entirely on state or ZIP code information will result in hundreds or thousands of records with the same index key value, causing the record matcher a lot of extra work and wasting execution time.

Evaluating Field Match Criteria Match Scores

If more than one field matching criterion is created and added to the record matcher, the final record match score may be constructed in the following ways:

- Use the average of the criteria match scores
- Use the weighted average of the criteria match scores
- Use the maximum criterion match score
- Use the minimum criterion match score
- Return a match score of 100% if all criteria results are greater than or equal to a threshold, otherwise return a score of 0%
- Return a match score of 100% if any criterion result is greater than or equal to a threshold, otherwise return a score of 0%

When using the weighted average to combine criteria results, you must also specify a weight for each field matching criterion. The total of the weights of the criteria being used must be 100. For example, imagine that two field matching criteria are defined (last name and ZIP code). If you want the last name criterion result to be four times more important than the ZIP code criterion result, assign a weight of 80 to the last name criterion and 20 to the ZIP code criterion.

Notes about Multi-level Matching

Multi-level matching in Centrus Merge/Purge is optional. If you don't wish to use it, all criteria are considered to be on the same level. Also, the record matcher rule and threshold value apply to all field match criteria. To implement basic (uni-level) record matching:

- Attach criteria to the record matcher with `QmpRecMatAddCrit`.

- Set the rule for combining all criteria match scores with [QmpRecMatSetRule](#).
- Set the record matcher threshold value with [QmpRecMatSetThreshold](#).

To implement multi-level record matching:

- Attach criteria by level to the record matcher with [QmpRecMatAddFullLvlCrit](#) or [QmpRecMatAddLvlCrit](#).
- Set the level rule for combining all criteria match scores when calling [QmpRecMatAddFullLvlCrit](#), or by calling [QmpRecMatSetLvlRule](#).
- Set a level's threshold value when calling [QmpRecMatAddFullLvlCrit](#), or by calling [QmpRecMatSetLvlThreshold](#).

Level 0 is the lowest criterion level. After the rule on level 0 is used, level 1 is consulted, then level 2, etc.

Priority of Field Comparison Operations

When two fields are compared, the record matcher performs these actions, in order:

1. Remove white space from strings (if this property is set).
2. Remove punctuation from strings (if this property is set).
3. Check if one or both fields are blank.
4. Perform string alignment (if this property is set).
5. Perform substringing (if appropriate).
6. Convert string text to lower case (if this property is set).
7. Match the fields using each of the algorithms attached to the field matching criterion.

Quick Reference

Function	Description	Page
QmpRecMatAddCrit	Adds a criterion to the record matcher.	703
QmpRecMatAddFullLvlCrit	Adds a criterion to the record matcher at a level. Also adds the rule for combining criteria scores and sets the level matching threshold.	705
QmpRecMatAddLvlCrit	Adds a criterion to the record matcher at a level.	707
QmpRecMatAddPreProcDataSrc	Adds a preprocessed data source to the record matcher.	708
QmpRecMatBuildFieldMaps	Builds field maps.	709
QmpRecMatClear	Clears a record matcher.	710

Function	Description	Page
QmpRecMatCompute	Invokes the record matcher compute method to compare two records.	712
QmpRecMatCreate	Creates a record matcher.	716
QmpRecMatFillWinSize	Gets the sliding window size.	717
QmpRecMatFillWinStats	Gets statistics on dynamic sliding window.	718
QmpRecMatGetCritAt	Gets a handle to a criterion at a particular index value in the record matcher collection of field match criteria.	719
QmpRecMatGetCritCnt	Gets number of criteria in the record matcher.	720
QmpRecMatGetCritWeight	Gets the weighting for a record matcher criterion	721
QmpRecMatGetGenSortPreProc	Retrieves “create sorted versions of the preprocessed data source” flag.	722
QmpRecMatGetIdxKeyCnt	Gets index key count.	723
QmpRecMatGetLvlCnt	Gets number of levels in the record matcher.	724
QmpRecMatGetLvlCritAt	Gets a handle to a criterion at a particular index in the record matcher collection of criteria (organized by level).	725
QmpRecMatGetLvlCritCnt	Gets number of criteria in the record matcher at a given level.	726
QmpRecMatGetLvlCritWeight	Gets the weighting for a criterion by level.	727
QmpRecMatGetLvlRule	Gets record matcher rule for combining the match scores of criteria at the same level.	728
QmpRecMatGetLvlThreshold	Gets the record matcher threshold for a given level.	729
QmpRecMatGetMatRes	Gets the match result object out of the record matcher.	730
QmpRecMatGetNthRecInterval	Gets record matcher Nth record event interval.	731
QmpRecMatGetRule	Gets criterion rule that record matcher will use.	732
QmpRecMatGetThreshold	Gets the threshold for the record matcher.	733
QmpRecMatIsValid	Tests whether a record matcher is valid.	734
QmpRecMatRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	735
QmpRecMatRemCrit	Removes specified criterion from record matcher.	736
QmpRecMatRemIdxKey	Removes a specified index key from the record matcher.	737
QmpRecMatRemLvl	Removes the specified criteria level from the record matcher.	738
QmpRecMatRemLvlCrit	Removes a specified criterion from the record matcher by level.	739

Function	Description	Page
QmpRecMatRemPreProcDataSrc	Removes a preprocessed data source from the record matcher.	740
QmpRecMatSetCritWeight	Sets the weighting for a criterion.	741
QmpRecMatSetGenSortPreProc	Builds sorted versions of the preprocessed data source.	742
QmpRecMatSetLvlCritWeight	Sets the weighting for a criterion at a level.	743
QmpRecMatSetLvlRule	Sets record matcher rule for combining the match scores of criteria at the same level.	744
QmpRecMatSetLvlThreshold	Sets the record matcher threshold for a given level.	746
QmpRecMatSetNthRecInterval	Sets record matcher Nth record event interval.	747
QmpRecMatSetRule	Sets criterion rule which record matcher will use.	748
QmpRecMatSetThreshold	Sets the threshold for the record matcher.	749
QmpRecMatSetWinSize	Sets the window size.	750
QmpRecMatUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	752
QmpRecMatUseDataLstSvc	Specifies the data list service object to be used by the record matcher.	753
QmpRecMatUseIdxKey	Specifies the index key object to be used by the record matcher.	754
QmpRecMatUseMatRes	Specifies the match result object to be used by the record matcher.	755

QmpRecMatAddCrit

ADDS A CRITERION TO THE RECORD MATCHER.

Syntax

```
void QmpRecMatAddCrit ( MpHnd in_hRecMat, MpHnd in_hCrit, int
in_iWeight, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hCrit
Handle to the criterion. *Input*.

in_iWeight
Weight to be assigned to criterion. Must be an integer between 0 and 100.
The default value is 0. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpRecMatAddCrit adds a field match criterion to the record matcher. A collection of field match criteria are used together by the record matcher to determine if two records match.

A field match criterion consists of the names of a pair of fields to be compared between records (such as a last name field) and the manner (algorithms) in which the fields are to be compared. The results from the individual field matches can be combined for the overall record match result using the following choices:

- Use the average of the results
- Use the weighted average of the results
- Use the maximum result
- Use the minimum result
- Use 100% if all results are greater than or equal to a threshold, otherwise 0%
- Use 100% if any result is greater than or equal to a threshold, otherwise 0%

When using the weighted average to combine the field results, you must also specify a weight for each field. The total of the weights must sum to 100.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons.

Example

See example on [page 712](#).

QmpRecMatAddFullLvlCrit

ADDS A CRITERION TO THE RECORD MATCHER AT A LEVEL. ALSO ADDS THE RULE FOR COMBINING CRITERIA SCORES AND SETS THE LEVEL MATCHING THRESHOLD.

Syntax

```
void QmpRecMatAddFullLvlCrit ( MpHnd in_hRecMat, MpHnd  

    in_hCrit, QMS_CRITERIA_RULE in_CritRule, int in_iThreshold,  

    int in_iLevel, int in_iWeight, QRESULT* out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*.

in_hCrit

Handle to the criterion to add. *Input*.

in_CritRule

Rule for combining criteria match scores at this level. *Input*.

Valid enums are:

QMS_CRITERIA_AVERAGE	Average of all criteria.
QMS_CRITERIA_WEIGHTED_AVERAGE	Weighted average of all criteria.
QMS_CRITERIA_MAX	Use the criterion that provided the highest score.
QMS_CRITERIA_MIN	Use the criterion that provided the lowest score.
QMS_CRITERIA_AND	All criteria must achieve a "threshold" score.
QMS_CRITERIA_OR	At least one criterion must achieve a "threshold" score.

in_iThreshold

Record matcher threshold for this level. *Input*.

in_iLevel

Level of criterion. *Input*.

in_iWeight

Weight of criterion (if using the weighted average of criteria). *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpRecMatAddFullLvlCrit adds a criterion to the record matcher at a level. It also specifies the level record matching threshold, the rule for combining criteria scores at this level, and the weight of the criterion (if combining criteria scores using the weighted average).

See Also

`QmpRecMatAddLvlCrit`, `QmpRecMatSetLvlThreshold`.

Example

```
/* add a level criteria using the full level criteria add */
QmpRecMatAddFullLvlCrit( hRecMat, hZipMatchCriterion,
    QMS_CRITERIA_AVERAGE, 50, 0, 2, &qres );
```

QmpRecMatAddLvlCrit

ADDS A CRITERION TO THE RECORD MATCHER AT A LEVEL.

Syntax

```
void QmpRecMatAddLvlCrit( MpHnd in_hRecMat, MpHnd in_hCrit,
                           int in_iWeight, int in_iLevel, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_hCrit
    Handle to the criterion to add. Input.
in_iWeight
    Weight of the criterion. Input.
in_iLevel
    Level of the criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

See Also

[QmpRecMatAddFullLvlCrit](#).

Example

```
QmpRecMatAddLvlCrit( hRecMat, hAddressMatchCriterion, 90, 2,
                        &qres );
```

QmpRecMatAddPreProcDataSrc

ADDS A PREPROCESSED DATA SOURCE TO THE RECORD MATCHER.

Syntax

```
void QmpRecMatAddPreProcDataSrc ( MpHnd in_hRecMat, MpHnd
                                  in_hDataSrc, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hDataSrc
Handle to the data source. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Allows an application to add a preprocessed data source to the record matcher. The record matcher currently can accept only one preprocessed data source. If you add a data source, you must remove it before adding another.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons.

See Also

[QmpRecMatRemPreProcDataSrc](#)

Refer to “[Preprocessed Data Source](#)” on page 37 for a discussion of what constitutes a preprocessed data source.

Example

See example on [page 712](#).

QmpRecMatBuildFieldMaps

BUILDS FIELD MAPS.

Syntax

```
void QmpRecMatBuildFieldMaps ( MpHnd in_hRecMat, QRESULT*
                               out_pResult );
in_hRecMat
    Handle to record matcher. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

An application would call this function if it were bypassing the record matcher “start” function (**QmpPhaseStart**), and instead invoking **QmpRecMatCompute** directly. This is quite common in real time applications that do not actually run the entire record matcher phase.

Example

```
hRec1 = QmpRecCreate( &qres );
hRec2 = QmpRecCreate( &qres );
QmpRecCopy( hRec1, hProtoRec, &qres );
QmpRecSetCharPtrFldByName( hRec1, "name", "John",      &qres );
QmpRecSetCharPtrFldByName( hRec1, "addr", "123 Main", &qres );
QmpRecSetCharPtrFldByName( hRec1, "zip",  "80027",     &qres );
QmpRecCopy( hRec2, hRec1, &qres );
QmpRecMatBuildFieldMaps( hRecMat, &qres );
iResult = QmpRecMatCompute( hRecMat, hRec1, hRec2, &qres );
if ( iResult < 100 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Compute failed"
);
```

QmpRecMatClear

CLEAR A RECORD MATCHER.

Syntax

```
void QmpRecMatClear ( MpHnd in_hRecMat, QRESULT* out_pResult
) ;
```

in_hRecMat
Handle to record matcher. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Clears the record matcher. This means setting it back to a default and empty state.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpDeclHnd( hRecMat );
QRESULT qres;
long lSize;
int iThreshold;
QMS_CRITERIA_RULE rule;

/* create record matcher */
hRecMat = QmpRecMatCreate( QMS_CRITERIA_AVERAGE, 90, QFALSE,
&qres );

/* Set/get miscellaneous properties */
QmpRecMatSetWinSize( hRecMat, 3, &qres );
lSize = QmpRecMatGetWinSize( hRecMat, &qres );
if ( lSize != 3 )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Window size is
wrong" );

QmpRecMatSetRule( hRecMat, QMS_CRITERIA_WEIGHTED_AVERAGE, &qres
);
rule = QmpRecMatGetRule( hRecMat, &qres );
if ( rule != QMS_CRITERIA_WEIGHTED_AVERAGE )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher rule
is wrong" );

QmpRecMatSetThreshold( hRecMat, 90, &qres );
iThreshold = QmpRecMatGetThreshold( hRecMat, &qres );
if ( iThreshold != 90 )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher
threshold is wrong" );
```

```
/* Clear the record matcher */
QmpRecMatClear( hRecMat, &qres );
```

QmpRecMatCompute

INVOKES THE RECORD MATCHER COMPUTE METHOD TO COMPARE TWO RECORDS.

Syntax

```
int QmpRecMatCompute ( MpHnd in_hRecMat, MpHnd in_hRec1, MpHnd
                      in_hRec2, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hRec1
Handle to first record. *Input*.

in_hRec2
Handle to second record. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the match result value if successful, or -1 if there is an error. A result other than `QRESULT_INFO_YES` is not a match.

Notes

Invokes the record matcher compute method to compare two records. This function is normally called internally by the record matching logic. However, it is available for use by applications as well. It could be used in a real-time application that needed to match a “target” record against a selection of possible matches in an existing database.

Example

This example illustrates the use of many record matcher functions, including `QmpRecMatCompute`.

```
/* declare handles */
int iName, iAddr, iZip, iResult, iWeight, iThreshold;
long lCount, lCount2;
QMS_CRITERIA_RULE rule;
QMS_EVERY_NTHREC_FUNC pEveryNthRecordClient;
QmpDeclHnd( hRecMat );
QmpDeclHnd( hAddressMatchCriterion );
QmpDeclHnd( hZipMatchCriterion );
QmpDeclHnd( hMatchByString );
QmpDeclHnd( hProtoRec );
QmpDeclHnd( hRec1 );
QmpDeclHnd( hRec2 );
QmpDeclHnd( hPreproTable );
QmpDeclHnd( hPreproDataSrc );
QmpDeclHnd( hIndexKey1 );
QmpDeclHnd( hIndexKey2 );
QmpDeclHnd( hMatResTable );
QmpDeclHnd( hMatRes );
QmpDeclHnd( hDITR );
QmpDeclHnd( hDataListSvc );
QmpDeclHnd( hDataListDefault );
QMS_TABLE_INDEXES* IndxInfoArray = NULL;
```

```

long lDefWinSize, lMaxWinSize;

/* create record matcher */
hRecMat = QmpRecMatCreate( QMS_CRITERIA_AVERAGE, 90, &qres );

/* Set/get miscellaneous properties */
QmpRecMatSetWinSize( hRecMat, 3, 3, &qres );
QmpRecMatFillWinSize( hRecMat, &lDefWinSize, &lMaxWinSize, &qres );
if ( lDefWinSize != 3 || lMaxWinSize != 3 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Window size is wrong" );
QmpRecMatSetRule( hRecMat, QMS_CRITERIA_WEIGHTED_AVERAGE, &qres );
rule = QmpRecMatGetRule( hRecMat, &qres );
if ( rule != QMS_CRITERIA_WEIGHTED_AVERAGE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher rule is wrong"
);
QmpRecMatSetThreshold( hRecMat, 90, &qres );
iThreshold = QmpRecMatGetThreshold( hRecMat, &qres );
if ( iThreshold != 90 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher threshold is
wrong" );

/* Set up prototype record and a couple of compare records to test Compute */
hProtoRec = QmpRecCreate( &qres );
iName = QmpRecAdd( hProtoRec, "name", &qres );
iAddr = QmpRecAdd( hProtoRec, "addr", &qres );
iZip = QmpRecAdd( hProtoRec, "zip", &qres );

/* create and set the data list service */
hDataListSvc = QmpDataLstSvcCreate( &qres );
hDataListDefault = QmpDataLstDefCreate( "Default", &qres );
QmpDataLstSvcSetRecProto( hDataListSvc, hProtoRec, &qres );
QmpDataLstSvcAddDataLst( hDataListSvc, hDataListDefault, &qres );
QmpRecMatUseDataLstSvc( hRecMat, hDataListSvc, &qres );

/* Set up the criteria */
hAddressMatchCriterion = QmpCritCreate( QMS_ALGORITHMS_AVERAGE, 90, &qres );
hZipMatchCriterion = QmpCritCreate( QMS_ALGORITHMS_AVERAGE, 90, &qres );
hMatchByString = QmpAlgCreate( QMS_ALGORITHM_TYPE_STRING, &qres );
QmpCritAddAlg( hAddressMatchCriterion, hMatchByString, 0, &qres );
QmpCritAddAlg( hZipMatchCriterion, hMatchByString, 0, &qres );
QmpCritSetPairPartFld( hAddressMatchCriterion, hProtoRec, "addr", 1, "addr",
1, &qres );
QmpCritSetPairSubStrFld( hZipMatchCriterion, hProtoRec, "zip", 1, 5, "zip",
1, 5, &qres );

/* Add criteria */
QmpRecMatAddCrit( hRecMat, hAddressMatchCriterion, 90, &qres );
QmpRecMatAddCrit( hRecMat, hZipMatchCriterion, 90, &qres );
lCount = QmpRecMatGetCritCnt( hRecMat, &qres );
if ( lCount != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of criteria" );

/* Set weights of criteria. */
QmpRecMatSetCritWeight( hRecMat, hAddressMatchCriterion, 70, &qres );
iWeight = QmpRecMatGetCritWeight( hRecMat, hAddressMatchCriterion, &qres );
if ( iWeight != 70 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher criteria weight
is wrong" );
QmpRecMatSetCritWeight( hRecMat, hZipMatchCriterion, 30, &qres );
iWeight = QmpRecMatGetCritWeight( hRecMat, hZipMatchCriterion, &qres );
if ( iWeight != 30 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher criteria weight is
wrong" );

/* Test Compute */
hRec1 = QmpRecCreate( &qres );
hRec2 = QmpRecCreate( &qres );
QmpRecCopy( hRec1, hProtoRec, &qres );

```

QmpRecMatCompute

```
QmpRecSetCharPtrFldByName( hRec1, "name", "John", &qres );
QmpRecSetCharPtrFldByName( hRec1, "addr", "123 Main", &qres );
QmpRecSetCharPtrFldByName( hRec1, "zip", "80027", &qres );
QmpRecCopy( hRec2, hRec1, &qres );
QmpRecMatBuildFieldMaps( hRecMat, &qres );
iResult = QmpRecMatCompute( hRecMat, hRec1, hRec2, &qres );
if ( iResult < 100 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Compute failed" );

/* Index keys */
hIndexKey1 = QmpIndxKeyCreate( &qres );
hIndexKey2 = QmpIndxKeyCreate( &qres );
QmpIndxKeySetRecProto( hIndexKey1, hProtoRec, &qres );
QmpIndxKeySetRecProto( hIndexKey2, hProtoRec, &qres );
QmpIndxKeyAddCompByName( hIndexKey1, "addr", QMS_INDXKEY_XFORM_IDENTITY, 1,
    100, &qres );
QmpIndxKeyAddCompByName( hIndexKey1, "zip", QMS_INDXKEY_XFORM_IDENTITY, 1,
    100, &qres );
QmpRecMatUseIndxKey( hRecMat, hIndexKey1, &qres );
QmpRecMatUseIndxKey( hRecMat, hIndexKey2, &qres );
lCount = QmpRecMatGetIndxKeyCnt( hRecMat, &qres );
if ( lCount != 2 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of index keys" );

/* test validity of record matcher */
if ( QmpRecMatIsValid( hRecMat, &qres ) == QFALSE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher is invalid" );

/* remove some index keys */
QmpRecMatRemIndxKey( hRecMat, hIndexKey1, &qres );
QmpRecMatRemIndxKey( hRecMat, hIndexKey2, &qres );
lCount = QmpRecMatGetIndxKeyCnt( hRecMat, &qres );
if ( lCount != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of index keys" );

/* Preprocessed data source */
hPreproTable = QmpTblCbCreate( "", "prepro.dbf", &qres );
hPreproDataSrc = QmpDataSrcPreCreate( "prepro.dbf", 2, &qres );
QmpDataSrcUseTbl( hPreproDataSrc, hPreproTable, &qres );
QmpRecMatAddPreProcDataSrc( hRecMat, hPreproDataSrc, &qres );

/* Use the match result */
hMatResTable = QmpTblCbCreate( "", "matres.dbf", &qres );
hMatRes = QmpMatResCreate( hMatResTable, &qres );
if ( QmpTblExists( hMatResTable, &qres ) ) {
    QmpTblDestroyRep( hMatResTable, &qres );
}
QmpRecMatUseMatRes( hRecMat, hMatRes, &qres );

/* Use the DITR */
hDITR = QmpTblCbCreate( "", "ditr.dbf", &qres );
QmpPhaseUseDITR( hRecMat, hDITR, &qres );

/* Register for events */
pEveryNthRecordClient = &EventHandlerForEveryNthRecord;
QmpRecMatRegEveryNthRecFunc( hRecMat, pEveryNthRecordClient, &qres );
QmpRecMatSetNthRecInterval( hRecMat, 5000, &qres );

/* remove criteria */
QmpRecMatRemCrit( hRecMat, hAddressMatchCriterion, &qres );
QmpRecMatRemCrit( hRecMat, hZipMatchCriterion, &qres );
lCount = QmpRecMatGetCritCnt( hRecMat, &qres );
if ( lCount != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of criteria" );

/* remove data source */
QmpRecMatRemPreProcDataSrc( hRecMat, 2, &qres );
```

```

/* Clear the record matcher */
QmpRecMatClear( hRecMat, &qres );

/* Just before destruction, get information on the table index */
QmpTblSetAutoOpenIdx( hPreprotoTable, QTRUE, &qres );
QmpTblOpen( hPreprotoTable, &qres );
lCount = QmpTblGetIdxCnt( hPreprotoTable, &qres );
if ( lCount < 5 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of indexes" );
IdxInfoArray = (QMS_TABLE_INDEXES*)malloc( lCount * sizeof( QMS_TABLE_INDEXES
));
lCount2 =QmpTblFillIdxInfo( hPreprotoTable, lCount, IdxInfoArray, &qres );
QmpTblClose( hPreprotoTable, &qres );
if (QMS_STRICTCMP( IdxInfoArray[4].szIndexExpression, "zipcode" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Index info is incorrect" );
free( IdxInfoArray );

/* Destroy objects */
QmpPhaseDestroy( hRecMat, &qres );
QmpCritDestroy( hAddressMatchCriterion, &qres );
QmpCritDestroy( hZipMatchCriterion, &qres );
QmpDataSrcDestroy( hPreprotoDataSrc, &qres );
QmpTblDestroy( hPreprotoTable, &qres );
QmpAlgDestroy( hMatchByString, &qres );
QmpRecDestroy( hProtoRec, &qres );
QmpRecDestroy( hRec1, &qres );
QmpRecDestroy( hRec2, &qres );
QmpIdxKeyDestroy( hIndexKey1, &qres );
QmpIdxKeyDestroy( hIndexKey2, &qres );
QmpMatResDestroy( hMatRes, &qres );
QmpTblClose( hMatResTable, &qres );
QmpTblClose( hDITR, &qres );
QmpTblDestroy( hMatResTable, &qres );
QmpTblDestroy( hDITR, &qres );
QmpDataLstSvcDestroy( hDataListSvc, &qres );
QmpDataLstDestroy( hDataListDefault, &qres );

```

QmpRecMatCreate

CREATES A RECORD MATCHER.

Syntax

```
MpHnd QmpRecMatCreate ( QMS_CRITERIA_RULE in_CriteriaRule,
                           int in_iThreshold, QRESULT* out_pResult );
```

in_CriteriaRule

Criterion rule to use. *Input*.

Valid enums are:

QMS_CRITERIA_AVERAGE	Average of all criteria.
QMS_CRITERIA_WEIGHTED_AVERAGE	Weighted average of all criteria.
QMS_CRITERIA_MAX	Use the criterion that provided the highest score.
QMS_CRITERIA_MIN	Use the criterion that provided the lowest score.
QMS_CRITERIA_AND	All criteria must achieve a “threshold” score.
QMS_CRITERIA_OR	At least one criterion must achieve a “threshold” score.

in_iThreshold

Record matching threshold. Records being compared must score at least as high as this threshold to be considered duplicates. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to record matcher if successful, or NULL if there is an error.

Notes

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons.

QmpRecMatFillWinSize

GETS THE SLIDING WINDOW SIZE.

Syntax

```
void QmpRecMatFillWinSize( MpHnd in_hRecMat, long*
    out_plDefWinSize, long* out_plMaxWinSize, QRESULT*
    out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

out_plDefWinSize
Default window size. *Input*.

out_plMaxWinSize
Maximum window size. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function gets the size of the sliding window in the record matcher.

The sliding window of the record matcher can be set to a fixed record size, or it can be set to grow (as needed) up to a maximum record size. A variable-size sliding window is called a dynamic sliding window. The sliding window will need to expand if a large number of records with the same key value come through the system. This will allow all records with the same index key value to be compared in the same sliding window, rather than relying on transitive closure to accomplish this.

Note: The dynamic sliding window may increase the number of matches, but at the cost of slower job execution. If the application defines a poor index key definition (one with extremely low cardinality), job execution time will be especially poor. For example, index key definitions based entirely on state or ZIP code information will result in hundreds or thousands of records with the same index key value, causing the record matcher a lot of extra work and wasting execution time.

Example

```
/* Configure a dynamic sliding window */
QmpRecMatSetWinSize( hRecMat, 3, 6, &qres );
QmpRecMatFillWinSize( hRecMat, &lDefWinSize, &lMaxWinSize, &qres );
if ( lDefWinSize != 3 || lMaxWinSize != 6 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Window size is wrong" );
```

QmpRecMatFillWinStats

GETS STATISTICS ON DYNAMIC SLIDING WINDOW.

Syntax

```
void QmpRecMatFillWinStats( MpHnd in_hRecMat, long*
                           out_lTimesGrown, long* out_lTimesNotGrown, long*
                           out_lMaxSizeGrown, QRESULT* out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*.

out_lTimesGrown

Number of times sliding window grew. *Output*.

out_lTimesNotGrown

Number of times the sliding window did *not* grow. *Output*.

out_lMaxSizeGrown

Maximum size sliding window has grown. *Output*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpRecMatFillWinStats provides information on the behavior of the sliding window. It should be called after the record matching phase has been run.

Example

```
QmpRecMatFillWinStats( in_hRecMat, &lTimesGrown,
                         &lTimesNotGrown, &lMaxSizeGrown, &qres );
```

QmpRecMatGetCritAt

GETS A HANDLE TO A CRITERION AT A PARTICULAR INDEX VALUE IN THE RECORD MATCHER COLLECTION OF FIELD MATCH CRITERIA.

Syntax

```
MpHnd QmpRecMatGetCritAt ( MpHnd in_hRecMat, long in_lIndex,
                           QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_lIndex
    Index to criterion in criteria collection.
out_pResult
    Result code. Output.
```

Return Value

Returns a handle to a criterion if successful, or `NULL` if there is an error.

Notes

Field matching criteria exist in an ordered, indexed collection inside the record matcher, and may be accessed by their index value.

`QmpRecMatGetCritAt` returns the handle to a criterion, given the criterion's index value in the record matcher criteria collection.

Example

```
QmpDeclHnd( hCriterion );
QmpDeclHnd( hRecMat );
QRESULT qres;
hCriterion = QmpRecMatGetCritAt ( hRecMat, 3, &qres );
```

QmpRecMatGetCritCnt

GETS NUMBER OF CRITERIA IN THE RECORD MATCHER.

Syntax

```
long QmpRecMatGetCritCnt ( MpHnd in_hRecMat, QRESULT*  
    out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns number of criteria if successful, or -1 if there is an error.

Notes

A criterion is a rule for determining if two records match. Records may be matched using more than one criterion. Criteria are initially added to the record matcher with **QmpRecMatAddCrit**.

Example

See example on [page 712](#).

QmpRecMatGetCritWeight

GETS THE WEIGHTING FOR A RECORD MATCHER CRITERION

Syntax

```
int QmpRecMatGetCritWeight ( MpHnd in_hRecMat, MpHnd in_hCrit,  
                           QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hCrit
Handle to a record matcher criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the weight value for a record matcher criterion if successful, or -1 if there is an error.

Notes

The value of each weight must be between 0 and 100. If the record matcher rule is `QMS_CRITERIA_WEIGHTED_AVERAGE`, the summed weights of all criteria must equal 100. The default criterion weight value is 0.

Example

See example on [page 712](#).

QmpRecMatGetGenSortPreProc

RETRIEVES “CREATE SORTED VERSIONS OF THE PREPROCESSED DATA SOURCE” FLAG.

Syntax

```
QBOOL QmpRecMatGetGenSortPreProc ( MpHnd in_hRecMat, QRESULT*  
    out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*.

out_pResult

Result code. *Output*.

Return Value

QTRUE if the preprocessed data source will be sorted, else QFALSE.

Notes

A preprocessed data source must always contain one table with the records sorted in S1K1 order, as well as a separate table for each index key definition, with the records sorted in index key order. These other sorted tables contain a subset of the fields in the original table. This subset is limited to the fields that will be matched and indexed on, plus the standard augmented fields found in the DITR.

QmpRecMatGetGenSortPreProc gets the flag that determines if the proper sorted tables are built for the preprocessed data source attached to the record matcher.

See Also

QmpRecMatSetGenSortPreProc.

Example

```
QmpDeclHnd( hRecMat );  
QBOOL bGenSortPreProc;  
QRESULT qres;  
  
bGenSortPreProc = QmpRecMatGetGenSortPreProc( hRecMat, &qres  
);
```

QmpRecMatGetIdxKeyCnt

GETS INDEX KEY COUNT.

Syntax

```
long QmpRecMatGetIdxKeyCnt (MpHnd in_hRecMat, QRESULT*  
                           out_pResult );  
  
in_hRecMat  
          Handle to record matcher. Input.  
  
out_pResult  
          Result code. Output.
```

Return Value

Returns index key count if successful, or -1 if there is an error.

Notes

This function returns the number of index keys that were given to the application with `QmpRecMatUseIdxKey`.

Example

See example on [page 712](#).

QmpRecMatGetLvlCnt

GETS NUMBER OF LEVELS IN THE RECORD MATCHER.

Syntax

```
long QmpRecMatGetLvlCnt ( MpHnd in_hRecMat, QRESULT*
                           out_pResult );
in_hRecMat
Handle to record matcher. Input.
out_pResult
Result code. Output.
```

Return Value

If successful, returns the number of levels of criteria in the record matcher.

Notes

After calling `QmpRecMatGetLvlCnt`, call `QmpRecMatGetLvlCritCnt` to find the number of criteria at each level, and then call `QmpRecMatGetLvlCritAt` to retrieve each criterion at the specified level.

See Also

`QmpRecMatGetLvlCritCnt`, `QmpRecMatGetLvlCritAt`.

Example

```
long lLevels;
long lCount;
int iWeight;
int i, j;
QmpDeclHnd( hGetCrit );

lLevels = QmpRecMatGetLvlCnt( hRecMat, &qres );
for ( i = 0; i < lLevels; i++ ) {
    lCount = QmpRecMatGetLvlCritCnt( hRecMat, 2, &qres );
    for ( j = 0; j < lCount; j++ ) {
        hGetCrit = QmpRecMatGetLvlCritAt( hRecMat, i, j,
                                           &qres );
        iWeight = QmpRecMatGetLvlCritWeight( hRecMat,
                                              hGetCrit, i, &qres );
    }
}
```

QmpRecMatGetLvlCritAt

GETS A HANDLE TO A CRITERION AT A PARTICULAR INDEX IN THE RECORD MATCHER COLLECTION OF CRITERIA (ORGANIZED BY LEVEL).

Syntax

```
MpHnd QmpRecMatGetLvlCritAt ( MpHnd in_hRecMat,
    long in_lIndex, int in_iLevel, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_lIndex
    Index value of criterion in the record matcher's collection of criteria. There
    is a separate collection of criteria for each level. Input.
in_iLevel
    Level of criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

If successful, the handle to the field match criterion at the specified index value and level.

Notes

For every level of field match criteria, there is a separate collection of criteria kept by the record matcher. The criterion level determines which collection of criteria to search; the index value determines which criterion in the collection is correct.

When called with level of zero, this function acts the same as **QmpRecMatGetCritAt**.

See Also

QmpRecMatAddLvlCrit, **QmpRecMatSetLvlThreshold**.

Example

```
QmpDeclHnd( hGetCrit );
hGetCrit = QmpRecMatGetLvlCritAt( hRecMat, 0, 2, &qres );
```

QmpRecMatGetLvlCritCnt

GETS NUMBER OF CRITERIA IN THE RECORD MATCHER AT A GIVEN LEVEL.

Syntax

```
long QmpRecMatGetLvlCritCnt ( MpHnd in_hRecMat,
                             int in_iLevel, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_iLevel
    Criterion level you are interested in. Input.
out_pResult
    Result code. Output.
```

Return Value

If successful, returns the number of criteria in the record matcher at a given level.

Notes

Call **QmpRecMatGetLvlCritCnt** to determine how many criteria are defined at a given level. After receiving this information, you could call **QmpRecMatGetLvlCritAt** to get each criterion at that level.

See Also

QmpRecMatGetLvlCnt.

Example

```
long lLevels;
long lCount;
int iWeight;
int i, j;
QmpDeclHnd( hGetCrit );

lLevels = QmpRecMatGetLvlCnt( hRecMat, &qres );
for ( i = 0; i < lLevels; i++ ) {
    lCount = QmpRecMatGetLvlCritCnt( hRecMat, 2, &qres );
    for ( j = 0; j < lCount; j++ ) {
        hGetCrit = QmpRecMatGetLvlCritAt( hRecMat, i, j,
                                         &qres );
        iWeight = QmpRecMatGetLvlCritWeight( hRecMat,
                                         hGetCrit, i, &qres );
    }
}
```

QmpRecMatGetLvlCritWeight

GETS THE WEIGHTING FOR A CRITERION BY LEVEL.

Syntax

```
int QmpRecMatGetLvlCritWeight ( MpHnd in_hRecMat, MpHnd
                               in_hCrit, int in_iLevel, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_hCrit
    Handle to the criterion. Input.
in_iLevel
    Level of criterion. Input.
out_pResult
    Result code. Output.
```

Return Value

If successful, returns a criterion weighting.

Notes

A weight is assigned to a criterion if the record matcher is using a weighted average to combine criterion match scores.

After retrieving the handle of a criterion using `QmpRecMatGetLvlCritAt`, call `QmpRecMatGetLvlCritWeight` with the handle to find the criterion's weight.

See Also

`QmpRecMatGetLvlCritAt`.

Example

```
int iWeight;

iWeight = QmpRecMatGetLvlCritWeight( hRecMat, hGetCrit, 2,
                                     &qres );
```

QmpRecMatGetLvlRule

GETS RECORD MATCHER RULE FOR COMBINING THE MATCH SCORES OF CRITERIA AT THE SAME LEVEL.

Syntax

```
QMS_CRITERIA_RULE QmpRecMatGetLvlRule ( MpHnd in_hRecMat, int
                                         in_iLevel, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_iLevel
Criteria level. *Input*.

out_pResult
Result code. *Output*.

Return Value

If successful, returns the rule the record matcher will use at the given level to combine criteria match scores:

QMS_CRITERIA_UNDEFINED
QMS_CRITERIA_AVERAGE
QMS_CRITERIA_WEIGHTED_AVERAGE
QMS_CRITERIA_MAX
QMS_CRITERIA_MIN
QMS_CRITERIA_AND
QMS_CRITERIA_OR

If no rule has been defined, QMS_CRITERIA_UNDEFINED is returned.

See Also

[QmpRecMatSetLvlRule](#).

Example

```
long lLevels;
QMS_CRITERIA_RULE rule;
int i;

lLevels = QmpRecMatGetLvlCnt( hRecMat, &qres );
for ( i = 0; i < lLevels; i++ ) {
    rule = QmpRecMatGetLvlRule( hRecMat, i, &qres );
}
```

QmpRecMatGetLvlThreshold

GETS THE RECORD MATCHER THRESHOLD FOR A GIVEN LEVEL.

Syntax

```
int QmpRecMatGetLvlThreshold ( MpHnd in_hRecMat, int
    in_iLevel, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_iLevel
Criteria level for the threshold in question. *Input*.

out_pResult
Result code. *Output*.

Return Value

If successful, returns the matching threshold which the record matcher will use at the given level.

Notes

See Also

[QmpRecMatSetLvlRule](#).

Example

```
long lLevels;
int iThreshold;
int i;

lLevels = QmpRecMatSetLvlCnt( hRecMat, &qres );
for ( i = 0; i < lLevels; i++ ) {
    iThreshold = QmpRecMatGetLvlThreshold( hRecMat, i,
        &qres );
    if ( iThreshold != 50 )
        QmpUtilChkresultCode( QRESULT_SEVERE_FAIL,
            "Record matcher threshold is wrong" );
}
```

QmpRecMatGetMatRes

GETS THE MATCH RESULT OBJECT OUT OF THE RECORD MATCHER.

Syntax

```
MpHnd QmpRecMatGetMatRes ( MpHnd in_hRecMat, QRESULT*
    out_pResult );
in_hRecMat
    Handle to record matcher. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns handle to match result object if successful, or `NULL` if there is an error.

Notes

This function retrieves the match result object used by the record matcher. The record matcher outputs the results of its successful record comparisons to a table called the match result table. This table is encapsulated by the match result object. The match result table has information about both records that were compared. Specifically, for every record comparison that met the record matcher threshold, the table contains a record with the following information:

source ID of record 1	long integer
primary key of record 1	long integer
data list ID of record 1	long integer
source ID of record 2	long integer
primary key of record 2	long integer
data list ID of record 2	long integer
result of the comparison	integer between 0 and 100; the higher the score, the better the match

The match result table is used by the dupe groups phase to construct the dupe groups table.

Example

See example on [page 712](#).

QmpRecMatGetNthRecInterval

GETS RECORD MATCHER NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpRecMatGetNthRecInterval ( MpHnd in_hRecMat, QRESULT*
```

```
    out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns current record count interval notification setting for the record matcher if successful, or -1 if there is an error.

Notes

This function gets the record matcher Nth record event interval.

The record matcher can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the record matcher processes 100 records, it will send out a notification to *all* registered clients.

Example

See example on [page 712](#).

QmpRecMatGetRule

GETS CRITERION RULE THAT RECORD MATCHER WILL USE.

Syntax

```
QMS_CRITERIA_RULE QmpRecMatGetRule ( MpHnd in_hRecMat,
```

```
    QRESULT* out_pResult );
```

in_hRecMat

Handle to record matcher. Input.

out_pResult

Result code. *Output*.

Return Value

Returns the record matcher's current criterion rule if successful, or QMS_CRITERIA_UNDEFINED if there is an error.

Notes

QmpRecMatGetRule gets the criterion rule which the record matcher will use.

See Also

QmpRecMatSetRule for setting the record matcher rule.

Example

```
QRESULT qres;
QmpDeclHnd( hRecMat );
long lSize;
QMS_CRITERIA_RULE rule;

/* create record matcher */
hRecMat = QmpRecMatCreate( QMS_CRITERIA_AVERAGE, 90, QFALSE,
    &qres );

/* Set/get miscellaneous properties */
QmpRecMatSetWinSize( hRecMat, 3, 3, &qres );
lSize = QmpRecMatGetWinSize( hRecMat, &qres );
if ( lSize != 3 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Window size is
        wrong" );

QmpRecMatSetRule( hRecMat, QMS_CRITERIA_WEIGHTED_AVERAGE, &qres
    );
rule = QmpRecMatGetRule( hRecMat, &qres );
if ( rule != QMS_CRITERIA_WEIGHTED_AVERAGE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher rule
        is wrong" );
```

QmpRecMatGetThreshold

GETS THE THRESHOLD FOR THE RECORD MATCHER.

Syntax

```
int QmpRecMatGetThreshold ( MpHnd in_hRecMat, QRESULT*  
    out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the record matcher threshold if successful, or -1 if there is an error. This may be any integer between 0 and 100. The default value is 75.

Notes

Two records will be considered duplicates if their match scores equal or exceed the record matcher threshold.

Example

See example on [page 712](#).

QmpRecMatIsValid

TESTS WHETHER A RECORD MATCHER IS VALID.

Syntax

```
QBOOL QmpRecMatIsValid ( MpHnd in_hRecMat, QRESULT*  
    out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if record matcher is valid, else QFALSE.

Notes

A valid record matcher is one whose criteria (if it has any assigned yet) are also valid. If weighted average is used as the criteria rule, the weights of the criteria must add up to 100. The record matcher threshold and window size must be in allowable ranges. The collection of index keys must be valid. If the record matcher uses a preprocessed data source, it must be valid. For definitions of the validity of these contained objects, see **QmpCritIsValid**, **QmpIndxKeyIsValid**, and **QmpDataSrcIsValid**.

Example

See example on [page 712](#).

QmpRecMatRegEveryNthRecFunc

REGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpRecMatRegEveryNthRecFunc ( MpHnd in_hRecMat,
                                    QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_Func
Pointer to event handler. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The record matcher can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise change a visual representation on a computer screen.

Example

See the example on [page 712](#).

QmpRecMatRemCrit

REMOVES SPECIFIED CRITERION FROM RECORD MATCHER.

Syntax

```
MpHnd QmpRecMatRemCrit ( MpHnd in_hRecMat, MpHnd in_hCrit,  
    QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hCrit
Handle to the criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns handle to removed criterion if successful, or NULL if there is an error.

Notes

This function removes one of the criteria added with **QmpAddCrit**. The criteria is not destroyed, but is simply removed from the list of record matcher field criteria.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons. This function should not be called while the record matcher is executing.

Example

See example on [page 712](#).

QmpRecMatRemIdxKey

REMOVES A SPECIFIED INDEX KEY FROM THE RECORD MATCHER.

Syntax

```
MpHnd QmpRecMatRemIdxKey ( MpHnd in_hRecMat, MpHnd
    in_hIdxKey, QRESULT* out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*.

in_hIdxKey

Handle to the index key. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to removed index key if successful, or NULL if there is an error.

Notes

Removes a specific index key from the collection of index keys in the record matcher. The index key is *not* deleted, only its membership in the collection is removed.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons. This function should not be called while the record matcher is executing.

Example

See example on [page 712](#).

QmpRecMatRemLvl

REMOVES THE SPECIFIED CRITERIA LEVEL FROM THE RECORD MATCHER.

Syntax

```
void QmpRecMatRemLvl ( MpHnd in_hRecMat, int in_iLevel,
    QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_iLevel
    Criteria level to remove. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpRecMatRemLvl is a convenience function; call it when you want to remove all of the field match criteria objects at a given level from the record matcher.

To remove an individual criterion at a given level, call **QmpRecMatRemLvlCrit**.

See Also

QmpRecMatAddLvlCrit, **QmpRecMatSetLvlThreshold**.

Example

```
long lLevels;
int iThreshold;
int i;

lCount = QmpRecMatGetLvlCritCnt( hRecMat, 2, &qres );
if ( lCount != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL,
        "Wrong number of criteria" );
lCount = QmpRecMatGetLvlCnt( hRecMat, &qres );
QmpRecMatRemLvl( hRecMat, lCount-1, &qres );
```

QmpRecMatRemLvlCrit

REMOVES A SPECIFIED CRITERION FROM THE RECORD MATCHER BY LEVEL.

Syntax

```
MpHnd QmpRecMatRemLvlCrit ( MpHnd in_hRecMat, MpHnd in_hCrit,
    int in_iLevel, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_hCrit
    Handle to the criterion to remove. Input.
in_iLevel
    Level of criterion to remove. Input.
out_pResult
    Result code. Output.
```

Return Value

Handle to removed criterion.

Notes

See Also

[QmpRecMatAddLvlCrit](#), [QmpRecMatSetLvlThreshold](#).

Example

```
QmpRecMatRemLvlCrit(hRecMat, hZipMatchCriterion, 2, &qres);
```

QmpRecMatRemPreProcDataSrc

REMOVES A PREPROCESSED DATA SOURCE FROM THE RECORD MATCHER.

Syntax

```
MpHnd QmpRecMatRemPreProcDataSrc ( MpHnd in_hRecMat, long  
    in_lDataSrcID, QRESULT* out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*.

in_lDataSrcID

ID of the data source to be removed. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to removed data source if successful, or `NULL` if there is an error.

Notes

Allows an application to remove a preprocessed data source from the record matcher. The record matcher currently can accept only one data source. If you add a data source, you must remove it before adding another.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons. This function should not be called while the record matcher is executing.

See Also

[QmpRecMatAddPreProcDataSrc](#).

Example

See example on [page 712](#).

QmpRecMatSetCritWeight

SETS THE WEIGHTING FOR A CRITERION.

Syntax

```
void QmpRecMatSetCritWeight ( MpHnd in_hRecMat, MpHnd  
    in_hCrit, int in_iWeight, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hCrit
Handle to criterion. *Input*.

in_iWeight
Weight to assign to criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Weights may be assigned to record matcher criteria. The value of each weight must be between 0 and 100. If the record matcher rule is QMS_CRITERIA_WEIGHTED_AVERAGE, the summed weights of all criteria must equal 100. The default criterion weight value is 0.

Example

See example on [page 712](#).

QmpRecMatSetGenSortPreProc

BUILDS SORTED VERSIONS OF THE PREPROCESSED DATA SOURCE.

Syntax

```
void QmpRecMatSetGenSortPreProc ( MpHnd in_hRecMat, QBOOL
                                in_bGenSorts, QRESULT* out_pResult );
in_hRecMat
Handle to record matcher. Input.
in_bGenSorts
The build flag. If QTRUE, the preprocessed data source is sorted. If QFALSE, the preprocessed data source is left alone. Input.
out_pResult
Result code. Output.
```

Return Value

None.

Notes

A preprocessed data source must always contain one table with the records sorted in S1K1 order, as well as a separate table for each index key definition, with the records sorted in index key order. These other sorted tables contain a subset of the fields in the original table. This subset is limited to the fields that will be matched and indexed on, plus the standard augmented fields found in the DITR.

QmpRecMatSetGenSortPreProc sets the flag that determines if the proper sorted tables are built for the preprocessed data source attached to the record matcher. This function allows you to attach an unsorted preprocessed data source to the record matcher, and let the Centrus Merge/Purge library create the appropriate sorted tables for you.

See Also

QmpRecMatGetGenSortPreProc.

Example

```
QmpDeclHnd( hRecMat );
QRESULT qres;

QmpRecMatSetGenSortPreProc( hRecMat, QTRUE, &qres );
```

QmpRecMatSetLvlCritWeight

SETS THE WEIGHTING FOR A CRITERION AT A LEVEL.

Syntax

```
void QmpRecMatSetLvlCritWeight ( MpHnd in_hRecMat, MpHnd
    in_hCrit, int in_iWeight, int in_iLevel, QRESULT*
    out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_hCrit
Handle to the criterion. *Input*.

in_iWeight
Weight of criterion. *Input*.

in_iLevel
Level of criterion. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

A weight is assigned to a criterion if the record matcher is using a weighted average to combine criterion match scores.

Calling **QmpRecMatSetLvlCritWeight** with a level of 0 is equivalent to calling **QmpRecMatSetCritWeight**.

See Also

QmpRecMatSetCritWeight.

Example

```
int iWeight;

QmpRecMatSetLvlCritWeight(hRecMat, hGetCrit, 80, 2, &qres);
iWeight = QmpRecMatGetLvlCritWeight( hRecMat, hGetCrit, 2,
    &qres );
if ( iWeight != 80 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong criteria
level weight" );
```

QmpRecMatSetLvlRule

SETS RECORD MATCHER RULE FOR COMBINING THE MATCH SCORES OF CRITERIA AT THE SAME LEVEL.

Syntax

```
void QmpRecMatSetLvlRule ( MpHnd in_hRecMat,
    QMS_CRITERIA_RULE in_Rule, int in_iLevel, QRESULT*
    out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_Rule
Criteria rule to apply. *Input*.
Valid enums are:

QMS_CRITERIA_AVERAGE
QMS_CRITERIA_WEIGHTED_AVERAGE
QMS_CRITERIA_MAX
QMS_CRITERIA_MIN
QMS_CRITERIA_AND
QMS_CRITERIA_OR

in_iLevel
Criteria level.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

See Also

[QmpRecMatGetLvlRule](#).

Example

```
rule = QmpRecMatGetLvlRule( hRecMat, 2, &qres );
if ( rule != QMS_CRITERIA_AVERAGE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher
        rule is wrong" );
QmpRecMatSetLvlRule( hRecMat, QMS_CRITERIA_WEIGHTED_AVERAGE,
    2, &qres );
rule = QmpRecMatGetLvlRule( hRecMat, 2, &qres );
if ( rule != QMS_CRITERIA_WEIGHTED_AVERAGE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Record matcher
        rule is wrong" );
QmpRecMatRemLvlCrit( hRecMat, hZipMatchCriterion, 2, &qres );
lCount = QmpRecMatGetLvlCritCnt( hRecMat, 2, &qres );
```

```
if ( lCount != 1 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of
                           criteria" );
hGetCrit = QmpRecMatGetLvlCritAt( hRecMat, 0, 2, &qres );
hGetCrit = QmpRecMatRemLvlCrit( hRecMat, hGetCrit, 2, &qres );
lCount = QmpRecMatGetLvlCritCnt( hRecMat, 2, &qres );
if ( lCount != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of
                           criteria" );
lCount = QmpRecMatGetLvlCnt( hRecMat, &qres );
QmpRecMatRemLvl( hRecMat, 2, &qres );
lCount = QmpRecMatGetLvlCnt( hRecMat, &qres );
QmpRecMatRemLvl( hRecMat, 1, &qres );
lCount = QmpRecMatGetLvlCnt( hRecMat, &qres );
```

QmpRecMatSetLvlThreshold

SETS THE RECORD MATCHER THRESHOLD FOR A GIVEN LEVEL.

Syntax

```
void QmpRecMatSetLvlThreshold ( MpHnd in_hRecMat, int
    in_iThreshold, int in_iLevel, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_iThreshold
    The record matcher threshold for the given level. Input.
in_iLevel
    Criteria level of the threshold. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

See Also

[QmpRecMatGetLvlRule](#).

Example

```
long lLevels;
int iThreshold;
int i;

lLevels = QmpRecMatSetLvlCnt( hRecMat, &qres );
for ( i = 0; i < lLevels; i++ ) {

    iThreshold = QmpRecMatSetLvlThreshold( hRecMat, i, 90,
        &qres );
    if ( iThreshold != 90 )
        QmpUtilChkresultCode( QRESULT_SEVERE_FAIL,
            "Record matcher threshold is wrong" );
}
```

QmpRecMatSetNthRecInterval

SETS RECORD MATCHER NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpRecMatSetNthRecInterval ( MpHnd in_hRecMat, long
                               in_lInterval, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_lInterval
    Record matcher Nth record event interval. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the record matcher Nth record event interval.

The record matcher can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the record matcher processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

Example

See example on [page 712](#).

QmpRecMatSetRule

SETS CRITERION RULE WHICH RECORD MATCHER WILL USE.

Syntax

```
void QmpRecMatSetRule ( MpHnd in_hRecMat, QMS_CRITERIA_RULE
in_Rule, QRESULT* out_pResult );
```

in_hRecMat

Handle to record matcher. *Input*

in_Rule

Criterion rule. *Input*.

Valid enums are:

QMS_CRITERIA_AVERAGE	Average of all criteria.
QMS_CRITERIA_WEIGHTED_AVERAGE	Weighted average of all criteria.
QMS_CRITERIA_MAX	Use the criterion that provided the highest score.
QMS_CRITERIA_MIN	Use the criterion that provided the lowest score.
QMS_CRITERIA_AND	All criteria must achieve a “threshold” score.
QMS_CRITERIA_OR	At least one criterion must achieve a “threshold” score.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpRecMatSetRule sets the criteria rule which the record matcher will use. The rule defines how the results from the collection of criteria will be accumulated to arrive at an overall match score for the records being compared. The criteria rule must be one of the typedef enumerations in QMS_CRITERIA_RULE. These values are shown above.

See Also

[“Data Types” on page 73](#) for information on Centrus Merge/Purge enumerated types.

QmpRecMatGetRule for the value of the rule set by this function.

Example

See example on [page 712](#).

QmpRecMatSetThreshold

SETS THE THRESHOLD FOR THE RECORD MATCHER.

Syntax

```
void QmpRecMatSetThreshold ( MpHnd in_hRecMat, int  
                           in_iThreshold, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input*.

in_iThreshold
Threshold to set. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The record matcher threshold may be any integer between 0 and 100. The default value is 75.

Two records will be considered matches if their match scores equal or exceed the record matcher threshold.

Example

See example on [page 712](#).

QmpRecMatSetWinSize

SETS THE WINDOW SIZE.

Syntax

```
void QmpRecMatSetWinSize( MpHnd in_hRecMat, long
    in_lDefWinSize, long in_lMaxWinSize, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_lDefWinSize
    Default window size. Input.
in_lMaxWinSize
    Maximum window size. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the size of the sliding window in the record matcher.

The sliding window of the record matcher can be set to a fixed record size, or it can be set to grow (as needed) up to a maximum record size. A variable-size sliding window is called a dynamic sliding window. The sliding window will need to expand if a large number of records with the same index key value come through the system. This will allow all records with the same key value to be compared in the same sliding window, rather than relying on transitive closure to accomplish this.

To configure a dynamic sliding window, call `QmpRecMatSetWinSize` with the maximum window size greater than the default window size. For example:

```
QmpRecMatSetWinSize( hRecMat, 4, 40, &qres );
```

To configure a static sliding window, or to turn off the dynamic sliding window, call `QmpRecMatSetWinSize` with both windows set to the same size. For example:

```
QmpRecMatSetWinSize( hRecMat, 4, 4, &qres );
```

Note: The dynamic sliding window may increase the number of matches, but at the cost of slower job execution. If the application defines a poor index key definition (one with extremely low cardinality), job execution time will be especially poor. For example, index key definitions based entirely on state or ZIP code information will result in hundreds or thousands of records with the same index key value, causing the record matcher a lot of extra work and wasting execution time.

Example

```
/* Set/get miscellaneous properties */
QmpRecMatSetWinSize( hRecMat, 3, 6, &qres );
QmpRecMatFillWinSize( hRecMat, &lDefWinSize, &lMaxWinSize, &qres );
if ( lDefWinSize != 3 || lMaxWinSize != 6 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Window size is wrong" );
```

QmpRecMatUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpRecMatUnregEveryNthRecFunc ( MpHnd in_hRecMat,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
```

in_hRecMat
Handle to record matcher. *Input.*

in_Func
Pointer to event handler. *Input.*

out_pResult
Result code. *Output.*

Return Value

None.

Notes

This function unregisters the specified event handler from the record matcher, so that the event handler will no longer be notified of every Nth record processed.

The record matcher can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise change a visual representation on a computer screen.

Since the record matcher makes a pass through the input records for each index sequence, event messages will be sent for each pass through the records. For example, if there are 1,000 input records, 3 sorted input record tables, and the Nth record interval is 10, the record matcher will make three passes through the 1,000 records, and will generate approximately $(1000 * 3)/10 = 300$ messages for each pass.

Example

See example on [page 712](#).

QmpRecMatUseDataLstSvc

SPECIFIES THE DATA LIST SERVICE OBJECT TO BE USED BY THE RECORD MATCHER.

Syntax

```
void QmpRecMatUseDataLstSvc(MpHnd in_hRecMat, MpHnd
    in_hDataLstSvc, QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_hDataLstSvc
    Handle to data list service. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Specifies the data list service object to be used by the record matcher. The data list service is used by the record matcher to help it decide if it should or should not perform certain comparisons. If two records are from the same data list, and the data list has a property of “do not allow intra-list comparisons”, the record matcher will not compare those two records. This is very useful when the record matcher is using a preprocessed data source that has already been de-duped. Using the data list service, it could determine that any two records from the preprocessed data source should *not* be compared, thus speeding up the processing.

Example

See example on [page 712](#).

QmpRecMatUseIdxKey

SPECIFIES THE INDEX KEY OBJECT TO BE USED BY THE RECORD MATCHER.

Syntax

```
void QmpRecMatUseIdxKey ( MpHnd in_hRecMat, MpHnd  
                          in_hIdxKey, QRESULT* out_pResult );  
  
in_hRecMat  
    Handle to record matcher. Input.  
  
in_hIdxKey  
    Handle to index key. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

Adds an index key to the set of index keys maintained by the record matcher object. This function can be called several times to build up a collection of index keys. Each new index key will result in a separate execution of the record matching process, using a different index ordering to run through the records. This collection of index keys given to the record matcher should be the same as those given to the data input phase. Using effective index keys will result in similar records being placed near each other so that they can be compared to each other in at least one ordering during the record matching phase. Even if the two records don't appear near enough to one another to be compared when ordered by one index key, some other ordering is likely to place the records near enough to one another that they will be compared.

The record matcher and its components (including criteria, algorithms, index keys, and preprocessed data sources) must remain intact through the dupe groups phase. This is necessary because the dupe groups phase uses the record matcher to make additional record comparisons.

Example

See the example on [page 712](#).

QmpRecMatUseMatRes

SPECIFIES THE MATCH RESULT OBJECT TO BE USED BY THE RECORD MATCHER.

Syntax

```
void QmpRecMatUseMatRes ( MpHnd in_hRecMat , MpHnd in_hMatRes ,
                           QRESULT* out_pResult );
in_hRecMat
    Handle to record matcher. Input.
in_hMatRes
    Handle to match result object. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function specifies the match result object to be used by the record matcher. The record matcher outputs the results of its record comparisons to a table called the match result table. This table is encapsulated in an object called the match result. The match result table has information on both records that were compared. Specifically, for every record comparison that met the record matcher threshold, the table contains a record with the following information:

primary key of record 1	long integer
source id of record 1	long integer
data list id of record 1	long integer
primary key of record 2	long integer
source id of record 2	long integer
data list id of record 2	long integer
result of the comparison	integer between 0 and 100; the higher the score, the better the match

The match result table is used as input to the dupe groups phase to construct the dupe groups table.

Example

See example on [page 712](#).

Function Class: QmpRpt*

GENERAL FUNCTIONS FOR MANIPULATING LIBRARY REPORTS.

Some of the ways you can configure a report using these functions include adding lines to the report header, generating a page header, and changing the page width and height.

Reports allow you to hone your management of processes. For example, your reports might show that by using the first five characters of the ZIP code field (substringing) your matches increased by 15%. In another example, a list-by-list report might show that one data source provided consistently better matches (cleaner data) than the others.

Quick Reference

Function	Description	Page
QmpRptFillStatSamp	fills a structure with the current statistical sampling properties of a report.	758
QmpRptSetFilename	Sets the output file name for the report.	760
QmpRptSetLinesPerPage	Sets the lines per page for the report.	761
QmpRptSetPageWidthInChars	Sets the page width for the report.	762
QmpRptSetStatSamp	Sets the statistical sampling properties of a report.	763
QmpRptSetSubtitle	Sets the subtitle for the report.	765
QmpRptSetTitle	Sets the title for the report.	766

QmpRptFillStatSamp

FILLS A STRUCTURE WITH THE CURRENT STATISTICAL SAMPLING PROPERTIES OF A REPORT.

Syntax

```
void QmpRptFillStatSamp ( MpHnd in_hRpt,
    QMS_OUTPUT_SAMPLING* out_StatSampling, QRESULT*
    out_pResult );
```

in_hRpt

Handle to report. *Input*.

out_StatSampling

Statistical sampling structure. *Output*.

Structure members are:

lMaxRecords	Maximum number of records to include.
lFirstRecordNum	Record number of first record to include.
lLastRecordNum	Record number of last record to include.
lInterval	Sampling interval (return every “nth” record).
lGroupSize	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record).
lMaxDupeGroups	Maximum number of dupe groups to include.
lDuplicatesFirst	First dupe group to include.
lDuplicatesLast	Last dupe group to include.
lDuplicatesInterval	Dupe group sampling interval.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpRptFillStatSamp fills a structure of type **QMS_OUTPUT_SAMPLING** with the statistical sampling properties currently set for a report.

Statistical sampling allows you to quickly filter output records in a controlled way. By including a small subset of records in a report or output table, you can evaluate the output before committing to a long processing run.

Statistical sampling for reports is optional. Call **QmpRptSetStatSamp** to turn statistical sampling on for a report and to set the sampling properties. If statistical sampling is used for a report, a statistical sampling section will appear in that report, as well as in the job summary report.

Example

```
QMS_OUTPUT_SAMPLING OutputSampling; /* statistical sampling structure */
QMS_OUTPUT_SAMPLING FillOutputSampling; /* statistical sampling structure */
QmpRptSetLinesPerPage( hNearMissReport, -1, &qres );
QmpRptSetPageWidthInChars( hNearMissReport, 79, &qres );
QmpMissRptSetPrintKeys( hNearMissReport, QTRUE, &qres );
```

```

QmpMissRptSetPrintSrc( hNearMissReport, QTRUE, &qres );
QmpMissRptUseDupGrp( hNearMissReport, hDupeGroups, &qres );
iAcceptance = ( iThreshHold > 90 ) ? iThreshHold : iThreshHold + 10;
QmpMissRptSetAcceptLvl( hNearMissReport, iAcceptance, &qres );
QmpPhaseSetRecProto( hNearMissReport, hProtoRec, &qres );

/* Print a Near Miss report */
sprintf( szStringPtr, "Rpt-NearMiss-%d.log", iThreshHold );
sprintf( pszTempPathBuffer, "%s%s%s", pszOutputPathBuffer, pszDelim,
        szStringPtr );
remove(pszTempPathBuffer);
QmpRptSetFilename( hNearMissReport, pszTempPathBuffer, &qres );
sprintf( szTempPtr, ": Near Miss Report for a Threshold of %d", iThreshHold);
strcat( szStringPtr, szTempPtr );
QmpRptSetTitle( hNearMissReport, szStringPtr, &qres );
QmpRptSetSubtitle( hNearMissReport, "Generated for Make-A-Wish Foundation",
        &qres );
QmpPhaseUseDITR( hNearMissReport, hDITR, &qres );
if ( bEveryNth ) {
    QmpMissRptRegEveryNthRecFunc( hNearMissReport, pEveryNthRecordClient, &qres
    );
    QmpMissRptSetNthRecInterval( hNearMissReport, 200, &qres );
}
QmpRptFillStatSamp( hNearMissReport, &FillOutputSampling, &qres );
if ( bUseSampling ) {
    OutputSampling.lMaxRecords = 1000;      /* max number of records to output */
    OutputSampling.lFirstRecordNum = 1;      /* first record to output */
    OutputSampling.lLastRecordNum = 1000;     /* last record to output */
    OutputSampling.lInterval = 0;            /* Sampling interval for output */
    OutputSampling.lGroupSize = 0;           /* size of group beginning at */
                                            /* each output interval */
    OutputSampling.lMaxDupeGroups = 5;       /* maximum number of Dupe Groups */
                                            /* to output. */
    OutputSampling.lDuplicatesFirst = 1;      /* first Dupe Group to output. */
    OutputSampling.lDuplicatesLast = 20;      /* last DupeGroup to output. */
    OutputSampling.lDuplicatesInterval = 2;   /* Dupe Group sampling interval. */
    QmpRptSetStatSamp( hNearMissReport, &OutputSampling, &qres );
}
QmpRptFillStatSamp( hNearMissReport, &FillOutputSampling, &qres );
iAcceptance = QmpMissRptGetAcceptLvl( hNearMissReport, &qres );
printf( "Starting Near Miss Report phase:\n" );
if ( bUsePrePro ) {
    QmpMissRptAddPreProcDataSrc( hNearMissReport, hPreProDataSrc, &qres );
}
QmpPhaseStart( hNearMissReport, &qres );

```

QmpRptSetFilename

SETS THE OUTPUT FILE NAME FOR THE REPORT.

Syntax

```
void QmpRptSetFilename ( MpHnd in_hRpt, const char*
    in_szFilename, QRESULT* out_pResult );
```

in_hRpt
Handle to report. *Input*.

in_szFilename
Name of the report output file. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Example

```
QmpRptSetFilename ( hDupsReport, (char*) pStringPtr, pResult
    );
```

QmpRptSetLinesPerPage

SETS THE LINES PER PAGE FOR THE REPORT.

Syntax

```
void QmpRptSetLinesPerPage ( MpHnd in_hRpt, int in_iLines,
                            QRESULT* out_pResult );
in_hRpt
    Handle to report. Input.
in_iLines
    Lines per page of the report. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpRptSetLinesPerPage sets the number of lines per page. The default value is 60.

If the value is positive, a header will be generated after this many lines are output. Setting the value to -1 limits header generation to the first page only.

When the header is created a form-feed character is generated. It is up to your text processor and printer to use the character.

Example

```
/*****************************************/
/* Report generation Phase.          */
/*****************************************/

QmpRptSetLinesPerPage( hDuplicatesReport, -1, &qres );
QmpRptSetPageWidthInChars( hDuplicatesReport, 79, &qres );
QmpDupsRptSetPrintKeys( hDuplicatesReport, QTRUE, &qres );
QmpDupsRptSetPrintSrc( hDuplicatesReport, QTRUE, &qres );
QmpDupsRptSetPrintMatRes( hDuplicatesReport, QTRUE, &qres );
QmpDupsRptUseDupGrp( hDuplicatesReport, hDupeGroups, &qres );
QmpPhaseSetRecProto( hDuplicatesReport, hProtoRec, &qres );
```

QmpRptSetPageWidthInChars

SETS THE PAGE WIDTH FOR THE REPORT.

Syntax

```
void QmpRptSetPageWidthInChars ( MpHnd in_hRpt, int
                               in_iWidth, QRESULT* out_pResult );
in_hRpt
    Handle to report. Input.
in_iWidth
    Width of the report in characters. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpRptSetPageWidthInChars sets the character width for a report page. The default value 79 works well for 10-point, fixed-width fonts in portrait mode. A value of 135 works well for landscape mode. Using fixed-width fonts is suggested.

Example

```
/*****************************************/
/* Report generation Phase.           */
/*****************************************/
QmpRptSetLinesPerPage( hDuplicatesReport, -1, &qres );
QmpRptSetPageWidthInChars( hDuplicatesReport, 79, &qres );
QmpDupsRptSetPrintKeys( hDuplicatesReport, QTRUE, &qres );
QmpDupsRptSetPrintSrc( hDuplicatesReport, QTRUE, &qres );
QmpDupsRptSetPrintMatRes( hDuplicatesReport, QTRUE, &qres );
QmpDupsRptUseDupGrp( hDuplicatesReport, hDupeGroups, &qres );
QmpPhaseSetRecProto( hDuplicatesReport, hProtoRec, &qres );
```

QmpRptSetStatSamp

SETS THE STATISTICAL SAMPLING PROPERTIES OF A REPORT.

Syntax

```
void QmpRptSetStatSamp ( MpHnd in_hRpt,
    QMS_OUTPUT_SAMPLING* in_StatSampling, QRESULT*
    out_pResult );
```

in_hRpt

Handle to report. *Input*.

in_StatSampling

Statistical sampling structure. *Input*.

Structure members are:

lMaxRecords	Maximum number of records to include.
lFirstRecordNum	Record number of first record to include.
lLastRecordNum	Record number of last record to include.
lInterval	Sampling interval (return every “nth” record). This <i>must</i> be greater than lGroupSize.
lGroupSize	Size of group beginning at every interval (i.e., return a group of “m” records starting at every “nth” record).
lMaxDupeGroups	Maximum number of dupe groups to include.
lDupesFirst	First dupe group to include.
lDupesLast	Last dupe group to include.
lDupesInterval	Dupe group sampling interval.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpRptSetStatSamp turns on statistical sampling for a report and sets the sampling properties using an application-supplied structure of type **QMS_OUTPUT_SAMPLING**. The structure members must be set before the function is called.

Statistical sampling allows you to quickly filter output records in a controlled way. By including a small subset of records in a report or output table, you can evaluate the output before committing to a long processing run.

Statistical sampling for reports is optional. Call **QmpRptFillStatSamp** to get the current statistical sampling properties of a report. If statistical sampling is used for a report, a statistical sampling section will appear in that report, as well as in the job summary report.

If you want to use either the interval or group size sampling properties, both properties must be set to a value greater than zero. It does not make sense to set a group size greater than an interval; doing so will cause an error.

Note: Statistical sampling can not be set for a job summary report, since this report is not record-based.

Example

```
/* statistical sampling structure */
QMS_OUTPUT_SAMPLING OutputSampling;

/* maximum number of records to output */
OutputSampling.lMaxRecords      = 1000;

/* first record to output */
OutputSampling.lFirstRecordNum   = 1;

/* last record to output */
OutputSampling.lLastRecordNum    = 1000;

/* Sampling interval for output */
OutputSampling.lInterval         = 0;

/* size of group beginning at each output interval */
OutputSampling.lGroupSize        = 0;

/* maximum number of Dupe Groups to output. */
OutputSampling.lMaxDupeGroups   = 5;

/* first Dupe Group to output. */
OutputSampling.lDuplicatesFirst  = 1;

/* last DupeGroup to output. */
OutputSampling.lDuplicatesLast   = 20;

/* Dupe Group sampling interval. */
OutputSampling.lDuplicatesInterval = 2;

QmpRptSetStatSamp( hDuplicatesReport, &OutputSampling, &qres
);
```

QmpRptSetSubtitle

SETS THE SUBTITLE FOR THE REPORT.

Syntax

```
void QmpRptSetTitle ( MpHnd in_hRpt, const char*
    in_szSubtitle, QRESULT* out_pResult );
```

in_hRpt
Handle to report. *Input*.

in_szSubtitle
Report subtitle. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpRptSetTitle supplies a line of text that appears in the header of a report. If this call is not made, or made with an empty *in_szSubtitle* value, the string is not included in the report header.

See Also

QmpRptSetTitle

Example

```
strcat( szStringPtr, ": Uniques Report for a Threshold of ");
strcat( szStringPtr, "90");
QmpRptSetTitle(hUniqsReport, szStringPtr, &qres );
QmpRptSetSubtitle(hUniqsReport, "Generated for Make-A-Wish
Foundation", &qres );
```

The code sample produces the subtitle in the following report header:

```
Centrus Merge/Purge version 1.0.0
Page: 1
Sagent Technology, Inc.
Uniques Report
```

```
Rpt-Uniques-90.log: Uniques Report for a Threshold of 90
Generated for Make-A-Wish Foundation
Date/Time: May 22, 1998 9:36 AM
-----
```

QmpRptSetTitle

SETS THE TITLE FOR THE REPORT.

Syntax

```
void QmpRptSetTitle ( MpHnd in_hRpt, const char* in_szTitle,
                      QRESULT* out_pResult );
in_hRpt
    Handle to report. Input.
in_szTitle
    Report title. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpRptSetTitle supplies a line of text that appears in the header of a report. If this call is not made, or made with an empty *in_szTitle* value, the string is not included in the report header.

Example

```
strcat ( pStringPtr, ": Uniques Report for a Threshold of " );
strcat ( pStringPtr, szMatcherThreshold );
QmpRptSetTitle ( hUniqsReport, (char*) pStringPtr, pResult );
```

The code sample produces the title in the following report header:

```
Centrus Merge/Purge version 1.0.0
Page: 1
Sagent Technology, Inc.
Uniques Report
```

```
Rpt-Uniques-90.log: Uniques Report for a Threshold of 90
Generated for Make-A-Wish Foundation
Date/Time: May 22, 1998 9:36 AM
-----
```

Function Class: QmpTbl*

GENERAL TABLE MANIPULATION FUNCTIONS.

Notes on Text Tables

Centrus Merge/Purge supports ASCII text tables with both fixed-width and character-delimited fields. These tables are used primarily for data input and table generation output.

Text tables used by the Centrus Merge/Purge library must have an accompanying format file. Centrus Merge/Purge will create this format file for text tables it creates; it is the user's responsibility to provide format files for text tables given to the library. Refer to [Appendix B](#) on page 907 for information about creating and editing format files. If Centrus Merge/Purge creates an output text table, it creates the format file in the same directory. The format file is given the same name as the table, with a .fmt extension.

Text tables do not have a default file extension. For example, if you specify an output text table to have the name "output", the file name will be Output, not Output.txt. Similarly, text tables used as input do not require a particular file extension.

Field-delimited output text tables created by the library use a comma delimiter and have field names in the first row. On NT/Win32 platforms the EOL character is CRLF, and on UNIX platforms the EOL character is LF.

Fixed-width output text tables created by the library have field widths set by the DITR record, and do not have field names in the first row. On NT/Win32 platforms the EOL character is CRLF, and on UNIX platforms the EOL character is LF.

Notes on Oracle Tables

Centrus Merge/Purge supports Oracle tables for Oracle versions 7.3 and greater. Machines using the **QmpTbl*** functions must have Oracle Client version 7.3 or greater installed on Windows NT or Sun Solaris platforms.

Centrus Merge/Purge Oracle tables are used for data input and table generation output. Indexing is not supported for Oracle tables. Updating existing data is facilitated by reading an input table, modifying the data, and writing a new output table.

Note: Traversing an oracle table backwards is an extremely time-consuming (slow) operation.

The Oracle-specific functions in the Centrus Merge/Purge library are:

- **QmpTblOrc1Create**
- **QmpTblOrc1SetCache**
- **QmpTblOrc1GetCache**
- **QmpTblSetWhereClause**

Quick Reference

Function	Description	Page
QmpTblAddNewRec	Adds a new blank record, making it the current record.	770
QmpTblAtBOF	Returns QTRUE if at beginning of table.	771
QmpTblAtEOF	Returns QTRUE if at end of table.	772
QmpTblCbCreate	Creates a CodeBase Merge/Purge table object.	773
QmpTblClearFlds	Clears the fields of a table.	774
QmpTblClose	Closes the table.	775
QmpTblCommitRec	Commits to the table any changes in the current record.	776
QmpTblCreateIdx	Creates an index.	777
QmpTblCreateRep	Creates the underlying representation of the table.	778
QmpTblDefineFlds	Defines the fields of a table.	779
QmpTblDelIdx	Deletes an index.	780
QmpTblDelRec	Deletes the current record.	781
QmpTblDestroy	Destroys a table.	782
QmpTblDestroyRep	Destroys the underlying representation of the Table.	783
QmpTblDTxtCreate	Creates a delimited-text Merge/Purge table object.	784
QmpTblExists	Tests whether the underlying table representation (the native database table) exists.	786
QmpTblFillByFieldRec	Fills the supplied fields from the current record.	787
QmpTblFillIdxInfo	Copies into an application-allocated array the index name and expression for the table indexes.	788
QmpTblFillMappedRec	Fills a mapped record.	789
QmpTblFillRec	Fills the field values from the current record.	790
QmpTblFTxtCreate	Creates a fixed-width text Merge/Purge table object.	791
QmpTblGetAutoOpenIdx	Finds out whether indexes will be opened when the table is opened.	793
QmpTblGetCnt	Gets the number of records in the table.	794
QmpTblGetDatabaseName	Gets name of database.	795
QmpTblGetExclusive	Gets exclusive use setting.	796
QmpTblGetFldCnt	Gets the number of fields in a table.	797
QmpTblGetFldName	Gets the field name of a particular field of a table	798
QmpTblGetFldNameVB	Get the field name of a particular field of a table (VB version)	799
QmpTblGetFlds	Gets the fields of a table.	800

Function	Description	Page
QmpTblGetIdxCnt	Gets the number of indexes for this table.	801
QmpTblGetPos	Gets the current position in the table.	802
QmpTblGetReadOnly	Gets read-only setting.	803
QmpTblGetTblName	Gets name of table.	804
QmpTblIsOpen	Tests whether table is open.	805
QmpTblMoveBy	Moves by an offset amount.	806
QmpTblMoveFirst	Moves to first record in table.	807
QmpTblMoveLast	Moves to last record in table.	808
QmpTblMoveTo	Moves to a particular record.	809
QmpTblOpen	Opens the specified table.	810
QmpTblOrclCreate	Creates an Oracle Merge/Purge table object.	811
QmpTblOrclGetCache	Gets the cache size of an Oracle Merge/Purge table object.	812
QmpTblOrclSetCache	Sets the cache size of an Oracle Merge/Purge table object.	813
QmpTblSeekRec	Seeks a specified record.	814
QmpTblSeekVal	Seeks a particular key value.	815
QmpTblSetAutoOpenIdx	Enables the indexes to be opened when the table is opened.	817
QmpTblSetDatabaseName	Sets name of database.	818
QmpTblSetExclusive	Sets exclusive use setting.	819
QmpTblSetIdx	Sets the current index.	820
QmpTblSetMappedRec	Sets a mapped record.	821
QmpTblSetReadOnly	Sets specified table to read only.	822
QmpTblSetRec	Sets the field values in the current record.	823
QmpTblSetTblName	Sets name of table.	824
QmpTblSetWhereClause	Sets the where clause of a RDBMS table (does nothing for file-based tables).	825

QmpTblAddNewRec

ADDS A NEW BLANK RECORD, MAKING IT THE CURRENT RECORD.

Syntax

```
QBOOL QmpTblAddNewRec ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if a record is added, else QFALSE.

Notes

`QmpTblAddNewRec` adds a new blank record to a table, making it the current table record.

Example

```
bIsRecordAdded = QmpTblAddNewRec (hTbl, pResult);
```

QmpTblAtBOF

RETURNS QTRUE IF AT BEGINNING OF TABLE.

Syntax

```
QBOOL QmpTblAtBOF ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

Returns QTRUE if at the beginning of the table, else QFALSE.

Notes

Beginning Of File (BOF) is not the position of the first record, just as End Of File (EOF) is just beyond the last record.

Example

```
bBegOfTable = QmpTblAtBOF ( hTbl, pResult );
```

QmpTblAtEOF

RETURNS QTRUE IF AT END OF TABLE.

Syntax

```
QBOOL QmpTblAtEOF ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

Returns QTRUE if at the end of the table, else QFALSE.

Notes

End Of File (EOF) is just after the last record.

Example

```
if ( QmpTblAtEOF( hTblInput, &qres ) ) {  
    QmpTblClose( hTblInput, &qres );  
    QmpTblDestroy( hTblInput, &qres );  
    return QFALSE;  
}
```

QmpTblCbCreate

CREATES A CODEBASE MERGE/PURGE TABLE OBJECT.

Syntax

```
MpHnd QmpTblCbCreate ( const char* in_pszDatabaseName, const
                         char* in_pszTableName, QRESULT* out_pResult );
in_pszDatabaseName
    Database name. Input.
in_pszTableName
    Table name. The name must either be a pure name with no file extension
    (e.g. table_alpha) or a name with a .dbf extension (e.g. table_beta.dbf). Either
    of these name conventions may include a full name path (e.g.
    c:\tables\table_beta.dbf). Input.
out_pResult
    Result code. Output.
```

Return Value

Returns handle to table if successful, or NULL if there is an error.

Notes

The CodeBase table type consists of a Centrus Merge/Purge wrapper around a standard CodeBase table. This is a fast access table with an unlimited potential file size. Use this if your data begins in DBF format.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

```
hDuplicatesTable = QmpTblCbCreate ( "", "duplicatesTable", pResult );
```

QmpTblClearFlds

CLEAR THE FIELDS OF A TABLE.

Syntax

```
void QmpTblClearFlds ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the fields of a table back to their initial state. The field count is set to 0.

None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpTblClearFlds (hTbl, pResult );
```

QmpTblClose

CLOSES THE TABLE.

Syntax

```
void QmpTblClose ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

QmpTblClose closes the physical representation of the table.

Example

```
QmpTblClose( hTblInput, &qres );  
QmpTblDestroy( hTblInput, &qres );
```

QmpTblCommitRec

COMMITS TO THE TABLE ANY CHANGES IN THE CURRENT RECORD.

Syntax

```
void QmpTblCommitRec ( MpHnd in_hTbl, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpTblCommitRec commits changes in the current record to the physical table file.

Example

```
/* 10.m.VIII commit the changes to the record in the */
/* DITR */
QmpTblCommitRec(DITR_Table, &Result);
if(!qmpSucceeded(Result)) {
DisplayResultMsg(Result, "Unable to commit enhancement "
    "change to table at dupe group %ld", group);
QmpDupGrpRecDestroy(hDupeRecord, &Result);
QmpRecDestroy(hRecord, &Result);
QmpRecDestroy(hSeekRecord, &Result);
QmpRecDestroy(hMasterRec, &Result);
QmpDupGrpRecDestroy(hMasterDupeRec, &Result);
QmpTblClose(Prepro_Table, &Result);
QmpTblClose(DITR_Table, &Result);
free(DupeGroupIDs);
return MPE_FAILURE;
}
```

QmpTblCreateIdx

Creates an index.

Syntax

```
void QmpTblCreateIdx ( MpHnd in_hTbl, const char*
    in_pszIdxName, const char* in_pszIdxExpr, QBOOL
    in_bUnique, QRESULT* out_pResult );
```

in_hTbl

Handle to table. *Input*.

in_pszIdxName

Index name. *Input*.

in_pszIdxExpr

Index expression. *Input*.

in_bUnique

Require uniques or not. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

QmpTblCreateIdx creates and names an index for the table based on the index expression supplied. The index expression qualifies which field or fields are included in the index. The *in_bUnique* parameter tells the table whether to include duplicates or to include only a single entry when multiples exist.

QmpTblCreateIdx creates the index, and **QmpTblSetIdx** activates the index. The index sorts the fields identified in the index expression in ascending order.

Example

```
QmpTblCreateIdx (hTbl, pszIdxName, pszIdxExpr, bUnique,
    pResult );
```

QmpTblCreateRep

CREATES THE UNDERLYING REPRESENTATION OF THE TABLE.

Syntax

```
void QmpTblCreateRep ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

`QmpTblCreateRep` creates a physical table (a file). Other functions such as `QmpTblCbCreate` create the object that encapsulates a physical table. The table object should be created first, followed by the physical table.

If the creation function fails, an error code is returned in the *out_pResult* parameter.

Example

```
QmpTblCreateRep (hTbl, pResult);
```

QmpTblDefineFlds

DEFINES THE FIELDS OF A TABLE.

Syntax

```
void QmpTblDefineFlds ( MpHnd in_hTbl, MpHnd in_hRec, QRESULT*  
    out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_hRec
Record containing field definitions. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpTblDefineFlds defines the fields of a table. The fields are located in the record passed to the function.

Table field name case is not preserved. For example, setting a field name to “ZIP Code” does not guarantee that the capital letters will be present if the field name is retrieved in the future.

Example

```
QmpTblDefineFlds (hTbl, hRec, pResult );
```

QmpTblDelIdx

DELETES AN INDEX.

Syntax

```
void QmpTblDelIdx ( MpHnd in_hTbl, const char*
                      in_pszIdxName, QRESULTT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_pszIdxName
Index name. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The table must be opened with index auto-open set to QTRUE. The default value for a table's index auto-open setting is QFALSE.

See Also

[QmpTblSetAutoOpenIdx](#), [QmpTblGetAutoOpenIdx](#).

Example

```
QmpTblDelIdx (hTbl, pszIdxName, pResult );
```

QmpTblDelRec

DELETES THE CURRENT RECORD.

Syntax

```
void QmpTblDelRec ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function deletes the current record.

Example

```
QmpTblDelRec (hTbl, pResult);
```

QmpTblDestroy

DESTROYS A TABLE.

Syntax

```
void QmpTblDestroy ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table.

out_pResult
Result code.

Return Value

None.

Notes

This function deletes the Centrus Merge/Purge table object, *not* the underlying table representation.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Destroying an object does not affect any associated objects (objects that are attached using **Qmp*Use*** or **Qmp*Add***). Only the references to the associated objects are destroyed.

Example

```
QmpTblDestroy ( (MpHnd)*(MpHnd*)ele->data, &Result );
```

QmpTblDestroyRep

DESTROYS THE UNDERLYING REPRESENTATION OF THE TABLE.

Syntax

```
void QmpTblDestroyRep ( MpHnd in_hTbl, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function destroys the underlying representation of a specified table (a physical table or file) and frees its associated resources.

Example

```
/* 9.c use data list svc */
QmpDupGrpsUseDataLstSvc(job->hDupeGroups,
    job->hDataListService, &Result);
if(!qmpSucceeded(Result)) {
    QmpTblDestroyRep(job->hDupeGroupsTable, &Result);
    QmpTblDestroy(job->hDupeGroupsTable, &Result);
    QmpDupGrpsDestroy(job->hDupeGroups, &Result);
    DisplayResultMsg(Result, "Unable to use data list service in
        dupe group phase");
    return MPE_FAILURE;
}
```

QmpTblDTxtCreate

CREATES A DELIMITED-TEXT MERGE/PURGE TABLE OBJECT.

Syntax

```
MpHnd QmpTblDTxtCreate ( const char* in_pszDatabaseName, const
                           char* in_pszTblName, QRESULT* out_pResult );
in_pszDatabaseName
  Database name. Input.
in_pszTblName
  Table name. Input.
out_pResult
  Result code. Output.
```

Return Value

Returns handle to table object if successful, or `NULL` if there is an error.

Notes

Delimited text files use a character such as comma or tab to delimit fields in a record.

Since records in a delimited text file vary in length, library functions which move within this type of file are limited or unimplemented.

The following table functions currently do not support delimited text tables:

- `QmpTblGetAutoOpenIdx`
- `QmpTblSetAutoOpenIdx`
- `QmpTblDelRec`
- `QmpTblMoveLast`
- `QmpTblSeekRec`
- `QmpTblSeekVal`
- `QmpTblCreateIdx`
- `QmpTblDelIdx`
- `QmpTblSetIdx`
- `QmpTblGetIdxCnt`
- `QmpTblFillIdxInfo`

The following functions have limits placed upon their use:

`QmpTblMoveTo` is limited to a move to the first record.

`QmpTblMoveBy` is limited to an offset of one record.

Example

```
QmpDeclHnd( hTable );
QmpDeclHnd( hIniFil );
QmpDeclHnd( hLog );
QmpDeclHnd( hMyRec );
const char* szFieldName;
hTable = QmpTblDTxtCreate( "", "1000csv.csv", &qres );
QmpTblOpen( hTable, &qres );
szFieldName = QmpTblGetFldName( hTable, 0, &qres );
printf( "Fieldname is: %s\n", szFieldName );
szFieldName = QmpTblGetFldName( hTable, 1, &qres );
printf( "Fieldname is: %s\n", szFieldName );
QmpTblClose( hTable, &qres );
QmpTblDestroy( hTable, &qres );
hMyRec = QmpRecCreate( &qres );
QmpRecAddByType( hMyRec, "First", QMS_VARIANT_STRING, 15, 0, &qres );
QmpRecAddByType( hMyRec, "Last", QMS_VARIANT_STRING, 15, 0, &qres );
QmpRecAddByType( hMyRec, "Addr", QMS_VARIANT_STRING, 35, 0, &qres );
hTableOut = QmpTblDTxtCreate( "", "test.csv", &qres );
QmpTblDefineFlds( hTableOut, hMyRec, &qres );
QmpTblCreateRep( hTableOut, &qres );
QmpTblClose( hTableOut, &qres );
QmpTblDestroy( hTableOut, &qres );
```

QmpTblExists

TESTS WHETHER THE UNDERLYING TABLE REPRESENTATION (THE NATIVE DATABASE TABLE) EXISTS.

Syntax

```
QBOOL QmpTblExists ( MpHnd in_htbl, QRESULT* out_pResult );  
in_htbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

Returns QTRUE if underlying table representation exists, else QFALSE.

Notes

This function tests whether the native database table exists.

Example

```
/* Destroy the physical representation of the record */  
/* matcher results file */  
  
if ( QmpTblExists( hMatResTable, pResult ) ) {  
    QmpTblDestroyRep( hMatResTable, pResult );  
}
```

QmpTblFillByFieldRec

FILLS THE SUPPLIED FIELDS FROM THE CURRENT RECORD.

Syntax

```
void QmpTblFillByFieldRec ( MpHnd in_hTbl, MpHnd io_hRec,  
                           QRESULT* out_pResult );
```

in_hTbl

Handle to table. *Input*.

io_hRec

Record to fill from current record. *Input, Output*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpTblFillByFieldRec fills a record passed into the function with the field values from the current record. This function is designed to handle the situation where the passed-in record has fewer fields than the table records.

Example

```
QmpTblFillByFieldRec (hTbl, hRec, pResult );
```

QmpTblFillIdxInfo

COPIES INTO AN APPLICATION-ALLOCATED ARRAY THE INDEX NAME AND EXPRESSION FOR THE TABLE INDEXES.

Syntax

```
long QmpTblFillIdxInfo ( MpHnd in_hTbl, long in_lMaxIdx,
                          QMS_TABLE_INDEXES* io_IndxArray, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
in_lMaxIdx
    Number of index structures to copy. Input.
io_IndxArray
    Output array of index information. Input, output.
out_pResult
    Result code. Output.
```

Return Value

Returns the number of indexes on this table if successful, else -1.

Notes

The application owns the array of QMS_TABLE_INDEXES, as well as the strings inside each array element.

Example

```
QmpTblSetAutoOpenIdx( hPreproTable, QTRUE, &qres );
QmpTblOpen( hPreproTable, &qres );
lCount = QmpTblGetIndxCnt( hPreproTable, &qres );
if ( lCount < 5 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Wrong number of indexes" );
IndxInfoArray = (QMS_TABLE_INDEXES*)malloc( lCount * sizeof( QMS_TABLE_INDEXES
));
lCount2 = QmpTblFillIdxInfo( hPreproTable, lCount, IndxInfoArray, &qres );
QmpTblClose( hPreproTable, &qres );
if (QMS_STRICTCMP( IndxInfoArray[4].szIndexExpression, "zipcode" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "Index info is incorrect" );
free( IndxInfoArray );
```

QmpTblFillMappedRec

FILLS A MAPPED RECORD.

Syntax

```
void QmpTblFillMappedRec( MpHnd in_hTbl, MpHnd in_hFldMap,
                           MpHnd in_hRec, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_hFldMap
Handle to field map. *Input*.

in_hRec
Handle of record to fill. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Once a field map has been created and defined, and a data source has been identified, call **QmpTblFillMappedRec** to fill the destination record with data from a source record (located in the data source's table).

Example

See example on [page 507](#).

QmpTblFillRec

FILLS THE FIELD VALUES FROM THE CURRENT RECORD.

Syntax

```
void QmpTblFillRec ( MpHnd in_hTbl, MpHnd io_hRec, QRESULT*  
                     out_pResult );
```

in_hTbl
Handle to table. *Input*.

io_hRec
Record to fill from current record. *Input, Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpTblFillRec fills a record passed into the function with the field values from the current record.

Example

```
/* Fetch the input table record into the */  
/* intermediate record */  
QmpTblFillRec( hInputTable, himRecord, pResult );
```

QmpTblTxtCreate

Creates a fixed-width text merge/purge table object.

Syntax

```
MpHnd QmpTblTxtCreate ( const char* in_pszDatabaseName, const
                           char* in_pszTblName, QRESULT* out_pResult );
in_pszDatabaseName
    Database name. Input.
in_pszTblName
    Table name. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns handle to table object if successful, or NULL if there is an error.

Notes

The following table functions currently do not support fixed width text tables:

```
QmpTblGetAutoOpenIdx
QmpTblSetAutoOpenIdx
QmpTblDelRec
QmpTblSeekRec
QmpTblSeekVal
QmpTblCreateIdx
QmpTblDelIdx
QmpTblSetIdx
QmpTblGetIdxCnt
QmpTblFillIdxInfo
```

Example

```
QmpDeclHnd( hTable );
QmpDeclHnd( hIniFil );
QmpDeclHnd( hMyRec );
QmpDeclHnd( hLog );
const char* szFieldName;
hTable = QmpTblTxtCreate( "", "1000.sdf", &qres );
QmpTblOpen( hTable, &qres );
szFieldName = QmpTblGetFldName( hTable, 0, &qres );
printf( "Fieldname is: %s\n", szFieldName );
szFieldName = QmpTblGetFldName( hTable, 1, &qres );
printf( "Fieldname is: %s\n", szFieldName );

QmpTblClose( hTable, &qres );
QmpTblDestroy( hTable, &qres );
hMyRec = QmpRecCreate( &qres );
QmpRecAddByType( hMyRec, "First", QMS_VARIANT_STRING, 15, 0, &qres );
```

QmpTblTxtCreate

```
QmpRecAddByType( hMyRec, "Last", QMS_VARIANT_STRING, 15, 0, &qres );
QmpRecAddByType( hMyRec, "Addr", QMS_VARIANT_STRING, 35, 0, &qres );
hTableOut = QmpTblTxtCreate( "", "test.sdf", &qres );
QmpTblDefineFlds( hTableOut, hMyRec, &qres );
QmpTblCreateRep( hTableOut, &qres );
QmpTblClose( hTableOut, &qres );
QmpTblDestroy( hTableOut, &qres );
```

QmpTblGetAutoOpenIdx

FINDS OUT WHETHER INDEXES WILL BE OPENED WHEN THE TABLE IS OPENED.

Syntax

```
QBOOL QmpTblGetAutoOpenIdx ( MpHnd in_hTbl, QRESULT*  
    out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if indexes are automatically opened, else QFALSE.

Notes

If **QmpTblGetAutoOpenIdx** returns QTRUE, the index file associated with a table will be opened when the table is opened. Otherwise, the index file is not opened simultaneously with the table.

The default value for a table's index auto-open setting is QFALSE.

Example

```
bIsTblAutoOpenIdx = QmpTblGetAutoOpenIdx(hDITR_Table,  
    &Result);
```

QmpTblGetCnt

GETS THE NUMBER OF RECORDS IN THE TABLE.

Syntax

```
long QmpTblGetCnt ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
    Handle to table. Input.  
out_pResult  
    Result code. Output.
```

Return Value

Returns number of records if successful, or -1 if there is an error.

Notes

This function gets the number of records in the table.

Example

```
long lTableRecCount = QmpTblGetCnt (hTbl, pResult );
```

QmpTblGetDatabaseName

GETS NAME OF DATABASE.

Syntax

```
const char* QmpTblGetDatabaseName ( MpHnd in_hTbl, QRESULT*
```

```
    out_pResult );  
in_hTbl  
Handle to table. Input.
```

```
out_pResult  
Result code. Output.
```

Return Value

Returns pointer to database name if successful, or `NULL` if there is an error.

Notes

This function gets the name of the table's database.

Example

```
pDatabaseName = QmpTblGetDatabaseName (hTbl, pResult );
```

QmpTblGetExclusive

GETS EXCLUSIVE USE SETTING.

Syntax

```
QBOOL QmpTblGetExclusive ( MpHnd in_hTbl, QRESULT* out_pResult
) ;
```

in_hTbl

Handle to table. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if the table is set to exclusive use, else QFALSE.

Notes

If a file is set to exclusive use, it may be used by only one agent at a time. A file's default exclusive use setting is QTRUE.

Example

```
/* 10.h try to get exclusive access for the */
/* DITR enhancement */

ExclusiveFlg = QmpTblGetExclusive(DITR_Table, &Result);
if(!qmpSucceeded(Result)) {
    DisplayResultMsg(Result, "Unable to get exclusive state for
        DITR");
    QmpTblClose(Prepro_Table, &Result);
    return MPE_FAILURE;
}
```

QmpTblGetFldCnt

GETS THE NUMBER OF FIELDS IN A TABLE.

Syntax

```
long QmpTblGetFldCnt ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
    Handle to table. Input.  
out_pResult  
    Result code. Output.
```

Return Value

Returns the number of fields if successful, or -1 if there is an error.

Notes

QmpTblGetFldCnt returns the number of fields in a table record.

Example

```
/* 5. get the number of fields in input */  
ninp_fields = (int)QmpTblGetFldCnt(cbtbl_input, &Result);
```

QmpTblGetFldName

GETS THE FIELD NAME OF A PARTICULAR FIELD OF A TABLE

```
const char* QmpTblGetFldName( MpHnd in_hTbl, int in_iFld,
    QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_iFld
Field index. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns name of specified field if successful, or NULL if there is an error.

Notes

QmpTblGetFldName returns the name of the Nth table field, where N is the field index.

Table field name case is not preserved. For example, setting a field name to "ZIP Code" does not guarantee that the capital letters will be present if the field name is retrieved in the future.

Example

```
name = strdup (QmpTblGetFldName (cbtbl_input, i, &Result));
```

QmpTblGetFldNameVB

GET THE FIELD NAME OF A PARTICULAR FIELD OF A TABLE (VB VERSION)

Syntax

```
void QmpTblGetFldNameVB( MpHnd in_hTbl, int in_iFld, char*
    io_szBuffer, long in_lSize, QRESULT* out_pResult );
```

in_hTbl

Handle to table. *Input.*

in_iFld

Field index. *Input.*

io_szBuffer

Buffer to put name into. *Input, Output.*

in_lSize

Size of buffer. *Input.*

out_pResult

Result code. *Output.*

Return Value

None.

Notes

QmpTblGetFldNameVB provides the name of the Nth table field, where N is the field index.

Table field name case is not preserved. For example, setting a field name to “ZIP Code” does not guarantee that the capital letters will be present if the field name is retrieved in the future.

Example

```
QmpTblGetFldNameVB(hTbl, iFld, szBuffer, lSize, pResult );
```

QmpTblGetFlds

GETS THE FIELDS OF A TABLE.

Syntax

```
long QmpTblGetFlds ( MpHnd in_hTbl, MpHnd io_hRec, QRESULT*  
                      out_pResult );  
  
in_hTbl  
        Handle to table. Input.  
  
io_hRec  
        Record containing field definitions. Input, Output.  
  
out_pResult  
        Result code. Output.
```

Return Value

Returns the number of fields contained in the record if successful, or -1 if there is an error.

Notes

QmpTblGetFlds fills a record passed to the function with table field definitions.

Table field name case is not preserved. For example, setting a field name to "ZIP Code" does not guarantee that the capital letters will be present if the field name is retrieved in the future.

Example

```
QmpTblGetFlds( hInputTable, himRecord, pResult );
```

QmpTblGetIdxCnt

GETS THE NUMBER OF INDEXES FOR THIS TABLE.

Syntax

```
long QmpTblGetIdxCnt ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

Returns the number of indexes for this table if successful, or -1 if there is an error.

Notes

QmpTblGetFlds gets the number of indexes for a table.

Example

```
long lIndexNumber;  
QmpDeclHnd(hTbl);  
QRESULT Result;  
  
lIndexNumber = QmpTblGetIdxCnt ( hTbl, &Result );
```

QmpTblGetPos

GETS THE CURRENT POSITION IN THE TABLE.

Syntax

```
long QmpTblGetPos ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the current position in the table if successful, or -1 if there is an error.

Notes

Example

```
long lCurrentTablePos = QmpTblGetPos (hTbl, pResult );
```

QmpTblGetReadOnly

GETS READ-ONLY SETTING.

Syntax

```
QBOOL QmpTblGetReadOnly ( MpHnd in_hTbl, QRESULT* out_pResult
) ;
```

in_hTbl

Handle to table. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if table is set to read-only, QFALSE if table is writable or if an error occurs.

Notes

When specifying a table to receive output data, you should be certain that the table is not read-only. This function provides a means to check the read/write status of a table.

See Also

[QmpTblSetReadOnly](#)

Example

```
/* can't modify if its read-only */
/* 10.i ensure DITR is not read-only */
if(QmpTblGetReadOnly(DITR_Table, &Result) ||
   !qmpSucceeded(Result)) {
    DisplayResultMsg(Result, "DITR is read only, or state is
unknown");
    QmpTblClose(Prepro_Table, &Result);
    return MPE_FAILURE;
}
```

QmpTblGetTblName

GETS NAME OF TABLE.

Syntax

```
const char* QmpTblGetTblName ( MpHnd in_hTbl, QRESULT*  
    out_pResult );
```

in_hTbl

Handle to table. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns pointer to table name if successful, or `NULL` if there is an error.

Example

```
strcat( szTempName, QmpTblGetTblName( hDITR, pResult ) );
```

QmpTblIsOpen

TESTS WHETHER TABLE IS OPEN.

Syntax

```
QBOOL QmpTblIsOpen ( MpHnd in_hTbl, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns QTRUE if table is open, QFALSE if closed or if an error occurs.

Example

```
bIsTblOpen = QmpTblIsOpen(hPrepro_Table, &Result);
```

QmpTblMoveBy

MOVES BY AN OFFSET AMOUNT.

Syntax

```
void QmpTblMoveBy ( MpHnd in_hTbl, long in_lOffset, QRESULT*  
                     out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_lOffset
Position offset amount. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpTblMoveBy moves the table record pointer by an offset amount. Positive offset values move the pointer forward, negative values move the pointer backward. The index is respected when moving the pointer.

Example

```
QmpTblMoveBy ( hTblInput, 1, &qres );
```

QmpTblMoveFirst

MOVES TO FIRST RECORD IN TABLE.

Syntax

```
void QmpTblMoveFirst ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

QmpTblMoveFirst moves the table record pointer to the first record in the table. The index is used for this operation.

Example

```
QmpTblMoveFirst ( hTblInput, &qres );
```

QmpTblMoveLast

MOVES TO LAST RECORD IN TABLE.

Syntax

```
void QmpTblMoveLast ( MpHnd in_hTbl, QRESULT* out_pResult );  
in_hTbl  
Handle to table. Input.  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

QmpTblMoveLast moves the table record pointer to the last record in the table. The index is used for this operation.

Example

```
QmpTblMoveLast (hTbl, pResult );
```

QmpTblMoveTo

MOVES TO A PARTICULAR RECORD.

Syntax

```
void QmpTblMoveTo ( MpHnd in_hTbl, long in_lPos, QRESULT*  
                    out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_lPos
Position (record) to move to. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpTblMoveTo moves the record pointer inside a table. The index is not used for this operation.

Example

```
QmpTblMoveTo (hTbl, lPos, pResult );
```

QmpTblOpen

OPENS THE SPECIFIED TABLE.

Syntax

```
void QmpTblOpen ( MpHnd in_hTbl, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Calling **QmpTblOpen** actually creates a physical table file. Before this function is called, the table exists only in a Centrus Merge/Purge internal representation.

Example

```
/* SetupDatabase - Open the database and its tables. */
/* Associate indexes with the tables */

void SetupDatabase ( )
{
    QmpTblSetDatabaseName( hInputTable,
        "\\\\Builder\\big23\\Development\\Merge-
        Purge\\Development\\Data\\TestData.mdb", pResult );
    QmpTblSetTblName( hInputTable, szTestTableName, pResult );
    QmpTblOpen( hInputTable, pResult );
```

QmpTblOrclCreate

CREATES AN ORACLE MERGE/PURGE TABLE OBJECT.

Syntax

```
MpHnd QmpTblOrclCreate (const char* in_szTblName, const char*
    in_szParams, QRESULT* out_pResult );

in_szTblName
    Name of the Oracle table. Input.

in_szParams
    Connection string. Input.

out_pResult
    Result code. Output.
```

Return Value

Returns a handle to the created table.

Notes

Oracle databases require a connection string in the format “UserName/Password@ConnectionString”. Centrus Merge-Purge Oracle tables require that the location of the Oracle driver be provided. Centrus Merge-Purge Oracle tables do not use the database name since this makes no sense in the Oracle “tablespace” scheme.

The Oracle create method will fail if memory cannot be allocated. Errors using the table will generally not occur until the table is used, it is opened, or its existence is checked.

See Also

[QmpTblCbCreate](#), [QmpTblDTxtCreate](#), [QmpTblFTxtCreate](#),
[QmpTblOrclGetCache](#), [QmpTblOrclSetCache](#).

Example

```
QmpDeclHnd( hTable );
const char *szFieldName;
hTable = QmpTblOrclCreate( "MyTable",
    "user/password@service", &qres );

QmpTblOpen( hTable, &qres );
szFieldName = QmpTblGetFldName( hTable, 0, &qres );
printf( "Fieldname is: %s\n", szFieldName );
szFieldName = QmpTblGetFldName( hTable, 1, &qres );
printf( "Fieldname is: %s\n", szFieldName );
QmpTblClose( hTable, &qres );
```

QmpTblOrclGetCache

GETS THE CACHE SIZE OF AN ORACLE MERGE/PURGE TABLE OBJECT.

Syntax

```
int QmpTblOrclGetCache (MpHnd in_hTbl, QRESULT* out_pResult);
```

in_hTbl
Handle to table. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the Oracle cache size for the table if successful, else 0.

Notes

Centrus Merge-Purge Oracle tables have an internal cache size that can be used to improve their performance. The cache size is a positive integer that indicates the number of records (Oracle rows) that will be cached for reading and writing.

Use **QmpTblOrclSetCache** to set the cache size to something other than the default. The Oracle get cache method will fail if the input table handle is invalid.

See Also

QmpTblOrclSetCache, **QmpTblOrclCreate**.

Example

```
QmpDeclHnd( hTable );  
  
hTable = QmpTblOrclCreate( "MyTable", "user/password@service",  
    &qres );  
iCacheSize = QmpTblOrclGetCache( hTable, &qres );  
printf( " Before setting, cache size is %d\n", iCacheSize );  
QmpTblOrclSetCache( hTable, 250, &qres );  
QmpTblOpen( hTable, &qres );  
iCacheSize = QmpTblOrclGetCache( hTable, &qres );  
printf( " Cache size is %d\n", iCacheSize );  
QmpTblClose( hTable, &qres );
```

QmpTblOrclSetCache

SETS THE CACHE SIZE OF AN ORACLE MERGE/PURGE TABLE OBJECT.

Syntax

```
void QmpTblOrclSetCache (MpHnd in_hTbl, int in_iCacheSize,
                        QRESULT* out_pResult);

in_hTbl
    Handle to table. Input.

in_iCacheSize
    Cache size. Value must be between 100 and 32,767. Input.

out_pResult
    Result code. Output.
```

Return Value

None.

Notes

Centrus Merge-Purge Oracle tables have an internal cache size that can be used to improve their performance. The cache size is a positive integer that indicates the number of records (Oracle rows) that will be cached for reading and writing.

This method should be called before the table has been opened (using **QmpTblOpen**) or created (using **QmpTblCreateRep**). This method should not be called while a table is currently being read from or written to.

It is not necessary to set the cache size for a table. If this method is not called for a table, the default internal cache size (100) will be used.

If you want to traverse an oracle table backwards, or if you periodically skip backwards in a table while reading, make your cache as large as possible for best performance. The Oracle set cache method will fail if the input table handle is invalid or if the cache size is invalid.

See Also

QmpTblOrclGetCache, **QmpTblOrclCreate**.

Example

```
QmpDeclHnd( hTable );

hTable = QmpTblOrclCreate( "MyTable",
                           "user/password@service", &qres );
QmpTblOrclSetCache( hTable, 250, &qres );
QmpTblOpen( hTable, &qres );
QmpTblClose( hTable, &qres );
```

QmpTblSeekRec

SEEKS A SPECIFIED RECORD.

Syntax

```
QBOOL QmpTblSeekRec ( MpHnd in_hTbl, const char*
    in_pszOperator, MpHnd in_hRec, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
in_pszOperator
    Pointer to operator to apply. Input.
    Currently only the “=” operator is supported.
in_hRec
    Record containing key values to use in seek. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns QTRUE if specified record is found, else QFALSE.

Example

```
/* seek to the duplicate in the DITR */
/* seek by equality */
if(QmpTblSeekRec(DITR_Table, "=", hSeekRecord, &Result) &&
    qmpSucceeded(Result)) {
    /* get data from the DITR */
    QmpTblFillRec(DITR_Table, hRecord, &Result);
    if(!qmpSucceeded(Result)) {
        DisplayResultMsg(Result, "Failed to fill in the dupe "
            "record from the DITR");
        QmpDupGrpRecDestroy(hDupeRecord, &Result);
        QmpRecDestroy(hRecord, &Result);
        QmpRecDestroy(hSeekRecord, &Result);
        QmpRecDestroy(hMasterRec, &Result);
        QmpDupGrpRecDestroy(hMasterDupeRec, &Result);
        QmpTblClose(DITR_Table, &Result);
        free(DupeGroupIDs);
        return MPE_FAILURE;
    }
}
```

QmpTblSeekVal

SEEKS A PARTICULAR KEY VALUE.

Syntax

```
QBOOL QmpTblSeekVal ( MpHnd in_hTbl, const char*
    in_pszOperator, const char* in_pszKeyVal, QRESULT*
    out_pResult );
```

in_hTbl

Handle to table. *Input*.

in_pszOperator

Operator to apply. *Input*.

Currently only the “=” operator is supported.

in_pszKeyVal

Actual key value being sought. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if specified record is found, else QFALSE.

Notes

QmpTblSeekVal searches for a record based upon the operator and key value passed into the function. The key is the “seek expression” which specifies what is being sought.

QmpTblSeekVal does not support text table searching.

For example, to seek a record based on the primary key and source ID field values, you would construct a key expression based on the field lengths and values, and pass the key into **QmpTblSeekVal**.

If the record being sought had the following field information:

Field	Field Length	Field Value
primary key	11	22
source ID	5	31

The key value string would look like this (underscores represent blank characters):

“_____22___31”

The seek also requires an index that contains the same field information as the key value string.

Example

```
QBOOL bIsRecordFound = QmpTblSeekVal (hTbl, pszOperator,  
    pszKeyVal, pResult );
```

QmpTblSetAutoOpenIdx

ENABLES THE INDEXES TO BE OPENED WHEN THE TABLE IS OPENED.

Syntax

```
void QmpTblSetAutoOpenIdx ( MpHnd in_hTbl, QBOOL
                           in_bAutoOpen, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
in_bAutoOpen
    Auto open index. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

If *in_bAutoOpen* = QTRUE, the index file associated with a table will be opened when the table is opened. Otherwise, the index file is not opened simultaneously with the table.

A table *must* be closed before calling **QmpTblSetAutoOpenIdx**.

The default value for a table's index auto-open setting is QFALSE.

Example

```
/* 10.g set auto open index */
QmpTblSetAutoOpenIdx(DITR_Table, QTRUE, &Result);
if(!qmpSucceeded(Result)) {

    /* auto open index has failed */
    DisplayResultMsg(Result, "Unable to set auto open index to
                           true");
    QmpTblClose(Prepro_Table, &Result);
    return MPE_FAILURE;
}
```

QmpTblSetDatabaseName

SETS NAME OF DATABASE.

Syntax

```
void QmpTblSetDatabaseName ( MpHnd in_hTbl, char* in_pszName,
                           QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_pszName
Name of database. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

For .dbf and text databases, the database name refers to the folder in which the table resides.

Example

```
QmpTblSetDatabaseName ( hInputTable,
                           "\\\\[Builder]\\big23\\Development\\Merge-
                           Purge\\Development\\Data\\TestData.mdb", pResult );
```

QmpTblSetExclusive

SETS EXCLUSIVE USE SETTING.

Syntax

```
void QmpTblSetExclusive ( MpHnd in_hTbl, QBOOL in_bExclusive,
                         QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
in_bExclusive
    Set table exclusive use to on (QTRUE) or off (QFALSE). Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

If a file is set to exclusive use, it may be used by only one agent at a time. A file's default exclusive use setting is QTRUE.

Example

```
/* clean up and close out */
/* set non-exclusive access */
QmpTblSetExclusive(DITR_Table, ExclusiveFlg, &Result);
if(!qmpSucceeded(Result)) {
    DisplayResultMsg(Result, "Failed to return exclusive setting
        for DITR, " "may lead to problems later");
}
QmpTblClose(Prepro_Table, &Result);
QmpTblSetReadOnly(Prepro_Table, ReadOnlyFlg, &Result);
QmpTblClose(DITR_Table, &Result);
DisplayMsg("Enhanced %d records", n_enh);
return MPE_SUCCESS;
```

QmpTblSetIdx

SETS THE CURRENT INDEX.

Syntax

```
void QmpTblSetIdx ( MpHnd in_hTbl, const char*
                      in_pszIdxName, QRESULTT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_pszIdxName
Index name. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpTblSetIdx sets the current index for the table that is passed into the function. Setting the current index causes the table fields to be ordered by the index expression.

Example

```
QmpTblSetIdx (hTbl, pszIdxName, pResult );
```

QmpTblSetMappedRec

SETS A MAPPED RECORD.

Syntax

```
void QmpTblSetMappedRec( MpHnd in_hTbl, MpHnd in_hFldMap,
                         MpHnd in_hRec, QRESULT* out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_hFldMap
Handle to field map. *Input*.

in_hRec
Handle of record to set. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Call `QmpTblSetMappedRec` to fill a record in the table with data from the provided record. The table record will be filled according to the provided field map.

Example

See example on [page 507](#).

QmpTblSetReadOnly

SETS SPECIFIED TABLE TO READ ONLY.

Syntax

```
void QmpTblSetReadOnly ( MpHnd in_hTbl, QBOOL in_bReadOnly,
    QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
in_bReadOnly
    Set table read only setting to on (QTRUE) or off (QFALSE). Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

When specifying a table to receive output data, you should be certain that the table is not read-only. This function provides a means to set the read/write status of a table.

Example

```
/* clean up and close out */
/* set non-exclusive access */
QmpTblSetExclusive(DITR_Table, ExclusiveFlg, &Result);
if(!qmpSucceeded(Result)) {
    DisplayResultMsg(Result, "Failed to return exclusive setting
        for DITR,
        "may lead to problems later");
}
QmpTblClose(Prepro_Table, &Result);
QmpTblSetReadOnly(Prepro_Table, ReadOnlyFlg, &Result);
QmpTblClose(DITR_Table, &Result);
DisplayMsg("Enhanced %ld records", n_enh);
return MPE_SUCCESS;
```

QmpTblSetRec

SETS THE FIELD VALUES IN THE CURRENT RECORD.

Syntax

```
void QmpTblSetRec ( MpHnd in_hTbl, MpHnd in_hRec, QRESULT*  
                     out_pResult );
```

in_hTbl
Handle to table. *Input*.

in_hRec
Record to set the current record from. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpTblSetRec sets the field values of the current record using the values of another record. The current record is located in the table that is passed to the function.

The record passed in to the function must have the same or fewer number of fields as the table. It is assumed that the fields of the two records occur in the same order.

Example

```
/* set record for new input */  
/* we have seeked to the record, now set it with */  
/* our newly modified hDupeRecord */  
QmpTblSetRec(DITR_Table, hRecord, &Result);  
if(!qmpSucceeded(Result)) {  
    DisplayResultMsg(Result, "Unable to set updated record in "  
                    "DITR, enhancement " "incomplete at dupe group %ld", group);  
}
```

QmpTblSetTblName

SETS NAME OF TABLE.

Syntax

```
void QmpTblSetTblName ( MpHnd in_hTbl, const char* in_pszName,  
                        QRESULT* out_pResult );  
  
in_hTbl  
Handle to table. Input.  
  
in_pszName  
Table name. Input.  
  
out_pResult  
Result code. Output.
```

Return Value

None.

Notes

QmpTblSetTblName should be called after the table object has been created, but *before* it is opened. Calling **QmpTblSetTblName** specifies the table that an object uses; the call does not change the physical table name.

If the name has no path, it will be placed in the current working directory. Otherwise, the path must exist. Setting the table name does not check if the folder exists; the check occurs when opening the file.

Example

```
QmpTblSetTblName ( hInputTable, szTestTableName, pResult );
```

QmpTblSetWhereClause

SETS THE WHERE CLAUSE OF A RDBMS TABLE (DOES NOTHING FOR FILE-BASED TABLES).

Syntax

```
void QmpTblSetWhereClause ( MpHnd in_hTbl, const char*
                            in_szWhereClause, QRESULT* out_pResult );
in_hTbl
    Handle to table. Input.
in_szWhereClause
    SQL query—do not include the keyword “where”. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

QmpTblSetWhereClause allows you to search an Oracle database using a SQL query.

QmpTblSetWhereClause must be called before opening a table (it won’t work if the table is open).

Note: Users of this function bear the responsibility of knowing SQL. Incorrectly using SQL statements in this function will yield strange or incorrect results. CMP does not verify the correctness of the SQL expression.

Example

```
const char* szSqlConnect = "scott/tiger@my_db";

QmpDeclHnd( hTable );
hTable = QmpTblOrclCreate( "customers", szSqlConnect, &qres );

// set where clause before opening table
QmpTblSetWhereClause( hTable, "zipcode = '90210' or zipcode =
    '69301'", &qres );
QmpTblOpen( hTable, &qres );

// count of records in table is only records meeting
// the where clause
unsigned long count = QmpTblGetCnt( hTable, &qres );

//... do other table operations as usual (MoveFirst,
// MoveNext, etc.)

QmpTblClose( hTable );
```


Function Class: QmpTblGen*

TABLE GENERATION PHASE FUNCTIONS.

Quick Reference

Function	Description	Page
QmpTblGenAddPreProcDataSrc	Specifies the preprocessed data source to be used by the table generation phase.*	828
QmpTblGenClear	Clears the table generation object (Sets back to its initial state).	829
QmpTblGenCreate	Creates a table generation object.	830
QmpTblGenGetDataDestCount	Gets number of data destinations in the table generation phase.	831
QmpTblGenIsValid	Tests whether a table generation object is valid.	832
QmpTblGenRemove	Removes a data destination from the table generation phase.	833
QmpTblGenRemPreProcDataSrc	Removes a preprocessed data source from the table generation phase.	834
QmpTblGenUseCons	Gives the data consolidation phase to the table generation phase.	835
QmpTblGenUseDataDest	Attaches a data destination object to the table generator.	836
QmpTblGenUseDataLstSvc	Specifies the data list service object to be used by the table generation phase.	837
QmpTblGenUseDupeGroup	Attaches a dupe groups object to the table generation phase.	838

QmpTblGenAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE TABLE GENERATION PHASE.*

Syntax

```
void QmpTblGenAddPreProcDataSrc( MpHnd in_hTableGen, MpHnd
                                in_DataSource, QRESULT* out_pResult );
in_hTableGen
    Handle to table generator object. Input.
in_DataSource
    Handle to preprocessed data source to be used. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the table generation phase.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

Example

```
/* Give the data destinations to the table generation */
/* phase */

QmpTblGenUseDataDest( hTableGen, hDataDestUniques, &qres );
QmpTblGenUseDataDest( hTableGen, hDataDestMasters, &qres );
QmpTblGenUseDataDest( hTableGen, hDataDestSubords, &qres );
QmpTblGenUseDataDest( hTableGen, hDataDestSupp, &qres );
QmpTblGenAddPreProcDataSrc( hTableGen, hPreProDataSrc, &qres );
lDestCount = QmpTblGenGetDataDestCount( hTableGen, &qres );
printf( "Starting Table Generation phase:\n" );
QmpPhaseStart( hTableGen, &qres );
```

QmpTblGenClear

CLEAR THE TABLE GENERATION OBJECT (SETS BACK TO ITS INITIAL STATE).

Syntax

```
void QmpTblGenClear ( MpHnd in_hTableGen, QRESULT* out_pResult
) ;
```

in_hTableGen
Handle to table generator. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function sets the table generation object back to its initial state.
None of the resources used by this object will be destroyed, but any references to them will be removed from the object.

Example

```
QmpTblGenClear ( hTableGen, pResult );
```

QmpTblGenCreate

CREATES A TABLE GENERATION OBJECT.

Syntax

```
MpHnd QmpTblGenCreate(QRESULT* out_pResult);  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to table generator if successful, or NULL if there is an error.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
hTableGenerator = QmpTblGenCreate( hCMPJob, pResult );
```

QmpTblGenGetDataDestCount

GETS NUMBER OF DATA DESTINATIONS IN THE TABLE GENERATION PHASE.

Syntax

```
long QmpTblGenGetDataDestCount( MpHnd in_hTableGen, QRESULT*  
                                out_pResult );
```

in_hTableGen
Handle to table generator. *Input*.

out_pResult
Result code. *Output*.

Return Value

Returns the number of data destination objects attached to the table generation phase if successful, or -1 if there is an error.

Notes

A data destination object encapsulates an output table. Every data destination object attached to the table generation phase represents an output table that will be generated by the job.

Example

```
/* Give the data destinations to the table generation */  
/* phase */  
  
QmpTblGenUseDataDest( hTableGen, hDataDestUniques, &qres );  
QmpTblGenUseDataDest( hTableGen, hDataDestMasters, &qres );  
QmpTblGenUseDataDest( hTableGen, hDataDestSubords, &qres );  
QmpTblGenUseDataDest( hTableGen, hDataDestSupp, &qres );  
QmpTblGenAddPreProcDataSrc( hTableGen, hPreProDataSrc, &qres );  
lDestCount = QmpTblGenGetDataDestCount( hTableGen, &qres );  
printf( "Starting Table Generation phase:\n" );  
QmpPhaseStart( hTableGen, &qres );
```

QmpTblGenIsValid

TESTS WHETHER A TABLE GENERATION OBJECT IS VALID.

Syntax

```
QBOOL QmpTblGenIsValid( MpHnd in_hTblGen, QRESULT* out_pResult  
);
```

in_hTblGen

Handle to table generator. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns QTRUE if table generator object is valid, else QFALSE.

Example

```
/* 11.n if table generation is valid, run table */  
/* generation. */  
  
if(QmpTblGenIsValid(hTableGen, &Result)) {  
    QmpPhaseStart(hTableGen, &Result1);  
}
```

QmpTblGenRemove

REMOVES A DATA DESTINATION FROM THE TABLE GENERATION PHASE.

Syntax

```
MpHnd QmpTblGenRemove( MpHnd in_hTblGen, MpHnd in_hDataDest,  
                           QRESULT* out_pResult );
```

in_hTblGen

Handle to table generator. *Input*.

in_hDataDest

Handle to data destination object to remove. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns handle to removed data destination object if successful, or NULL if there is an error.

Notes

QmpTblGenRemove removes a data destination object from the table generator and returns the object handle.

Example

```
hRemDataDest = QmpTblGenRemove(hTblGen, hDataDest, pResult);
```

QmpTblGenRemPreProcDataSrc

REMOVES A PREPROCESSED DATA SOURCE FROM THE TABLE GENERATION PHASE.

Syntax

```
void QmpTblGenRemPreProcDataSrc ( MpHnd in_hTblGen, long  
                                  in_lDataSourceID, QRESULT* out_pResult );  
  
in_hTblGen  
        Handle to table generator. Input.  
in_lDataSourceID  
        Index to preprocessed data source to be removed. Input.  
out_pResult  
        Result code. Output.
```

Return Value

None.

Notes

Allows an application to remove a preprocessed data source from the table generation phase. The table generation phase currently can accept only one preprocessed data source. If you add a data source, you must remove it before adding another.

See Also

[QmpTblGenAddPreProcDataSrc](#).

Example

```
QmpTblGenRemPreProcDataSrc (hTblGen, lDataSourceID, pResult  
                           );
```

QmpTblGenUseCons

GIVES THE DATA CONSOLIDATION PHASE TO THE TABLE GENERATION PHASE.

Syntax

```
void QmpTblGenUseCons( MpHnd in_hTblGen, MpHnd in_hCons,
                      QRESULT* out_pResult );
in_hTblGen
    Handle to table generation phase. Input.
in_hCons
    Handle to data consolidation phase. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The table generation phase requires the data consolidation phase for consolidated record output and updated duplicates.

Example

```
QmpTblGenUseCons ( hTblGen, hCons, &qres );
```

QmpTblGenUseDataDest

ATTACHES A DATA DESTINATION OBJECT TO THE TABLE GENERATOR.

Syntax

```
void QmpTblGenUseDataDest( MpHnd in_hTblGen, MpHnd
                           in_hDataDest, QRESULT* out_pResult );
in_hTblGen
    Handle to table generator. Input.
in_hDataDest
    Handle to data destination object to use. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

A data destination object encapsulates an output table. Every data destination object attached to the table generation phase represents an output table that will be generated by the job.

A table generator can have multiple data destination objects attached to it. Call **QmpTblGenUseDataDest** for each data destination to be attached.

Example

```
/* Give the data destinations to the table generation */
/* phase otherwise configure table generation phase */

QmpTblGenUseDataDest( hTableGenerator, hDataDestInclUniques,
                      pResult );
QmpTblGenUseDataDest( hTableGenerator, hDataDestSuppUniques,
                      pResult );
QmpTblGenUseDataDest( hTableGenerator, hDataDestInclMasters,
                      pResult );
QmpTblGenUseDataDest( hTableGenerator, hDataDestSuppMasters,
                      pResult );
QmpTblGenUseDataDest( hTableGenerator, hDataDestInclSubords,
                      pResult );
QmpTblGenUseDataDest( hTableGenerator, hDataDestSuppSubords,
                      pResult );
```

QmpTblGenUseDataLstSvc

SPECIFIES THE DATA LIST SERVICE OBJECT TO BE USED BY THE TABLE GENERATION PHASE.

Syntax

```
void QmpTblGenUseDataLstSvc( MpHnd in_hTblGen, MpHnd
    in_hDataLstSvc, QRESULT* out_pResult );
in_hTblGen
    Handle to table generator. Input.
in_hDataLstSvc
    Handle to data list service object. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The data list service comprises one or more data lists objects. A data list object contains the specification of the criteria for assigning a record membership to a single data list. An application typically uses several data lists in a single data list service. The data list service manages and coordinates the activities of the collection of data lists contained in it.

Example

```
/* Give table generator the data list service object */

QmpTblGenUseDataLstSvc ( hTableGenerator, hDataListService,
    pResult );
```

QmpTblGenUseDupeGroup

ATTACHES A DUPE GROUPS OBJECT TO THE TABLE GENERATION PHASE.

Syntax

```
void QmpTblGenUseDupeGroup( MpHnd in_hTblGen, MpHnd  
    in_hDupeGroups, QRESULT* out_pResult );  
  
in_hTblGen  
    Handle to table generator. Input.  
  
in_hDupeGroups  
    Handle to dupe groups object to use. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

None.

Notes

The table generation phase uses the dupe groups object to determine table membership.

Example

```
/* Give the table generator the dupe groups object */  
QmpTblGenUseDupeGroup( hTableGen, hDupeGroups, &qres );
```

Function Class: QmpTime*

FUNCTIONS FOR MANIPULATING TIME OBJECTS.

Timer objects are used as “stopwatches” in a Centrus Merge/Purge application. They are useful for determining the execution time for tracts of code, as well as for determining when parts of code start and stop executing.

Using a timer object consists of creating the object, then starting and stopping it at the appropriate times. Query library functions, such as `QmpTimeGetElapsedTime` and `QmpTimeGetStopTime`, are used to retrieve the timer values from the object.

Phase objects inherit the attributes of the timer object, and are timers themselves. (See Figure 5: “Centrus Merge/Purge Object Hierarchy” on [page 65](#).) Once a phase has been created, its handle may be passed to the `QmpTime*` functions to retrieve the phase start time, stop time, elapsed time, etc.

Quick Reference

Function	Description	Page
<code>QmpTimeCreate</code>	Creates a timer.	840
<code>QmpTimeDestroy</code>	Destroys a timer.	842
<code>QmpTimeGetElapsedTime</code>	Gets elapsed time.	844
<code>QmpTimeGetElapsedTimeToNow</code>	Gets elapsed time to now.	846
<code>QmpTimeGetStartTime</code>	Gets start time.	848
<code>QmpTimeGetStopTime</code>	Gets stop time.	850
<code>QmpTimeIsValid</code>	Tests whether a timer is valid.	852
<code>QmpTimeStartTiming</code>	Starts a timer object.	853
<code>QmpTimeStopTiming</code>	Stops a timer object.	855

QmpTimeCreate

CREATES A TIMER.

Syntax

```
MpHnd QmpTimeCreate ( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns handle to timer if successful, or NULL if there is an error.

Notes

This function creates a timer object.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

See Also

[QmpTimeDestroy](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
unsigned short* pusElapsedMinutes;
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */
QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
    pResult );
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,
    pusElapsedMinutes, pusElapsedSeconds, pResult );
```

```
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
                     pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,
                     pResult );
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeDestroy

DESTROYS A TIMER.

Syntax

```
void QmpTimeDestroy ( MpHnd in_hTime, QRESULT* out_pResult );
```

in_hTime
Handle to timer. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

See Also

`QmpTimeCreate`

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
unsigned short* pusElapsedMinutes;
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */

QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
```

```
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
                     pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
                     pResult );
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,
                       pusElapsedMinutes, pusElapsedSeconds, pResult );
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
                     pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,
                     pResult );
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeGetElapsedTime

GETS ELAPSED TIME.

Syntax

```
void QmpTimeGetElapsedTime (MpHnd in_hTime, unsigned short*
                           out_pusElapsedDays, unsigned short* out_pusElapsedHours,
                           unsigned short* out_pusElapsedMinutes, unsigned short*
                           out_pusElapsedSeconds, QRESULT* out_pResult );
```

in_hTime
Handle to timer. *Input*.

out_pusElapsedDays
Elapsed days. *Output*.

out_pusElapsedHours
Elapsed hours. *Output*.

out_pusElapsedMinutes
Elapsed minutes. *Output*.

out_pusElapsedSeconds
Elapsed seconds. *Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function returns the elapsed time between the time the timer object was started and stopped.

See Also

[QmpTimeGetElapsedTimeToNow](#), [QmpTimeGetStartTime](#),
[QmpTimeGetStopTime](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
unsigned short* pusElapsedMinutes;
```

```
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */

QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
    pResult );
QmpTimeGetElapsedTime (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
    pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,
    pResult );
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeGetElapsedTimeToNow

GETS ELAPSED TIME TO NOW.

Syntax

```
void QmpTimeGetElapsedTimeToNow ( MpHnd in_hTime, unsigned
                                short* out_pusElapsedDays, unsigned short*
                                out_pusElapsedHours, unsigned short* out_pusElapsedMinutes,
                                unsigned short* out_pusElapsedSeconds, QRESULT* out_pResult
                                );
```

in_hTime
Handle to timer. *Input*.

out_pusElapsedDays
Elapsed days. *Output*.

out_pusElapsedHours
Elapsed hours. *Output*.

out_pusElapsedMinutes
Elapsed minutes. *Output*.

out_pusElapsedSeconds
Elapsed seconds. *Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function returns the elapsed time between the time the timer object was started and the time this function was invoked.

See Also

[QmpTimeGetElapsedTime](#), [QmpTimeGetStartTime](#),
[QmpTimeGetStopTime](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
```

```
unsigned short* pusElapsedMinutes;
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */

QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
    pResult );
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,
    pusElapsedMinutes, pusElapsedSeconds, pResult );
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
    pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,
    pResult );
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeGetStartTime

GETS START TIME.

Syntax

```
void QmpTimeGetStartTime ( MpHnd in_hTime, unsigned short*
    out_pusStartYear, unsigned short* out_pusStartMonth,
    unsigned short* out_pusStartDay, unsigned short*
    out_pusStartHour, unsigned short* out_pusStartMinute,
    unsigned short* out_pusStartSecond, QRESULT* out_pResult );
```

in_hTime
Handle to timer. *Input*.

out_pusStartYear
Start year. *Output*.

out_pusStartMonth
Start month. *Output*.

out_pusStartDay
Start day. *Output*.

out_pusStartHour
Start hour. *Output*.

out_pusStartMinute
Start minute. *Output*.

out_pusStartSecond
Start second. *Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function returns the time at which the timer object was started.

See Also

[QmpTimeGetElapsedTime](#), [QmpTimeGetElapsedTimeToNow](#),
[QmpTimeGetStopTime](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
```

```
unsigned short* pusStartSecond, pusStopSecond;  
  
unsigned short* pusElapsedDays;  
unsigned short* pusElapsedHours;  
unsigned short* pusElapsedMinutes;  
unsigned short* pusElapsedSeconds;  
  
hTimer = QmpTimeCreate (pResult);  
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );  
QmpTimeStartTiming (hTimer, pResult );  
  
/* other application code here */  
  
QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,  
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,  
    pResult );  
QmpTimeStopTiming (hTimer, pResult );  
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,  
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,  
    pResult );  
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,  
    pusElapsedMinutes, pusElapsedSeconds, pResult );  
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,  
    pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,  
    pResult );  
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeGetStopTime

GETS STOP TIME.

Syntax

```
void QmpTimeGetStopTime ( MpHnd in_hTime, unsigned short*
    out_pusStopYear, unsigned short* out_pusStopMonth, unsigned
    short* out_pusStopDay, unsigned short* out_pusStopHour,
    unsigned short* out_pusStopMinute, unsigned short*
    out_pusStopSecond, QRESULT* out_pResult );
```

in_hTime
Handle to timer. *Input*.

out_pusStopYear
Stop year. *Output*.

out_pusStopMonth
Stop month. *Output*.

out_pusStopDay
Stop day. *Output*.

out_pusStopHour
Stop hour. *Output*.

out_pusStopMinute
Stop minute. *Output*.

out_pusStopSecond
Stop second. *Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function returns the time at which the timer object was stopped.

See Also

[QmpTimeGetElapsedTime](#), [QmpTimeGetElapsedTimeToNow](#),
[QmpTimeGetStartTime](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
```

```
unsigned short* pusStartSecond, pusStopSecond;  
  
unsigned short* pusElapsedDays;  
unsigned short* pusElapsedHours;  
unsigned short* pusElapsedMinutes;  
unsigned short* pusElapsedSeconds;  
  
hTimer = QmpTimeCreate (pResult);  
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );  
QmpTimeStartTiming (hTimer, pResult );  
  
/* other application code here */  
  
QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,  
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,  
    pResult );  
QmpTimeStopTiming (hTimer, pResult );  
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,  
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,  
    pResult );  
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,  
    pusElapsedMinutes, pusElapsedSeconds, pResult );  
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,  
    pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,  
    pResult );  
QmpTimeDestroy (hTimer, pResult );
```

QmpTimelsValid

TESTS WHETHER A TIMER IS VALID.

Syntax

```
QBOOL QmpTimeIsValid ( MpHnd in_hTime, QRESULT* out_pResult );
in_hTime
    Handle to timer. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns QTRUE if timer is valid, else QFALSE.

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
unsigned short* pusElapsedMinutes;
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */

QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
    pResult );
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,
    pusElapsedMinutes, pusElapsedSeconds, pResult );
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
    pusStopDay, pusStopHour, pusStopMinute, pusStopSecond,
    pResult );
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeStartTiming

STARTS A TIMER OBJECT.

Syntax

```
void QmpTimeStartTiming ( MpHnd in_hTime, QRESULT* out_pResult
    );
in_hTime
    Handle to timer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function starts the timer object.

See Also

[QmpTimeStopTiming](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
unsigned short* pusElapsedMinutes;
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */

QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
    pResult );
```

QmpTimeStartTiming

```
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,
    pusElapsedMinutes, pusElapsedSeconds, pResult );
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
    pusStopDay, pussStopHour, pussStopMinute, pussStopSecond,
    pResult );
QmpTimeDestroy (hTimer, pResult );
```

QmpTimeStopTiming

STOPS A TIMER OBJECT.

Syntax

```
void QmpTimeStopTiming ( MpHnd in_hTime, QRESULT* out_pResult
    );
in_hTime
    Handle to timer. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function stops the timer object.

See Also

[QmpTimeStartTiming](#)

Example

```
QmpDeclHnd(hTimer);
QRESULT* pResult;
QBOOL bIsTimerValid;

unsigned short* pusStartYear, pusStopYear;
unsigned short* pusStartMonth, pusStopMonth;
unsigned short* pusStartDay, pusStopDay;
unsigned short* pusStartHour, pusStopHour;
unsigned short* pusStartMinute, pusStopMinute;
unsigned short* pusStartSecond, pusStopSecond;

unsigned short* pusElapsedDays;
unsigned short* pusElapsedHours;
unsigned short* pusElapsedMinutes;
unsigned short* pusElapsedSeconds;

hTimer = QmpTimeCreate (pResult);
bIsTimerValid = QmpTimeIsValid (hTimer, pResult );
QmpTimeStartTiming (hTimer, pResult );

/* other application code here */

QmpTimeGetElapsedTimeToNow (hTimer, pusElapsedDays,
    pusElapsedHours, pusElapsedMinutes, pusElapsedSeconds,
    pResult );
QmpTimeStopTiming (hTimer, pResult );
QmpTimeGetStartTime (hTimer, pusStartYear, pusStartMonth,
    pusStartDay, pusStartHour, pusStartMinute, pusStartSecond,
    pResult );
```

QmpTimeStopTiming

```
QmpTimeGetElapsedTime (hTimer, pusElapsedDays, pusElapsedHours,
    pusElapsedMinutes, pusElapsedSeconds, pResult );
QmpTimeGetStopTime (hTimer, pusStopYear, pusStopMonth,
    pusStopDay, pussStopHour, pussStopMinute, pussStopSecond,
    pResult );
QmpTimeDestroy (hTimer, pResult );
```

Function Class: QmpUniqsRpt*

UNIQUES REPORT FUNCTIONS.

Quick Reference

Function	Description	Page
QmpUniqsRptAddPreProcDataSrc	Specifies the preprocessed data source to be used by the uniques report phase.*	858
QmpUniqsRptCreate	Creates a uniques report.	859
QmpUniqsRptGetNthRecInterval	Get uniques report Nth record event interval.	860
QmpUniqsRptRegEveryNthRecFunc	Registers the event handler to be notified of every Nth record processed.	861
QmpUniqsRptSetNthRecInterval	Sets uniques report Nth record event interval.	862
QmpUniqsRptSetPrintKeys	Specifies whether to print a primary key column in the report.	863
QmpUniqsRptSetPrintSrc	Specifies whether to print a record source ID column in the report.	864
QmpUniqsRptUnregEveryNthRecFunc	Unregisters the event handler to be notified of every Nth record processed.	865
QmpUniqsRptUseDupGrp	Specifies the dupe groups object to use for the uniques report.	866

QmpUniqsRptAddPreProcDataSrc

SPECIFIES THE PREPROCESSED DATA SOURCE TO BE USED BY THE UNIQUES REPORT PHASE.*

Syntax

```
void QmpUniqsRptAddPreProcDataSrc ( MpHnd in_hUniqsRpt, MpHnd
                                    in_hDataSrc, QRESULT* out_pResult );
in_hUniqsRpt
    Handle to uniques report object. Input.
in_hDataSrc
    Handle to preprocessed data source to be used. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

*Only for data sources of type QMS_DATSRC_TYPE_PREPRO.

Use this function to specify a preprocessed data source for use by the uniques report phase.

Note that if *anything* is wrong with the preprocessed data source (missing fields, misspelled field names, incorrect data list IDs), an error will be generated and the data source will not be used.

Example

```
printf( "Starting Uniques Report phase:\n" );
QmpUniqsRptAddPreProcDataSrc( hUniquesReport, hPreProDataSrc,
                               &qres );
QmpPhaseStart( hUniquesReport, &qres );
```

QmpUniqsRptCreate

CREATES A UNIQUES REPORT.

Syntax

```
MpHnd QmpUniqsRptCreate ( QRESULT* out_pResult );  
out_pResult  
    Result code. Output.
```

Return Value

Returns handle to uniques report if successful, or `NULL` if there is an error.

Notes

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Example

```
hUniquesReport = QmpUniqsRptCreate (pResult );
```

QmpUniqsRptGetNthRecInterval

GET UNIQUES REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
long QmpUniqsRptGetNthRecInterval ( MpHnd in_hUniqsRpt,
                                    QRESULT* out_pResult );
in_hUniqsRpt
    Handle to uniques report. Input.
out_pResult
    Result code. Output.
```

Return Value

Returns uniques report Nth record event interval if successful, or -1 if there is an error.

Notes

This function gets the uniques report phase Nth record event interval.

The uniques report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to get that interval.

If, for example, the interval is set to every 100th record, then each time the uniques report phase processes 100 records, it will send out a notification to *all* registered clients.

See Also

[QmpUniqsRptRegEveryNthRecFunc](#), [QmpUniqsRptSetNthRecInterval](#),
[QmpUniqsRptUnregEveryNthRecFunc](#)

Example

```
lUniqsRptNthRecInterval = QmpUniqsRptGetNthRecInterval
                           (hUniqsRpt, pResult );
```

QmpUniqsRptRegEveryNthRecFunc

Registers the event handler to be notified of every Nth record processed.

Syntax

```
void QmpUniqsRptRegEveryNthRecFunc( MpHnd in_hUniqsRpt,
                                     QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hUniqsRpt
    Handle to uniques report. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code.
```

Return Value

None.

Notes

This function registers the event handler for being notified of every Nth record processed. The uniques report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
if ( bEveryNth ) {
    QmpUniqsRptRegEveryNthRecFunc( hUniquesReport,
                                    pEveryNthRecordClient, &qres );
    QmpUniqsRptSetNthRecInterval( hUniquesReport, 200, &qres );
}
```

QmpUniqsRptSetNthRecInterval

SETS UNIQUES REPORT NTH RECORD EVENT INTERVAL.

Syntax

```
void QmpUniqsRptSetNthRecInterval ( MpHnd in_hUniqsRpt, long
                                  in_lInterval, QRESULT* out_pResult );
in_hUniqsRpt
    Handle to uniques report. Input.
in_lInterval
    uniques report Nth record event interval. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function sets the uniques report phase Nth record event interval.

The uniques report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. A client may not wish to be notified of every single record, but would like an “occasional” notification at regular intervals. This function allows the client to set that interval.

If, for example, the interval is set to every 100th record, then each time the uniques report phase processes 100 records, it will send out a notification to *all* registered clients. If one client changes this interval, it affects the notification for all clients.

See Also

QmpUniqsRptGetNthRecInterval, **QmpUniqsRptRegEveryNthRecFunc**,
QmpUniqsRptUnregEveryNthRecFunc

Example

```
if ( bEveryNth ) {
    QmpUniqsRptRegEveryNthRecFunc( hUniquesReport,
                                    pEveryNthRecordClient, &qres );
    QmpUniqsRptSetNthRecInterval( hUniquesReport, 200, &qres );
}
```

QmpUniqsRptSetPrintKeys

SPECIFIES WHETHER TO PRINT A PRIMARY KEY COLUMN IN THE REPORT.

Syntax

```
void QmpUniqsRptSetPrintKeys (MpHnd in_hUniqsRpt, QBOOL
                             in_bPrintKeys, QRESULT* out_pResult );
in_hUniqsRpt
    Handle to uniques report object. Input.
in_bPrintKeys
    If QTRUE, print record primary keys. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

The primary key is a number (generally a record number) that is unique to a data source. It is usually the counter value when the records are read in by the data input phase. If a job has one data source, the primary key uniquely identifies a record. If a job has multiple data sources, the primary key and source ID values uniquely identify a record.

The following is an example of a uniques report with a primary key column. The column header is named “Key”.

Source	Key	Data List	FIRSTNAME	LASTNAME
1	5	1	ALVLIDA	LABERCCO
1	6	2	ALVILDAR	ALBERIC
1	10	2	ANHON	AJLMARALES

Example

```
QmpUniqsRptSetPrintKeys ( hUniquesRpt, QTRUE, pResult );
```

QmpUniqsRptSetPrintSrc

SPECIFIES WHETHER TO PRINT A RECORD SOURCE ID COLUMN IN THE REPORT.

Syntax

```
void QmpUniqsRptSetPrintSrc (MpHnd in_hUniqsRpt, QBOOL  

    in_bPrintSources, QRESULT* out_pResult );
```

in_hUniqsRpt
 Handle to uniques report object. *Input*.

in_bPrintSources
 If QTRUE, print source IDs. *Input*.

out_pResult
 Result code. *Output*.

Return Value

None.

Notes

The source ID field identifies the data source the record came from.

The following is an example of a uniques report with a source ID column. The column header is named “Source”.

Source	Key	Data List	FIRSTNAME	LASTNAME
1	5	1	ALVLIDA	LABERCCO
1	6	2	ALVILDAR	ALBERIC
1	10	2	ANHON	
AJLMARALES				

Example

```
QmpUniqsRptSetPrintSrc (hUniquesRpt, QTRUE, pResult );
```

QmpUniqsRptUnregEveryNthRecFunc

UNREGISTERS THE EVENT HANDLER TO BE NOTIFIED OF EVERY NTH RECORD PROCESSED.

Syntax

```
void QmpUniqsRptUnregEveryNthRecFunc ( MpHnd in_hUniqsRpt,
                                         QMS_EVERY_NTHREC_FUNC in_Func, QRESULT* out_pResult );
in_hUniqsRpt
    Handle to uniques report. Input.
in_Func
    Pointer to event handler. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function unregisters the specified event handler from the uniques report phase, so that the event handler will no longer be notified of every Nth record processed.

The uniques report phase can send an event (a notification) to one or more client functions which contains information about the number of records processed. It is during the processing of the EveryNthRecord event that a client application can make additional calls back into the library (for instance, to stop the phase). It is also a convenient time to update progress meters or otherwise update a visual representation on a computer screen.

Example

```
QmpUniqsRptUnregEveryNthRecFunc (hUniqsRpt,
                                    pUniqsRptEventHandler, pResult );
```

QmpUniqsRptUseDupGrp

SPECIFIES THE DUPE GROUPS OBJECT TO USE FOR THE UNIQUES REPORT.

Syntax

```
void QmpUniqsRptUseDupGrp( MpHnd in_hUniqsRpt, MpHnd  
    in_hdUpGrps, QRESULT* out_pResult );
```

in_hUniqsRpt
Handle to uniques report object. *Input*.

in_hdUpGrps
Handle to dupe groups object. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

The records whose dupe group ID = 0 will be output to the report.

Example

```
QmpUniqsRptUseDupGrp ( hUniqsRpt, hDupeGroups, pResult );
```

Function Class: QmpUtil*

GENERAL-PURPOSE UTILITY FUNCTIONS AND MACROS.

Quick Reference

Function	Description	Page
QmpDeclHnd (Macro)	Allocate an object handle and initialize it to NULL.	868
QmpUtilChkresultCode	Checks the specified result code for success or failure.	869
QmpUtilFailed (Macro)	Tests whether a result code indicates failure.	870
QmpUtilGetMessage (Macro)	Isolates and returns the value of the message digits of a QRESULT code.	871
QmpUtilGetSeverity (Macro)	Isolates and return the value of the severity digit of a QRESULT code.	872
QmpUtilGetSuccessFlag (Macro)	Isolates and returns the value of the success flag digit of a QRESULT code.	873
QmpUtilPutFileData	Outputs data to a file.	874
QmpUtilPutFileMsg	Output a message to a file.	875
QmpUtilSucceeded (Macro)	Tests whether a QRESULT code indicates success.	876

QmpDeclHnd (Macro)

ALLOCATE AN OBJECT HANDLE AND INITIALIZE IT TO NULL.

Syntax

```
QmpDeclHnd ( handle );  
handle  
Object handle name. Input.
```

Return Value

None.

Notes

QmpDeclHnd is a convenient way to allocate an object handle and initialize it to NULL in one step. It is not a good idea to pass an uninitialized object handle to a Centrus Merge/Purge function.

QmpDeclHnd(hMatCor);
expands to:

```
MpHnd hMatCor = NULL;
```

Example

```
QmpDeclHnd( hRecMat );
```

QmpUtilChkresultCode

CHECKS THE SPECIFIED RESULT CODE FOR SUCCESS OR FAILURE.

Syntax

```
void QmpUtilChkresultCode ( QRESULT* in_Result, const char*
                           in_szMsg );
in_Result
    Result code to check. Input.
in_szMsg
    A user-specified message to append to the standard message. Input.
```

Return Value

None.

Notes

This function checks the parameter *in_Result* for success or failure. If it is a failure result code, the associated error message is looked up, and a formatted message is printed to all “registered” destinations in the Centrus Merge/Purge error processing system. This error output respects both the error reporting level and the error fatality level. This means that if the error is less severe than the user-set reporting level, it will not be displayed. If the error is less severe than the user-set fatality level, the program will continue executing.

C programmers can call this function to check the result codes returned by their C function calls in their applications. Use of this function by C applications is not required, and the error reporting system can be used as described starting on [page 897](#).

Example

```
QmpLogSetErrRptLvl( hLog, QMS_ERROR_SEVERE, &qres );
QmpUtilChkresultCode ( QRESULT_WARN_IGNORE_RECORD, "No error
                       should print" );
QmpUtilChkresultCode ( QRESULT_WARN_NO_DUPEGROUPS, "No error
                       should print" );
QmpUtilChkresultCode ( QRESULT_SEVERE_TOO_LARGE, "**** Should
                       print error" );
QmpUtilChkresultCode ( QRESULT_SEVERE_NOT_FOUND, "**** Should
                       print error" );
```

QmpUtilFailed (Macro)

TESTS WHETHER A RESULT CODE INDICATES FAILURE.

Syntax

```
QmpUtilFailed (QRESULT in_Result);  
in_Result  
Result code. Input.
```

Return Value

QmpUtilFailed evaluates to 0 if the result code indicates failure, non-zero if it indicates success.

Notes

This macro is used to test a result code returned from a library function call.

Example

```
if (QmpUtilFailed (Result)) {  
/* process error condition */  
}
```

QmpUtilGetMessage (Macro)

ISOLATES AND RETURNS THE VALUE OF THE MESSAGE DIGITS OF A QRESULT CODE.

Syntax

```
QmpUtilGetMessage (QRESULT in_Result);  
in_Result  
Result code. Input.
```

Return Value

QmpUtilGetFacility evaluates to the value of the message digits of the QRESULT value.

Example

```
int iMessage = QmpUtilGetMessage (Result);
```

QmpUtilGetSeverity (Macro)

ISOLATES AND RETURN THE VALUE OF THE SEVERITY DIGIT OF A QRESULT CODE.

Syntax

```
QmpUtilGetSeverity (QRESULT in_Result);  
in_Result  
Result code. Input.
```

Return Value

QmpUtilGetSeverity evaluates to the severity digit of the QRESULT value.

Example

```
int iSeverity = QmpUtilGetSeverity (Result);
```

QmpUtilGetSuccessFlag (Macro)

ISOLATES AND RETURNS THE VALUE OF THE SUCCESS FLAG DIGIT OF A QRESULT CODE.

Syntax

```
QmpUtilGetSuccessFlag (QRESULT in_Result);  
in_Result  
Result code. Input.
```

Return Value

QmpUtilGetSuccessFlag evaluates to the success flag digit of the QRESULT value.

Example

```
int iSuccessFlag = QmpUtilGetSuccessFlag (Result);
```

QmpUtilPutFileData

OUTPUTS DATA TO A FILE.

Syntax

```
void QmpUtilPutFileData (const char* in_szFilename, const
                        char* in_szFormat, TYPE UNDEFINED in_argument, ... );
in_szFilename
    File name for data output. Input.
in_szFormat
    Format string. This string resembles a printf format string and contains
    both the number of in_argument parameters to be output to the file and
    their types. Input.
in_argument
    Argument to be output to file. The number of in_argument parameters
    passed in to the function is defined in the format string in_szFormat.
    Input.
```

Return Value

None.

Notes

Since a variable number of arguments are used, there is no **QRESULT** parameter at the end of the parameter list.

See Also

QmpUtilPutFileMsg

Example

```
/* Example 1 */
/* Dump the application match result object */

printf( "Begin Dump \n" );
QmpUtilPutFileData( "Results.log", "\nDumping match result
object for a threshold of %d.\n", lMatcherThreshold );

/* Example 2 */
QmpUtilPutFileData( "Results.log", "%s processed %d records\n",
in_szName, in_lRecordCount );
```

QmpUtilPutFileMsg

OUTPUT A MESSAGE TO A FILE.

Syntax

```
void QmpUtilPutFileMsg (const char* in_szFilename, const char*
    in_szMsg, QRESULT* out_pResult );
in_szFilename
    File name. Input.
in_szMsg
    Message to be output to the file. Input.
out_pResult
    Result code. Output.
```

Return Value

None.

Example

```
QmpUtilPutFileMsg("Results.log", "\nDumping duplicate ranking
results.\n", pResult );
```

QmpUtilSucceeded (Macro)

TESTS WHETHER A QRESULT CODE INDICATES SUCCESS.

Syntax

```
QmpUtilSucceeded (QRESULT in_Result);  
in_Result  
Result code. Input.
```

Return Value

QmpUtilSucceeded evaluates to 0 if the result code indicates success, non-zero if it indicates failure.

Notes

This macro is used to test a result code returned from a library function call.

Example

```
if (QmpUtilSucceeded (Result)) {  
/* continue processing */  
}
```

Function Class: QmpVar*

FUNCTIONS FOR MANIPULATING VARIANT OBJECTS.

The variant is a Centrus Merge/Purge object that is capable of storing a variable's value in any one of several formats. It is useful for storing a piece of data in a single location without being concerned about whether that data is a long, float, string, char, or some other type. It achieves this by encapsulating a union of data types inside an object, and then providing interface functions to allow applications to set that object to a value, and to retrieve that value. It contains a type field and a union of variables of different types.

The variant has a number of different uses. Among them is putting data into and getting data out of Centrus Merge/Purge record fields. The Centrus Merge/Purge record object (accessed through `QmpRec*` functions) is basically an array of variant objects representing the record fields, which allows them to contain different types of data.

The variant allows applications to put data into them in a number of different formats, and to extract data in the same or other formats. Implicit conversion of the contained data is done to change the type.

Applications must destroy any variant object they create, but they must not destroy any variant that is returned as the value of a function. As with all Centrus Merge/Purge objects, the application only destroys the objects it creates with a `Qmp*Create` function.

Quick Reference

Function	Description	Page
<code>QmpVarChgType</code>	Changes the type of the variant.	879
<code>QmpVarCreate</code>	Creates a variant.	880
<code>QmpVarDestroy</code>	Destroys a variant.	882
<code>QmpVarFillDateStruct</code>	Fills a QDATESTRUCT structure from a variant.	883
<code>QmpVarFillVarStruct</code>	Fills a QVARSTRUCT from a variant object.	885
<code>QmpVarGetDouble</code>	Gets a double from a variant.	886
<code>QmpVarGetFloat</code>	Gets a float from a variant.	887
<code>QmpVarGetLong</code>	Gets long from a variant.	888
<code>QmpVarGetString</code>	Gets string from a variant.	889
<code>QmpVarGetStringVB</code>	Gets string from a variant (Visual Basic version).	890
<code>QmpVarSetDateStruct</code>	Sets variant to a QDATESTRUCT.	891
<code>QmpVarSetDouble</code>	Sets variant to a double.	892
<code>QmpVarSetFloat</code>	Sets variant to a float.	893

Function	Description	Page
QmpVarSetLong	Sets variant to a long.	894
QmpVarSetString	Set variant to a string.	895
QmpVarSetVarStruct	Sets variant to a QVARSTRUCT.	896

QmpVarChgType

CHANGES THE TYPE OF THE VARIANT.

Syntax

```
void QmpVarChgType ( MpHnd in_hVar, QMS_VARIANT_TYPE  
                      in_NewType, QRESULT* out_pResult );
```

in_hVar

Handle to variant. *Input*.

in_NewType

Type variant is being changed to. *Input*.

Variant enums are:

QMS_VARIANT_EMPTY	Uninitialized CQmsVariant.
-------------------	----------------------------

QMS_VARIANT_BOOLEAN	
---------------------	--

QMS_VARIANT_CHAR	
------------------	--

QMS_VARIANT_DOUBLE	
--------------------	--

QMS_VARIANT_FLOAT	
-------------------	--

QMS_VARIANT_INT	
-----------------	--

QMS_VARIANT_LONG	
------------------	--

QMS_VARIANT_STRING	
--------------------	--

QMS_VARIANT ULONG	
-------------------	--

QMS_VARIANT USHORT	
--------------------	--

QMS_VARIANT_VOIDSTAR	Generic pointer to void
----------------------	-------------------------

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpVarChgType changes the type of the variant. The data represented inside the variant is also changed to the new type. If the application were to call **QmpVarFillVarStruct**, the “varType” field of that structure and the contents of the structure union would show that, indeed, the variant had been changed to the new type.

Example

See example on [page 880](#).

QmpVarCreate

CREATES A VARIANT.

Syntax

```
MpHnd QmpVarCreate ( QRESULT* out_pResult );
out_pResult
    Result code. Output.
```

Return Value

Returns handle to variant if successful, or NULL if there is an error.

Notes

This function is used to create a variant.

Any object created by the application with a **Qmp*Create** function must be destroyed by the application with the corresponding **Qmp*Destroy** function.

Example

```
const char* pszString = NULL;
char buffer[TEST_BUF_SIZE];
float fValue = 0.0;
long lValue = 0;
QVARSTRUCT* pVarStruct = 0;
QVARSTRUCT varStruct;

QmpDeclHnd( hVar );
hVar = QmpVarCreate( &qres );

QmpVarSetString( hVar, "hello", &qres );
pszString = QmpVarGetString( hVar, &qres );
if (strcmp( "hello", pszString ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
        differ" );

QmpVarGetStringVB( hVar, buffer, sizeof(buffer), &qres );
if (strcmp( "hello", buffer ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
        differ" );

QmpVarSetString( hVar, "99.99", &qres );
fValue = QmpVarGetFloat( hVar, &qres );
if ( fabs( fValue - 99.99 ) > 0.00001 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
        differ" );

QmpVarSetFloat( hVar, (float)88.88, &qres );
lValue = QmpVarGetLong( hVar, &qres );
if ( lValue != 88 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
        differ" );

QmpVarSetLong( hVar, 77, &qres );
pszString = QmpVarGetString( hVar, &qres );
```

```
if (strcmp( "77", pszString ) != 0 )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
differ" );

QmpVarChgType( hVar, QMS_VARIANT_DOUBLE, &qres );
pVarStruct = QmpVarGetVarStruct( hVar, &qres );
if ( pVarStruct->varType != QMS_VARIANT_DOUBLE )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
differ" );
if ( fabs( pVarStruct->uVal.doubleValue - 77.0 ) > 0.00001 )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
differ" );

varStruct.varType = QMS_VARIANT_LONG;
varStruct.uVal.longValue = 10;
QmpVarSetVarStruct( hVar, &varStruct, &qres );
pVarStruct = QmpVarGetVarStruct( hVar, &qres );
if ( pVarStruct->varType != QMS_VARIANT_LONG )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant types
differ" );
if ( pVarStruct->uVal.longValue != 10 )
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values
differ" );

QmpVarDestroy( hVar, &qres );           /* destroy variant */
```

QmpVarDestroy

DESTROYS A VARIANT.

Syntax

```
void QmpVarDestroy ( MpHnd in_hVar, QRESULT* out_pResult );
```

in_hVar
Handle to variant. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function destroys the specified variant.

Any object created by the application with a `Qmp*Create` function must be destroyed by the application with the corresponding `Qmp*Destroy` function.

Destroying an object does not affect any associated objects (objects that are attached using `Qmp*Use*` or `Qmp*Add*`). Only the references to the associated objects are destroyed.

Do *not* destroy variants that are returned as values of functions. The application does *not* own these variants. The application only owns the variants that it has created with `QmpVarCreate`. For example, the application should not destroy variant handles which received their values from `QmpRecGetFldByHnd` and `QmpRecGetFldByName`.

Example

See example on [page 880](#).

QmpVarFillDateStruct

FILLS A QDATESTRUCT STRUCTURE FROM A VARIANT.

Syntax

```
void QmpVarFillDateStruct ( MpHnd in_hVar, QDATESTRUCT*
                           out_pDateStruct, QRESULT* out_pResult );
in_hVar
    Handle to variant. Input.
out_pDateStruct
    Pointer to the QDATESTRUCT structure. Output.
    The structure contains one member:
    

---


    char ccyyymmdd[9]; Date plus one byte for null character.


---


out_pResult
    Result code. Output.
```

Return Value

None.

Notes

This function copies the date information from a variant into a QDATESTRUCT structure.

Example

```
const char* pszString = NULL;
char buffer[TEST_BUF_SIZE];
float fValue = 0.0;
long lValue = 0;
double dValue = 0;
QVARSTRUCT varStruct;
QDATESTRUCT dateStruct;

/* Create variant object */
QmpDeclHnd( hVar );
hVar = QmpVarCreate( &qres );

/* Set/Get values in the variant */
QmpVarSetString( hVar, "hello", &qres );
pszString = QmpVarGetString( hVar, &qres );
if (strcmp( "hello", pszString ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
QmpVarGetStringVB( hVar, buffer, sizeof(buffer), &qres );
if (strcmp( "hello", buffer ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
QmpVarSetString( hVar, "99.99", &qres );
fValue = QmpVarGetFloat( hVar, &qres );
if ( fabs( fValue - 99.99 ) > 0.00001 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
dValue = QmpVarGetDouble( hVar, &qres );
if ( fabs( dValue - 99.99 ) > 0.00001 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
QmpVarSetFloat( hVar, (float)88.88, &qres );
lValue = QmpVarGetLong( hVar, &qres );
if ( lValue != 88 )
```

```
QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );

QmpVarSetDouble( hVar, (double)88.88, &qres );
lValue = QmpVarGetLong( hVar, &qres );
if ( lValue != 88 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );

QmpVarSetLong( hVar, 77, &qres );
pszString = QmpVarGetString( hVar, &qres );
if ( strcmp( "77", pszString ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
QmpVarChgType( hVar, QMS_VARIANT_DOUBLE, &qres );
QmpVarFillVarStruct( hVar, &varStruct, &qres );
if ( varStruct.varType != QMS_VARIANT_DOUBLE )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
if ( fabs( varStruct.uVal.doubleValue - 77.0 ) > 0.00001 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
varStruct.varType = QMS_VARIANT_LONG;
varStruct.uVal.longValue = 10;
QmpVarSetVarStruct( hVar, &varStruct, &qres );
QmpVarFillVarStruct( hVar, &varStruct, &qres );
if ( varStruct.varType != QMS_VARIANT_LONG )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant types differ" );
if ( varStruct.uVal.longValue != 10 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
strcpy( dateStruct.ccyyymmdd, "19990101" );
QmpVarSetDateStruct( hVar, &dateStruct, &qres );
strcpy( dateStruct.ccyyymmdd, "00000000" );
QmpVarFillDateStruct( hVar, &dateStruct, &qres );
if ( strcmp( dateStruct.ccyyymmdd, "19990101" ) != 0 )
    QmpUtilChkresultCode( QRESULT_SEVERE_FAIL, "variant values differ" );
QmpVarDestroy( hVar, &qres ); /* destroy variant */
```

QmpVarFillVarStruct

FILLS A QVARSTRUCT FROM A VARIANT OBJECT.

Syntax

```
void QmpVarFillVarStruct ( MpHnd in_hVar, QVARSTRUCT*  
    out_pVarStruct, QRESULT* out_pResult );
```

in_hVar
Handle to variant. *Input*.

out_pVarStruct
Pointer to the QVARSTRUCT structure. *Output*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

This function copies the information from a variant object into a QVARSTRUCT structure. The QVARSTRUCT is a variant C structure that the application can manipulate without using a variant object wrapper.

Example

See example on [page 883](#).

QmpVarGetDouble

GETS A DOUBLE FROM A VARIANT.

Syntax

```
double QmpVarGetDouble ( MpHnd in_hVar, QRESULT* out_pResult
) ;
```

in_hVar

Handle to variant. *Input*.

out_pResult

Result code. *Output*.

Return Value

Returns a double value from the variant if successful, or -1.0 if there is an error.

Notes

QmpVarGetDouble gets a double value from a variant, regardless of how the data was originally placed into the variant. This function attempts to convert the value that it returns to the calling program to the best double value representing its contents.

Example

See example on [page 883](#).

QmpVarGetFloat

GETS A FLOAT FROM A VARIANT.

Syntax

```
float QmpVarGetFloat ( MpHnd in_hVar, QRESULT* out_pResult );  
in_hVar  
Handle to variant. Input.  
out_pResult  
Result code. Output.
```

Return Value

Returns a float value from the variant if successful, or -1.0 if there is an error.

Notes

QmpVarGetFloat gets a float value from a variant, regardless of how the data was originally placed into the variant. This function attempts to convert the value that it returns to the calling program to the best long value representing its contents.

For example, the long number 99 is returned as the float 99. Also, a variant which has a string value of “99” will return the float value, 99.

Example

See example on [page 880](#).

QmpVarGetLong

GETS LONG FROM A VARIANT.

Syntax

```
long QmpVarGetLong ( MpHnd in_hVar, out_pResult );  
in_hVar  
Handle to variant. Input.  
out_pResult  
Result code. Output.
```

Return Value

Returns a long value from a variant if successful, or -1 if there is an error.

Notes

QmpVarGetLong gets a long value from a variant, regardless of how the data was originally placed into the variant. This function attempts to convert the value that it returns to the calling program to the best long value representing its contents.

For example, the floating point number 99.99 is returned as the long 99. Also, a variant which has a string value of “99” will return the long value, 99.

Example

See example on [page 880](#).

QmpVarGetString

GETS STRING FROM A VARIANT.

Syntax

```
const char* QmpVarGetString( MpHnd in_hVar, QRESULT*  
    out_pResult );  
  
in_hVar  
    Handle to variant. Input.  
  
out_pResult  
    Result code. Output.
```

Return Value

Returns `const` pointer to string if successful, or `NULL` if there is an error.

Notes

`QmpVarGetString` gets a string from a variant, regardless of how the data was originally placed into the variant. This function attempts to convert the value that it returns to the calling program to the best “spelling” of its contents. For example, the floating point number 99.99 is returned as the string, “99.99”.

A variant which has a boolean value of `QTRUE` will return “TRUE”.

Example

See example on [page 880](#).

QmpVarGetStringVB

GETS STRING FROM A VARIANT (VISUAL BASIC VERSION).

Syntax

```
void QmpVarGetStringVB( MpHnd in_hVar, char* io_szBuffer, long  
                        in_lSize, QRESULT* out_pResult );  
  
in_hVar  
        Handle to variant. Input.  
  
io_szBuffer  
        Buffer to put message into. Input.  
  
in_lSize  
        Size of buffer. Input.  
  
out_pResult  
        Result code. Output.
```

Return Value

None.

Notes

`QmpVarGetStringVB` gets a string from a variant, regardless of how the data was originally placed into the variant. This function attempts to convert the value that it returns to the calling program to the best “spelling” of its contents. For example, the floating point number 99.99 is returned as the string, “99.99”.

A variant which has a boolean value of QTRUE will return “TRUE”.

This function is tailored for Visual Basic programs. It does not return the string as the function value. Instead it places the string into the buffer that the calling program provides.

Example

See example on [page 880](#).

QmpVarSetDateStruct

SETS VARIANT TO A QDATESTRUCT.

Syntax

```
void QmpVarSetDateStruct ( MpHnd in_hVar, QDATESTRUCT*  
    in_pDateStruct, QRESULT* out_pResult );
```

in_hVar
Handle to variant. *Input*.

in_pDateStruct
Pointer to QDATESTRUCT. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpVarSetDateStruct fills a variant object with the date information contained in the QDATESTRUCT parameter. QDATESTRUCT is a C structure that the application can manipulate without using a date object wrapper.

Example

See example on [page 883](#).

QmpVarSetDouble

SETS VARIANT TO A DOUBLE.

Syntax

```
void QmpVarSetDouble ( MpHnd in_hVar, double in_dVal, QRESULT*  
                      out_pResult );
```

in_hVar
Handle to variant. *Input*.

in_dVal
Input double value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpVarSetDouble sets a variant to a double value. The variant type becomes QMS_VARIANT_DOUBLE. The application can retrieve this variant value as a double by calling **QmpVarGetDouble**. It can retrieve this value as a long by calling **QmpVarGetLong**, or as a string by calling **QmpVarGetString** or **QmpVarGetStringVB**.

Example

See example on [page 883](#).

QmpVarSetFloat

SETS VARIANT TO A FLOAT.

Syntax

```
void QmpVarSetFloat ( MpHnd in_hVar, float in_fVal, QRESULT*  
                      out_pResult );
```

in_hVar
Handle to variant. *Input*.

in_fVal
Input float value. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpVarSetFloat sets a variant to a floating point value. The variant type becomes QMS_VARIANT_FLOAT. The application can retrieve this variant value as a float by calling **QmpVarGetFloat**. It can retrieve this value as a long by calling **QmpVarGetLong**, or as a string by calling **QmpVarGetString** or **QmpVarGetStringVB**.

Example

See example on [page 880](#).

QmpVarSetLong

SETS VARIANT TO A LONG.

Syntax

```
void QmpVarSetLong ( MpHnd in_hVar, long in_lVal, QRESULT*  
                     out_pResult );
```

in_hVar
Handle to variant. *Input.*

in_lVal
Input long value. *Input.*

out_pResult
Result code. *Output.*

Return Value

None.

Notes

QmpVarSetLong sets a variant to a long value. The variant type becomes `QMS_VARIANT_LONG`. The application can retrieve this variant value as a long by calling **QmpVarGetLong**. It can retrieve this value as a float by calling **QmpVarGetFloat**, or as a string by calling **QmpVarGetString** or **QmpVarGetStringVB**.

Example

See example on [page 880](#).

QmpVarSetString

SET VARIANT TO A STRING.

Syntax

```
void QmpVarSetString ( MpHnd in_hVar, const char*  
    in_pszString, QRESULT* out_pResult );
```

in_hVar
Handle to variant. *Input*.

in_pszString
Input string. *Input*.

out_pResult
Result code. *Output*.

Return Value

None.

Notes

QmpVarSetString sets a variant to a string. The variant type becomes QMS_VARIANT_STRING. The application can retrieve this variant value as a string by calling **QmpVarGetString** or **QmpVarGetStringVB**. The application can retrieve this value as a long by calling **QmpVarGetLong**, or as a float by calling **QmpVarGetFloat**.

Example

See example on [page 880](#).

QmpVarSetVarStruct

SETS VARIANT TO A QVARSTRUCT.

Syntax

```
void QmpVarSetVarStruct ( MpHnd in_hVar, QVARSTRUCT*
```

```
    in_pVarStruct, QRESULT* out_pResult );
```

in_hVar

Handle to variant. *Input*.

in_pVarStruct

Pointer to QVARSTRUCT. *Input*.

out_pResult

Result code. *Output*.

Return Value

None.

Notes

QmpVarSetVarStruct sets the contents of a variant object to be those of the passed in QVARSTRUCT structure.

The QVARSTRUCT is a variant C structure that the application can manipulate without using a variant object wrapper. **QmpVarSetVarStruct** fills a variant object with the variable and type information contained in the QVARSTRUCT parameter.

Example

See example on [page 880](#).

Appendix A

QRESULT Return Codes

Explanation of Return Codes

Each QRESULT return code is an 8-digit decimal number. The first digit represents the success code:

1 = success

2 = failure

The second digit represents the severity code:

0 = informative

1 = warning

2 = severe

3 = fatal

The third, fourth, and fifth digits represent the facility code, but are currently unused.

The sixth, seventh, and eighth digits represent the message number, which varies by facility.

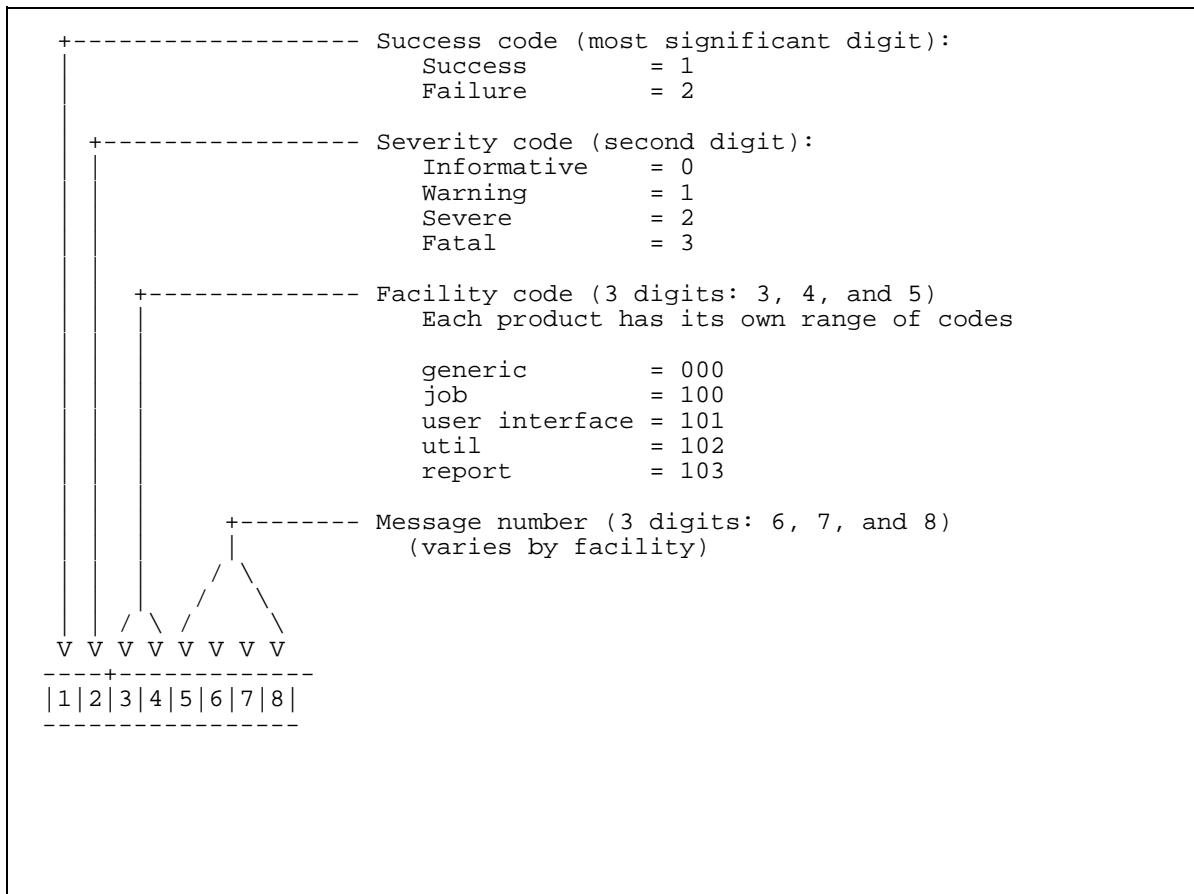
Typical QRESULT return codes look like this:

1100000 = success of a method in the job facility.

2100001 = failure due to out of range parameter

2101002 = failure due to bad pointer

The following diagram shows each digit's significance:



Success (Informational) Return Codes

QRESULT return code	QRESULT value	QRESULT message
QRESULT_INFO_YES	10000000	“Informative - Yes”
QRESULT_INFO_NO	10000001	“Informative - No”
QRESULT_INFO_SUCCEED	10000002	“Informative”
QRESULT_INFO_ENDOFFILE	10000003	“Informative - end of file”
QRESULT_INFO_ENDOFENUM	10000004	“Informative - end of enum”

Failure (Warning) Return Codes

QRESULT return code	QRESULT value	QRESULT message
QRESULT_WARN_IGNORE_RECORD	11000000	“Warning - Record was ignored”
QRESULT_WARN_STOP_EXECUTE	11000001	“Warning - Stop execution of Data List Service”
QRESULT_WARN_NO_DUPEGROUPS	11000002	“Warning - No dupe groups exist”
QRESULT_WARN_DUPGRP_TOO_BIG	11000003	“Warning - Too many members in dupe group”
QRESULT_WARN_UNIMPLEMENTED	11000004	“Informative - unimplemented”
QRESULT_WARN_DEMO_EXCEEDED	11000005	“Warning - volume of records exceeds demo library limit”
QRESULT_WARN_DATASOURCES_EXCEEDED	11000006	“Warning - PreProcessed data source should contain a single data source”
QRESULT_WARN_NO_SUBORDS	11000007	“Warning - Dupe group has no subordinate records”
QRESULT_WARN_NO_DUPID_INDEX	11000008	“Warning - PreProcessed data source has no dupgroupid index”
QRESULT_WARN_DAL_USER_CANCEL	11000009	“Warning - user canceled the DAL open dialog”
QRESULT_WARN_METHOD_RETIRING	11000010	“Warning - THIS METHOD IS SLATED FOR RETIREMENT”
QRESULT_WARN_VERIFICATION	11000011	“Warning - Verification failed”
QRESULT_WARN_NO_CONSOL_TABLE	11000012	“Warning - No consolidated record table is available”
QRESULT_WARN_DEFAULT_DATE_HINT	11000013	“Warning - default Date format hint used”

QRESULT return code	QRESULT value	QRESULT message
QRESULT_WARN_PREPRO_SORT_RECORDS	11000014	“Warning - Preprocessed table and sorted table record counts do not match”
QRESULT_WARN_PREPRO_SORT_FIELDS	11000015	“Warning - Sorted table field counts do not match field count”
QRESULT_WARN_PREPRO_SORT_FIELDS_TY PE	11000016	“Warning - Preprocessed table and sorted table field properties do not match”
QRESULT_WARN_SORT_READ	11000017	“Warning - Generating a table on a sort object which has had NextItem called.”
QRESULT_WARN_PREPRO_SORT_FLD_EXIST	11000018	“Warning - A sorted table field is missing”
QRESULT_WARN_PREPRO_SORT_FLD_SRCID	11000019	“Warning - The existing sorted table SrcId does not match”
QRESULT_WARN_NO_DUPGRP_ID_EXIST	11000020	“Warning - The specified dupegroup doesn't exist”
QRESULT_WARN_REC_FIELD_COUNTS	11000021	“Warning - Field count does not match fmt file”
QRESULT_WARN_DISK_USE_FAILURE	11000022	“Warning - disk may fill due to this job”
QRESULT_WARN_INVALID_STATSAMP	11000023	“Warning - invalid statistical sampling information: turning off sampling”
QRESULT_WARN_ORACLE_CONSTRAINT	11000024	“Warning - The Oracle table has no primary key defined, so you will not be able to write to the table”

Failure (Severe) Return Codes

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_FAIL	22000000	“Severe”
QRESULT_SEVERE_ONE_PREPRO	22000001	“Severe - Only one Preprocessed table allowed”
QRESULT_SEVERE_PREPRO_TABLE_MATCH	22000002	“Severe - DITR and Preprocessed table do not match”
QRESULT_SEVERE_BAD_INPUT	22000003	“Severe - bad input”
QRESULT_SEVERE_FILE_CLOSE	22000004	“Severe - could not close file”
QRESULT_SEVERE_FILE_OPEN	22000005	“Severe - could not open file”
QRESULT_SEVERE_FILE_READ	22000006	“Severe - could not read from file”

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_FILE_WRITE	22000007	"Severe - could not write to file"
QRESULT_SEVERE_INVALID_ARG	22000008	"Severe - invalid argument"
QRESULT_SEVERE_INVALID_OBJECT	22000009	"Severe - invalid object"
QRESULT_SEVERE_NOT_FOUND	22000010	"Severe - not found"
QRESULT_SEVERE_OUT_OF_BOUNDS	22000011	"Severe - out of bounds"
QRESULT_SEVERE_OUT_OF_MEMORY	22000012	"Severe - out of memory"
QRESULT_SEVERE_TOO_LARGE	22000013	"Severe - too large"
QRESULT_SEVERE_TOO_LOW	22000014	"Severe - input too low"
QRESULT_SEVERE_TABLE_TYPE	22000015	"Severe - table type is not supported for this object"
QRESULT_SEVERE_WRONG_VARIANT	22000016	"Severe - wrong variant"
QRESULT_SEVERE_INVALID_CHILD	22000017	"Severe - invalid child object"
QRESULT_SEVERE_DUPLICATE_ENTRY	22000018	"Severe - duplicate entry"
QRESULT_SEVERE_NOT_IMPLEMENTED	21000019	"Severe - Function not implemented!"
QRESULT_SEVERE_DUPLICATE_ALG	22000020	"Severe - Duplicate algorithm detected"
QRESULT_SEVERE_INVALID_CLIENT	22000021	"Severe - Event client is invalid"
QRESULT_SEVERE_NO_EVENT_HANDLER	22000022	"Severe - Event handler not found"
QRESULT_SEVERE_DEFAULTCTOR	22000024	"Severe - Default constructor should not be used"
QRESULT_SEVERE_PHASE_CANNOT_START	22000025	"Severe - Phase is not ready to start"
QRESULT_SEVERE_PHASE_NOT_FOUND	22000026	"Severe - Phase is not part of job"
QRESULT_SEVERE_INDEX_CREATION	22000027	"Severe - could not create table index"
QRESULT_SEVERE_INDEX_NAME_CREAT	22000028	"Severe - could not create unique index name"
QRESULT_SEVERE_INVALID_INDXKEY	22000029	"Severe - index key object is invalid"
QRESULT_SEVERE_INVALID_RECMAT	22000030	"Severe - record matcher object is invalid"
QRESULT_SEVERE_INVALID_INDXGEN	22000031	"Severe - index generator object is invalid"
QRESULT_SEVERE_NO_TABLE_NAME	22000032	"Severe - table does not have a name"
QRESULT_SEVERE_NO_TABLE_FIELDS	22000033	"Severe - no table fields are defined"
QRESULT_SEVERE_TABLE_NOT_READY	22000034	"Severe - table is not yet ready for use"
QRESULT_SEVERE_TABLE_IS_OPEN	22000035	"Severe - table is open. Operation not allowed."

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_TABLE_NOT_INIT	22000036	"Severe - table is not initialized"
QRESULT_SEVERE_TABLE_NOT_OPEN	22000037	"Severe - table is not open. Operation not allowed."
QRESULT_SEVERE_TABLE_REMOVE	22000038	"Severe - table could not be removed."
QRESULT_SEVERE_TABLE_MOVE	22000039	"Severe - table move to new current record failed. "
QRESULT_SEVERE_TABLE_OPEN	22000040	"Severe - table open failed. "
QRESULT_SEVERE_TABLE_CLOSE	22000041	"Severe - table close failed. "
QRESULT_SEVERE_TABLE_COMMIT	22000042	"Severe - table commit failed. "
QRESULT_SEVERE_TABLE_ADD	22000043	"Severe - table add record failed. "
QRESULT_SEVERE_TABLE_DELETE	22000044	"Severe - table delete failed. "
QRESULT_SEVERE_TABLE_FIELD_LARGE	22000045	"Severe - table field name is too large. "
QRESULT_SEVERE_TOO_MANY_HANDLES	22000046	"Severe - too many handles for index keys. "
QRESULT_SEVERE_TABLE_INDEX_NOT_FOUND	22000047	"Severe - table index does not exist. "
QRESULT_SEVERE_INVALID_RECORD_NUMBER	22000048	"Severe - invalid table record number. "
QRESULT_SEVERE_LAST_INDEX	22000049	"Severe - last index cannot be deleted"
QRESULT_SEVERE_INVALID_RECORD	22000050	"Severe - prototype record is invalid"
QRESULT_SEVERE_INVALID_STATE	22000051	"Severe - invalid state"
QRESULT_SEVERE_INVALID_TYPE	22000052	"Severe invalid type"
QRESULT_SEVERE_LOCKED_RECORD	22000053	"Severe - Record is locked"
QRESULT_SEVERE_INVALID_INDEX_NAME	22000054	"Severe - invalid index name"
QRESULT_SEVERE_TABLE_NOT_FOUND	22000055	"Severe - table was not found"
QRESULT_SEVERE_INVALID_MATCH_RESULT	22000056	"Severe - match result object is invalid"
QRESULT_SEVERE_INVALID_LICENSE	22000057	"Severe - invalid license (either missing or bad password)"
QRESULT_SEVERE_TABLE_SEEK	22000058	"Severe - seek on table failed"
QRESULT_SEVERE_INVALID_ENV	22000059	"Severe - invalid or undefined environment variable"
QRESULT_SEVERE_DAL_LIB_NOT_READY	22000060	"Severe - database library is not yet ready for use"
QRESULT_SEVERE_DATABASE_NOT_READY	22000061	"Severe - database is not yet ready for use"

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_DATABASE_IS_OPEN	22000062	“Severe - database is open. Operation not allowed.“
QRESULT_SEVERE_REC_READ	22000063	“Severe - could not read record.“
QRESULT_SEVERE_NO_NAME	22000064	“Severe - object has no name.“
QRESULT_SEVERE_INVALID_DATLST	22000065	“Severe - Data List is invalid“
QRESULT_SEVERE_DUPES_NOT_READY	22000066	“Severe - Duplicates are not computed yet“
QRESULT_SEVERE_INVALID_DUPEGROUPID	22000067	“Severe - invalid Dupe Group ID“
QRESULT_SEVERE_INVALID_DATADEST	22000068	“Severe - table generation object is invalid“
QRESULT_SEVERE_DATLST_SUPPRESS	22000069	“Severe - Cannot perform action on suppression Data List“
QRESULT_SEVERE_DATLST_ACTION	22000070	“Severe - Invalid Data List action“
QRESULT_SEVERE_DATLST_NO_DFLT	22000071	“Severe - No default Data List defined“
QRESULT_SEVERE_DATLST_DFLT	22000072	“Severe - Default Data List already defined“
QRESULT_SEVERE_INVALID_EXPRESS	22000073	“Severe - Invalid field expression“
QRESULT_SEVERE_DATLST_REMOVE_DFLT	22000074	“Severe - Cannot remove default Data List while others are in service“
QRESULT_SEVERE_INVALID_DATSRCID	22000075	“Severe - invalid Data Source ID“
QRESULT_SEVERE_INVALID_STATSAMP	22000076	“Severe - invalid statistical sampling information“
QRESULT_SEVERE_NO_FILL_FUNCTOR	22000077	“Severe - Data Input fill functor not found“
QRESULT_SEVERE_NO_DATSRC	22000078	“Severe - No data sources found“
QRESULT_SEVERE_WRONG_DATSRC	22000079	“Severe - Data Source is of wrong type“
QRESULT_SEVERE_NO_DUPEGROUP_SCORES	22000080	“Severe - No dupe groups scores are available“
QRESULT_SEVERE_NO_DAL_EXTEN	22000081	“Severe - No DAL extensions found“
QRESULT_SEVERE_NO_ALG	22000082	“Severe - No algorithms set for criterion“
QRESULT_SEVERE_BAD_TIME	22000083	“Severe - Time object is invalid“
QRESULT_SEVERE_SEEK_EXPRESSION	22000084	“Severe - Seek Expression Empty“
QRESULT_SEVERE_PHASE_NOT_USABLE	22000085	“Severe - Phase cannot be used in this operation“

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_REC_FIELD_EXISTS	22000086	"Severe - Adding field that already exists."
QRESULT_SEVERE_INVALID_FLDMAP	22000087	"Severe - invalid field map"
QRESULT_SEVERE_TABLE_ADDED_TWICE	22000088	"Severe - table may only be added once."
QRESULT_SEVERE_TEST_UNEXPECTED	22000089	"Severe - a test encountered unexpected result."
QRESULT_SEVERE_DEFAULT_CONSTRUCTOR	22000090	"Severe - the default constructor should not be called."
QRESULT_SEVERE_REC_FIELD_COUNTS	22000091	"Severe - the field counts are unexpected."
QRESULT_SEVERE_REC_FIELD_HANDLE	22000092	"Severe - the field could not be found"
QRESULT_SEVERE_REC_FIELD_ADDED	22000093	"Severe - a field could not be added"
QRESULT_SEVERE_EMPTY_MEMBERS	22000094	"Severe - a required collection is empty"
QRESULT_SEVERE_REMOVED	22000095	"Severe - an entity could not be removed"
QRESULT_SEVERE_CALL_ONCE	22000096	"Severe - function should be called only once"
QRESULT_SEVERE_KEYDIST_INVALID	22000097	"Severe - the Key Distance algorithm is invalid"
QRESULT_SEVERE_TABLE_DBF	22000098	"Severe - table required to be of type 'dbf'"
QRESULT_SEVERE_DUPEGROUP_INTERNAL	22000099	"Severe - Dupe Group internal inconsistency error"
QRESULT_SEVERE_DUPEGROUP_ARRAY	22000100	"Severe - Dupe Group member array is invalid"
QRESULT_SEVERE_DUPEGROUP_EMPTY	22000101	"Severe - Dupe Group has no members"
QRESULT_SEVERE_DUPEGROUP_FILL	22000102	"Severe - Dupe Group Dupe Group fillmembers failed"
QRESULT_SEVERE_RECMAT_THRESHOLD	22000103	"Severe - Record Match result outside allowable range"
QRESULT_SEVERE_NO_DATADEST	22000104	"Severe - no data destinations have been specified"
QRESULT_SEVERE_DAL_MISSING_COMP	22000105	"Severe - Missing DAL FMT file"
QRESULT_SEVERE_DAL_NO_TABLES	22000107	"Severe - no DAL tables found in the database"

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_DAL_ERROR	22000108	“Severe - error opening DAL database”
QRESULT_SEVERE_NO_INPUT_RECORDS_ER ROR	22000109	“Severe - No Data Input records.”
QRESULT_SEVERE_NO_PREPRO_RECORDS_E RROR	22000110	“Severe - No Pre-processed Data records.”
QRESULT_SEVERE_MASTER_NOT_FOUND	22000111	“Severe - Master duplicate not found”
QRESULT_SEVERE_SUBORD_NOT_FOUND	22000112	“Severe - Subordinate duplicate not found”
QRESULT_SEVERE_INVALID_WINDOW	22000113	“Severe - Sliding Window is invalid”
QRESULT_SEVERE_VERIFICATION	22000114	“Severe - Verification failed”
QRESULT_SEVERE_INVALID_CONSOL_CRIT	22000115	“Severe - Invalid data consolidation criterion”
QRESULT_SEVERE_INVALID_DATE_HINT	22000116	“Severe - Date format hint not found”
QRESULT_SEVERE_WRONG_ADD	22000117	“Severe - use different Add field function”
QRESULT_SEVERE_INVALID_CB_TABLE	22000118	“Severe - CodeBase Table is invalid”
QRESULT_SEVERE_TABLE_RECLEN	22000119	“Severe - Table record is invalid”
QRESULT_SEVERE_TABLE_EXTEN	22000120	“Severe - Table extension is invalid”
QRESULT_SEVERE_SORT_PREPARE	22000121	“Severe - Sort PrepareData() has not been called”
QRESULT_SEVERE_SORT_DONE	22000122	“Severe - Sort data has all been read”
QRESULT_SEVERE_PREPRO_SORT_VERSION S	22000123	“Severe - Preprocessed table and sorted table do not match”
QRESULT_SEVERE_PREPRO_SORT_RECORDS	22000124	“Severe - Preprocessed table and sorted table record counts do not match”
QRESULT_SEVERE_PREPRO_SORT_FIELDS	22000125	“Severe - Sorted table field counts do not match match field count”
QRESULT_SEVERE_PREPRO_SORT_FIELDS_ TYPE	22000126	“Severe - Preprocessed table and sorted table field properties do not match”
QRESULT_SEVERE_PREPRO_SORT_FLD_EXI ST	22000127	“Severe - A sorted table field is missing”
QRESULT_SEVERE_DISK_FULL	22000128	“Severe - disk is full”
QRESULT_SEVERE_INVALID_DISK_USAGE	22000129	“Severe - disk usage data is invalid or uninitialized”
QRESULT_SEVERE_INVALID_DATE_FIELD	22000130	“Severe - The date field does not match the date format”

QRESULT return code	QRESULT value	QRESULT message
QRESULT_SEVERE_INVALID_TRIM_FIELD	22000131	“Severe - The trim field value is invalid”
QRESULT_SEVERE_METHOD_RETired	22000132	“Severe - THIS METHOD HAS RETIRED”
QRESULT_SEVERE_NO_TABLE_PARAMS	22000133	“Severe - Table requires connection string parameters”
QRESULT_SEVERE_REC_FIELD_NAME	22000134	“Severe - The field name is invalid”
QRESULT_SEVERE_ORACLE_CONSTRAINT	22000135	“Severe - The Oracle table has no primary key defined”
QRESULT_SEVERE_TABLE_COPY	22000136	“Severe - The table copy operation failed”

Failure (Fatal) Return Codes

QRESULT return code	QRESULT value	QRESULT message
QRESULT_FATAL_BAD_POINTER	23000000	“Fatal - bad pointer”
QRESULT_FATAL_DEAD_OBJECT	23000001	“Fatal - dead (unallocated) object”
QRESULT_FATAL_DTOR	23000002	“Fatal - destructor failed”
QRESULT_FATAL_GET_DIR	23000003	“Fatal - could not get current directory”

Last Chance Return Code

This message is output if an error is generated, but hasn't been added to the error utility routine, `LookUpErrorMessage`.

QRESULT return code	QRESULT value	QRESULT message
QRESULT_WARN_NO_AVAILABLE		“There is no message defined for this error”

Appendix B

Formatting ASCII Files

This section describes how to format ASCII text files to use as source data. In a Windows environment, Centrus Merge/Purge CLI uses the application Centrus Data Formatter to automate the process of creating and updating format files. There is no corresponding application in UNIX, so creating and updating UNIX format files must be done manually. See “[Formatting ASCII Files on UNIX](#)” on page 915 for information on formatting ASCII text in a UNIX environment.

Formatting Text Files in Windows

Centrus Data Formatter is a 32-bit Microsoft Windows application designed to automate the process of creating and updating format files. Centrus Data Formatter reads field and record information from the existing format file, if there is one, or from the actual ASCII text file. This information is then displayed, along with a section of the first part of the text file so that you can see how the data is interpreted by Centrus Merge/Purge. You can accept Centrus Data Formatter’s settings or make changes as necessary. When you’re satisfied that your data is correctly described, Centrus Data Formatter updates any existing format file or creates a new one. Your ASCII text file is then ready for Centrus Merge/Purge to process.

About ASCII Text Files

ASCII files are document files encoded in a universally recognized text format. This type of file is useful for transferring data between programs and computing platforms that could not otherwise understand each other's file formats. Most data and text processing programs give you the option of exporting files in ASCII text format, usually with a file name extension of **.TXT**.

You can describe an ASCII text file containing addresses or other structured data by answering two questions:

- How are the fields and records defined?
- What, if any, is the end of line character?

Defining Fields

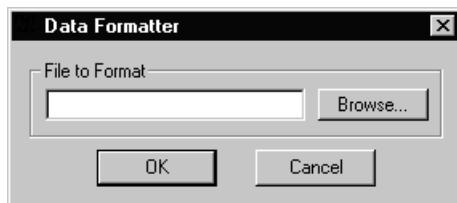
Fields are typically defined in one of two ways. *Delimited files* contain information separated by special characters, called delimiters. For example, the comma and tab characters are commonly used delimiters. The width of the fields defined by the delimiters can vary. *Fixed width* files, by contrast, contain fields that are always a constant width and are arranged in columns. The width of each field must be defined.

Some additional information is needed to complete the description of the file's fields. Each field may contain either *numeric* or *character* data. If the data is numeric, the number of decimal places must be entered. Each fixed-width field should have a name, type (numeric or character) and width. Delimited fields need only name and type. Finally, fixed width or delimited files may have field names rather than data in the first row or record.

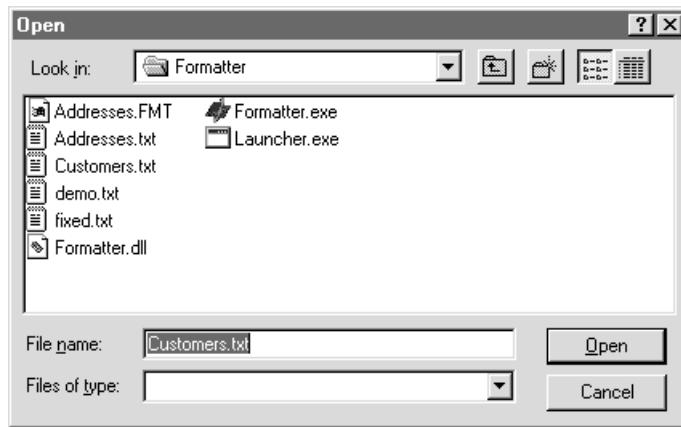
Delimited files also define field widths, which are useful when importing files to another format. Note that the entire field value is always read, regardless of the defined width.

Using Centrus Data Formatter

When you start Centrus Data Formatter, the **Data Formatter** dialog box appears.

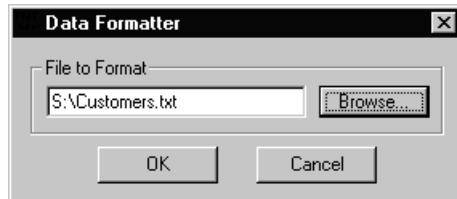


1. Enter the path and file name of the file you want to format or click **Browse...** to search for the file. The **Open** dialog box appears.



2. Select a text file and click **Open**. Centrus Data Formatter looks for the selected text file's format file—a file with the same name as the selected text file, but with an extension of .FMT. If Centrus Data Formatter finds a format file, it uses the information contained in the format file to read the text file. If Centrus Data Formatter does *not* find a format file, it attempts to extract format information directly from the text file.

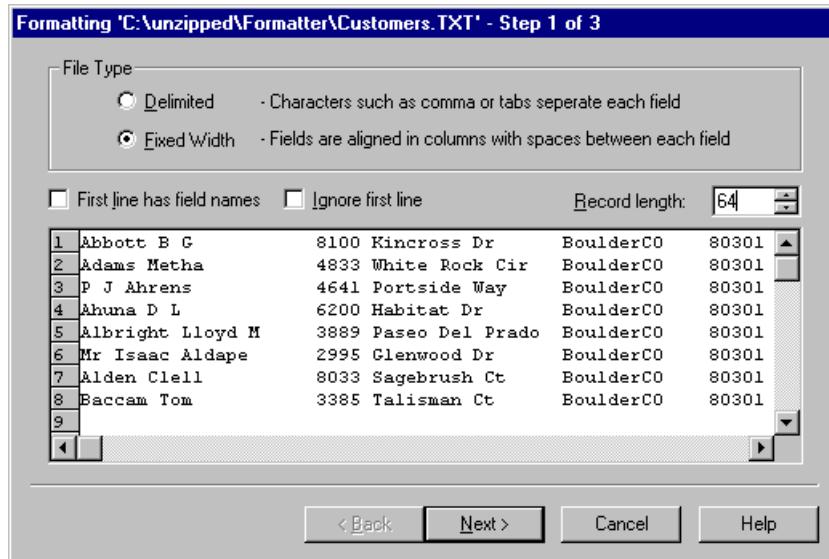
The **Data Formatter** dialog box reappears with the file you selected displayed.



3. Click **OK**.

Step 1

The **Formatting ‘filename’ – Step 1 of 3** dialog box appears.



When the Data Formatter opens the file, it attempts to determine the **File Type** – either *Fixed* or *Delimited*. The data in the Preview window at the bottom of the dialog is displayed differently, depending on what type of file you are using.

If there are no line terminators in the file, the Data Formatter prompts you to enter the record length. You can either select a record length or have the Data Formatter calculate it for you by searching for the most likely logical breaks in the file.

1. Accept the default **FileType**, or change if necessary.
2. Accept or change the following values:
 - **First line has field names** - the Data Formatter searches the first line of the file for field names.
 - **Ignore first line** - the Data Formatter ignores the first line of the file away without checking it for field names.
 - **Record length (for fixed width records only)** - the length of the records in the file. Valid values are between 1 and 8192 characters.

In a file with no line terminators, the Data Formatter re-processes the file with the new **Record length** and re-displays the records.

In a file with line terminators, the Data Formatter increases or decreases the size of the last field depending on the value in **Record length**.

The Preview window at the bottom of the dialog box shows a portion of the text file, with the first 30 records shown as columns and rows. You can view additional fields and records by scrolling.

In a file with no line terminators, each row is displayed with the exact number of characters as the **Record length** value. You can visually

determine the record length of non-terminated records by looking at the rows displayed.

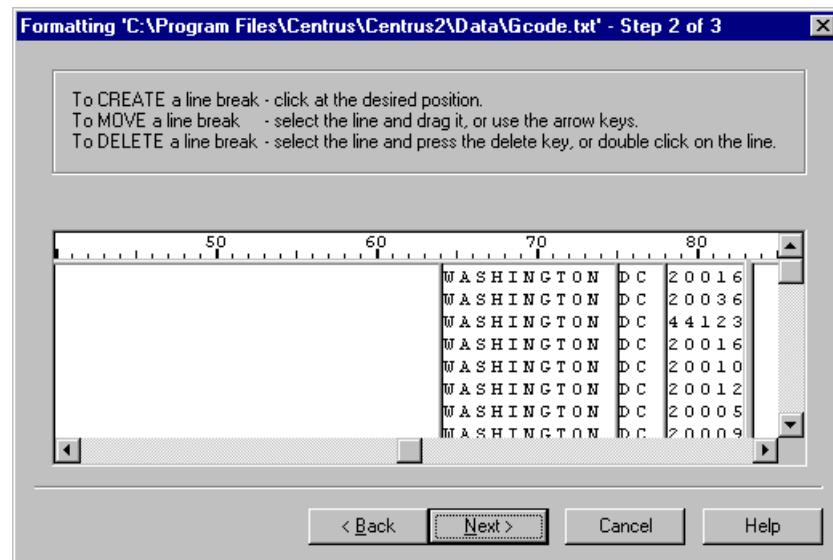
In a file with terminators, records are parsed and the width of the last field is affected by the **Record length** value.

3. Click **Next>**. The **Formatting 'filename' – Step 2 of 3** dialog box appears.

Step 2

Fixed Width

The following dialog box appears if the file's **File Type** is *Fixed Width*.

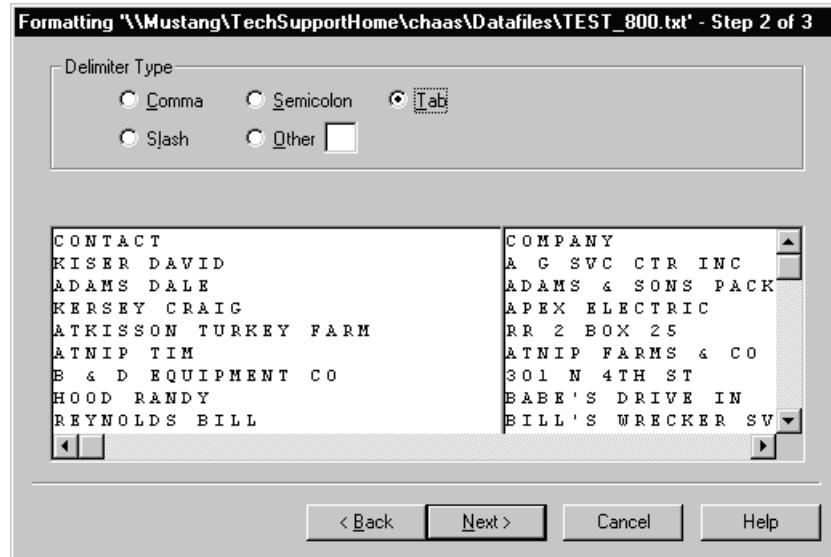


As you can see, the field breaks that Centrus Data Formatter determined are shown in the Preview window of this dialog box.

1. Follow the directions at the top of the dialog box to Create, Move, or Delete a line break.
2. Click **Next>**. The **Formatting 'filename' – Step 3 of 3** dialog box appears.

Delimited

If the file is *Delimited*, the dialog box shown below appears.



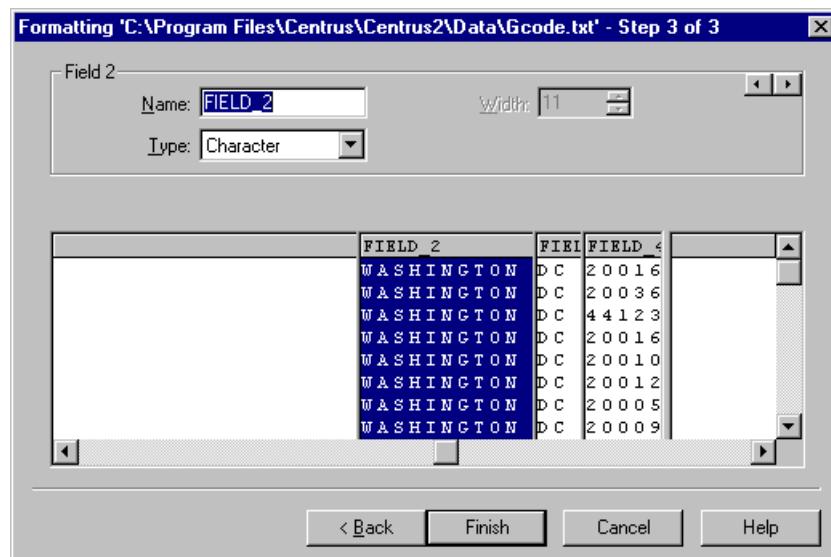
1. Accept the default **Delimiter Type** or change as needed.

You can enter any punctuation character as a delimiter in *Other*, except the characters for space (), quote ("), period (.), minus (-), or underscore (_).

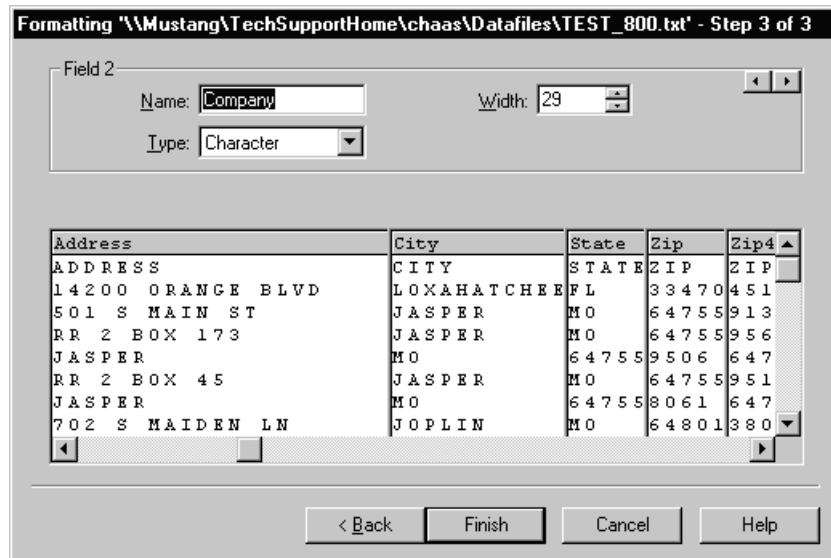
2. Click **Next >**. The **Formatting 'filename' - Step 3 of 3** dialog box appears.

Step 3

The following dialog box appears if the file's **File Type** is *Fixed Width*.



The following dialog box appears if the file's **File Type** is *Delimited*.



1. Select fields to change either in the Preview window or using the left/right arrows at the top of the dialog box.
2. Change the **Field Name** (helpful is you will be mapping fields in Centrus Desktop). Field names must begin with an alphabetic character or underscore, following by alphanumeric or underscore characters.
3. Make the following changes as needed:
 - **Type** - field type, either Character or Numeric.
 - **Decimals (Numeric fields only)** - the number of decimal places in the numeric field.
 - **Width (Delimited files only)** - changes the width of the selected field, padding or truncating the field. Changing this value also changes the **Record length**. Valid values are 1 - 2048.
4. Click **Finish** to write the .FMT file.

Note: If an .FMT file already exists, it will be overwritten.

About Format Files

The format file created by Centrus Data Formatter has the same filename as the ASCII address file, but uses the extension **.FMT** rather than **.TXT**. Since Centrus Data Formatter automates the creation and editing of format files, there's no need for you to "hand edit" format files. But understanding the structure of a format file may aid you in using Centrus Data Formatter. The basic structure is:

```
<0 or more lines of comments, each line beginning with a slash (/) or a
semicolon (;)>
TYPE=<delimiter type>
EOL=<end of line character>
FLN=<if the first line contains field names> (Line not present otherwise)
<field name>TAB<field type>TAB<field width>.<number of decimal places>
```

Delimited File Example

A sample format file for a delimited text file is shown below.

```
; Comments begin with semicolons (;) or slashes (/).
; Blank lines are ignored.
; Size is not required, as this is a comma delimited file.
TYPE=Comma
EOL=CRLF
FLN=N
First_Name      Character      15
Last_Name       Character      15
Address         Character      20
City            Character      15
State           Character      5
ZIP             Character      5
```

Sample data records are shown below. Note that all fields are not present for each record, but blank fields between two existing fields are still delimited. Note, too, that each field is enclosed by quotation marks, indicating literal strings. This prevents any commas that may be present within fields from being used as delimiters.

```
"Bob", "Doe", "123 Main St", "Nome", "AK", ""
"Carol", "Jones", "321 28th Ave, Apt. 22", "", "", "80301"
"Ted", "Tyler", "345 Mission St", "Longmont", "CO", "80503"
"Alice", "Brown", "PO BOX 123", "Boulder", "CO", ""
```

Fixed Width File Example

A sample format file for a fixed width text file is shown below.

```
; Comments begin with semicolons (;) or slashes (/).
; Blank lines are ignored.
; Field size IS required; this is a fixed width file.
; Note that numeric fields have decimals defined.

TYPE=Fixed
EOL=CRLF
FLN=N

First_Name          Character      15
Last_Name           Character      15
Address             Character      20
City                Character      15
State               Character      5
ZIP                 Character      5
Latitude            Numeric       11.6
Longitude           Numeric       11.6
Block_Grp           Character      10
Match_Cd            Character      10
Loc_Code            Character      10
```

A portion of some sample data records is shown below. Note that all fields are not present for each record. Blank fields between two existing fields are filled with blank spaces, preserving the layout of the data.

Bob	Doe	123	Main St	Nome	AK
Carol	Jones	321	28th Ave		80301
Ted	Tyler	345	Mission St	Longmont	CO 80503

Formatting ASCII Files on UNIX

This section describes how to format ASCII text files to use as source data in a UNIX environment. Centrus Merge/Purge uses the application Centrus Data Formatter to automate the process of creating and updating format files (.fmt) in a Windows environment. These format files describe the format of an ASCII file. There is no corresponding application in UNIX, so you must create the format file in a text editor.

Editing the Format Template

Create a file containing the text in **Template.fmt** (described on [page 918](#)). In your ASCII text file determine the type, end of line character, and whether the first line contains field names. Alter your .fmt file to reflect these properties.

For each field in the ASCII text file, create a field property line containing the field name, field type, field width and optional date format.

Use the commented lines as guides. Remember, the first character will usually suffice with a single space to separate values. But using the full keywords and lining up columns will make a file that is easier to read.

Note: If the database that created the ASCII text file allows you to export a ‘format’ file, you may be able to convert this file to a Centrus Merge/Purge .fmt file using a script.

ASCII File Properties

The format file specifies which type of file, which end of line characters to expect and whether the file has a first line containing field names.

- Type: Fixed, Tab, Comma, Semicolon or Slash.
You can also use “TYPE=” to set the delimiter to any punctuation character (except space, quote, period, minus, plus, underbar)
- EOL: LF, Return or CRLF
- FLN: Yes or No

Record Field Properties

The following are the Record Field properties:

Name	Type	Field width	optional date format
------	------	-------------	----------------------

- Name: May be any alphanumeric characters. It is best to keep names to 10 characters or less.
- Type: Numeric, Character or Date.
- Field width: may be in the form number, or number.number. In the first case the number is treated as a long, in the second as a floating point value. Decimals are ignored for Character fields.
- Date Format: this is a hint that tells Centrus Merge/Purge how to expect your fields containing dates to appear. The date format keywords are Day, Month, Century and Year.

If a date field in your data file looks like 02-14-98 then use mm-dd-yy, if it looks like 02/14/1998 use mm/dd/ccyy, and if it looks like Feb 2, 1998 then use mmm dd, ccyy.

General Guidelines

Follow these guidelines when creating a format file for an ASCII text file:

- Any line beginning with a ';' is considered a comment.
- First character is usually all that is needed.

The notable exceptions are 'Se' and 'Sl' which need to the first two characters to avoid ambiguity. In the examples below the barest minimum valid setting is underlined.

- The delimiter in the format file is whitespace, either spaces or tabs. The following lines are valid and equivalent:

FIRSTNAME	CHARACTER	15
FIRSTNAME	CHARACTER	15
FIRSTNAME	C 1	5

Sample Format File

The following is a sample format file:

```
; Centrus Merge/Purge ASCII format template
; Duplicates_Fix.txt

TYPE=Fixed
EOL=CRLF
FLN=No

FIRSTNAME    Character   15
LASTNAME     Character   15
ADDRESS       Character   35
CITY          Character   20
STATE         Character   2
ZIPCODE       Character   9
DATE          Date        8           mm-dd-yy
```

Template(fmt File

The following is a format template file for the ASCII Address File:

```
;TYPE=Comma
;TYPE=Fixed
;TYPE=Tab
;TYPE=Fixed
;TYPE=;
;TYPE=/
;EOL=CRLF
;EOL=LF
;EOL=RETURN
;FLN=y
;FLN=n
;FIELDNAME TYPE          WIDTH DATEFORMAT
;FIELDNAME      CHARACTER    15
;FIELDNAME      NUMBER       15
;FIELDNAME      NUMBER       15.7
;FIELDNAME      DATE         8      mm/dd/yy
;FIELDNAME      DATE         8      mm-dd-yy
;FIELDNAME      DATE         8      mmm dd, ccyy
```

Glossary

This glossary contains terms that are used frequently throughout the Centrus Merge/Purge library. When a term is defined using other terms that are also defined in the Glossary, those secondary terms are shown in *italics*.

A

Algorithm	See <i>field matching algorithm</i> .
Augmented fields	When Centrus Merge/Purge reads in source data, it copies the fields and adds its own augmented fields which help track the data throughout processing. See “ Augmented Fields ” on page 39 for a list and description of these fields.

B

Batch application	The Centrus Merge/Purge library allows the creation of applications that process many records from multiple input data sources in batch (non-interactive) mode.
Best record/best record criteria	The user-specified criteria which allows the Centrus Merge/Purge library to sort duplicates records from “best” to “worst” for the purpose of duplicate ranking. Currently source input list priority is the only available criteria.

Break group	The logical collection of all records that have the same index key value. Break groups are not used in the Centrus Merge/Purge library; instead, a <i>sliding window</i> method is used.
--------------------	--

C

Cardinality	In Centrus Merge/Purge, cardinality describes the number of unique key values for a given index. In a high-cardinality index, a large number of unique index key values are generated. Generally, higher-cardinality indexes are desired because they expose more true duplicates.
CMP	Centrus Merge/Purge
Character frequency algorithm	A type of <i>field matching algorithm</i> in which the frequency of characters appearing in the fields being compared is considered in evaluating how closely the fields match. This algorithm is particularly well suited for matching fields with character transpositions.
Cross-field comparison	Comparing two different fields between records (such as comparing the first name from one record to the last name of another record).

D

Data consolidation	The process of consolidating duplicate records into a “best record” by copying the “best” individual fields from the duplicates and creating a new record.
Data destination	A data destination is an object that surrounds an individual output table that is created and filled by the <i>table generator</i> . A data destination table may be any of the Centrus Merge/Purge table types.
Data input phase	The data input phase reads in data records in preparation for the <i>record matching</i> phase. It works with the <i>index key definitions</i> to create additional fields and field values for the data records, if necessary. The data input phase allows an application to give Centrus Merge/Purge its table (or tables) of input records for performing batch merge/purge. This is the first phase in most Centrus Merge/Purge applications, since the records must be input into the system before they can be ordered, matched, or correlated.
Data Input Table Repository (DITR)	The DITR is the database table populated by the data input phase.

Data list	A logical grouping of sets of records. Data lists are independent of the physical <i>data sources</i> from which the records come. Data lists can span <i>data sources</i> , and a single <i>data source</i> can have records contained in multiple data lists. The sets of records comprising a data list can come from one or more data sources and can include all or only a portion of the records from the individual data sources. Some examples of data lists are: a) all records from a particular data source, b) records for which a certain field contains a certain value, and c) records can not be included in any other data list.
Data list service	The data list service is an object type that performs data list-related functions for clients of data lists, such as a data input phase, record matcher, or pre-processed data source. The data list service is an aggregate of one or more data list objects. The data input phase uses this service to do initial data list assignment of new records. The record matcher uses it to determine if two records from the same data list should be compared. The preprocessed data source uses it to do data list assignment of reference data records at the beginning of an application using data lists.
Data source	An object containing a source of data records for input into either the data input phase or the record matching phase.
De-dupe	To detect duplicate records and either delete them or mark them as duplicates.
Dupe/Duplicate	A record that is not unique. It is considered to be the same entity (for example, person, firm, household) as at least one other data record in the set of <i>data sources</i> .
Dupe group	A collection of <i>duplicates</i> , all of which are <i>duplicates</i> of one another.
Duplicate candidate	A record which may be a duplicate of another record, but whose final <i>duplicate</i> status has not been completely determined. Centrus Merge/Purge creates a set of <i>duplicate candidates</i> during <i>record matching</i> as the pool from which actual <i>duplicates</i> are determined during the dupe groups phase.
Duplicate groups phase/dupe groups phase	The Centrus Merge/Purge library <i>phase</i> that determines master and subordinate records for a <i>dupe group</i> , and calculates the match score between a dupe group master and its subordinates. This information is stored in a dupe groups table. The dupe groups phase may, at the request of the application, remove any low-ranking member records from the dupe group.

Duplicates report/duplicates report phase

The duplicates report phase generates a report that contains duplicate records.

E

Edit distance algorithm

A type of *field matching algorithm* in which the number of characters that have to be changed in order to change the contents of one field into the contents of another is considered in evaluating how closely the fields match.

F

Field matching algorithm

The programmed logic that Centrus Merge/Purge uses to determine how well fields from two records match. Examples of *field matching algorithms* are *soundex* and *edit distance*. You can specify that one or more field matching algorithms be applied to any pair of record fields being compared. *Field matching algorithms* are part of the *field matching criteria* that you define. Field matching criteria include one or more algorithms.

Field matching criterion

A user-specified criterion which describes to Centrus Merge/Purge which fields from each record to compare, what algorithms to use in the comparison, and the weights to be associated with the outcomes of each algorithm. The record matcher uses the field matching criteria to determine if two records match.

Fielding

Identifying and extracting the individual fields out of a record or record field (for example, obtaining the street name out of a general address field).

Filtering

The process of selecting for inclusion certain records based on specified criteria. Input and output data records can be filtered. Filtering can be based upon single-field content, or by membership in a data list. See also *suppression*.

Functor

A function in Centrus Merge/Purge wherein the user supplies a C or Visual Basic callback.

I

Index key definition

The record ordering definition provided by the user that Centrus Merge/Purge uses to produce record orderings in preparation for record matching. The *index key definition* is a combination of fields and sub-fields used in the record ordering. All input records are processed for duplicates once for each *index key definition* you have specified.

Index key value generation/Index key values	Centrus Merge/Purge index key objects can generate index key values for records based on the records' field values and the <i>index key definitions</i> .
Indexed table	A database table that has been indexed according to user-defined <i>index key definitions</i> .
J	
Job/job configuration	The sum-total of the information which describes the input and output data, procedures, steps, configuration settings, algorithms, and reports for a the Centrus Merge/Purge library application execution. Typically, a user of the Centrus Merge/Purge library defines a variety of jobs to accomplish different goals. Jobs can be saved, edited, and executed at the user's discretion. The scheduling of a job is not part of the job configuration, but exists outside and independently of a <i>job</i> .
Job summary report/job summary report phase	The job summary report phase generates a report that summarizes the characteristics of the Merge/Purge job being executed.
K	
Keyboard distance algorithm	A type of <i>field matching algorithm</i> similar to the <i>edit distance algorithm</i> , in which the match result is reduced for each character that needs to be changed in proportion to the physical distance between the actual character and the desired character on a keyboard.
L	
List-by-list report/list-by-list report phase	The list-by-list report phase generates a report that compares data list membership. The report is actually a matrix that displays how many records any two data lists have in common. One use of this report is determining how many records from an in-house list match those in a rented list, so that a refund may be obtained for the duplicate records.
M	
Master duplicate	The <i>duplicate</i> record considered to be the "best" as defined by the user-supplied <i>best record</i> criteria. All other <i>duplicates</i> of the record are considered to be <i>subordinate duplicates</i> .

Match record	In a Centrus Merge/Purge library <i>real-time</i> application, the application provides a record to the Merge/Purge Library that is tested for uniqueness. The record is referred to as the <i>match record</i> .
Match result	A value representing the degree to which fields or records match, ranging from 0% for no match up to 100% for an exact match.
Match result object/table	In Centrus Merge/Purge, the match result object contains a match result table which the record matcher populates with information on potential duplicate matches.
Merge	The act of outputting the <i>unique</i> records from the set of <i>data sources</i> to an output file, leaving the original input <i>data sources</i> intact. See <i>Purge</i> .
Merge/Purge	The industry name for the process of bringing together multiple <i>data sources</i> , detecting <i>duplicate</i> and <i>unique</i> records, and either <i>merging</i> the data sources into an output database, or <i>purging</i> the <i>duplicates</i> from the original input <i>data sources</i> .
Multi-key record matching	Several <i>index key definitions</i> can be specified by the user for record ordering. The <i>index key definitions</i> are used to order records from <i>data sources</i> . The records are then processed by the <i>record matcher</i> , one pass over the records for each <i>index key definition</i> in the <i>index key</i> order, to detect <i>duplicate</i> records. The results of these passes are then merged together by the <i>dupe groups</i> phase to produce a list of record <i>duplicates</i> .

N

Numeric algorithm	A type of <i>field matching algorithm</i> in which two numeric fields are compared (such as a zip code). Either the numbers match exactly (100%) or not at all (0%).
Numeric string algorithm	A type of <i>field matching algorithm</i> created to compare address lines. It first matches numeric data in the string with the <i>numeric algorithm</i> . If the numeric data matches at 100%, the alphabetic data is matched using the <i>edit distance</i> and <i>character frequency algorithms</i> . The final match score is the average of the <i>edit distance</i> and <i>character frequency algorithms</i> .

O

Object	The basic functional unit of the Centrus Merge/Purge library. The library contains functions that allow the user to create, manipulate, and destroy objects. C programmers access objects through handles that are provided through object creation functions.
---------------	--

Output table	An output table is a user-specified database table that contains the records that are the results of processing a job with Centrus Merge/Purge.
P	
Phase	A phase is an object in the process of executing. All phases have objects, but not all objects have phases. The Merge/Purge Library phases are data input, record matching, table generation, dupe groups, and several report generation phases. Some phases may require other phases to have been previously run before they can be executed.
Preprocessed data source	A data source that contains everything needed to be used directly by the <i>record matching</i> , <i>dupe groups</i> , <i>report</i> , and <i>table generation phases</i> . It is not constructed, or intended, to be input record-by-record by the data input phase.
Primary key	When Centrus Merge/Purge reads in source data, every record is assigned a primary key value which helps in tracking the record through processing. If you have a thousand records in your database, primary key values of 1 – 1000 will be assigned.
Purge	The act of removing <i>subordinate duplicate</i> records from the original input <i>data sources</i> , or marking them as “deleted”. See <i>Merge</i> .
Q	
QmpAlg	A Centrus Merge/Purge <i>object</i> that stores <i>field matching algorithm</i> information.
QmpCrit	A Centrus Merge/Purge <i>object</i> that stores <i>field match criteria</i> information.
QmpDataDest	A Centrus Merge/Purge <i>object</i> that encapsulates and stores information about output tables.
QmpDataInp	A Centrus Merge/Purge <i>object</i> that stores information about the data input process. When the object is executing, it is called the <i>data input phase</i> .
QmpDataLst	A Centrus Merge/Purge <i>object</i> that stores data list information. A single QmpDataLst object defines an individual data list.
QmpDataLstSvc	A Centrus Merge/Purge <i>object</i> that stores information about the data list service. A single QmpDataLstSvc object defines an individual data list service.
QmpDataSrc	A Centrus Merge/Purge <i>object</i> that encapsulates and stores information about data input tables.
QmpDupGrpRec	A Centrus Merge/Purge <i>object</i> that stores <i>dupe groups</i> record information.

QmpDupGrps	A Centrus Merge/Purge <i>object</i> that encapsulates <i>dupe group</i> information and stores information about the process of determining dupe groups. When the object is executing, it is called the <i>dupe groups phase</i> .
QmpDupsRpt	A Centrus Merge/Purge <i>object</i> that encapsulates a <i>duplicates report</i> file and stores information about the process of creating a duplicates report. When the object is executing, it is called the <i>duplicates report phase</i> .
QmpGlb	A Centrus Merge/Purge “umbrella” <i>object</i> created to store diverse and unrelated information in one neat package. It currently handles library version, system log object, and license information.
QmpIdxKey	A Centrus Merge/Purge <i>object</i> that stores <i>index key definition</i> information.
QmpIniFil	A Centrus Merge/Purge <i>object</i> that stores .INI file information (such as the file path).
QmpJobRpt	A Centrus Merge/Purge <i>object</i> that encapsulates a <i>job summary report</i> file and stores information about the process of creating a job summary report. When the object is executing, it is called the <i>job summary report phase</i> .
QmpListByListRpt	A Centrus Merge/Purge <i>object</i> that encapsulates a <i>list-by-list report</i> file and stores information about the process of creating a list-by-list report. When the object is executing, it is called the <i>list-by-list report phase</i> .
QmpLog	A Centrus Merge/Purge <i>object</i> that controls access to and stores information about the error and trace systems. The QmpLog object is also used to output data to a file.
QmpMatRes	A Centrus Merge/Purge <i>object</i> that controls access to and stores information about the match result repository table.
QmpMatResRec	A Centrus Merge/Purge <i>object</i> that stores match result repository table record information. The match result repository table is accessed through a match result object.
QmpObj	A Centrus Merge/Purge root <i>object</i> that most other objects inherit characteristics from.
QmpPhase	A Centrus Merge/Purge <i>object</i> that stores information about a job <i>phase</i> . It is a root object that the other phase objects inherit characteristics from.
QmpRec	A Centrus Merge/Purge <i>object</i> that stores table record information. It is a root object that other record objects inherit characteristics from.
QmpRecMat	A Centrus Merge/Purge <i>object</i> that stores information about the <i>record matching</i> process. When the object is executing, it is called the <i>record matching phase</i> .

QmpRpt	A Centrus Merge/Purge <i>object</i> that encapsulates a report file and stores information about the process of creating a report. It is a root object that other report objects inherit characteristics from.
QmpTbl	A Centrus Merge/Purge <i>object</i> that controls access to and stores information about a table.
QmpTblGen	A Centrus Merge/Purge <i>object</i> that stores information about the table generation process. When the object is executing, it is called the <i>table generation phase</i> .
QmpTime	A Centrus Merge/Purge <i>object</i> that stores “stop-watch” information. A timer object may be started and stopped at any time, and may be queried for the start, stop, and elapsed time values.
QmpUniqsRpt	A Centrus Merge/Purge <i>object</i> that encapsulates a <i>uniques report</i> file and stores information about the process of creating a uniques report. When the object is executing, it is called the <i>uniques report phase</i> .
QmpVar	A Centrus Merge/Purge <i>object</i> that contains a variant variable (a union that is designed to contain one of several types of data). It has a type field and a unioned data field. It is a general purpose “bag” into which you can put a data value of nearly any type.

R

Ranking	The process of ordering <i>duplicate</i> records according to the <i>best record criteria</i> . The record that ranks highest is called the <i>master duplicate</i> , and all other duplicates of this record are called <i>subordinate duplicates</i> .
Real-time application	The Centrus Merge/Purge library allows the creation of applications and jobs which process individual data records interactively in <i>real-time</i> (as opposed to <i>batch</i>) to identify <i>duplicate candidates</i> .
Record matcher/record matching phase	The Centrus Merge/Purge library <i>phase</i> that compares records to determine whether they are <i>duplicates</i> . The record matching phase processes the records in multiple passes, once per <i>index key definition</i> , to examine the input records in different orders.
Record matching	The process of comparing two records to see if they are the same. <i>Record matching</i> is controlled by user-definable <i>field matching criteria</i> . In the Centrus Merge/Purge library, a <i>record matcher</i> is the entity that performs this activity.
Record prototype	The user supplied description of the fields in a record including their names, lengths, and types. The Centrus Merge/Purge library uses the record prototype to parse field expressions from <i>index key definitions</i> to generate <i>index key values</i> .

Report phase	The Centrus Merge/Purge library <i>phase</i> that generates a report and writes it to a file. The report types include duplicate, unique, job summary, and list-by-list.
---------------------	--

S

Sliding window	A data structure used by the <i>record matcher</i> to move sequentially through the records, comparing all of the records that are encompassed by the window to one another.
-----------------------	--

Sliding window size	The user-specified size of the <i>sliding window</i> , establishing the number of records that the window spans for record comparisons. The application may define a static or dynamic window size. Dynamic sliding windows may grow or shrink (up to application-defined parameters), depending upon the size of window needed.
----------------------------	--

Soundex algorithm	A type of <i>field matching algorithm</i> that compares two fields based on their pronunciation.
--------------------------	--

Source ID	The data source identifier, automatically assigned when you define your data sources. The number of Source IDs (SrcIDs) equals the number of data sources you have defined in the project. In Centrus Merge/Purge reports, the Source ID identifies the data source in which the duplicate record was found.
------------------	--

String matching algorithm	A type of <i>field matching algorithm</i> in which field strings are compared character by character for either an exact string match or no string match.
----------------------------------	---

Subordinate duplicate	Any <i>duplicates</i> that are not <i>master duplicates</i> . These are <i>duplicates</i> that are not the best of the set of <i>duplicates</i> as defined by the user-supplied <i>best record criteria</i> .
------------------------------	---

Suppression	Specifically excluding certain records based on user-supplied <i>suppression</i> criteria. Output data records can be suppressed. <i>Suppression</i> can be based upon single field content, or by single field comparison to data lists. See also <i>filtering</i> .
--------------------	---

T

Table generator/table generation phase	The Centrus Merge/Purge library <i>phase</i> that creates output tables. These tables contain unique records, master duplicates, subordinate duplicates, and various combinations of these record types (such as “survivors”—uniques and master duplicates). Additionally, this phase allows filtering of output based on each record’s list membership.
---	--

Threshold value	The record matching threshold value determines which records should be retained. Any record whose field matching results meet or exceed the record matching threshold is identified as a match. For example, if you set a 90% matching threshold, records that are found to have an 85% likelihood of matching are not recorded.
------------------------	--

U

Unique/unique record	A record which the Merge/Purge Library has determined has no duplicates in the input or reference records. It, and it alone, identifies a particularly entity (e.g., person, firm, household). No other records identify this same entity.
-----------------------------	--

Uniques report/uniques report phase	The uniques report phase generates a report that contains unique records (records that have no duplicates).
--	---

V

Variant	A Centrus Merge/Purge object that is capable of storing a variable's value in any one of several formats. It is useful for storing a piece of data in a single location without limiting whether that data is a long, float, string, char, or some other type.
----------------	--

W

Weighting factors	Two types exist: <i>field matching algorithm</i> weighting factors and <i>field matching criteria</i> weighting factors:
--------------------------	--

Field matching criteria weighting factors are coefficients that can be associated with each record's field matching results to provide a relative degree of importance across all of its field matching criteria results. This allows applications to "value" some record field match results more than others. These weighting factors are defined through the record matcher. The sum of the weighting factors must be 100.

Field matching criteria weighting factors are coefficients that can be associated with each record's field matching results to provide a relative degree of importance across all of its field matching criteria results. This allows applications to "value" some record field match results more than others. These weighting factors are defined through the record matcher. The sum of the weighting factors must be 100.

Work file

The Centrus Merge/Purge library's internal representation of the data which has been processed. *Work files* store intermediate results which are used internally for efficient record processing. The *work file* representation facilitates *record matching*, *duplicate detection*, *duplicate ranking*, and *data consolidation*. The Merge/Purge Library allows the saving of *work files* between job runs to optimize processing of subsequent jobs.

Index

A

algorithms

- character frequency 40, 920
- edit distance 107, 922
- keyboard distance 40, 107, 923
- numeric 41, 924
- numeric string 924
- soundex 41, 88, 107, 928
- string comparison 41, 107
- string matching 928

ASCII files

- formatting on UNIX 915

augmented field

B

batch application

- components 57
- creating a 57–64
- flow of 59

C

character frequency algorithm

cmpsimpl

- explanation 66–67
- flow 68–71

CodeBase table

48, 773

constants

- NULL 65–66, 97, 868
- OUTPUT_* 97, 267
- QFALSE 102
- QMS_MAX_PATH 101
- QRESULT_YES 64, 104
- QSEVERITY_FAIL 102
- QSEVERITY_SUCCEED 102
- QTRUE 102

contacting technical support

D

data destination

36, 50, 225

data input phase

19, 279

Data Input Table Repository (DITR)

- explanation of 49

data list

- explanation of 42

data list service

44, 349

data source

functor 36

preprocessed

- augmented fields 37

- explanation of 37

table 36, 81

data types

MpHnd 65, 73

QBOOL 74

QMS_ALGORITHM_TYPE 75

QMS_ALGORITHMS_RULE 74

QMS_ALIGN_OPTION 75

QMS_CRITERIA_RULE 77

QMS_DATLST_ACTION 78

QMS_DATLST_OUTPUT_PREF 79

QMS_DATLST_TYPE 80

QMS_DATSRC_TYPE 81

QMS_ERROR_ACTION 83

QMS_ERROR_HANDLER 84

QMS_ERROR_LEVEL 85

QMS_EVENT_TYPE 86

QMS_EVERY_NTHREC_FUNC 86

QMS_FILL_REC_FUNC 87

QMS_FLDEVAL_OPER 88

QMS_INDXKEY_XFORM_TYPE 88

QMS_MATCH_GROUPING 89

QMS_PHASE_STATE 91

QMS_PHASE_TYPES 90

QMS_REPORT_TEXT_POSITION

92

QMS_REPORT_TITLE_POSITION

92

QMS_VARIANT_TYPE 95

QRESULT 95

QVARSTRUCT 95

dupe

see duplicate

dupe group

6, 14, 25, 43, 171, 349, 439,

447, 448, 450, 456, 457, 458, 469,

470, 474, 495, 617

ID 170, 434, 447, 494, 616, 866

table 49, 429, 448, 449, 469, 486, 730

dupe groups phase

6, 14, 19, 25, 37, 49, 50,

178, 482, 483, 484, 485, 504, 602,

625

duplicate

master 56, 238, 245, 261, 262, 273, 274,

- 435, 439, 454, 455
 subordinate 26, 36, 56, 171, 239, 246,
 263, 275, 435, 436, 439, 465, 495,
 496, 617
- duplicate candidate** 3, 14
- duplicate group**
see dupe group
- duplicate groups phase.** *See* dupe groups phase
- duplicates report** 30, 493
- E**
- edit distance algorithm** 107, 922
- error checking** 104, 897–898
- error log file** 585, 596
- event**
 notification facility 50
 Nth record interval 50
- F**
- field matching**
 algorithm
 combining results of multiple algorithms 41
 criteria 40
- formatting**
 ASCII files 908, 915
- functor** 36
- functor data source** 36
- H**
- handles** 64–66, 73, 104
See also QmpDeclHnd macro
- header file** 64, 104
- I**
- index generation phase** 519
- index key**
 definition 6, 9–12, 20
 value 56
- installing Centrus Merge/Purge** 4
- J**
- job summary report** 27, 31, 50, 558
- K**
- keyboard distance algorithm** 40, 107, 923
- L**
- list-by-list report** 28, 32, 50, 574, 575
- M**
- macros**
 QmpDeclHnd 97, 868
 QmpUtilFailed 99, 870
 QmpUtilGetMessage 98, 871
 QmpUtilGetSeverity 98, 872
 QmpUtilGetSuccessFlag 98, 873
 QmpUtilSucceeded 98, 876
 QMS_CHDIR 99
 QMS_GETCWD 99
 QMS_MAX 100
 QMS_MAX3 100
 QMS_MIN 100
 QMS_MIN3 100
 QMS_STRICTCMP 101
 QMS_STRICTNICMP 101
- master duplicate** 56, 238, 245, 261, 262,
 273, 274, 435, 439, 454, 455
- match result repository** 49
- matching algorithms** 12
- merge/purge**
 how Centrus Merge/Purge works 6–7
 how traditional merge/purge works 5
- MpHnd data type** 65, 73
- N**
- Nth record event interval** 50
- NULL constant** 65–66, 97, 868
- numeric algorithm** 41, 924
- numeric string algorithm** 924
- O**
- object hierarchy chart** 65
- objects, using** 64–66
- OUTPUT_* constant** 97, 267
- P**
- phases**
 data input 19, 279
 dupe groups 6, 14, 19, 25, 37, 49, 50,
 178, 482, 483, 484, 485, 504, 602, 625
 duplicate groups. *See* dupe groups
 index generation 519
 record matching 6, 9, 14, 20
 table generation 17, 34, 37, 50, 56, 836
- preprocessed data source**

- augmented fields 37
 - explanation of 37
- primary key field 37, 174, 437, 499, 620

- Q**
- QBOOL data type 74
- QFALSE constant 102
- QmpDeclHnd macro 97, 868
- QmpUtilFailed macro 99, 870
- QmpUtilGetMessage macro 98, 871
- QmpUtilGetSeverity macro 98, 872
- QmpUtilGetSuccessFlag macro 98, 873
- QmpUtilSucceeded macro 98, 876
- QMS_ALGORITHM_TYPE data type 75
- QMS_ALGORITHMS_RULE data type 74
 - 74
- QMS_ALIGN_OPTION data type 75
- QMS_CHDIR macro 99
- QMS_CRITERIA_RULE data type 77
- QMS_DATLST_ACTION data type 78
- QMS_DATLST_OUTPUT_PREF data type 79
 - 79
- QMS_DATLST_TYPE data type 80
- QMS_DATSRC_TYPE data type 81
- QMS_ERROR_ACTION data type 83
- QMS_ERROR_HANDLER data type 84
- QMS_ERROR_LEVEL data type 85
- QMS_EVENT_TYPE data type 86
- QMS_EVERY_NTHREC_FUNC data type 86
 - 86
- QMS_FILL_REC_FUNC data type 87
- QMS_FLDEVAL_OPER data type 88
- QMS_GETCWD macro 99
- QMS_IDXKEY_XFORM_TYPE data type 88
 - 88
- QMS_MATCH_GROUPING data type 89
 - 89
- QMS_MAX macro 100
- QMS_MAX_PATH constant 101
- QMS_MAX3 macro 100
- QMS_MIN macro 100
- QMS_MIN3 macro 100
- QMS_PHASE_STATE data type 91
- QMS_PHASE_TYPES data type 90
- QMS_REPORT_TEXT_POSITION data type 92
 - 92
- QMS_REPORT_TITLE_POSITION data type 92
 - 92
- QMS_STRICTCMP macro 101
- QMS_STRNICMP macro 101
- QMS_VARIANT_TYPE data type 95
- QRESULT data type 95
- QRESULT return codes explanation 104, 897–898
- failure (fatal) 906
- failure (severe) 900–906
- failure (warning) 899
- last chance return code 906
 - success (informational) 899
- QRESULT_YES constant 64, 104
- QSEVERITY_FAIL constant 102
- QSEVERITY_SUCCEED constant 102
- QTRUE constant 102
- QVARSTRUCT data type 95

- R**
- record matching phase 6, 9, 14, 20
- record, prototype 269, 311, 319, 320, 321, 322, 323, 324, 329, 367, 373, 386, 401, 437
- reports
 - duplicates 30, 493
 - job summary 27, 31, 50, 558
 - list-by-list 28, 32, 50, 574, 575
 - uniques 28, 34, 860, 863, 864

- S**
- sliding window
 - explanation 6, 9–12, 20
 - setting window size 717, 750
- soundex algorithm 41, 88, 107, 928
- source ID 37, 344, 437
- string comparison algorithm 41, 107
- string matching algorithm 928
- subordinate duplicate 26, 36, 56, 171, 239, 246, 263, 275, 435, 436, 439, 465, 495, 496, 617

- T**
- table data source 36, 81
- table generation phase 17, 34, 37, 50, 56, 836
- tables
 - CodeBase 48, 773
 - Data Input Table Repository (DITR)
 - explanation of 49
 - explanation of 48
- technical support
 - contacting 4
- template.fmt file 916
- threshold value 42, 74, 203, 733
- trace log file 52

U

uniques report 28, 34, 860, 863, 864

UNIX

 formatting ASCII files on 915

V

variant 95, 95–96, 675, 877