

**Mumps Programming Language
Interpreter, Compiler,
&
MDH C++ Class Library
User's Guide**

Kevin C. O'Kane

kc.okane@gmail.com
okane@uni.edu

<http://threadsafebooks.com/>
<http://www.cs.uni.edu/~okane/>

January 29, 2026

Table of Contents

1 Distribution Introduction.....	13
1.1 DISTRO FILE STRUCTURE.....	13
1.2 INTERPRETER VS COMPILER VS MDH TOOLKIT.....	15
1.3 SYSTEM SOFTWARE REQUIREMENTS.....	15
1.4 MUMPS LANGUAGE RESOURCES.....	16
2 Installation.....	17
2.1 USING AN INSTALLER.....	17
2.2 COMPILING FROM SOURCE CODE.....	18
2.3 COMPILATION OPTIONS.....	19
2.4 MATH OPTIONS.....	19
2.4.1 Hardware Based Math.....	19
2.4.2 Extended Precision Math.....	20
2.5 OTHER CONFIGURE OPTIONS.....	20
3 Global Array Database Overview.....	22
3.1 HISTORY.....	22
3.2 THE TREE STRUCTURED MEDICAL RECORD.....	22
3.3 GLOBAL ARRAY REPRESENTATION.....	23
4 Native Database Global Array Option.....	25
4.1 OVERVIEW.....	25
4.2 NATIVE DATABASE CONFIGURATION.....	25
4.2.1 --with-maxglobal=val.....	25
4.2.2 --with-cache=VAL.....	25
4.2.3 --with-block=blksize.....	26
5 Sqlite3 Database Global Array Option.....	27
5.1 OVERVIEW.....	27
5.2 GLOBAL ARRAY TO SQLITE3 RELATIONAL DATABASE MAPPING.....	27
5.3 SQLITE3 DATABASE CONFIGURATION.....	29
5.3.1 Sqlite3 Database File.....	29
5.3.2 Sqlite2 Compile Options.....	29
5.3.2.1 --with-dbname=name.....	29
5.3.2.2 --with-index_size=number.....	29
5.3.2.3 --with-data_size=nbr.....	29
5.3.2.4 --with-index-max=nbr.....	29
5.3.2.5 --with-dbfile=name.....	29
5.4 SQLITE3 PERFORMANCE TUNING.....	29
5.5 SQLITE3 COMMAND LINE INTERPRETER.....	30
5.6 SQLITE3 DATABASE LANGUAGE EXTENSIONS.....	30
5.6.1 \$zSqlite.....	30
5.6.2 \$zSqlite("begin transaction").....	30
5.6.3 \$zSqlite("commit transaction").....	30
5.6.4 \$zSqlite("savepoint",[savepoint_name]).....	30
5.6.5 \$zSqlite("rollback",[savepoint]).....	31
5.6.6 \$zSqlite("pragma",option).....	31
5.6.7 \$zsqlOpen.....	31
5.6.8 \$zNative.....	31
6 Interpreter & Compiler.....	32

6.1 INTERPRETER.....	32
6.1.1 Running the Mumps CLI Interpreter.....	32
6.1.2 Interpreting a Mumps Program.....	32
6.2 COMPILER.....	33
6.2.1 How to Compile and Run a Mumps Program.....	33
6.2.2 Compiler Error Messages.....	34
6.2.3 Global Array Storage in Compiled Programs.....	34
6.2.4 Compiler Performance and Interoperability.....	34
7 Interpreter & Compiler Z Functions, and System Variables.....	35
7.1 SYSTEM VARIABLES.....	35
7.1.1.1 \$zProgram.....	35
7.2 DATABASE FUNCTIONS.....	35
7.2.1 \$zGlobal(<i>string</i>).....	35
7.2.2 \$zDBfile.....	35
7.2.3 \$zDBname.....	35
7.2.4 \$zIndexsize.....	35
7.2.5 \$zIndexmax.....	35
7.2.6 \$zDatasize.....	35
7.2.7 \$zDatabase.....	35
7.3 BASH FUNCTIONS.....	36
7.3.1 \$zbasename(arg1[,arg2]).....	36
7.3.2 \$zfiletest(arg1,arg2).....	36
7.4 MATH FUNCTIONS.....	36
7.4.1 \$zabs(arg) absolute value.....	36
7.4.2 \$zacos(arg) arc cosine.....	36
7.4.3 \$zasin(arg) Arc sine.....	36
7.4.4 \$atan(arg) Arc tangent.....	37
7.4.5 \$zcos(arg) Cosine.....	37
7.4.6 \$zexp(arg) Exponential.....	37
7.4.7 \$zexp2(arg) Exponential base 2.....	37
7.4.8 \$zexp10(arg) Exponential base 10.....	37
7.4.9 \$zlog(arg) Natural log.....	37
7.4.10 \$zlog2(arg) Base 2 log.....	37
7.4.11 \$zlog10(arg) Base 10 log.....	37
7.4.12 \$zpow(arg1,arg2) Power function.....	37
7.4.13 \$zsqr(<i>arg</i>) Square root.....	37
7.4.14 \$zsin(arg) Sine function.....	37
7.4.15 \$ztan(arg) Tangent function.....	37
7.5 DATE FUNCTIONS.....	37
7.5.1 \$zdate(or \$zd) formatted date string.....	37
7.5.2 \$zd1 numeric internal date.....	37
7.5.3 \$zd2(<i>InternalDate</i>) date conversion.....	38
7.5.4 \$zd3(<i>Year,Month,Day</i>) Julian date.....	38
7.5.5 \$zd4(<i>Year,DayOfYear</i>) Julian to Gregorian.....	38
7.5.6 \$zd5(<i>Year, Month, Day</i>) comma listed date.....	38
7.5.7 \$zd6 hour:minute.....	38
7.5.8 \$zd7 hyphenated date.....	38
7.5.9 \$zd8 hyphenated date with time.....	38
7.6 SPECIAL PURPOSE FUNCTIONS.....	38
7.6.1 \$zb(arg) remove blanks.....	38
7.6.2 \$zchdir(directory_path) change directory.....	38
7.6.3 \$zCurrentFile Current Mumps File.....	38
7.6.4 \$zdump[(filename)] dump global arrays.....	39

7.6.5 \$zrestore[(arg)] restore globals.....	39
7.6.6 \$zfile(arg) file exists test.....	39
7.6.7 \$zflush flush Btree buffers.....	39
7.6.8 \$zgetenv(arg) get environment variable.....	39
7.6.9 \$zhtml(arg) encode HTML string.....	39
7.6.10 \$zhit global array cache hit ratio.....	39
7.6.11 \$zlower(string) convert to lower case.....	39
7.6.12 \$znormal(arg1[,arg2]) word normalization.....	40
7.6.13 \$zNoBlanks(arg) remove all blanks.....	40
7.6.14 \$zpad(arg1,arg2) left justify with padding.....	40
7.6.15 \$zseek(arg).....	40
7.6.16 \$zsrand(arg).....	41
7.6.17 \$zstem(arg).....	41
7.6.18 \$zsystem(arg).....	41
7.6.19 \$ztell.....	41
7.6.20 \$zu(expression).....	41
7.6.21 \$zwi(arg).....	41
7.6.22 \$zwn extract words from buffer.....	41
7.6.23 \$zwp extract words from buffer.....	41
7.6.24 \$zws(string) initialize internal buffer.....	41
7.7 SCAN FUNCTIONS.....	42
7.7.1 \$zzScan.....	42
7.7.2 \$zzScanAlnum.....	42
7.7.3 \$zzInput(var).....	42
7.8 VECTOR AND MATRIX FUNCTIONS.....	43
7.8.1 \$zzAvg(vector).....	43
7.8.2 \$zzCentroid(gblMatrix,gblRef).....	43
7.8.3 \$zzCount(gblVector).....	44
7.8.4 \$zzMax(gbl).....	44
7.8.5 \$zzMin(gbl).....	44
7.8.6 \$zzMultiply(gbl1,gbl2,gbl3).....	45
7.8.7 \$zzSum(gblVector).....	45
7.8.8 \$zzTranspose(gblMatrix1,gblMatrix2).....	45
7.9 TEXT PROCESSING FUNCTIONS.....	45
7.9.1 Similarity Functions.....	45
7.9.1.1 \$zzCosine(gbl1,gbl2).....	45
7.9.1.2 \$zzSim1(gbl1,gbl2).....	45
7.9.1.3 \$zzDice(gbl1,gbl2).....	45
7.9.1.4 \$zzJaccard(gbl1,gbl2).....	45
7.9.2 \$zzBMGSearch(arg1,arg2).....	46
7.9.3 \$zPerlMatch(string,pattern).....	47
7.9.4 \$zReplace(string,pattern,replacement).....	48
7.9.5 \$zShred(string,length).....	48
7.9.6 \$zShredQuery(string,length).....	48
7.9.7 \$zzSoundex(s1).....	50
7.9.8 \$zSmithWaterman(s1,s2,algn,mat,gap,noMatch,match).....	50
7.9.9 \$zzIDF(global,doccunt).....	51
7.9.10 Correlation Functions.....	51
7.9.10.1 \$zzTermCorrelate(global1,global2).....	51
7.9.10.2 \$zzDocCorrelate(gblref1,gblref2,mthd,thrshld).....	53
7.9.11 Stop and Synonym Functions.....	53
7.9.11.1 \$zStopInit(arg).....	53
7.9.11.2 \$zStopLookup(word).....	53

7.9.11.3 \$zSynInit(fileName).....	53
7.9.11.4 \$zSynLookup(word).....	53
7.10 GTK RELATED FUNCTIONS.....	55
7.10.1 \$z~mdh~toggle~button~set~active(ToggleButtonReference,intVal).....	55
7.10.2 \$z~mdh~dialog~new~with~buttons(ParentWindowRef,dialog).....	55
7.10.3 \$z~mdh~entry~get~text(EntryReference).....	55
7.10.4 \$z~mdh~entry~set~text(EntryReference,value).....	55
7.10.5 \$z~mdh~text~buffer~set~text(TextBufferReference,string).....	55
7.10.6 \$z~mdh~label~set~text(LabelReference,string).....	55
7.10.7 \$z~mdh~tree~selection~get~selected(TreeModelReference,column).....	55
7.10.8 \$z~mdh~tree~store~clear(TreeStoreReference).....	55
7.10.9 \$z~mdh~tree~level~add(TreeStoreReference,treeDepth,index,data[,...]).....	55
7.10.10 \$z~mdh~spin~button~get~value(SpinButtonReference).....	55
7.10.11 \$z~mdh~spin~button~set~value(SpinButtonReference,number).....	55
7.10.12 \$z~mdh~widget~hide(widgetReference).....	55
7.10.13 \$z~mdh~widget~show(widgetReference).....	56
8 Interpreter & Compiler Implementation Notes.....	57
8.1 SOURCE CODE FORMAT.....	57
8.2 MODULO OPERATOR.....	57
8.3 GOTO COMMAND.....	57
8.4 NOTES ON ARITHMETIC PRECISION.....	57
8.5 \$FNUMBER().....	57
8.6 EXPONENTIAL FORMAT NUMBERS.....	57
8.7 EXTENDED ARITHMETIC PRECISION.....	57
8.8 FLOATING POINT PRECISION.....	57
8.9 INTEGER PRECISION.....	58
8.10 EXTENDED PRECISION PERFORMANCE.....	58
8.11 ROUNDING.....	58
8.12 NEW COMMAND.....	58
8.13 RUNTIME SYMBOL TABLE.....	58
8.14 FORMS OF THE NEW COMMAND.....	58
8.14.1 New Command with No Arguments.....	58
8.14.2 New Command with Arguments.....	60
8.14.3 New Command with Comma List of Variable Names.....	60
8.14.4 New Command with Parenthesized List of Variable Names.....	61
8.15 KILL COMMAND.....	61
8.16 FOR COMMAND EXTENSIONS.....	61
8.16.1 Break and Quit.....	62
8.17 LOCK COMMAND WITH SQL.....	64
8.18 LOCK COMMAND IN NATIVE DATABASE MODE.....	65
8.19 NAKED INDICATOR.....	65
8.20 JOB COMMAND.....	65
8.21 FILE NAMES CONTAINING DIRECTORY INFORMATION.....	65
8.22 FILE NAMES.....	65
8.23 ARRAY INDEX COLLATING SEQUENCE.....	65
8.24 SUBROUTINE & FUNCTION CALLS.....	66
8.25 \$FNUMBER() FUNCTION.....	67
8.26 \$SELECT() FUNCTION.....	67
8.27 COMPILING LARGE PROGRAMS.....	67
8.28 EMBEDDED EXPRESSIONS.....	68
8.29 INLINE C++ CODE (COMPILER ONLY).....	68
8.30 FUNCTIONS.....	68
8.30.1 Call by Value.....	69
8.30.2 Call by Reference.....	69

8.31 SHELL COMMANDS.....	71
8.31.1 shell/p.....	71
8.31.2 shell/g.....	71
8.31.3 shell.....	71
8.32 DATABASE <i>EXPR</i>	71
8.33 ZHALT RETURN_CODE.....	72
9 The Multi-Dimensional and Hierarchical Database Toolkit.....	73
9.1 INTRODUCTION.....	73
9.2 INSTALLATION.....	73
9.2.1 Global Array Database.....	73
9.3 COMPILING PROGRAMS.....	74
9.4 WRITING C++ MDH PROGRAMS.....	74
9.5 MDH IMPLEMENTATION OF GLOBALS.....	74
9.6 GLOBAL ARRAYS AS C++ TREES AND MATRICES.....	75
9.6.1 Traditional Matrix.....	75
9.6.2 Alternative Matrix View.....	76
9.6.3 Tree View.....	77
9.7 ACCESSING GLOBAL ARRAYS.....	78
9.8 GLOBAL ARRAY INDICES.....	78
9.9 NAVIGATING GLOBALS.....	79
9.10 LOCKING THE DATA BASE.....	82
9.11 INVOKING THE MUMPS INTERPRETER.....	82
9.12 MISCELLANEOUS FUNCTIONS AND SYSTEM VARIABLES.....	83
9.12.1 GTK / Glade functions.....	83
9.12.1.1 void mdh_tree_level_add(GtkTreeStore *tree, int depth, char * col1 [, char *col2 ...]);.....	83
9.12.1.2 int mdh_dialog_new_with_buttons(GtkWindow *win, char * text).....	83
9.12.1.3 int mdh_toggle_button_get_active(GtkToggleButton *b).....	83
9.12.1.4 char * mdh_entry_get_text(GtkEntry *e, char * txt).....	83
9.12.1.5 void mdh_toggle_button_set_active(GtkToggleButton *b, int v).....	83
9.12.1.6 void mdh_entry_set_text(GtkEntry *e, char * txt).....	83
9.12.1.7 void mdh_text_buffer_set_text(GtkTextBuffer *t, char * txt).....	83
9.12.1.8 void mdh_label_set_text(GtkLabel *l, char * txt).....	83
9.12.1.9 void mdh_widget_hide(GtkWidget *w).....	83
9.12.1.10 void mdh_widget_show(GtkWidget *w).....	83
9.12.1.11 char * mdh_tree_selection_get_selected(GtkTreeSelection *t, int col, char *txt).....	84
9.12.1.12 void mdh_tree_store_clear(GtkTreeStore *t).....	84
9.12.1.13 double mdh_spin_button_get_value(GtkSpinButton *s).....	84
9.12.1.14 void mdh_spin_button_set_value(GtkSpinButton *s, double v).....	84
9.12.1.15 \$z~mdh~toggle~button~get~active(ToggleButtonReference).....	84
9.12.2 Boyer-Moore-Gosper Functions.....	84
9.12.3 cvt().....	86
9.12.4 xecute() and command().....	86
9.12.5 ErrorMessage().....	86
9.12.5.1 Error Exceptions.....	86
9.12.6 HitRatio().....	87
9.12.7 Hashing functions.....	87
9.12.8 Dump Global Array Database.....	88
9.12.9 Stream Output Function ostream.....	88
9.12.10 Smith-Waterman Alignment Function.....	88
9.12.11 Stop list functions: StopINIT(), StopLookup().....	89
9.12.12 int \$test.....	89
9.12.13 Xecute().....	90
9.12.14 Zseek(), Ztell().....	90

10 Class mstring.....	91
10.1 MSTRING FUNCTIONS.....	91
10.1.1 Ascii Function.....	91
10.1.2 begins Function.....	91
10.1.3 c_str Function.....	91
10.1.4 decorate Function.....	91
10.1.5 EncodeHTML Function.....	91
10.1.6 ends Function.....	92
10.1.7 Eval Function.....	92
10.1.8 Extract Function.....	92
10.1.9 Find Function.....	92
10.1.10 Horolog Function.....	92
10.1.11 Justify Function.....	93
10.1.12 Length Function.....	93
10.1.13 mcvT Function.....	93
10.1.14 Pattern Function.....	93
10.1.15 Perl Function.....	94
10.1.16 Piece Function.....	94
10.1.17 ReadLine Function.....	94
10.1.18 replace Function.....	95
10.1.19 ScanAlnum Function.....	95
10.1.20 shred Function.....	95
10.1.21 ShredQuery Function.....	95
10.1.22 Stem Function.....	96
10.1.23 Synonym Functions: SymInit(), SYN().....	97
10.1.24 SymGet SymPut Functions.....	97
10.1.25 s_str Function.....	97
10.1.26 Translate.....	97
10.1.27 Token Function.....	97
10.1.28 Translate Function.....	98
10.1.29 Basic mstring Examples.....	98
10.2 ASSIGNMENT FROM OTHER DATA TYPES.....	99
10.3 ARITHMETIC OPERATIONS ON MSTRING.....	100
10.4 MISCELLANEOUS MSTRING RULES.....	102
11 Overloaded mstring Operators.....	103
11.1 ADDITION.....	103
11.2 SUBTRACTION.....	103
11.3 MULTIPLICATION.....	103
11.4 DIVISION.....	104
11.5 INCREMENT/DECREMENT.....	104
11.6 UNARY OPERATIONS.....	104
11.7 MODULO.....	104
11.8 CONCATENATION.....	104
11.8.1 Relational.....	105
12 Class Global.....	107
12.1 ASSIGNMENT OPERATORS ON GLOBALS.....	107
12.2 ARITHMETIC OPERATORS ON GLOBALS.....	108
12.3 ACCESSING THE VALUE STORED IN A GLOBAL ARRAY ELEMENT.....	109
12.4 DIRECT Btree ACCESS.....	110
12.5 FUNCTIONS ON CLASS GLOBAL.....	111
12.5.1 Data().....	111
12.5.2 TreePrint().....	112

12.5.3 UnLock.....	113
12.5.4 Count.....	113
12.5.5 GlobalGet(), GlobalData(), GlobalSet().....	114
12.5.6 double HitRatio(void).....	114
12.5.7 Kill.....	115
12.5.8 Length.....	115
12.5.9 Max.....	115
12.5.10 Merge.....	115
12.5.11 Min.....	116
12.5.12 Multiply.....	116
12.5.13 Name.....	117
12.5.14 Order.....	117
12.5.15 Avg.....	118
12.5.16 Locks.....	119
12.5.17 GlobalClose.....	119
12.5.18 Query functions.....	119
12.5.19 Similarity functions.....	122
12.5.20 Transpose.....	125
12.5.21 Centroid.....	125
12.5.22 Correlation Functions.....	126
12.5.23 IDF.....	130
12.5.24 Sum.....	130
12.5.25 Btree.....	131
13 Overloaded global Operators.....	133
13.1 ASSIGNMENT.....	133
13.2 ADDITION.....	133
13.3 SUBTRACTION.....	133
13.4 MULTIPLICATION.....	134
13.5 DIVISION.....	134
13.6 INCREMENT/DECREMENT.....	135
13.7 UNARY.....	135
13.8 RELATIONAL.....	135
13.9 CASTS.....	136
14 GTK Desktop GUI Apps.....	138
14.1 GLADE GUI DESIGN TOOL.....	138
14.2 GTK EXAMPLE.....	139
14.2.1 Glade Design Tool.....	139
14.2.2 Building A Mumps App from The Glade XML File.....	141
14.2.2.1 gtk1.h.....	142
14.2.2.2 gtk2.h.....	142
14.2.2.3 gtk3.h.....	142
14.2.2.4 gtk4.h.....	143
14.2.2.5 gtk.mps.....	143
14.2.2.6 on.toggle1.toggled.mps.....	143
15 Pattern Matching.....	144
15.1 MUMPS 95 PATTERN MATCHING.....	144
15.2 USING PERL REGULAR EXPRESSIONS.....	144
15.3 EXAMPLES.....	145
16 Mumps Language Basics.....	146
16.1 MUMPS SYNTAX WARNING.....	146

16.2 VARIABLES.....	146
16.3 STRING CONSTANTS.....	147
16.4 NUMERIC CONSTANTS.....	148
16.5 MIXED STRINGS & NUMERIC CONSTANTS.....	148
16.6 LOGICAL VALUES.....	148
16.7 ARRAYS.....	149
16.8 HIERARCHICAL DATA.....	150
16.9 MUMPS COMMANDS.....	152
16.10 POSTCONDITIONALS.....	153
16.11 OPERATOR PRECEDENCE.....	154
16.12 OPERATORS.....	154
16.13 COMMANDS.....	156
16.14 SYNTAX RULES.....	158
16.15 SYNTAX EXTENSIONS.....	158
16.16 BLOCKS.....	159
16.17 QUIT.....	161
16.18 BREAK.....	162
16.19 CLOSE.....	163
16.20 DO.....	163
16.21 ELSE.....	163
16.22 FOR.....	163
16.23 GOTO.....	165
16.24 HALT.....	165
16.25 HANG.....	165
16.26 IF.....	165
16.27 IF AND ELSE.....	166
16.28 JOB.....	166
16.29 KILL.....	166
16.30 LOCK.....	166
16.31 MERGE.....	166
16.32 NEW.....	167
16.33 OPEN, USE AND UNIT NUMBERS.....	167
16.34 I/O FORMAT CODES.....	168
16.35 READ.....	168
16.36 SET.....	168
16.37 DATABASE TRANSACTION COMMANDS.....	169
16.38 USE.....	169
16.39 VIEW.....	169
16.40 WRITE.....	169
16.41 XECUTE.....	169
16.42 Z... COMMANDS.....	170
16.43 NAVIGATING ARRAYS.....	170
16.44 INDIRECTION.....	171
16.45 SUBROUTINES.....	171
16.46 FUNCTIONS.....	173
16.47 BUILTIN FUNCTIONS & VARIABLES.....	173
16.47.1 Intrinsic Special Variables.....	174
16.47.2 Intrinsic Functions.....	174
16.47.2.1 \$Ascii().....	174
16.47.2.2 \$Char().....	175
16.47.2.3 \$Data(var).....	175
16.47.2.4 \$Extract(e1,i2[,i3]).....	175
16.47.2.5 \$Find(e1,e2[,i3]).....	175
16.47.2.6 \$FNumber(a,b[,c]).....	176
16.47.2.7 \$Get(var[,default]).....	176
16.47.2.8 \$Justify(str,fld[,dec]).....	176

16.47.2.9 \$Length(exp[,str]).....	176
16.47.2.10 \$Name(arrayVar[,int]).....	177
16.47.2.11 \$Order(vn[,1]).....	177
16.47.2.12 \$Piece(str,pat[,i3[,i4]]).....	177
16.47.2.13 \$QLength(string).....	178
16.47.2.14 \$QSubscript(string, int).....	178
16.47.2.15 \$QQuery(array ref).....	178
16.47.2.16 \$Random(int).....	179
16.47.2.17 \$REverse(str).....	179
16.47.2.18 \$Select(texp1:exp1[,...]).....	179
16.47.2.19 \$STack(intexp1[,...]).....	179
16.47.2.20 \$Test(entryRef).....	179
16.47.2.21 \$TRanslate(exp1[,exp2[,exp3]).....	179
16.47.2.22 \$View().....	180
16.47.2.23 \$Z...().....	180
16.48 PROGRAMMING EXAMPLE.....	180
17 Licenses.....	184
17.1 GNU LICENSES.....	184
17.1.1 GNU General Public License.....	184
17.1.2 GNU Free Documentation License.....	189
17.1.3 GNU LESSER GENERAL PUBLIC LICENSE.....	194
17.2 PERL COMPATIBLE REGULAR EXPRESSION LIBRARY LICENSE.....	201

Index of Figures

Figure 1 Directory Map of Mumps Distro.....	15
Figure 2 Tree Structured Medical Record.....	22
Figure 3 Relational Database Medical Record.....	23
Figure 4 Example C++ Code.....	34
Figure 5 \$Zb() Examples.....	38
Figure 6 \$Zseek() Examples.....	40
Figure 7 \$Zwi() Examples.....	42
Figure 8 Scan Functions Examples.....	43
Figure 9 \$zzAvg() Example.....	43
Figure 10 \$zzCentroid() Example.....	44
Figure 11 \$zzCount() Example.....	44
Figure 12 \$zzMax() Example.....	44
Figure 13 \$zzMin() Example.....	44
Figure 14 Similarity Formulae.....	45
Figure 15 Similarity Functions.....	46
Figure 16 \$zzBMGSearch() Example.....	46
Figure 17 \$zzMultiply() Example.....	47
Figure 18 \$zzSum() Example.....	47
Figure 19 \$zzTranspose() Example.....	48
Figure 20 \$zPerlMatch() Example.....	48
Figure 21 \$zReplace() Example.....	49
Figure 22 \$zShred() Example.....	49
Figure 23 \$ShredQuery() Example.....	50
Figure 24 \$zSmithWaterman() Example.....	51
Figure 25 \$zzIDF() Example.....	51

Figure 26 \$zTermCorrelate() Example.....	53
Figure 27 \$zDocCorrelate()Example.....	54
Figure 28 Stop List Functions.....	55
Figure 29 new Command without Arguments.....	60
Figure 30 new Command with Comma List.....	61
Figure 31 new Command with Parenthesized List.....	61
Figure 32 Subroutine/Function Calls.....	67
Figure 33 Local Functions.....	69
Figure 34 Call by Value Functions.....	69
Figure 35 Call by Reference Functions.....	70
Figure 36 Function Return Values.....	70
Figure 37 Shell Command Example.....	71
Figure 38 Global Array as a Matrix.....	76
Figure 39 Global Array as Matrix with Numeric Subscripts.....	76
Figure 40 Global Array as Matrix with Additional Nodes.....	77
Figure 41 Global Array as Sparse Matrix.....	77
Figure 42 Tabular View of Tree.....	77
Figure 43 Global Array Tree.....	78
Figure 44 Navigating Global Arrays - Data().....	79
Figure 45 Navigating Global Arrays - Order().....	80
Figure 46 Global Array Navigation Example.....	80
Figure 47 Hierarchical Global Array Example.....	82
Figure 48 Boyer-Moore Example.....	84
Figure 49 Exceptions Examples.....	87
Figure 50 Smith-Waterman Example.....	89
Figure 51 Mumps Pattern Codes.....	93
Figure 52 Shred Function.....	95
Figure 53 ShredQuery.....	96
Figure 54 mstring Examples.....	99
Figure 55 mstring Assighment Examples.....	100
Figure 56 mstring Arithmetic Operations.....	102
Figure 57 mstring Operator Overloads.....	106
Figure 58 Exceptions.....	107
Figure 59 Example Global Array Arithmetic.....	109
Figure 60 Accessing Global Array Data Example.....	109
Figure 61 BTREE Example.....	111
Figure 62 TreePrint.....	112
Figure 63 TreePrint Output.....	113
Figure 64 Count Example.....	114
Figure 65 Max Example.....	115
Figure 66 Multiply Example.....	117
Figure 67 Name Example.....	117
Figure 68 Order Example.....	118
Figure 69 Avg Example.....	119
Figure 70 MeSH Headings.....	121
Figure 71 Query Functions Example.....	122
Figure 72 Sim1 Example.....	123
Figure 73 Jaccard Example.....	124
Figure 74 Dice Example.....	124
Figure 75 Cosine Example.....	125
Figure 76 Transpose Example.....	125
Figure 77 Centroid Example.....	126
Figure 78 TermCorrelate Example.....	129

Figure 79 DocCorrelate Example.....	130
Figure 80 IDF Example.....	130
Figure 81 Sum Example.....	131
Figure 82 Global Array Operator Overloads.....	136
Figure 83 Glade Canvas.....	138
Figure 84 Toggle Button Screen 1.....	139
Figure 85 Toggle Button Screen 2.....	140
Figure 86 Toggle Button Screen 3.....	140
Figure 87 Toggle Button Screen 4.....	141

1 Distribution Introduction

The Mumps language portion of the distro:

mumps/Mumps-Language-Processors/

consists of two subdirectories:

Legacy-Mumps-Interpreter/

Mumps-Interpreter-Compiler-Library/

The first sub-directory contains the original version of this dialect of Mumps written around 1980 and is no longer in use. The second sub-directory contains the current version of the code.

The current version contains three components:

1. The Mumps Interpreter
2. The Mumps Compiler
3. The Multi-Dimensional and Hierarchical (MDH) Toolkit

1.1 Distro File Structure

The Figure 1 gives the directory tree structure of the distribution. The main directory is named *mumps* and contains two sub-directories:

1. *Mumps-Language-Processors*
Contains the Mumps interpreter source code and installers.
2. *Mumps-Projects*
Contains programming examples including:
 - A genetic sequence search tool;
 - A compiler to generate GTK3 based GUIs with Mumps and Glade;
 - A collectio of information storage & retrieval modules and databases
 - An optimum binary tree example.

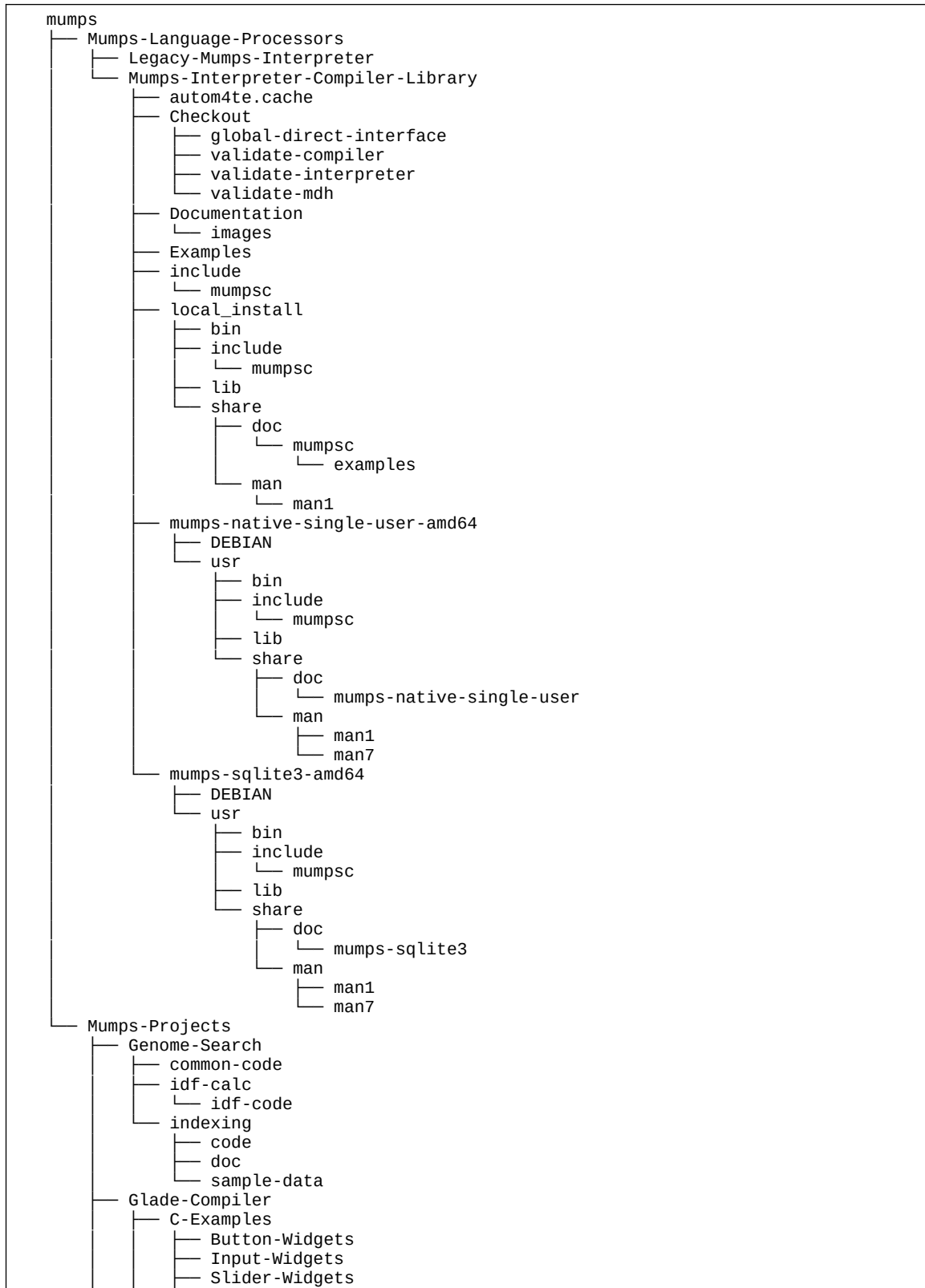
The directory *Mumps-Language-Processors* contains two parts:

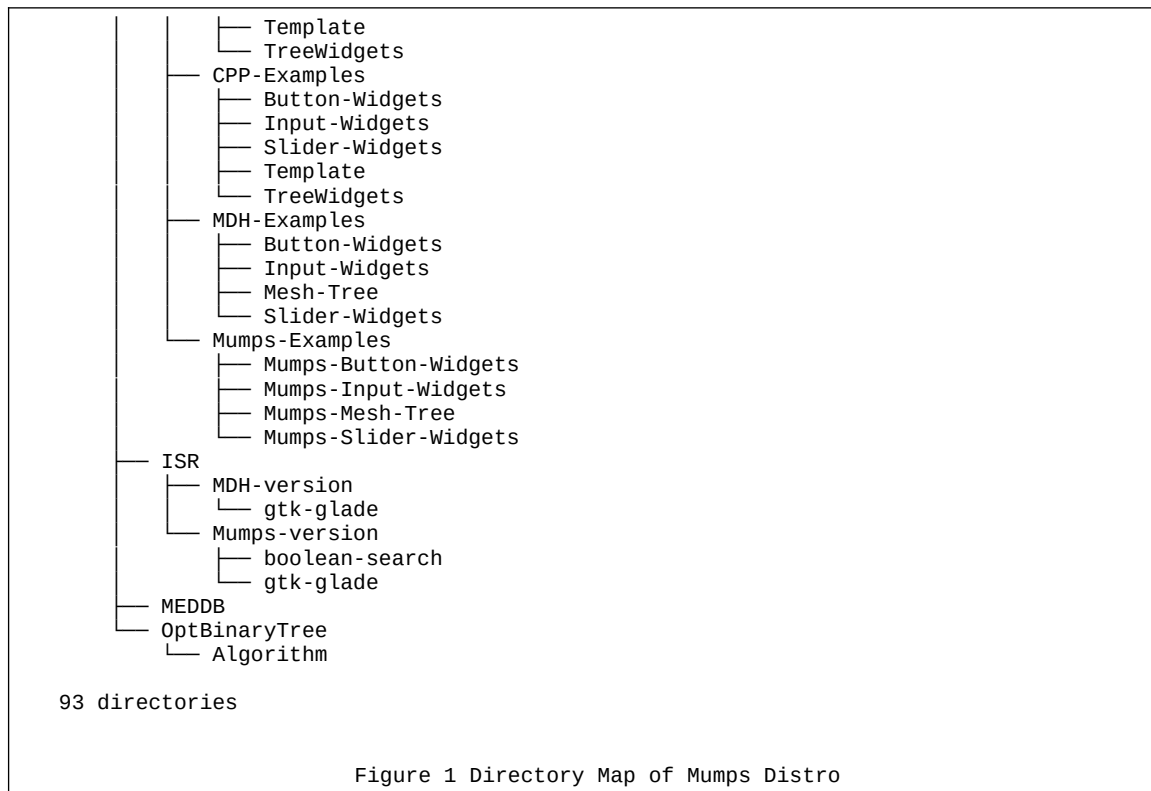
1. *Legacy-Mumps-Interpreter*
Contains the original, unmaintained, Mumps iterpreter.
2. *Mumps-Interpreter-Compiler-Library*
Contains the current Mumps interpreter, compiler, MDH Toolkit and installers.

The *Mumps-Interpreter-Compiler-Library* contains directories:

1. *Checkout*
Contains code to check the interpreter, compiler, and toolkit.
2. *Documentation*
Contains several docymentaion files including this one.
3. *Examples*
Some short Mumps examples.
4. *include*
The include files for the source code.

5. *local_install*
Contains a local version of the installed code used for building the installers.
6. *mumps-native-single-user-amd64*
Contains files needed to build a Debian installer for the native single user file system.
7. *mumps-sqlite3-amd64*
Contains files needed to build a Debian installer for the Sqlite3 version of the database.
8. The directory also contains source code, object files, installers. Makefiles, configure and Bash scripts.





1.2 Interpreter vs Compiler vs MDH Toolkit

The interpreter executes code directly from source text parsing each line as it is executed. Mumps source code to be dynamically created and executed.

On the other hand, the compiler translates your Mumps program to a C++ file which is subsequently compiled and linked with Mumps run-time libraries to produce binary executables. The compiler supports most of the features supported by the interpreter but not all. Compiler code is usually faster but, since both use the same database engines, database intensive programs run at about the same speed whether compiled or interpreted.

When the compiler translates Mumps to C++, the resulting C++ programs can be edited and other additional code introduced which might not be available in original Mumps.

The MDH toolkit is a collection of C++ macros, objects, methods and run-time libraries that permit you to write and run C++ programs using many of the database and string processing features of Mumps.

1.3 System Software Requirements

Building mumps requires that your system have certain software packages installed. These packages are standard libraries used by Mumps and are available through the Synaptic Package Manager, *apt-get* and other sources. The build script and the installers provided automatically install these if they are not present on your system.

The required software includes the following:

1. Linux Debian based version such as Debian, Ubuntu or Mint. The Windows WSL (Windows Subsystem for Linux) implementation with Ubuntu may also be used.
2. The *g++/gcc* compilers and related libraries.

3. The **pcre** (Perl Compatible Regular Expression) development libraries. The **pcre** libs should be in **/usr/lib** and the include files in **/usr/include**. Be certain to install the **pcre development** libraries.
4. The *bash shell* interpreter located in **/bin**.
5. The GNU **readline** and **readline-dev** packages.
6. **autoconf**
7. The following libraries are needed for the extended precision mathematics. If they are not installed by default, you will need to do so. Be sure to install the **development** versions of the libraries:
 - a) The GNU Multiple precision floating point computation library

<http://www.mpfr.org/libmpfr-dev>
 - b) The GNU Multiprecision arithmetic library development tools

<https://gmplib.org/libgmp-dev>

1.4 Mumps Language Resources

1. Mumps by Example:
<http://71.174.62.16/MDC/example/index.htm>
2. The Mumps Programming Language
https://www.amazon.com/Mumps-Programming-Language-Kevin-OKane/dp/1438243383/ref=tmm_pap_title_0?ie=UTF8&sr=1-1

2 Installation

The Mumps interpreter, compiler and MDH Toolkit are all installed at the same time either by means of a pre-built installer or compilation from source code. The options discussed below apply to all three versions of the code.

In the pre-built installers, options are, for the most part, already set and may not be changed. If you build the system from source code, you may change the options. The file ***build-mumps.script*** contains invocations of ***configure*** which propagates option selections through the code prior to compilation. For the most part, you will probably not want to alter these. But if you do, see the discussions below.

2.1 Using an Installer

Do **not** run *configure/make*. You must use the scripts provided and these will invoke *configure* and *make* using the correct build parameters.

The software uses two database systems to store the Mumps global arrays. The main supported versions of the interpreter, compiler, and MDH library use the Sqlite3 database. This is the default and the one contained in the installers provided on the web sites. The old native installer, however, is still present.

If you have an old or incompatible version already installed, you may need to remove it prior to installing the new version with one of the commands:

```
sudo dpkg --remove mumps-sqlite3 or
sudo dpkg --remove mumps-native-single-user
```

You may either build the interpreter/compiler/library from source code or use an installer. When you build from source code, you may select options such as which database system to use. The Sqlite3 version installer can be downloaded from the web page and also the distro. The native database installer is provided as part of the distro.

Due to variations in libraries from one generation of Linux to the next, installers may be labeled with the name and version of the Linux distro they were created with as needed.

Basically, the installers install the same code but one installs the Sqlite3 database while the other installs the native b-tree database.

To install, use a command similar to the following. Use only one of these. If you previously installed with a different database option, you will need to remove it first (see above).

```
sudo dpkg --install mumps-sqlite3-amd64.deb
sudo dpkg --install mumps-native-single-user-amd64.deb
```

If this is the first time you have installed the system, you may need to run an additional command to update system software. The distro uses a number of standard Unix libraries that are often not part of the a base system install. The following command should install these in the even the install above gave missing package error messages. If no messages, the following should be unnecessary.

```
sudo apt install -f
```

Alternatively, if the graphical installer ***gdebi*** is available, in an explorer window, double click on the installed ***.deb*** file. The installer ***gdebi*** should automatically install missing packages.

If you use the Sqlite3 version, be sure to take note of the database creation script noted below.

2.2 Compiling from Source Code

If you want to compile and build the system yourself, you need to first install any system libraries you may be missing. The Mumps code uses a number of libraries that are not usually installed by default on many systems. The script provided consists of **apt** statements that will install these"

```
sudo get-software.script
```

Next, configure, compile, and build (**make**) by running the following script :

```
build-mumps.script
```

This will compile and create an installer for the Sqlite3 version of the interpreter, compile, and MDH libraries. The installer will be named **mumps-sqlite3-amd64.deb**. During execution of this script, you will be asked for the **root** password.

Alternatively, if you want to build the legacy native system, type:

```
build-mumps.script native
```

The installer built will be named **mumps-native-single-user-amd64.deb**. The database used by this version will be stored in a file named **mumps.sqlite**. During execution of this script, you will be asked for the **root** password.

The native database version that uses the native b-tree database which is stored in files named **key.dat** and **data.dat**).

The native data base is faster than the Sqlite3 database. For comparison, in a test involving extended database activity, the native database version took 4-9 seconds while the disk based Sqlite3 database version with default settings took 11,060 seconds. This was due to the many additional writes done by Sqlite3 to enhance integrity.

However, very significant performance improvement of the Sqlite3 database may achieved by using **pragma** statements (see below) which reduce integrity but increase speed. When using the **pragma** tuning below, the Sqlite3 database performs at speeds comparable to the native database.

Without performance tuning, the Sqlite3 version is ACID compliant while the native version is not.

After you have installed the system, you need to create a Sqlite3 database if you are using the Sqlite3 version. The database will be automatically created at first use if you are using the legacy native version.

You create a Sqlite3 database with the command:

```
mumps-sqlite3-amd64.deb
```

This will create the file **mumps.sqlite** in the current directory.

The **mumps.sqlite** database or a symbolic link to it MUST be present in the directory from which you run mumps.

The **mumps.sqlite** database may be moved to other directories and you may create symbolic links to it.

Multiple instances of mumps programs may access the same Sqlite3 database (or symbolic link to same) concurrently.

However, only one instance may access the native database at once.

2.3 Compilation Options

The ***build-mumps.script*** script contains an invocation of the ***configure*** script with the basic options pre-set. Other additional options are described below.

Do not run configure directly. Use the provided script.

2.4 Math Options

Arithmetic in this Mumps distribution can be performed either by hardware or by a library of extended precision software.

In extended precision mode, the precision of both floating point and integer numbers can be significantly larger than is the case with standard hardware arithmetic with minimal performance penalty.

In this version of Mumps, as is the case with many others, numeric values are stored in variables as character strings. When a variable participates in an arithmetic operation, the value is converted to a numeric format, the operation performed (for example, addition), and the result converted back to character string. Not only are numeric values stored in variables as strings, but also, intermediate results are in string format.

2.4.1 Hardware Based Math

In hardware math mode, integer and floating point numbers are processed by your machine's arithmetic processing hardware. Floating point numbers are treated as either *long double* or *double* values and integers are treated as either signed 64-bit *long long* or signed 32-bit *long* integer values.

To enable hardware math, you must specify the following as a *configure* option:

--with-hardware-math=yes

Integer arithmetic may be performed in *int* (32 bit) or *long long* (64 bits in the gcc compiler) mode. The default is *long long*. The *int* mode may be turned on with the *configure* option:

--with-int-32

If the above is not specified, *long long* is used.

Floating point arithmetic may be performed in either *long double* or *double* mode. The *long double* mode may be enabled with the *configure* option:

--with-long-double

If the above is not specified, floating point arithmetic will be performed in *double* mode.

All numeric values are stored internally as strings. They are converted to binary numeric integer or floating point format just prior to an arithmetic operation and then converted back to strings.

By default, the string format of a floating point number will have with 8 digits of precision. This can be altered by *configure* using the *--with-float_digits* option (default is 8). For example, if you want 16 digits of precision, add

--with-float-digits=16

to the *configure* parameters. The number of digits specified should be consistent with the hardware data type (*double* or *long double*).

On x86 architectures, *long double* is usually implemented as an 80 bit number with a sign bit, an 15 bit exponent and 63 bit fractional part with a range of approximately 3.65×10^{-4951} to 1.18×10^{4932} while *double* is implemented as a 64 bit number.

2.4.2 Extended Precision Math

Extended precision is available through use of the GNU multiple precision arithmetic library¹ and the GNU MPFR library². For integers, this means effectively unlimited precision. For floating point numbers, the exponent is 64 bits and the fraction is user specified (default of 72 bits in Mumps - this option may be set by *configure*).

Hardware arithmetic will be selected during compilation of the interpreter if specified as a configuration option:

--with-hardware-math=no

If extended precision is used, the number of bits in the fraction of a floating point number can be set with:

--with-float-bits=value

where *value* is the number of bits. The default value is 72. The number of decimal digits for a given number of bits (nbits) is approximately:

$$\log_{10}(2^{nbits})$$

Thus, 72 bits corresponds to approximately 21 decimal digits.

For extended precision floating point numbers, the number of digits of precision to print is controlled by:

--with-float-digits=value

where *value* is the number of digits. The default is 8.

The number of digits specified should be consistent with the number of bits in the fraction. If the number of digits specified is too large, random low-order digits will appear in numbers.

If extended precision mode is in effect, integer numbers have no upper or lower bound.

2.5 Other Configure Options

The configure step as is typical contains many options. Specifying these causes modification to the source code and changes the final product. These must be set correctly. Not all combinations work.

The distribution, as noted above, contains several *bash* script files with pre-configured *configure* commands. For the most part, you probably don't want to write your own *configure* options except in limited cases.

With the exception of global array database options which are discussed separately in Section 3, the set of options to *configure* are:

¹ <http://www.mpfr.org/>

² <http://gmplib.org/manual/index.html>

configure prefix=

The directory where the runtime modules will be stored. If this is not specified, the default location is in a directory named **local_install** in the mumps distro directory.

--with-ibuf=

Maximum size of an interpreted program [default: 32000].

--with-strmax=

Maximum internal string size [default: 4096].

--with-locale=locale

Locale information [default: en_US.UTF-8].

--with-terminate-on-error

Halt interpreter on error [default: off]

--with-float-bits=val

Number of bits in an extended precision floating point fractional part (72).

--with-float-digits=val

Number of decimal digits to print in an extended precision floating point number (20).

--with-hardware-math={yes|no}

Use hardware arithmetic facilities.

--with-no-inline

Do not use inline functions.

--with-profile

Enable profiler (run *gprof mumps gmon.out > stats*).

3 Global Array Database Overview

3.1 History

Global arrays are unique to Mumps. They are a disk resident tree structured database whose elements and structure appear in programs as indexed arrays. The indices may be numeric or text and the arrays are not declared. In an array, each successive index describes part of the path from root to a leaf node. Data may be stored at any node.

Mumps was developed in the late 1960s for use on small computers. It was originally an interpreted language similar to BASIC which was also developed in the 1960s.

Unlike BASIC, Mumps was designed to be a medical database language. Its only basic data type was string although strings containing numbers could be used with arithmetic operators.

The main challenge Mumps was designed to satisfy was the storage of hierarchical medical records. Another hierarchical system developed at about the same time is IBM's Information Management System (IMS) which is still in use.

3.2 The Tree Structured Medical Record

The medical record structure, as envisioned by Mumps was a tree where contents were organized hierarchically as shown in Figure 2.

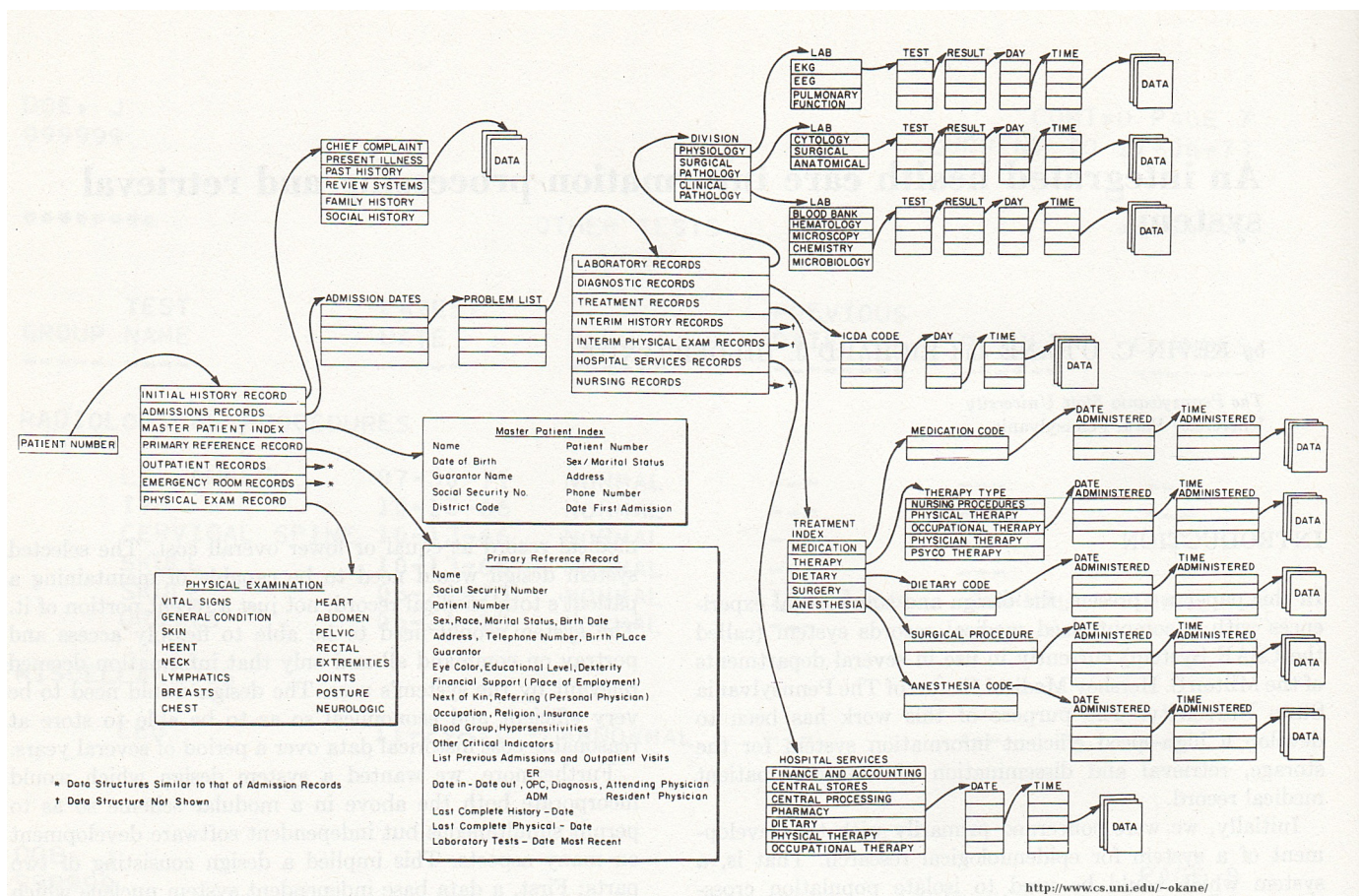


Figure 2 Tree Structured Medical Record

The alternative is usually a relational structure such as that exemplified by the entity-relationship diagram given in Figure 3.

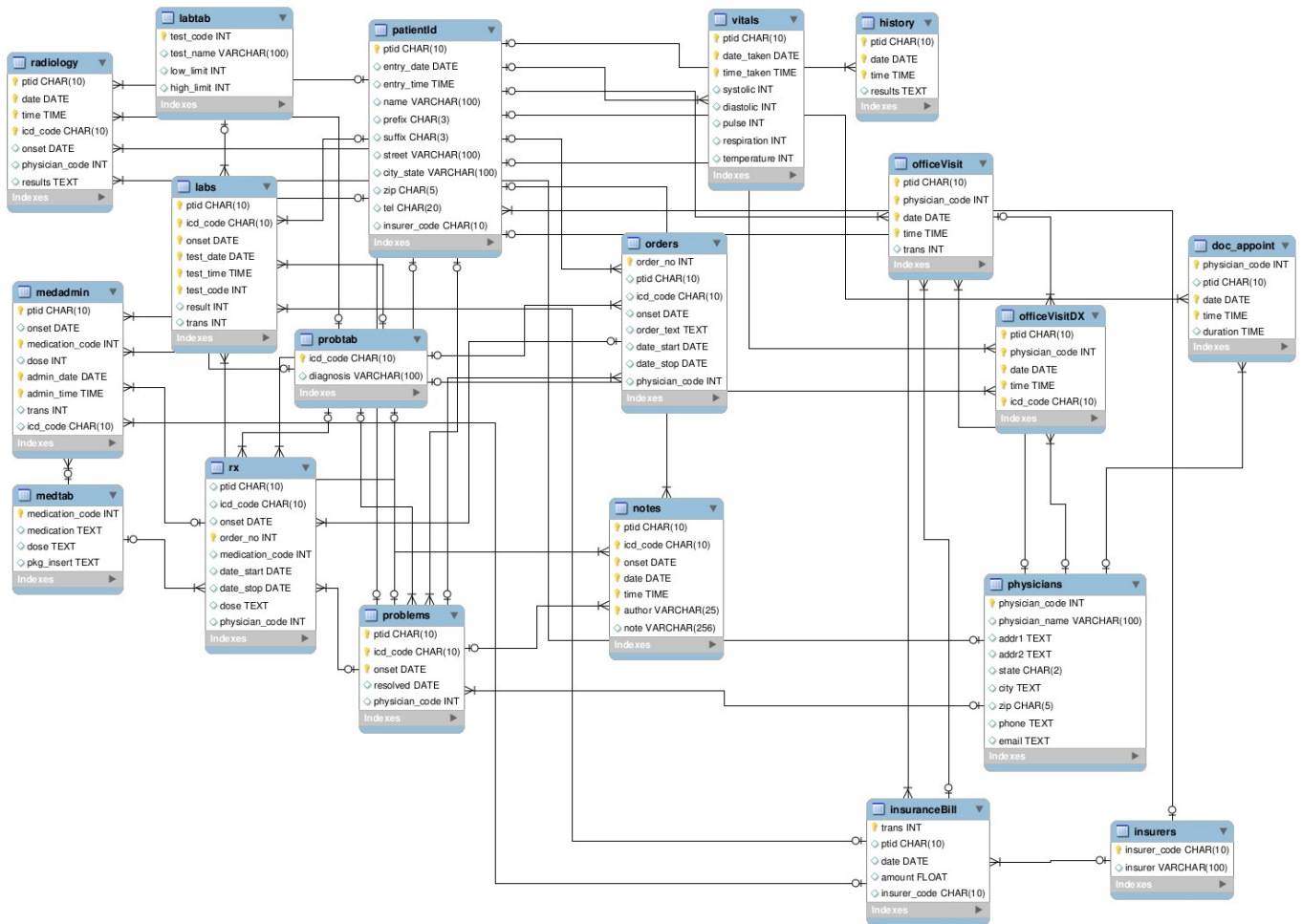


Figure 3 Relational Database Medical Record

The essential philosophical difference is that tree structured data tends to be more patient and problem centric while a relational approach is more data centric. That is, a tree structure organizes the data mainly according to patient medical complaint (problem). Thus, the medications, lab tests, notes, procedures, etc., are connected to a clinical issue and greatly improve the ability to audit what was done and why for each condition.

The relational structure, however, loses the semantic connection between data and the purpose of the data. This, regardless of problem being treated, all the lab tests, medications, notes, etc. for all patients are lumped together so that it becomes difficult to connect the cause (clinical problem) with effect (clinical procedure).

From a patient management point of view, the hierarchical view is better. From a financial management point of view gross aggregation of the data is usually preferred.

Trees were implemented in Mumps in structures known as **global arrays**. Global arrays are sparse disk resident trees represented in the language as string indexed arrays. Array notation follows the convention of FORTRAN and consists of an array name followed by a parenthesized list of indices. Indices trace a path through the global array tree for the named global array from the root to the final leaf. The height of the tree is variable depending on the path.

3.3 Global Array Representation

Global arrays are distinguished from ordinary volatile memory arrays by being preceded by a circumflex (^) character. While memory resident arrays disappear when a program ends, global arrays persist and are accessible to other programs in the system. A typical global array reference might appear as:

```
^patient(ptid, "hx", "PRR")
```

Data may be stored at any global array node either intermediate or terminal.

In this implementation of Mumps, global arrays may be stored either in an Sqlite3 relational database or in a native, single user B-tree based file system.

In the distribution, there are two database options for storage of the global arrays. The legacy option is called the native database. The other option is to use Sqlite3. Of the two, the Sqlite3 is the more reliable but transactions, since it is ACID (atomicity, consistency, isolation, durability) compliant, can be much slower.

4 Native Database Global Array Option

4.1 Overview

The native database option is fast with a minimum of overhead and it can efficiently manage very large databases however it lacks a number of features normally found on modern database systems:

1. It is sensitive to system crashes and programming errors.
2. It does a minimum of checkpointing.
3. It maintains part of the global array tree in volatile memory.

If the host system crashes or the program using the global arrays terminates unexpectedly, the contents of the entire global array database are likely to be lost.

However, in applications where speed is important and, in the event of a crash, the program can be re-run, the native database works well.

4.2 Native Database Configuration

The native database is a *single user* B-tree based global array facility where the global arrays are stored in one directory, usually the one in which the Mumps program is itself running. In this mode, only one *read-write* Mumps program may access the global arrays in a given directory at a time although other Mumps programs may run concurrently in other directories operating on other global array data sets. This is the fastest but most restrictive option. Globals arrays are stored in two files: ***key.dat*** and ***data.dat***.

4.2.1 --with-maxglobal=val

Maximum length of a global array reference in the native database. A global reference length includes the array name, all indices, plus parentheses and commas.

4.2.2 --with-cache=VAL

Native global database in-memory cache size. The number is the number of blocks (see: *--with-block*) to maintain in memory. Not used by Sqlite3.

The **only** legal values for this parameter are:

9
17
33
65
129
257
513
1025
2049
4097
8193
16385
32769
65537
131073
262145
524289
1048577

4.2.3 --with-block=blksize

Native global Btree block size.

The native Btree database consists of two files: the tree file (*key.dat*) containing the actual Btree and the data file (*data.dat*) containing stored data. The maximum size of the Btree file is dependent on the block size. The block sizes listed below each have a PAGE_SHIFT value and this ultimately determines the maximum file size as shown. The basic internal disk address is effectively 31 bits (signed 32 bit quantity) but, depending upon the block size, some number of bits at the low-order end of a block address are always zero. For example, if the block size is 1024, the final 10 bits of an address are always zeros. As only the significant 31 bits are stored, the true address is not 31 bits but 41 bits thus a file size of 2 terabytes is possible.

The only legal values for this parameter are:

1024
2048
4096
8192
16384
32768
65536
131072
262144

The block size determines the internal PAGE_SHIFT factor:

1024	→	PAGE_SHIFT 10
2048	→	PAGE_SHIFT 11
4096	→	PAGE_SHIFT 12
8192	→	PAGE_SHIFT 13
16384	→	PAGE_SHIFT 14
32768	→	PAGE_SHIFT 15
65536	→	PAGE_SHIFT 16
131072	→	PAGE_SHIFT 17
262144	→	PAGE_SHIFT 18
524288	→	PAGE_SHIFT 19
1048576	→	PAGE_SHIFT 20
2097152	→	PAGE_SHIFT 21

PAGE_SHIFT 10 corresponds to MBLOCK 1024 and a max Btree file size of 2 TB
PAGE_SHIFT 11 corresponds to MBLOCK 2048 and a max Btree file size of 4 TB
PAGE_SHIFT 12 corresponds to MBLOCK 4096 and a max Btree file size of 8 TB
PAGE_SHIFT 13 corresponds to MBLOCK 8192 and a max Btree file size of 16 TB
PAGE_SHIFT 14 corresponds to MBLOCK 16384 and a max Btree file size of 32 TB
PAGE_SHIFT 15 corresponds to MBLOCK 32768 and a max Btree file size of 64 TB
PAGE_SHIFT 16 corresponds to MBLOCK 65536 and a max Btree file size of 128 TB

The separate data file may grow to a max of 2**64 bytes for all settings.

5 Sqlite3 Database Global Array Option

5.1 Overview

If data integrity, remote, and multi-user access are important, the Sqlite is better. In this option, the interpreter and compiler will use Sqlite3 to store the global arrays.

While this option is slower than the native database option, due to relational data base system overhead, using a relational database has *significant advantages* with regard to reliability and flexibility. These include:

1. All database transactions are ACID (*Atomicity, Consistency, Isolation, Durability*) compliant.
2. SQL commands such as Begin Transaction, Commit and Rollback are available.
3. The Mumps global arrays can be queried with SQL commands from non-Mumps environments.
4. SQL views of the Mumps database may be constructed.
5. The Mumps global array database can be remote and distributed.
6. Mumps programs can execute SQL commands on any accessible database table.
7. Multiple concurrent Mumps programs may run at the same time.

The Sqlite3 option requires the database file to be pre-formatted prior to use. All global array data stored in global arrays is in character format. SQL comparisons between global array stored data and data from another SQL table using another data type may require casting operators.

5.2 Global Array to Sqlite3 Relational Database Mapping

There are advantages and disadvantages to storing global arrays in a relational database. The primary disadvantage is that the dynamic hierarchical nature of the Mumps database is not well suited to the tabular structure of a relational database where overall access is usually slower and more rigidly structured.

On the other hand, the Sqlite3 relational database provides flexible multi-user, robust, is fully ACID (*Atomicity, Consistency, Isolation, Durability*) compliant and provides a complete suite of transaction processing functions not otherwise available in the Mumps language definition.

A further advantage is that global array data may be interrogated and manipulated by ordinary, standard SQL commands.

When using the Sqlite3 option, the Mumps interpreter maps global array references to a multi-column relational database table with the name **mumps** (this can be changed). The columns of the table are named **a1, a2, ...**³ so forth. By default, there are 12 columns numbered **a1** through **a12**.

The first column (**a1**) contains the name of the global array. Columns **a2** through **a11** contain global array indices. The last column (**a12**) contains the value stored at the global array reference, if any.

Thus, by default, in this case, the maximum number of indices allowed in a global array reference is 10. The name of the global array and each index may have up to 256 characters and the value stored may have up to 512 characters⁴.

For example, the Mumps code:

```
set ^birds(1,2,3,4,5)="ducks"
```

would map to the Sqlite3 database table named *mumps* as follows:

³ The actual number of columns is set by a compile option discussed below in section 5.3.2 .

⁴ The size limits for indices and data stored are also set by a compiler option.

birds											
a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12
birds	1	2	3	4	5						ducks

Where the values for **a6** through **a9** are *null*.

In most modern database systems, the *null* columns do not seriously impact speed. Alternative methods to implement trees in relational databases introduce very high levels of overhead. Further, the **mumps** table is indexed by the global array name and the first index (that is, by the first two columns of the table) for faster access.

Testing indicates that performance is, in fact, quite good.

If your program instantiates array elements as shown in the following Mumps code:

```

set ^birds(1)="all"
set ^birds(1,2)="flying"
set ^birds(1,2,3)="water"
set ^birds(1,2,3,4)="large"
set ^birds(1,2,3,4,5)="ducks"
set ^birds(1,3)="flightless"
set ^birds(1,3,3)="water"
set ^birds(1,3,3,4)="large"
set ^birds(1,3,3,4,5)="penguins"

```

The relational table will look like⁵:

a1	a2	a3	a4	a5	a6	...	a10	a11	a12
birds	1					...			all
birds	1	2				...			flying
birds	1	2	3			...			water
birds	1	2	3	3		...			large
birds	1	3	3	4	5	...			ducks
birds	1	3				...			flightless
birds	1	3	3			...			water
birds	1	3	3			...			large
birds	1	3	3	5		...			penguins

Mumps access requests produce the expected results:

```

write ^birds(1)           => all
write ^birds(1,2)         => flying
write ^birds(1,2,3)       => water
write ^birds(1,2,3,4)     => large
write ^birds(1,2,3,4,5)   => ducks

write $order(^birds(1,2)) => 3

```

⁵ Table row order may differ but this is not important.

```
write $order(^birds(1,2,"")) => 3
```

The row-wise index duplication seen in the above is also present in B-tree implementations.

5.3 Sqlite3 Database Configuration

5.3.1 Sqlite3 Database File

In order for Mumps to store and retrieve global arrays in Sqlite3, there must be a pre-existing database file named (by default) ***mumps.sqlite*** which is accessible to the instance of Mumps being executed. Normally, that means the file is in the directory where you started Mumps from. Links, however, may be used if the database file is located in another directory.

You may create or re-initialize ***mumps.sqlite*** with the script file:

mumps-sql-db-create

which is installed by the installers. The file ***mumps.sqlite*** contains the Sqlite3 database and must be present prior to starting Mumps if you are using the Sqlite3 database option. If not, Mumps will halt.

In previous releases, this file was named differently. The current installer should remove the older version of this (named ***mumps-sql-db-create.script***) but if not, type:

```
sudo rm /usr/bin/mumps-sql-db-create.script
```

5.3.2 Sqlite2 Compile Options

Options contained in install script ***build-mumps.script*** can be used to set the maximum number of indices, the maximum number of characters per Mumps global array index, and the maximum number of characters that can be stored at a node. These are:

5.3.2.1 --with-dbname=name

Default name of the Sqlite3 mumps database table name [default: ***mumps***].

5.3.2.2 --with-index_size=number

Maximum number of characters in each Sqlite3 global array index.

5.3.2.3 --with-data_size=nbr

Maximum number of data characters stored for an Sqlite3 global array reference (final column).

5.3.2.4 --with-index-max=nbr

Maximum number of database columns. Default is 14. Maximum is 32. A value of 14 permits a global array name, 12 levels of indexing, and a data value.

5.3.2.5 --with-dbfile=name

Name of Sqlite3's database file stored in the users directory [default: ***mumps.sqlite***]

5.4 Sqlite3 Performance Tuning

By default, as is the case with all ACID compliant database systems, Sqlite3 journals and verifies all file updates. This leads to considerable I/O overhead and reduction of the speed at which transactions that alter the database take place. This insures that a system crash will not result in data loss.

In applications where data loss protection is not required, that is, those applications where the database software can be re-run in the event of a crash without loss, the strict update policy of Sqlite3 can be overridden with significant speed improvement.

To achieve this, execute the following Mumps commands prior to using the database:

```
s i=$zsqlite("pragma","journal_mode=off")
s i=$zsqlite("pragma","synchronous=off")
```

These will temporarily reduce overhead during database updates resulting in significant speed improvement in operations altering the database.

5.5 Sqlite3 Command Line Interpreter

"**sqlite3**" is a terminal-based front-end to the SQLite library that can evaluate queries interactively and display the results in multiple formats. **sqlite3** can also be used within shell scripts and other applications to provide batch processing features."⁶

If your global arrays are stored in **sqlite3**, they can be queried and manipulated by SQL commands using the **sqlite3** CLI. For example, using the example above:

```
sqlite3 mumps.sqlite
sqlite> select a10 from mumps where a1='birds' and a2='1' and a3='2';
flying
sqlite>
```

Similarly, SQL **views** may be established on the **mumps** table to facilitate access in other ways by other SQL users.

5.6 Sqlite3 Database Language Extensions

If Sqlite3 relational database storage for globals is enabled, the following functions and builtin variables are available in the Mumps interpreter, compiler, and MDH Toolkit. If the native database is in use, these, with the exception of **\$zNative**, are ignored. Except as noted, a return value of zero (0) indicates success while a non-zero result indicates failure.

5.6.1 \$zSqlite

\$zsqlite with no arguments returns **true** (1) if globals are being stored in Sqlite3, **false** (0) otherwise.

5.6.2 \$zSqlite("begin transaction")

Sends a **BEGIN TRANSACTION**; command to Sqlite3.

5.6.3 \$zSqlite("commit transaction")

Sends a **COMMIT TRANSACTION**; command to Sqlite3.

5.6.4 \$zSqlite("savepoint"[*savepoint_name*])

If the second argument is omitted, send a **SAVEPOINT default**; command to Sqlite3. If the second argument is present, send a **SAVEPOINT savepoint**; command to Sqlite3 where '**savepoint**' is the value passed as the second argument. See Sqlite3 documentation for details.

⁶ From the Sqlite3 Linux man page.

5.6.5 \$zSqlite("rollback",[savepoint])

If the second argument is omitted, send a **ROLLBACK TRANSACTION**; to default; command to Sqlite3. If the second argument is present, send a **ROLLBACK TRANSACTION savepoint**; command to Sqlite3 where '**savepoint**' is the value passed as the second argument.

5.6.6 \$zSqlite("pragma",option)

A **PRAGMA** command will be sent to Sqlite with *option* as its argument. If the **PRAGMA** results in a returned value, it will be the returned result of the function. Otherwise, the function will return 0 (success) or 1 (failure).

Some example **PRAGMA** commands:

```
s i=$zsqlite("pragma","mmap_size=20000000")
s i=$zsqlite("pragma","cache_size=-1000000")
s i=$zsqlite("pragma","journal_mode=off")
s i=$zsqlite("pragma","synchronous=off")
```

Using the **synchronous_mode=off** and **journal_mode=off** can improve speed considerably at the expense of crash integrity. When the **cache_size** is preceded by a minus sign, the value is interpreted as kilobytes. A positive value is interpreted as number of memory pages.

Memory-mapped I/O (**mmap**) allows the database file to be directly mapped into the process's address space, eliminating the need for explicit read and write system calls and reducing data copying between kernel and user space. This can significantly improve performance for read operations, especially when the database is large and fits within available memory.

5.6.7 \$zsqlOpen

Returns **true** if a connection to the SQL server is open, **false** otherwise.

5.6.8 \$zNative

\$znative returns **true** (1) if globals are being stored in the native global array. **False** (0) otherwise.

6 Interpreter & Compiler

6.1 Interpreter

6.1.1 Running the Mumps CLI Interpreter

To run the command line interpreter from a terminal window, type:

mumps

Any Mumps commands you enter will be executed immediately. To exit the interpreter, type **H**[alt] or *control-d*.

In interactive mode, you will be presented with a prompt (>). Any Mumps command may be typed for immediate execution (including a **goto** or **do** command with a file name reference pointing to a file to be loaded and executed).

The keyboard *up arrow* and *down arrow* keys may be used to cycle through and display commands previously entered during this session. You may line edit previously entered commands. To execute, hit *enter*.

Command line input to Mumps follows GNU ***readline*** conventions.

To exit the Mumps CLI, use the **Halt** (**h** or **H** or **halt**) command or **^d**.

6.1.2 Interpreting a Mumps Program

Mumps source code programs are ASCII files that can be created by any ASCII text editor.

However, avoid using word processing editors as they may embed hidden formatting characters into the text. These will cause errors.

One way to interpret a Mumps program is to pass the name of the program as an argument to the Mumps interpreter:

mumps progname.mps

This will invoke the Mumps interpreter and pass the name of the program to it for execution. The program file passed need only be readable – execute permission is not required.

Alternatively, you may run a Mumps program directly source file by identifying the Mumps interpreter on the first source code line and making the file executable. In this case, the Mumps program will have the following as its first line:

#!/usr/bin/mumps

The pound sign (#) makes the line a comment to Mumps but **#!** is recognized by the Linux shell as introducing the name of the program (***mumps*** in this case) to receive the source file text. The default location of the Mumps interpreter is: ***/usr/bin/mumps***.

The Mumps source file must be made executable:

chmod u+x progname.mps

where **progrname.mps** is the name of your mumps source file.

Example:

```
#!/usr/bin/mumps
  for i=1:1:10 do
    . write "Hello World ",i,!
  halt
```

You may execute the program by typing **progrname.mps** to your terminal prompt. The program above will write **Hello World**, followed by a number ten times.

Lines beginning (column 1) with pound sign (#) are ignored by the Mumps Interpreter and compiler.

6.2 Compiler

The Mumps Compiler is invoked with the script file **mumpsc**. This executable script will translate a Mumps program to an intermediate C++ file and compile the result using the Mumps runtime libraries. The result will be an executable binary. The intermediate C++ file is not deleted and can be inspected or edited. If available, the intermediate file will be processed by **astyle** by default to improve readability.

Most, but not all, Mumps language features are available in the compiler.

The **mumpsc** script may be used to compile a C++ intermediate file from a previous translation from Mumps to C++. You may edit the intermediate file.

Compiled programs source code, in nearly all cases, must begin with the zmain command as the first executable command. Omitting this line will result in many errors.

You may edit the C++ file created by the Mumps Compiler and include calls to other routines. You may compile the result (the C++ file) to a binary executable also using **mumpsc**.

You should not pass a Mumps compiler generated C++ file directly to the C++ compiler because the C++ file needs access to libraries which the **mumpsc** script command automatically provides.

If you use the Mumps Compiler, you should avoid using the **xecute** command and the indirection operator (@) as these invoke the Mumps interpreter and greatly increase overhead.

6.2.1 How to Compile and Run a Mumps Program

In order to be compiled, programs written in Mumps must have the extension **.mps**.

When you compile a Mumps program, a C++ translation of your program is created and written to the disk with the same name but with the **.cpp** extension. The C++ translation is then compiled and linked with run-time libraries to build an executable binary.

You may compile a Mumps program by using the executable script **mumpsc**. To compile a Mumps program using the script, type:

```
mumpsc myprog.mps
```

The actual program that translates Mumps to C++ is named **mumps2c**. You may run this program standalone:

```
mumps2c myprog.mps
```

The result will be a file named **myprog.cpp**. You may edit or modify this file and then compile it to binary executable with the **mumpsc** script.

Both ordinary C++ programs that use the MDH class libraries, and C++ programs created by the Mumps compiler, make use of the same Mumps compile and runtime libraries. The **mumpsc** script passes libraries

these to the **gpp** compiler. Compiling Mumps related code with the **gpp** compiler without these libraries will result in errors.

6.2.2 Compiler Error Messages

Generally speaking, in most cases you will receive syntax error messages from the Mumps compiler. These will identify the error and the line number in the original Mumps program containing the error.

However, in some cases, an error may not be detected by the Mumps compiler but, instead, by the C++ compiler.

Consequently, if you get ++ compiler error messages, the line number on the error message will refer to the line number in the C++ translation of your Mumps program. To connect this to a line number in your Mumps program, look into the generated **.cpp** file at the line number given by the C++ error message and then back track to the nearest prior commented Mumps source line - this shows the original in your Mumps programs that caused the problem.

For example, if you get a message from the C++ compiler saying that you have an error at line 1234 in the C++ module, open the C++ file and move to line 1234. At that location you may see something like⁷:

```
/*=====*
svPtr->LineNumber=4; //      write "the sum is: ",total,!
/*=====*/
    if (svPtr->out_file[svPtr->io]==NULL) ErrorMessage("Write to input file",svPtr->LineNumber);
    svPtr->hor[svPtr->io]+=fprintf(svPtr->out_file[svPtr->io],"%s","the sum is: ");
    if (sym_(SYMGET,(unsigned char *) "total",(unsigned char *) tmp0,svPtr)==NULL)
        VariableNotFound(svPtr->LineNumber);
    svPtr->hor[svPtr->io]+=fprintf(svPtr->out_file[svPtr->io],"%s",tmp0);
    fprintf(svPtr->out_file[svPtr->io],"\\n"); svPtr->hor[svPtr->io]=0; svPtr->ver[svPtr->io]++;
```

Figure 4 Example C++ Code

Note that each original line of Mumps code and its line number in the original Mumps file appear in a comment prior to the C++ translation of the line. Note that the translation of a line of Mumps code normally results in multiple lines of C++ code.

Generally speaking, you may receive C++ error messages if you reference non-existent labels or subroutines, or incorrectly specify indented do blocks (see below).

6.2.3 Global Array Storage in Compiled Programs

Global arrays will be stored in Sqlite3 or the native Btree database depending on which installer you used to build the interpreter. Global arrays created by compiled programs are interchangeable with global arrays created by the interpreter.

6.2.4 Compiler Performance and Interoperability

Compiled modules, exclusive of database access, execute faster than the same code executing on the interpreter. However, programs with large amounts of database or I/O activity will run at about the same speed as interpreted versions of the same program.

One advantage of full compilation is interoperability with other languages and with the host operating system. Programs written in C++ have full access to all system features and can be manually edited to improve performance.

⁷ The code in the example does not contain an error. It is for example purposes only.

7 Interpreter & Compiler Z Functions, and System Variables

\$zfunctions and systems variables are extensions added by the implementer and not covered by the standard. Thus, many if not all of the following extensions may not be supported, or supported differently, in other implementations. These are available with few exceptions, in both the compiler and interpreter version of Mumps. Several are available in the MDH Toolkit but usually with a different syntax.

Implementation note: you may add new **\$z** functions by modifying the function **zfcn()** located in the source file *interp-zfcns.cpp.in*.

7.1 System Variables

7.1.1.1 \$zProgram

Returns a string with the name of the currently executing program.

7.2 Database Functions

7.2.1 \$zGlobal(string)

Returns the name of the global array alphabetically following the value in *string*. Returns an empty string if there is none. Returns the name of the alphabetically first global array name if an empty *string* is passed.

7.2.2 \$zDBfile

Returns the name of the file containing the Sqlite3 data base or an empty string if Sqlite3 is not in use.

7.2.3 \$zDBname

Returns the name of the Sqlite3 table in which the global arrays are stored or an empty string if Sqlite3 is not in use.

7.2.4 \$zIndexsize

Returns the maximum number of indices permitted in a global array if Sqlite3 is in use. An empty string otherwise.

7.2.5 \$zIndexmax

Returns the maximum string length of a global array index element if Sqlite3 is in use or an empty string otherwise.

7.2.6 \$zDatasize

Returns the maximum string length of a data element that may be stored in a global array if Sqlite3 is in use or the empty string otherwise.

7.2.7 \$zDatabase

Returns a number indicating the database server:

- 1 Native database server with disk resident files
- 2 Sql database with disk resident files
- 3 Sql database with memory resident files (transient)

7.3 Bash Functions

7.3.1 \$zbasename(arg1[,arg2])

Returns a result equivalent of the Bash function *basename*

```
$zbasename("/home/jsmith/base.wiki") yields base.wiki
$zbasename("/home/jsmith/") yields jsmith
$zbasename("/") yields /

$zbasename("/home/jsmith/base.wiki", ".wiki") yields base
$zbasename("/home/jsmith/base.wikia", "ki") yields base.wi
$zbasename("/home/jsmith/base.wiki", "base.wiki") yields base.wiki
```

7.3.2 \$zfiletest(arg1,arg2)

Performs a Bash style check on a file name. The first argument is the name of a file and the second is a parameter that determines the type for file check. If the check condition is *true*, a one (1) is returned, zero (0) otherwise. The following are legal values for the second argument:

- a True if FILE exists.
- b True if FILE exists and is a block-special file.
- c True if FILE exists and is a character-special file.
- d True if FILE exists and is a directory.
- e True if FILE exists.
- f True if FILE exists and is a regular file.
- g True if FILE exists and its SGID bit is set.
- h True if FILE exists and is a symbolic link.
- k True if FILE exists and its sticky bit is set.
- p True if FILE exists and is a named pipe (FIFO).
- r True if FILE exists and is readable.
- s True if FILE exists and has a size greater than zero.
- t True if file descriptor FD is open and refers to a terminal.
- u True if FILE exists and its SUID (set user ID) bit is set.
- w True if FILE exists and is writable.
- x True if FILE exists and is executable.
- O True if FILE exists and is owned by the effective user ID.
- G True if FILE exists and is owned by the effective group ID.
- L True if FILE exists and is a symbolic link.
- N True if FILE exists and has been modified since it was last read.
- S True if FILE exists and is a socket.

7.4 Math Functions

The following C/C++ math functions are available in M2. Their arguments and return values are the same as the correspondingly named C++ functions.

7.4.1 \$zabs(arg) absolute value

Function returns the absolute value of its numeric argument.

7.4.2 \$zacos(arg) arc cosine

Computes the inverse cosine (arc cosine) of the input value. Arguments must be in the range -1 to 1.

7.4.3 \$zasin(arg) Arc sine

Computes the inverse sine (arc sine) of the argument **arg**. Arguments must be in the range -1 to 1.

7.4.4 **\$atan(arg)** Arc tangent

Computes the inverse tangent (arc tangent) of the input value.

7.4.5 **\$zcos(arg)** Cosine

Computes the cosine of the argument **arg**. Angles are specified in radians.

7.4.6 **\$zexp(arg)** Exponential

Calculates the exponential of **arg**, that is, *e* raised to the power *arg* (where *e* is the base of the natural system of logarithms, approximately 2.71828).

7.4.7 **\$zexp2(arg)** Exponential base 2

Calculates 2 raised to the power *arg*.

7.4.8 **\$zexp10(arg)** Exponential base 10

Calculates 10 raised to the power *arg*.

7.4.9 **\$zlog(arg)** Natural log

Returns the natural logarithm of **arg**, that is, its logarithm base *e* (where *e* is the base of the natural system of logarithms, 2.71828...).

7.4.10 **\$zlog2(arg)** Base 2 log

Returns the base 2 logarithm of *arg*.

7.4.11 **\$zlog10(arg)** Base 10 log

Returns the base 10 logarithm of **arg**.

7.4.12 **\$zpow(arg1,arg2)** Power function

Calculates **arg1** raised to the exponent **arg2**.

7.4.13 **\$zsqrt(arg)** Square root

Function returns the square root of its numeric argument.

7.4.14 **\$zsin(arg)** Sine function

Computes the sine of the argument **arg**. Angles are specified in radians.

7.4.15 **\$ztan(arg)** Tangent function

Computes the tangent of **arg**.

7.5 Date functions

7.5.1 **\$zdate(or \$zd)** formatted date string

Function returns the system date and time in standard system printable format. This includes: day of week, month, day of month, time (hour:minute:second), and year (4 digits).

7.5.2 **\$zd1** numeric internal date

Returns the number of seconds since January 1, 1970 - a standard used in Linux. This number may be used to accurately correlate events.

7.5.3 **\$zd2(InternalDate)** date conversion

Translates the Linux time from \$ZD1 into standard system printable format. The argument is a Linux format time value.

7.5.4 **\$zd3(Year,Month,Day)** Julian date

Returns the day of the year (Julian date) for the Gregorian date argument.

7.5.5 **\$zd4(Year,DayOfYear)** Julian to Gregorian

Returns the Gregorian date for the Julian date argument.

7.5.6 **\$zd5(Year, Month, Day)** comma listed date

Returns a string consisting of the year, a comma, the day of year, and the number of days since Sunday (Monday is 1).

7.5.7 **\$zd6** hour:minute

Returns a string consisting of the hour, a colon, and the minute.

7.5.8 **\$zd7** hyphenated date

Returns a string consisting of the year, hyphen, month, hyphen, and day of month. If an argument is given in the form of the number of seconds since Jan 1, 1970, the result returned will reflect the argument date.

7.5.9 **\$zd8** hyphenated date with time

Returns a string consisting of the year, hyphen, month, hyphen, and day of month, comma, and time in HH:MM format. If an argument is given in the form of the number of seconds since Jan 1, 1970, the result returned will reflect the argument date.

7.6 Special Purpose Functions

The following special purpose functions are available:

7.6.1 **\$zb(arg)** remove blanks

Function returns a string in which all leading blanks have been removed and all multiple blanks have been replaced by single blanks. See also **\$zNoBlanks()**. Figure 5 gives examples.

```
1 #!/usr/bin/mumps
2 set a="   abc   xyz   123   "
3 write $zb(a),"***",!
```

output:

```
abc xyz 123 ***
```

Figure 5 **\$Zb()** Examples

7.6.2 **\$zchdir(directory_path)** change directory

Function changes the current directory to the path specified. If the operation succeeds, a zero is returned. If it fails, -1 is returned.

7.6.3 **\$zCurrentFile** Current Mumps File

Returns the name of the currently executing Mumps program file (if any) or blank.

7.6.4 **\$zdump[(filename)]** dump global arrays

Function dumps the globals to a sequential ASCII file in the current directory. If an argument is given, it is taken as the name of the file to which the globals will be written. If the argument is omitted, a file name is constructed from the system date of the form **number.dmp** where **number** is the value of the C++ **time()** function at the time of the dump.

The dump file is a pure ASCII text file. Each entry in the global array is represented by two lines. The first line is the global array reference and the second line is the store value. In the global array reference, parentheses and commas are replaced by the "~" character. Thus, if you wish to use this facility, you may not include the "~" character in a global array index.

The function **\$zrestore()** reloads the global arrays from a dump file (see below).

\$zdump and **\$zrestore** do not work when SQL is used for the global array store.

7.6.5 **\$zrestore[(arg)]** restore globals

Function restores the globals from a dump file produced by **\$zdump**. If an argument is given, it is taken as the name of the dump file otherwise, the default name **dump** is used.

\$zdump and **\$zrestore** do not work when SQL is used for the global array store.

7.6.6 **\$zfile(arg)** file exists test

Function returns a zero or one indicating if the file given as the argument exists.

7.6.7 **\$zflush** flush Btree buffers

Function flushes all modified native global array handler buffers to disk. The function should only be used with the native globals. After flushing, all updates to the btree file system have been committed. In cases where the internal buffers are very large, this function may take several seconds to execute. The function returns the empty string. Flushing the buffers is a precaution against system failure which would otherwise result in corruption of the global arrays.

7.6.8 **\$zgetenv(arg)** get environment variable

Returns the contents of the environment variable specified as *arg* or the empty string if the variable is not found.

7.6.9 **\$zhtml(arg)** encode HTML string

Function encodes its argument in the form necessary to be a cgi-bin parameter. That is, alphabets remain unchanged, blanks become plus signs and all other characters become hexadecimal values, preceded by a percent sign.

7.6.10 **\$zhit** global array cache hit ratio

Function calculates and returns the native global array cache hit ratio. This number ranges between zero and one. A value of one indicates all requests were satisfied from the cache while a value of zero indicates no requests were satisfied from the cache. Calling this function resets the hit ratio to zero. A higher value for the hit ratio indicates better database performance.

7.6.11 **\$zlower(string)** convert to lower case

Function returns the input string with alphabets converted to lower case.

7.6.12 \$znormal(*arg1*[,*arg2*]) word normalization

Function converts the word passed as argument 1 to lower case and removes any embedded punctuation. If a second argument is given, the word is truncated to the length specified by this argument. If no second argument is given, words are truncated to 25 characters if their length exceeds 25 characters.

7.6.13 \$zNoBlanks(*arg*) remove all blanks

Returns *arg* with all blanks removed. See also: **\$zb**.

7.6.14 \$zpad(*arg1*,*arg2*) left justify with padding

Function left justifies the first argument in a string whose length is given by the second argument, padding to the right with blanks.

7.6.15 \$zseek(*arg*)

Function takes one argument (a positive integer) which is a byte offset in the currently active (use) file. The command moves the file pointer to that location in the file. **\$zseek()** may only be used on files opened with **old** attribute. Figure 6 gives examples.

```
1  #!/usr/bin/mumps
2  open 1:"tdb,new"
3  for j=1:1:1000 do
4    . use 1
5    . set i=$ztell
6    . set ^a(j)=i
7    . write "***** ",j,!
8
9  close 1
10 open 1:"tdb,old"
11 for j="":$order(^a(j)):"" do
12   . use 1
13   . set i=$zseek(^a(j))
14   . read a
15   . use 5
16   . write a,!
```

output:

```
***** 1
***** 10
***** 100
***** 1000
***** 101
***** 102
***** 103
***** 104
***** 105
***** 106
***** 107
***** 108
***** 109
***** 11
***** 110
***** 111
...

```

Figure 6 \$Zseek() Examples

7.6.16 \$srand(arg)

Seed the random number generator. The value passed as the argument will seed the internal random number generator. If the random number generator is re-seeded with the same seed, the sequence of random numbers produced by **\$random** will be the same. The value passed must be a positive integer.

7.6.17 \$zstem(arg)

Returns an word English word stem of the argument. This function attempts to remove common endings from words and return a root stem.

7.6.18 \$zsystem(arg)

Executes "arg" in a system shell. Returns -1 (fork failed) or the return code of the execution of the argument. See also the **shell** command.

7.6.19 \$ztell

Function returns the byte offset in the currently open file. Similar to the C++ **ftello** function. Note: The offset returned is for the file most recently made the default i/o file by the **use** command. **\$ztell** may be used on either a file opened as **new**, **old** or **append**. (See example under **\$zseek** above)

7.6.20 \$zu(expression)

Function returns 1 if the expression is numeric, 0 otherwise.

7.6.21 \$zwi(arg)

Function loads an internal buffer with the string given as the argument. The alphabetic characters of the argument are converted to lower case. The contents of this buffer are returned by the **\$zwn** and **\$zwp** functions. Figure 7 gives examples.

7.6.22 \$zwn extract words from buffer

Function returns successive words from the internal buffer delimited by blanks. When no more words remain, it returns an empty string (string of length zero). Returned words are converted to lower case. See **\$zwi**.

7.6.23 \$zwp extract words from buffer

Function returns successive words from an internal buffer delimited by blanks and punctuation characters. When no more words remain, it returns an empty string (string of length 0). Returned words are converted to lower case. See **\$zwi**.

7.6.24 \$zws(string) initialize internal buffer

Initializes the parse buffer but does not convert "string" to lower case as is the case with **\$zwi**

```
1 #!/usr/bin/mumps
2 set i="now, is the time, for all good"
3 set %=$zwi(i)
4 for w=$zwp write w,!
5 write "-----",!
6 set %=$zwi(i)
7 for w=$zwn write w,!
```

output:

```
now
,
```

```

is
the
time
,
for
all
good
-----
now,
is
the
time,
for
all
good

```

Figure 7 \$Zwi() Examples

7.7 Scan Functions

7.7.1 \$zzScan

7.7.2 \$zzScanAlnum

7.7.3 \$zzInput(var)

The functions return the next word in the current input stream delimited by white space. Words are restricted to a maximum length of 1023. Successive calls return successive words. When there are no more input words, an empty string is returned and **\$test** is set to *false*.

If only part of a line is scanned as a result of these functions, a subsequent **read** command will begin at the white space following the last word returned.

If scanning input from stdin (i/o unit 5), you may signal end of file with a *control-d* on a separate line by itself. This will result terminate the scan and **\$test** will be set to false.

\$zzScan returns all words delimited by whitespace with no conversion. Words may contain any *printable* ASCII character.

\$zzScanAlnum processes words before returning them according to the following rules:

- If a word begins with a number or punctuation, it is not returned.
- Non alpha-numeric characters are removed.
- Alpha characters are converted to lower case.

Both functions will advance to additional lines as needed. If a word exceeds 1023 bytes, the results are undefined. See Figure 8 for an example.

```

for the input line:
now -- __ ?? !@#$$%^&*()_+= IS 2for the time for

      for set i=$zzScan quit:'$test write i,!
output:
now

```

```

--
??
!@#$$%^&*()_+=
IS
2for
the
time
for

for set i=$zzScanAlnum quit:'$test write i,!

```

output:

```

now
the
time
for

```

Figure 8 Scan Functions Examples

\$zzInput(var) reads an entire input line, converts all characters to lower case, separates the words, removes punctuation (as defined by the C *ispunct()* function except hyphen), and stores the words into a numerically indexed array whose name is the value of the variable or constant passed as the argument. The function returns the number of elements in the array. A return of zero indicates no input was obtained (end of file). As the array created by the function could be quite large, you should probably **kill** it when no it is longer needed. The maximum line length permitted is twice the system parameter *MAX_STR* (9,000 bytes by default).

7.8 Vector and Matrix Functions

7.8.1 \$zzAvg(vector)

Computes and returns the average of the numeric values in the vector. For example, see Figure 9.

```

1 #!/usr/bin/mumps
2 for i=1:1:10 set ^a(99,i)=i
3 set i=$zzAvg(^a(99))
4 write "average=",i,!

```

Figure 9 \$zzAvg() Example

The above writes 5.5

7.8.2 \$zzCentroid(gblMatrix,gblRef)

A centroid vector *gblRef* is calculated for the invoking two dimensional global array *gblMatrix*. The centroid vector is the average value for each for each column of the matrix. Any previous contents of the global array named to receive the centroid vector are lost. The global array *gblMatrix* must contain at least two dimensions. See Figure 10 for an example. The matrix must be a top level global array.

```

1 #!/usr/bin/mumps
2 for i=0:1:10 do
3 . for j=1:1:10 do
4 .. set ^A(i,j)=5
5 set %=$zzCentroid(^A,^B)
6 for i=1:1:10 write ^B(i),!

```

output:

```

5

```

```
5
5
5
5
5
5
5
5
5
5
```

Figure 10 \$zzCentroid() Example

7.8.3 \$zzCount(gblVector)

Counts the number of nodes that contain a value in the global array reference and any descendants. For example, see Figure 37.

```
1 #!/usr/bin/mumps
2 kill ^a
3 for i=1:1:10 set ^a(99,i)=i
4 set i=$zzCount(^a(99))
5 write "count=",i,!
```

writes: count=10

Figure 11 \$zzCount() Example

7.8.4 \$zzMax(gbl)

Computes and returns the maximum numeric value in the vector and any descendants. See Figure 12 for an example.

```
1 #!/usr/bin/mumps
1 for i=1:1:10 set ^a(99,i)=i
2 set i=$zzMax(^a(99))
3 write "max=",i,!
```

output:

10

Figure 12 \$zzMax() Example

The above writes the largest value stored in the vector.

7.8.5 \$zzMin(gbl)

Returns the minimum numeric value stored in the vector and any descendants. See Figure 13 for an example.

```
1 #/usr/bin/mumps
2 for i=1:1:10 set ^a(99,i)=i*2
3 set i=$zzMin(^a(99))
4 write "min=",i,!
```

output:

2

Figure 13 \$zzMin() Example

7.8.6 \$zzMultiply(gbl1,gbl2,gbl3)

Multiplies the first and second matrix leaving the result in the third. The ordinary rules of algebra apply. Figure 17 gives an example. The arguments *gbl1* and *gbl2* must be top level, two dimensional arrays.

7.8.7 \$zzSum(gblVector)

Computes and returns the sum of the numeric values stored in the vector. For example, see Figure 18.

7.8.8 \$zzTranspose(gblMatrix1,gblMatrix2)

Transposes the first global array matrix leaving the result in the second. For example, see Figure 19. the argument *gblMatrix1* must be a top level, two dimensional array.

7.9 Text Processing Functions

The following functions are used in connection with experiments in information storage and retrieval.

7.9.1 Similarity Functions

7.9.1.1 \$zzCosine(gbl1,gbl2)

7.9.1.2 \$zzSim1(gbl1,gbl2)

7.9.1.3 \$zzDice(gbl1,gbl2)

7.9.1.4 \$zzJaccard(gbl1,gbl2)

These compute the Cosine, Sim1, Dice and Jaccard similarity coefficients between document vectors given as the first and second arguments. Both arguments are numeric global array vectors. The formulae are given in Figure 14 and an example in code is given in Figure 15. The formulae calculate the similarities between two global array vector *gbl1* and global array vector *gbl2*. The vectors need not be of equal length. Missing elements are interpreted as zero. The vectors should be top level vectors.

$$\begin{aligned} \text{Similarity}_{\text{Dice}}(i, j) &= \frac{2 \sum_{k=1}^{k=t} \text{Term}_{ik} \cdot \text{Term}_{jk}}{\sum_{k=1}^{k=t} \text{Term}_{ik} + \sum_{k=1}^{k=t} \text{Term}_{jk}} \\ \text{Similarity}_{\text{Jaccard}}(i, j) &= \frac{\sum_{k=1}^{k=t} \text{Term}_{ik} \cdot \text{Term}_{jk}}{\sum_{k=1}^{k=t} \text{Term}_{ik} + \sum_{k=1}^{k=t} \text{Term}_{jk} - \sum_{k=1}^{k=t} (\text{Term}_{ik} \cdot \text{Term}_{jk})} \\ \text{Similarity}_{\text{Cosine}}(i, j) &= \frac{\sum_{k=1}^{k=t} \text{Term}_{ik} \cdot \text{Term}_{jk}}{\sqrt{\sum_{k=1}^{k=t} \text{Term}_{ik}^2 \cdot \sum_{k=1}^{k=t} \text{Term}_{jk}^2}} \\ \text{Similarity}_{\text{Sim1}}(i, j) &= \sum_{k=1}^{k=t} \text{Term}_{ik} \cdot \text{Term}_{jk} \end{aligned}$$

Figure 14 Similarity Formulae

```

1  #!/usr/bin/mumps
2  kill ^A
3  kill ^B
4
5  set ^A("1")=3
6  set ^A("2")=2
7  set ^A("3")=1
8  set ^A("4")=0
9  set ^A("5")=0
10 set ^A("6")=0
11 set ^A("7")=1
12 set ^A("8")=1
13
14 set ^B("1")=1
15 set ^B("2")=1
16 set ^B("3")=1
17 set ^B("4")=0
18 set ^B("5")=0
19 set ^B("6")=1
20 set ^B("7")=0
21 set ^B("8")=0
22
23 write "Cosine=", $zzCosine(^A, ^B), !
24 write "Siml=", $zzSiml(^A, ^B), !
25 write "Dice=", $zzDice(^A, ^B), !
26 write "Jaccard=", $zzJaccard(^A, ^B), !

```

output:

```

Cosine=0.75
Siml=6
Dice=1
Jaccard=1

```

Figure 15 Similarity Functions

7.9.2 \$zzBMGSearch(arg1,arg2)

Boyer-Moore-Gosper Function returns the number of non-overlapping occurrences of *arg1* in *arg2*.

These functions, were obtained from

`ftp://ftp.uu.net/usenet/comp.sources.unix/volume5/bmgsubs.Z`

and were written by Jeffrey Mogul (Stanford University), based on code written by James A. Woods (NASA Ames, an agency of the U.S. Government) and are thus believed to be in the public domain. Figure 16 gives an example.

```

1  #!/usr/bin/mumps
2  set key="now"
3  set str="now is the now of the now in the know"
4  write $zBMGSearch(key, str), !

```

output:

```

4

```

Figure 16 \$zzBMGSearch() Example

7.9.3 \$zPerlMatch(string,pattern)

Applies the Perl **pattern** to **string** and returns 1 if the pattern fits and 0 otherwise. The **\$zPerlMatch** function has the side effect of creating variables in the local symbol table to hold backreferences, the equivalent concept of **\$1**, **\$2**, **\$3**, ... in Perl. Up to nine backreferences are currently supported, and can be accessed through the same naming scheme as Perl (**\$1** through **\$9**). These variables remain defined up to a subsequent call to **\$zPerlMatch**, at which point they are replaced by the backreferences captured from that invocation. Undefined backreferences are cleared between invocations; that is, if a match operation captured five backreferences, then **\$6** through **\$9** will contain the empty string. Figure 20 contains examples (long lines wrapped).

```
1  #/usr/bin/mumps
2  set ^d("1","1")=2
3  set ^d("1","2")=3
4  set ^d("2","1")=1
5  set ^d("2","2")=-1
6  set ^d("3","1")=0
7  set ^d("3","2")=4
8
9  set ^e("1","1")=5
10 set ^e("1","2")=-2
11 set ^e("1","3")=4
12 set ^e("1","4")=7
13 set ^e("2","1")=-6
14 set ^e("2","2")=1
15 set ^e("2","3")=-3
16 set ^e("2","4")=0
17
18 set %=$zzMultiply(^d,^e,^f)
19
20 for i="":$order(^f(i)):"" do
21 . for j="":$order(^f(i,j)):"" do
22 .. write i," ",j," ",^f(i,j),!
```

output:

```
1 1 -8
1 2 -1
1 3 -1
1 4 14
2 1 11
2 2 -3
2 3 7
2 4 7
3 1 -24
3 2 4
3 3 -12
3 4 0
```

Figure 17 \$zzMultiply() Example

```
1 #!/usr/bin/mumps
2 for i=1:1:10 set ^a(99,i)=i
3 set i=$zzSum(^a(99))
4 write "sum=",i,!
```

output:

```
55
```

Figure 18 \$zzSum() Example

```

1  #!/usr/bin/mumps
2  kill ^f
3
4  set ^d("1","1")=2
5  set ^d("1","2")=3
6  set ^d("2","1")=4
7  set ^d("2","2")=0
8
9  set %=$zzTranspose(^d,^f)
10
11 for i="":$order(^f(i)):"" do
12 . for j="":$order(^f(i,j)):"" do
13 .. write i," ",j," ",^f(i,j),!

```

output:

```

1 1 2
1 2 4
2 1 3
2 2 0

```

Figure 19 \$zzTranspose() Example

```

1  #!/usr/bin/mumps
2  write "Please enter a telephone number:",!
3  read phonenum
4
5  set p="^(1-)?(\(?\d{3}\)?)?(-| )?\d{3}-?\d{4}$"
6  if $zperlmatch(phonenum,p) do
7  . write "+++ This looks like a phone number.",!
8  . write "The area code is: ",$2,!
9  else do
10 . write "--- This didn't look like a phone number.",!

```

output:

```

Please enter a telephone number:
(123) 456-7890
+++ This looks like a phone number.
The area code is: (123)

```

```

Please enter a telephone number:
(123) 456-7890
+++ This looks like a phone number.

```

Figure 20 \$zPerlMatch() Example

7.9.4 \$zReplace(string,pattern,replacement)

The regular expression in *pattern* is evaluated on *string* and, if there is a match, the matching section is replaced by *replacement*. Figure 21 contains an example. In the first part, the word 'is' is replaced by 'IS'. In the second part, a match is sought for any content between two sets of matching brackets ([...]). The matched section is in back reference \$2. This is then used as a pattern to be replaced.

7.9.5 \$zShred(string,length)

7.9.6 \$zShredQuery(string,length)

The **\$zShred()** function segments the input argument **string** into fragments of **length** size upon successive calls. The function returns a string of length zero when there are no more fragments of size **length** remaining (thus, short fragments at the end of a string are not returned).

\$zShred copies the input string to an internal buffer upon the first call. Subsequent calls retrieve from this buffer. When the buffer is consumed, the function will copy the contents of the next string submitted to the buffer. Figure 22 contains an example.

```
1 #!/usr/bin/mumps
2 set a="now is the time for all"
3 set a=$zReplace(a,"is","IS")
4 write a,!
5
6 set a="[[now is the time]]"
7 if $zPerlMatch(a,"(\[\[)(.)(\]\])") do
8 . set a=$zReplace(a,$2,"ABC")
9 . write a,!
```

output:

```
now IS the time for all
[[ABC]]
```

Figure 21 \$zReplace() Example

```
1 #!/usr/bin/mumps
2 set a="now is the time for all good men to "
3 set a=a_"come to the aid of the party"
4 for do quit:j=""
5 . set j=$zShred(a,5)
6 . if j="" quit
7 . write j,!
```

output:

```
nowis
theti
mefor
allgo
odmen
to com
etoth
eaido
fthep
```

Figure 22 \$zShred() Example

The **\$zShredQuery** function segments **length** shifted copies of the input **string** into fragments of size **length** upon successive calls. That is, the function first returns all the fragments of size **length** of the **string** in the same manner as **\$zShred**. However, it then shifts the starting point of the input string to the right by one and returns all the fragments of size **length** relative to the shifted starting point. If repeatedly called, it repeats this process a total of **length** times. When there are no more combinations, the empty string is returned as shown in Figure 23.

```

1 #!/usr/bin/mumps
2 set a="now is the time for all good men to come to "
3 set a=a_"the aid of the party"
4 for do quit:j=""
5 . set j=$zShredQuery(a,5)
6 . if j="" quit
7 . write j,!

```

output:

nowis	tothe	goodm
theti	aidof	entoc
mefor	thepa	ometo
allgo	wisth	theai
odmen	etime	dofth
tocom	foral	epart
etoth	lgood	isthe
eaido	mento	timef
fthep	comet	orall
owist	othea	goodm
hetim	idoft	entoc
efora	hepar	ometo
llgoo	isthe	theai
dment	timef	dofth
ocomo	orall	epart

Figure 23 \$ShredQuery() Example

7.9.7 \$zSoundex(s1)

Returns the Soundex code for the argument string as follows:

1. All letters are converted to lower case;
2. Non-alphabetic characters are removed;
3. Adjacent duplicate letters are replaced by a single occurrence;
4. The first letter is retained;
5. The letters b, f, p, and v are replaced by the number 1;
6. The letters c, g, j, k, q, s, x, and z are replaced by the number 2;
7. The letters d and t are replaced by the number 3;
8. The letter l is replaced by the number 4;
9. The letters m and n are replaced by the letter 5;
10. the letter r is replaced by the number 6;
11. The is truncated to four characters.

7.9.8 \$zSmithWaterman(s1,s2,algn,mat,gap,noMatch,match)

Computes the Smith Waterman score between two strings. Result returned is the highest alignment score achieved. String lengths are limited by **STR_MAX** in the interpreter. If you compare very long strings (>100,000 characters), you may exceed stack space. This can be increased under Linux with the command:

```
ulimit -s unlimited
```

Figure 24 gives an example.

```

1 #!/usr/bin/mumps
2 set s1="now is the time"
3 set s2="now i th time"
4 set i=$zSmithWaterman(s1,s2,1,0,-1,-1,2)

```

```

5 write "score=",i,!
output:

1 now- is the time 16
  ::: :: ::: :::::
1 now i- th time 16

score=23

```

Figure 24 \$zSmithWaterman() Example

Parameters:

If *algn* is zero, no printout of alignments is produced. If *algn* is not zero, a summary of the alternative alignments will be printed.

If *mat* is zero, intermediate matrices will not be printed.

The parameters *gap*, *noMatch* and *match* are the gap and mismatch penalties (negative integers) and the match reward (a positive integer).

If insufficient memory is available, a segmentation violation will be raised. Try increasing your stack size.

7.9.9 \$zzIDF(global,doccount)

Calculates the Inverse Document Frequency score of words contained in the argument *global*. The parameter *doccount* is the total number of documents. The index of each element of the *global* vector is a word and the value stored is the number of times the word occurs in the collection. Figure 25 gives an example. The vector argument *global* must be a top level array.

```

1 #!/usr/bin/mumps
2 set ^a("now")=2
3 set ^a("is")=5
4 set ^a("the")=6
5 set ^a("time")=3
6 set j=4
7 set %=$zzIDF(^a,j)
8 for i="":$order(^a(i)):"" write i," ",^a(i),!

output:

is 0.7
now 2.0
the 0.4
time 1.4

```

Figure 25 \$zzIDF() Example

7.9.10 Correlation Functions

7.9.10.1 \$zzTermCorrelate(global1,global2)

Calculates the Term-Term co-occurrence matrix for the Document-Term matrix in *global1*. The result is placed in *global2*.

A Term-Term matrix has terms (words) as the indices of its rows and columns. A Term-Term matrix gives, for each position, the degree to which the term corresponding to the row is similar to the term corresponding to the column. The diagonal, which is the degree a term is related to itself, is ignored. Both operands must be top level arrays.

In both the doc-doc and term-term matrices, the upper and lower diagonal matrices are mirror images of one another. Figure 26 gives an example. The order of words in the output will depend upon which data base facility is in use and what it's collating settings are. The Native global array handler collates according to ASCII-7.

```
1  #!/usr/bin/mumps
2  kill ^A,^B
3
4  set ^A("1","computer")=5
5  set ^A("1","data")=2
6  set ^A("1","program")=6
7  set ^A("1","disk")=3
8  set ^A("1","laptop")=7
9  set ^A("1","monitor")=1
10
11 set ^A("2","computer")=5
12 set ^A("2","printer")=2
13 set ^A("2","program")=6
14 set ^A("2","memory")=3
15 set ^A("2","laptop")=7
16 set ^A("2","language")=1
17
18 set ^A("3","computer")=5
19 set ^A("3","printer")=2
20 set ^A("3","disk")=6
21 set ^A("3","memory")=3
22 set ^A("3","laptop")=7
23 set ^A("3","USB")=1
24
25 set %=$zzTermCorrelate(^A,^B)
26
27 for i="":$order(^B(i)):"" do
28 . write i,!
29 . for j="":$order(^B(i,j)):"" do
30 .. write ?10,j," ",^B(i,j),!
```

output:

USB			monitor 1		monitor
	computer 1		printer 1		computer 1
	disk 1		program 1	language	data 1
	laptop 1		computer 1		disk 1
	memory 1		laptop 1		laptop 1
	printer 1		memory 1		program 1
computer			printer 1		printer
	USB 1		program 1		
	data 1	laptop			USB 1
	disk 2		USB 1		computer 2
	language 1		computer 3		disk 1
	laptop 3		data 1		language 1
	memory 2		disk 2		laptop 2
	monitor 1		language 1		memory 2
	printer 2		memory 2	program	program 1
	program 2		monitor 1		
data			printer 2		computer 2
	computer 1		program 2		data 1
	disk 1	memory			disk 1
	laptop 1		USB 1		language 1
	monitor 1		computer 2		laptop 2
	program 1		disk 1		memory 1
disk			language 1		monitor 1
	USB 1		laptop 2		printer 1
	computer 2		printer 2		
	data 1		program 1		
	laptop 2				
	memory 1				

Figure 26 \$zTermCorrelate() Example

7.9.10.2 \$zDocCorrelate(gblref1,gblref2,mthd,thrshld)

A square Document-Document matrix *gblref2* is calculated from the Document-Term matrix *gblref1* according to method *mthd* (Cosine, Sim1, Dice, Jaccard). The value of elements in the Document-Document matrix will not exceed threshold (*thrshld*) and the cells associated with corresponding document numbers will not exist.

A Document-Document matrix has document id's as its row and column indices. A cell in the matrix indicates the degree to which the row document is related to the column document. The diagonal is ignored. Figure 27 gives an example.

7.9.11 Stop and Synonym Functions

7.9.11.1 \$zStopInit(arg)

7.9.11.2 \$zStopLookup(word)

7.9.11.3 \$zSynInit(fileName)

7.9.11.4 \$zSynLookup(word)

A call to **\$zStopInit(file_name)** will open and load a file of stop words into a C++ container. The file should consist of one word per line. If the file cannot be opened or there is insufficient memory to hold the list of words, the program will halt with an error message. **\$zStopInit()** converts all words to lower case.

```

1 #!/usr/bin/mumps
2 kill ^A,^B
3
4 set ^A("1","computer")=5
5 set ^A("1","data")=2
6 set ^A("1","program")=6
7 set ^A("1","disk")=3

```

```

8  set ^A("1","laptop")=7
9  set ^A("1","monitor")=1
10
11 set ^A("2","computer")=5
12 set ^A("2","printer")=2
13 set ^A("2","program")=6
14 set ^A("2","memory")=3
15 set ^A("2","laptop")=7
16 set ^A("2","language")=1
17
18 set ^A("3","computer")=5
19 set ^A("3","printer")=2
20 set ^A("3","disk")=6
21 set ^A("3","memory")=3
22 set ^A("3","laptop")=7
23 set ^A("3","USB")=1
24
25 set %=$zzDocCorrelate(^A,^B,"Cosine",.5)
26
27 for i="":$order(^B(i)):"" do
28 . write i,!
29 . for j="":$order(^B(i,j)):"" do
30 .. write ?10,j," ",^B(i,j),!

```

output:

```

1
      2 0.887096774193548
      3 0.741935483870968
2
      1 0.887096774193548
      3 0.701612903225806
3
      1 0.741935483870968
      2 0.701612903225806

```

Figure 27 \$zDocCorrelate()Example

A call to **\$zStopLookup(word)** will return 1 if *word* is in the stop list, 0 otherwise. Words presented to **\$zStopLookup(word)** should be in lower case.

\$SynInit() opens a synonym file. The file should consist of two or more words per line separated by from one another by one blank. The words are treated as synonyms with the first word on each line as the primary synonym. The primary synonym may be a code or category number. This word or code will be returned if any of the remaining words are passed as arguments to **\$SynLookup()**. Figure 28 gives an example.

These functions are peculiar to this implementation..'

Assume that the file "stop" contains the word "and"

```

set %=$zStopInit("stop")
if $zStopLookup("and") write "yes",!

```

Writes yes

Assume that the file "synonyms" contains a line with the text:

compression compressions compress compressed compresses

```
set %=$zSynInit("synonyms")
write $zSynLookup("compressions"),!
```

output:

compression

Figure 28 Stop List Functions

7.10 GTK Related Functions

7.10.1 `$z~mdh~toggle~button~set~active(ToggleButtonReference,intVal)`

Sets the button to active if intVal is 1, inactive if the value is 0.

7.10.2 `$z~mdh~dialog~new~with~buttons(ParentWindowRef,dialog)`

Raises a Gtk Dialog window displaying the contents of *dialog* with buttons **Yes** and **No**. Returns 1 if **Yes** is clicked; 0 if **No** is clicked; and -1 if the box is dismissed.

7.10.3 `$z~mdh~entry~get~text(EntryReference)`

Returns the current string contents of the referenced Entry box.

7.10.4 `$z~mdh~entry~set~text(EntryReference,value)`

Sets the contents of the named entry box to *value*.

7.10.5 `$z~mdh~text~buffer~set~text(TextBufferReference,string)`

Sets the contents of the referenced text buffer to the value of string.

7.10.6 `$z~mdh~label~set~text(LabelReference,string)`

Sets the text contents of the label referenced to string. Triggers a value changed signal.

7.10.7 `$z~mdh~tree~selection~get~selected(TreeModelReference,column)`

Returns value in designated column of referenced TreeModel.

7.10.8 `$z~mdh~tree~store~clear(TreeStoreReference)`

Clears (deletes) the contents of the referenced TreeStore.

7.10.9 `$z~mdh~tree~level~add(TreeStoreReference,treeDepth,index,data[,...])`

Add index at tree level treeDepth to column 1 of TreeStore. Add additional data items in successive columns.

7.10.10 `$z~mdh~spin~button~get~value(SpinButtonReference)`

Returns the current value of the referenced SpinButton.

7.10.11 `$z~mdh~spin~button~set~value(SpinButtonReference,number)`

Sets the current value of the referenced spin button to number.

7.10.12 `$z~mdh~widget~hide(widgetReference)`

Hides the widget from view.

7.10.13 `$z~mdh~widget~show(widgetReference)`

Displays (un-hides) the widget.

8 Interpreter & Compiler Implementation Notes

8.1 Source Code Format

C++ and C code were formatted using:

```
astyle -A6 -s6 *.cpp *.c
```

C++ generated by the Mumps compiler is formatted in the same manner if *astyle* is available.

8.2 Modulo Operator

The modulo operator (#) returns results that are the same as the C/C++ modulo operator (%). Some Mumps documentation shows the Mumps modulo returning results that are different than what would be expected from C/C++ modulo.

8.3 Goto Command

If you use a **goto** command, all **do** command pending returns are canceled. That is if you invoke a section of code by means of a **do** and the section of code executes a **goto** command, the return to the line the **do** was on is canceled as well as any other pending returns.

You may not use a **goto** in a compiled program block.

8.4 Notes on Arithmetic Precision

See section 2.4 on page 19 for additional details.

8.5 \$fnumber()

The builtin function **\$fnumber()** only works on numbers that can be represented in a 64 bit floating point variable.

8.6 Exponential format numbers

All numbers represented in exponential format are treated as floating point numbers. If exponential format constants are used in expressions, they must be enclosed in quotes:

```
set i="1.23e3"*5
```

8.7 Extended Arithmetic Precision

If enabled, Mumps will use the GNU *bignum* integer and MPFR floating point packages (this can be enabled/disabled by a *configure* in the *build...* script).

8.8 Floating Point Precision

When using extended precision MPFR numbers, floating point values have a default fractional precision of 72 bits. This can be changed with the *--with-float-bits=val* configure option. The maximum number of printed decimal digits is, by default, 20. This can be changed with the *--with-float-digits=val* configure option. The number of meaningful decimal digits that can be printed depends upon the number of bits in the fractional part of the floating point number. More bits mean more decimal digits can be printed.

If MPFR is not present, standard hardware *double* precision is used.

8.9 Integer Precision

There is no effective limit to integer precision except string length and memory when the extended precision *bignum* package is in use. Otherwise, precision is the same as the hardware *long*.

8.10 Extended Precision Performance

Extended precision arithmetic results in slower performance. The amount is dependent on how much arithmetic a program does, whether it is mainly integer or floating point (floating point is slower), and, in the case of fixed length numbers, how large the numbers are. Larger numbers result in slower computations.

8.11 Rounding

The *\$justify()* function is useful to round lengthy repeating decimal floating point numbers to a more reasonable value.

8.12 New Command

The **new** command functions differently than in the 1995 standard. The following details its behavior.

8.13 Runtime Symbol Table

The **new** command controls the internal run time symbol table. Upon entering a block by means of a **do** command, a new layer of the symbol table is created. Upon exit, the layer is discarded and the previous layer becomes the current layer.

When a program begins, an initial or base layer is created in the symbol table. In the absence of any **new** commands, newly created variables are stored at this base or initial layer.

When a variable is retrieved, all layers are searched beginning with the most recently created layer and progressing through to older layers until the initial layer is reached.

In the absence of any **new** commands, only the initial or base layer will contain variables.

8.14 Forms of the New Command

There are three forms of the **new** command based on the arguments provided. The first has no arguments, the second has a list of arguments consisting of variable names separated from one another by commas, and, finally, the third has an argument consisting of a parenthesized comma separated list of variable names. For example:

```
new
new a, b, c
new (a, b, c)
```

8.14.1 New Command with No Arguments

A **new** command with no arguments cause the system to copy all variables from all layers to the current layer.

Until the current block is exited, all access to any variable known at the time of the **new** command will access the copy of the variable, not the original. Upon exit from the block, the copies are deleted⁸.

Any variable created whose name was not known when the **new** command was executed, will be created and stored at the lowest base layer of the symbol table and, consequently, not deleted upon exit from the block that contained the **new** command.

⁸ A block is any sequence of code entered as a result of a **do** command.

If a **new** command is executed in a block that invokes a block which itself executes a **new** command, the **new** command in the second block makes a copy of the invoking block's variables along with any variables created by the invoking block after executing its **new** command. If, in the symbol table stack, a variable appears at several layers, only the most recent version will be copied.

An example is given in Figure 29. In this example, variables **i**, **j**, and **k** are created at the beginning of the program. The function **test1** is then called.

Initially, in **test1**, the variables have the same values that they did in the main function. The variable **i** is changed. The **new** command is executed and a copy of all the variables currently known (**i,j,k**) is made to the current layer. The values of **i**, **j**, and **k** are altered the function **test2** is called.

The values of the variables on entry to **test2** are the same as they were in **test1**. Another **new** command is executed making another copy of the variables. These are altered and a new variable, **y**, not previously known at any level (and thus stored at the base level) is created. Return is made to **test1**.

In **test1** the values of the variables are printed and it can be seen that they have reverted to the values they had prior to entering **test2**. Return is made to the main function.

In the main function the variables have reverted to the values they had prior to the invocation of **test1** with the exception of **i** which was altered in **test1** prior to execution of the **new** command. It retains the value it received in **test1**.

Note also that the variable **y** now exists at the main function level since, when it was created in **test1**, it was not in the group of variables copied to the symbol table level for **test1**. Thus, it was created at the base level of the symbol table.

However, when **y** was altered in **test2**, only the copy made by the **new** command in **test2** was altered, not the original.

```
#!/usr/bin/mumps
    set i=10
    set j=20
    set k=30
    do test1
    write "Main: expect 100 20 30 50: ",i," ",j," ",k," ",y,!
    halt

test1 write "test1: expect 10 20 30: ",i," ",j," ",k,!
    set i=100
    new
    set i=11,j=22,k=33,y=50
    do test2
    write "test1: expect 11 22 33 50 : ",i," ",j," ",k," ",y,!
    quit

test2 write "test2: expect 11 22 33 50: ",i," ",j," ",k," ",y,!
    new
    set i=12,j=23,k=34,y=55
    write "test2: expect 12 23 34 55 : ",i," ",j," ",k," ",y,!
    quit

root@AMD6 validate new01.mps

test1: expect 10 20 30: 10 20 30
test2: expect 11 22 33 50: 11 22 33 50
test2: expect 12 23 34 55 : 12 23 34 55
```

```
test1: expect 11 22 33 50 : 11 22 33 50
Main: expect 100 20 30 50: 100 20 30 50
```

Figure 29 **new** Command without Arguments

8.14.2 New Command with Arguments

There are two forms of the **new** command that take arguments.

The first has a list of arguments consisting of variable names separated from one another by commas:

```
new a,b,c
```

The second has an argument consisting of a parenthesized, comma separated list of variable names:

```
new (a,b,c)
```

If a variable is named in the list that does not exist, it is created in the current symbol table layer with a value of the empty string.

8.14.3 New Command with Comma List of Variable Names

If the **new** command argument is a list of one or more variable names, it means that the variables listed will be copied to the current symbol table level and, eventually, discarded when the current block is exited⁹.

If a variable whose name appears in the list exists at several layers in the symbol table stack, only the most recent will be copied.

Any reference to any variable not in the argument list will be satisfied by searching through the symbol table stack for the most recent instance of it. See Figure 30.

If a variable is mentioned in the argument list that does not exist, it is ignored.

```
#!/usr/bin/mumps
  set i=10
  set j=20
  set k=30
  do test1
  write "Main: expect 100 20 30 50: ",i," ",j," ",k," ",y,!
  halt

test1 write "test1: expect 10 20 30: ",i," ",j," ",k,!
  set i=100
  new i,j
  set i=11,j=22,k=33,y=50
  do test2
  write "test1: expect 11 23 34 55 : ",i," ",j," ",k," ",y,!
  quit

test2 write "test2: expect 11 22 33 50: ",i," ",j," ",k," ",y,!
  new i
  set i=12,j=23,k=34,y=55
  write "test2: expect 12 23 34 55 : ",i," ",j," ",k," ",y,!
  quit

root@AMD6 validate # new02.mps
```

⁹ A block is any sequence of code entered as a result of a **do** command.

```

test1: expect 10 20 30: 10 20 30
test2: expect 11 22 33 50: 11 22 33 50
test2: expect 12 23 34 55 : 12 23 34 55
test1: expect 11 23 34 55 : 11 23 34 55
Main: expect 100 20 30 50: 100 20 34 55

```

Figure 30 **new** Command with Comma List

8.14.4 New Command with Parenthesized List of Variable Names

If the **new** command argument list consists of a parenthesized list of one or more variable names, it means to make a copy of the most recent versions of all known variables except for the variable named in the list. This is similar to the no-argument version except the one or more variables known at the time of command execution will not be copied to the current symbol table layer.

When the block containing the **new** command is exited, the copies of the variables are discarded but any changes to this variables given in the argument list are not¹⁰.

See Figure 31.

```

#!/usr/bin/mumps
  set i=10
  set j=20
  set k=30
  do test1
  write "Main: expect 11 22 30 50: ",i," ",j," ",k," ",y,!
  halt

test1 write "test1: expect 10 20 30: ",i," ",j," ",k,!
      new (i,j)
      set i=11,j=22,k=33,y=50
      do test2
      write "test1: expect 11 23 34 55 : ",i," ",j," ",k," ",y,!
      quit

test2 write "test2: expect 11 22 33 50: ",i," ",j," ",k," ",y,!
      new i
      set i=12,j=23,k=34,y=55
      write "test2: expect 12 23 34 55 : ",i," ",j," ",k," ",y,!
      quit

root@AMD6 validate # new03.mps

test1: expect 10 20 30: 10 20 30
test2: expect 11 22 33 50: 11 22 33 50
test2: expect 12 23 34 55 : 12 23 34 55
test1: expect 11 23 34 55 : 11

```

Figure 31 **new** Command with Parenthesized List

8.15 Kill Command

For non-globals, the **kill** command operates only on the current symbol table level.

8.16 For Command Extensions

The **for** command accepts extensions such as the following:

```
for i=$order(^a(i)) ...
```

¹⁰ Note: if one or more of the variables in the argument list are themselves copies from a lower layer but not the base layer, they will eventually be discarded.

```
for i=init:$order(^a(i)):final ...
```

In the first example, the variable *i* will assume all the index values of the global array in collating sequence order.

In the second, the first value of *i* will be the next higher collating sequence value of the index above *init* and subsequent values will be the values in collating sequence order of the global array up to but not including *final*.

8.16.1 Break and Quit

In this version, the **break** command has a non-standard use. Originally intended as a means of interrupting a program for debugging purposes, in this implementation it is used in loop control.

A **quit** in a single line **for** terminates processing of the **for**. If there are multiple **for** commands, it terminates the nearest:

```
for i=1:1:10 write i,! if i>5 quit
      writes 1 through 6 only.
```

```
for i=1:1:10 for j=1:1:10 write j,! if j>5 quit
      writes 1 through 6 ten times.
```

A **break** may *NOT* be used in a single line **for** command. It may *ONLY* be used in an indented block that was introduced by a **do** command.

In an indented block, **quit** and **break** have special meanings:

A **quit** ends further processing of the block in which it appears and returns control to the line containing the invoking **do** at a point just after the **do**. Processing of the line containing the invoking **do** resumes. If there are more commands on the line, they are executed.

A **break** ends further processing of the block in which it appears but does not return the line containing the invoking **do**. Instead, execution moves to the line following the block which the **do** invoked.

Examples:

```
for i=1:1:10 do write " continuing"
. write !,i
. if i>5 quit
. write " ",i
write !,"done",!
```

writes

```
1 1 continuing
2 2 continuing
3 3 continuing
4 4 continuing
5 5 continuing
6 continuing
7 continuing
8 continuing
9 continuing
10 continuing
done
```

In this example, the block is invoked 10 times. After each invocation, the remainder of the line containing the **for** is executed producing the instances of the word "continuing". Each block invocation prints the value of *i*. When the value of *i* is greater than 5, the block executes the **quit** command thus

returning to the invoking line early. When the value of "i" is 5 or less, the full block is executed and return is made to the invoking line at block end. When the **for** command finishes execution, control is passed to the line following the **for** and "done" is printed.

```
set i=9
if i>0 do write " continuing"
. write !,i
. if i>5 quit
. write " ",i
write !,"done",!
```

writes:

```
9 continuing
done
```

In this example, the block is entered, the value of "i" is printed but, because "i" is greater than 5, the **quit** is executed and control is returned to the invoking **do** and the word "continuing" is printed. Now, the line being completely executed, control passes to the line following the block and "done" is printed.

```
for i=1:1:10 do write " mark " do write " end of line",!
. write i
. if i>5 quit
. write "X"
```

writes:

```
1X mark 1X end of line
2X mark 2X end of line
3X mark 3X end of line
4X mark 4X end of line
5X mark 5X end of line
6 mark 6 end of line
7 mark 7 end of line
8 mark 8 end of line
9 mark 9 end of line
10 mark 10 end of line
```

In this example, multiple **do** commands are shown. Note the two blanks following each. Each **do** invokes the block following the line containing the **do**

On the other hand, the **break** command terminates the the block in which it is contained but execution does not return to the line containing the invoking **do** but, instead, continues with the line following the block:

```
for i=1:1:10 do write " continuing"
. write !,i
. if i>5 break
. write " ",i
write !,"done",!
```

writes:

```
1 1 continuing
2 2 continuing
3 3 continuing
4 4 continuing
5 5 continuing
6
done
```

```

set i=9
if i>0 do write " continuing"
. write !,i
. if i>5 break
. write " ",i
write !,"done",!

writes:
9
done

for i=1:1:10 do write " mark " do write " end of line",!
. write i
. if i>5 break
. write "X"
write !

writes:

    1X mark 1X end of line
    2X mark 2X end of line
    3X mark 3X end of line
    4X mark 4X end of line
    5X mark 5X end of line
    6

```

In these examples, execution of the **break** can be seen to terminate the current block and move to the line following the block.

```

for i=1:1:10 do
. for j=1:1:5 do
.. write j,!
.. if j>3 break

```

The above write 1 through 4 ten times.

Note: the contents of **\$test** revert to their former value when exiting an indented block by means of **break** or **quit**:

```

if l=1 do
. write "test 1: ",$test,!
. if l=2 write "wow",!
. else write "not wow",!
. write "test 2: ",$test,!
write "test 3: ",$test,!

writes:

    test 1: 1
    not wow
    test 2: 0
    test 3: 1

```

If you exit a block with a **goto**, the value of **\$test** is not restored.

8.17 Lock Command with SQL

Locks are not needed if you are using Sqlite3 for global array storage as SQL transaction commands can achieve the same or better effect.

When using SQL for the backend global array store, the **Lock** should not be used. Instead, use the more modern native SQL transaction processing commands (*BEGIN*, *COMMIT*, *ROLLBACK*, etc.) to achieve the same effect with far greater integrity.

8.18 Lock Command in Native Database Mode

The *lock* command has no effect on the single user database as there are no other users to lock datya access to.

8.19 Naked indicator

This version of Mumps does not support the naked indicator.

It was originally included in early versions of Mumps because of the binary mapping of an n-way tree which was used at the time to store the global arrays. The naked indicator was a short-hand to the interpreter to allow it to search for a global without stating at the top of the tree each time thus resulting in faster access. That is no longer the case with B-tree based access methods.

8.20 Job command

The **JOB** command results in a C/C++ *fork()* function to be executed thus creating a child process. The child process will attempt to execute the argument to the **JOB** command. The **JOB** command may be used in the native B-tree user mode but only one process may access the globals. The Sqlite3 version has no such restriction.

The child process must end with a **HALT** command or the child process will hang.

8.21 File Names Containing Directory Information

When invoking a file name containing directory information (forward slash in Linux) with the **DO** or **GOTO** commands, the file name itself **must** be enclosed in quotes. For example:

```
set x="""^/home/user/xxx.mps"" goto @x
goto @""^/home/user/xxx.mps""
```

Note the extra quotes. These are required in order to embed the file name in quotes.

8.22 File Names

File names should conform to variable naming conventions except that the first character of a file name may not be the percent sign (%) character. The first character must be alphabetic. File names may only contain letters, digits and the percent sign. The underscore character may not be used.

8.23 Array Index Collating Sequence

Array index collating sequences for both global and local array is ASCII. That is, for the *\$query()* and *\$order()* functions, all array indices will be presented in the same order as ASCII strings. Thus, in an array with 15 elements whose indices range from 1 to 15, the indices will be presented as:

```
1 10 11 12 13 14 15 2 3 4 5 6 7 8 9
```

You may achieve numeric ordering by storing the indices padded to left with blanks such as:

```
for i=1:1:15 set ^a($justify(i,8))=i
set i="" for set i=$order(^a(i)) quit:i="" write +i," "
```

the indices will now be presented as:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Note the the *+i* in the **write** command has the effect of converting the string to a number with no leading blanks.

8.24 Subroutine & Function Calls

Subroutines and functions may be performed in several ways as shown in Figure 32. Values returned from functions invoked by a **do** command are ignored. In standard Mumps, the **\$\$** form is used only with function invocations.

Caution: be certain to include a **halt** or other exit in your program *prior* to any functions that may appear at the end of your code. If the **halt** is not present, function code will be entered and any passed variables will be undefined.

```
#!/usr/bin/mumps
# calls.mps

    set i=10
    do fcn(i)
    do fcn(5)
    do $$fcn(i)
    do $$fcn(5)
    set k=$$fcn(5)
    write "returned k=",k,!

    set i=10
    do fcn^ext.mps(i)
    do fcn^ext.mps(5)
    do $$fcn^ext.mps(i)
    do $$fcn^ext.mps(5)
    set k=$$fcn^ext.mps(5)
    write "returned k=",k,!

    do fcn^ext1.mps
    do fcn^ext1.mps
    do $$fcn^ext1.mps
    do $$fcn^ext1.mps
    set k=$$fcn^ext1.mps
    write "returned k=",k,!

    halt

fcn(x) write "in fcn(x) value passed is ",x,!
      quit x

-----

#!/usr/bin/mumps
# ext.mps

fcn(x) write "in fcn(x) value passed is ",x,!
      quit x

-----

#!/usr/bin/mumps
# ext1.mps

fcn    write "in fcn ext1.mps",!
      set x=22
      quit x
```

output results:

```
in fcn(x) value passed is 10
in fcn(x) value passed is 5
in fcn(x) value passed is 10
in fcn(x) value passed is 5
in fcn(x) value passed is 5
returned k=5
in fcn(x) value passed is 10
in fcn(x) value passed is 5
in fcn(x) value passed is 10
in fcn(x) value passed is 5
in fcn(x) value passed is 5
returned k=5
in fcn ext1.mps
in fcn ext1.mps
in fcn ext1.mps
in fcn ext1.mps
in fcn ext1.mps
returned k=22
```

Figure 32 Subroutine/Function Calls

8.25 \$Fnumber() Function

The **\$fnumber()** is implemented via the C function **strfmon()** which provides much greater flexibility when dealing with differing locales and, especially, currencies. The default locale is **en_US.UTF-8** but this can be set with the **configure** option:

```
--with-locale=location-information
```

You may use **\$fnumber()** with the legacy Mumps parameters or use it with a pattern parameter designed for **strfmon()**.

If you use the **strfmon()** parameter option, the function takes two arguments. The first must be a number consisting of only numeric characters. The second is a character string conforming to a **strfmon()** pattern but preceded by an asterisk to distinguish the pattern from those used by the legacy Mumps function of the same name. The **strfmon()** function is well documented but here are some examples:

```
set x=12345.6789
write $fn(x,"*%!n")    ==> 12,345.68
write $fn(x,"*%n")     ==> $12,345.68
write $fn(x,"*%i")     ==> USD 12,345.68
write $fn(x,"*%n3")    ==> $12,345.683
write $fn(x,"*%20n")   ==>                $12,345.68
```

8.26 \$Select() Function

All arguments of the **\$select()** function are evaluated. In standard Mumps, they are evaluated individually until one is true or all are false.

8.27 Compiling Large Programs

When compiling¹¹ large programs, especially if SQL is enabled, there may be a warning about **variable tracking** from the gcc/g++ compiler. You may ignore this.

¹¹ Using the compiler is not presently recommended.

8.28 Embedded Expressions

In several extended Mumps commands, the figure `&~exp~` may appear. The expression *exp* is evaluated and the result replaces the figure. For example:

```
set x="ls -lh"
shell &~x~
```

8.29 Inline C++ Code (Compiler Only)

Lines that begin with a plus (+) sign in column 1 are inserted as-is into C++ programs. Usually, these will be lines of C++ code. For example, if you have a line of Mumps code you want to execute 1000 times, the Mumps code would be:

```
for i=1:1:1000 do
. write "abc",!
```

This could be written as:

```
+   for(int i=0; i < 1000; i++) {
+       write "abc",!
+   }
```

The C++ *for* loop is considerably faster. Note the omission of the “.” indent.

Mumps code will not have access to the C++ variable unless you use a *declared* variable into which you place the C++ variables value:

```
        declare val
+   for(int i=0; i < 1000; i++) {
+       sprintf(val, "%s", i);
+       write val,!
+   }
```

8.30 Functions

This is the form of subroutine was originally used in Mumps. There are no parameters passed to the subroutine and the subroutine shares the same *namespace* as the calling program hence, as seen in the example in Figure 33, the values of the variables *i*, *j*, and *k* are accessible to the subroutine and any changes to them are available in the calling program.

Variables created in the subroutine in the normal manner by a **set** or **read** command, unless the subject of a **kill** command, are available to the calling routine.

Variables created in the subroutine as a result of a **new** command are destroyed upon return and are not available to the calling routine.

```
zmain
set i=10
set j=20
set k=30
write "main program: ",i," ",j," ",k,!
do test
write "main program: ",i," ",j," ",k,!
write "main program x=",x,!
write "main program $data(y)=", $data(y),!
halt
```

```

test
    write "sub-program: ",i," ",j," ",k,!
    set i=11
    set j=22
    set k=33
    set x=22
    new y
    set y=33
    quit

```

which produces the following output:

```

main program: 10 20 30
sub-program: 10 20 30
main program: 11 22 33
main program x=22
main program $data(y)=0

```

Figure 33 Local Functions

8.30.1 Call by Value

This form of subroutine call was introduced later in the evolution of Mumps. It permits parameters to be passed to the subroutine but the subroutine maintains a separate name space for values passed to it as parameters. Variables from the calling program are visible to the called program. Variables created by the called program become available to the calling program upon return (except if they are **killed** prior to return or created by a **new** command), and variables created in the called program are deallocated upon return and are thus not visible to the calling program. Changes to parameters passed to the called program do not change the corresponding arguments in the calling program.

```

zmain
    set i=10
    set j=20
    set k=30
    write "main program: ",i," ",j," ",k,!
    do test(i,j,k)
    write "main program: ",i," ",j," ",k,!
    halt

test(a,b,c)
    write "sub-program: ",a," ",b," ",c,!
    set a=11
    set b=22
    set c=33
    quit

```

which produces the following output:

```

main program: 10 20 30
sub-program: 10 20 30
main program: 10 20 30

```

Figure 34 Call by Value Functions

8.30.2 Call by Reference.

Same as the above but 'call by reference' permitted. That is, changes to parameters made by the called program cause changes to the corresponding arguments in the calling program. Note the "." in front of the variables in the 'do' command that are to be passed by reference. Both call by reference and call by value arguments may be mixed in the same 'do' statement.

```
#!/usr/bin/mumps
  zmain
  set i=10
  set j=20
  set k=30
  write "main program: ",i," ",j," ",k,!
  do test(.i,.j,.k)
  write "main program: ",i," ",j," ",k,!
  halt

test(a,b,c)
  write "sub-program: ",a," ",b," ",c,!
  set a=11
  set b=22
  set c=33
  quit
```

which produces the following output:

```
main program: 10 20 30
sub-program: 10 20 30
main program: 11 22 33
```

Figure 35 Call by Reference Functions

In each of the examples, the subroutine and calling program are actually part of the same C++ function. In effect, subroutines of the type shown above are similar to the old Basic **gosub** facility. Functions such as shown above may also return values:

An example recursive factorial computation is shown in Figure 36.

```
#!/usr/bin/mumps
  zmain
  set i=$$factorial(5)
  write "factorial=",i,!
  halt

factorial(a)
  write "sub-program: a=",a,!
  if a<2 quit 1
  set b=$$factorial(a-1)
  write "a=",a," b=",b,!
  quit a*b

sub-program: a=5
sub-program: a=4
sub-program: a=3
sub-program: a=2
sub-program: a=1
a=2 b=1
a=3 b=2
a=4 b=6
a=5 b=24
factorial=120
```

Figure 36 Function Return Values

8.31 Shell Commands

The **shell** command passes the remainder of the line to a shell for execution (**sh** in Linux). Shell output will appear on **stdout**. The command sets **\$test** to false if the *fork()* fails, true otherwise.

8.31.1 shell/p

The **shell/p** form passes the remainder of the line to a shell for execution but opens a pipe **from** the shell **to** Mumps unit number 6. All **stdout** output from the shell is directed to unit number 6 and can be read with any of the input commands or functions in association with the **use** command.

8.31.2 shell/g

The **shell/g** form passes the remainder of the line to a shell for execution (**sh** in Linux) and opens an output pipe **from** the Mumps program **to** the shell as Mumps unit number 6. Data **written** to this unit becomes **stdin** to the shell. Output from the shell is written to **stdout**. Remember to **close** unit number 6 to signal end-of-file to the shell.

8.31.3 shell

With no qualifier, the **shell** command passes the remainder of the command line to a shell. Input or output from the shell come from or go to **stdin** or **stdout**, respectively.

For example:

```
shell sort dictionary.tmp | uniq -c | sort -nr > dictionary.s
```

The Linux shell created will do the following:

1. The file *dictionary.tmp*, a collection of words, will be sorted by **sort** and the output piped to **uniq**
2. **uniq** counts duplicate entries and pipes its output consisting of a count and a word to **sort**
3. **sort** sorts the result numerically by number of duplicates in reverse order and writes its output to *dictionary.s*.

```
1 shell/p sort dictionary.tmp | uniq -c | sort -nr
2 open 1:"dictionary.s,new"
3 for do
4   . use 6
5   . read line
6   . if '$test break
7   . use 1
8   . write line,!
9 close 1
```

Figure 37 Shell Command Example

The above does the same but the output will be presented to Mumps unit 6 which reads and writes the result to the file named *dictionary.s*

8.32 Database *expr*

By default, Native database file *key.dat* and *data.dat* are stored in the directory current when a program is invoked.

The **database** command may be used to set the name of the files to be used to store the native global arrays. The expression will be evaluated and the resulting name will become the name, suffixed *.key* and *.dat*, of the files in which the native global arrays are stored. The expression may contain directory information. For example:

```
database "/home/user/data/mumps"
```

will cause the system to access files:

```
/home/user/data/mumps.key  
/home/user/data/mumps.dat
```

This command **must** be issued prior to any attempt to access the global arrays. It only works with the native B-tree database option.

8.33 Zhalt return_code

The **zhalt** command will terminate the current program with a return error code given by its argument. Example:

```
if a=0 zhalt 99
```

The value of **\$?** in the BASH environment will be 99.

9 The Multi-Dimensional and Hierarchical Database Toolkit

9.1 Introduction

The MDH (Multi-Dimensional and Hierarchical) Database Toolkit is a C++ Linux-based, open sourced, toolkit of software that supports fast, flexible, multi-dimensional and hierarchical storage, retrieval and manipulation of information in data bases in a manner similar to the Mumps language.

The package is written in C and C++ and is available under the GNU GPL/LGPL licenses in source code form.

The distribution kit contains demonstration implementations of text and sequence retrieval tools that function with very large genomic data bases and illustrate the toolkit's capability to manipulate massive data sets of genomic information.

The toolkit is distributed as part of the Mumps Compiler for Linux.

The toolkit is a solution to the problem of manipulating very large, character string indexed, multi-dimensional, sparse matrices. It is based on Mumps (also referred to as *M*), a general purpose programming language that originated in the mid 60's at the Massachusetts General Hospital. The toolkit supports access to the SQLite relational data base server, the Perl Compatible Regular Expression Library, and the Glade GUI builder.

The principal database feature in this project is the ***global array*** which permits direct, efficient manipulation of multi-dimensional arrays of effectively unlimited size.

A global array is a persistent, sparse, undeclared, multi-dimensional, string indexed data disk based structure. A global array may appear anywhere an ordinary array reference is permitted and data may be stored at leaf nodes as well as intermediate nodes in the data base array. The number of subscripts in an array reference is limited only by the system's maximum length array reference restriction with all subscripts expanded to their string values. The toolkit includes several functions to traverse the data base and manipulate the arrays.

The toolkit makes the data base and function set available as C++ classes and also permits execution of legacy Mumps scripts. To use the toolkit, you install the MDH and Mumps distribution. kit and related code.

9.2 Installation

The class libraries and related functions along with the Mumps Compiler and Interpreter must be installed before attempting to use the MDH package. See Section 2

The distribution contains a number of example programs written both in Mumps and C++/MDH.

9.2.1 Global Array Database

The global array database will be stored in either a SQLite3 or native database depending on the selections made during installation of the distro described in Section 3 above.

As is the case with the Mumps compiler and interpreter, the database file will be stored either in files named *key.dat* and *data.dat* (native database option) or *mumps.sqlite* for the SQLite3 option. Database files are normally local to the directory containing the running program but links to files in other directories are permitted.

The file *mumps.sqlite* must be initialized prior to use as described in Section 5.3 above.

SQLite3 database programs are subject to the same optimization as interpreter or compiler programs. See Section 5.4 above.

9.3 Compiling Programs

To compile a C++ program that uses the MDH (Multi-Dimensional and Hierarchical) library, use the command:

```
mumpsc myprog.cpp
```

This will invoke the **g++** compiler and make available the necessary libraries. The result will be a program named *myprog* which is executable. The script **mumpsc** is part of the Mumps Compiler which must be installed prior to using the toolkit.

Note: the **mumpsc** command is also used to compile Mumps source code to C++ and then to binary executables so Mumps programs may be executed directly rather than by the Mumps interpreter. The file ending (.cpp or .mps) determines behavior.

9.4 Writing C++ MDH Programs

In order to use global arrays or other MDH features, each C++ program must include the MDH header file at the beginning of the program:

```
#include <mumpsc/libmpscpp.h>
```

This header and related library code is installed on your system when you install the Mumps Compiler/Interpreter software.

9.5 MDH Implementation of Globals

The main feature of the MDH is its implementation of Mumps global arrays as a C++ class. This implementation also includes a number of builtin functions to manipulate and traverse global arrays as well as a class of string, **mstring**, which imitates the behavior of Mumps strings.

Global arrays are undimensioned, string indexed, disk resident data structures whose size is limited only by available disk space.

For example:

```
patient("1234", "Labs", "hct", "31 May 2022 03:05:56 PM EDT") = 44;12
```

where a node or cell in the global array *patient*, indexed by the four strings shown, is assigned the value

The resulting global array *patient* node can be viewed either as a leaf node in a four level tree (where the array indices select the tree path) or as a cell in a four dimensional matrix.

Global arrays are derived from a C++ **class**. Instances must be declared in your C++ program as instances of class **global**.

For example, to create the global array named **gbl**, use the following:

```
global gbl("gbl");
```

The instance consists of two parts: the name of the global array object and the name of the global array on disk associated with this object. Normally, these are the same.

In the above example, they are both **gbl**. Note that the disk name of the global is enclosed in a parenthesized character string expression following the object name.

The value in the expression need not (but usually does) match the name of the object. The name given in the parenthesized character string is the disk name of the global array. The global array object is associated

¹² Blanks in the Mumps code have been inserted for clarity. In actual Mumps, they would be errors. Where appropriate, blanks are permitted in C++ code.

with the disk name when the object is created. When the program object is destroyed (for example, at program termination), the disk based global array persists.

Note: programs that use global arrays MUST close the array file system with the **GlobalClose;** command before exiting. Failure to do so may corrupt the file system.

Global objects may be created through declarations as shown above or dynamically:

```
global *gptr;  
gptr = new global ("gbl_name");  
(*gptr)("1","2","3") = "test";
```

which is equivalent to:

```
global g("gbl_name");  
g("1","2","3") = "test";
```

Each **global** declaration creates a global array as an object or instance of the **global** class. Each global array you use must be first declared as an object of the **global** class. Global names can be any valid C/C++ variable name.

A global array will typically have one or more subscripts as discussed below. These will be of type **mstring**, or a null terminated array of **char**. Subscripts of global arrays must evaluate to printable characters in the range of decimal 32 (space) to, but not including, decimal 126 (tilde ~).

Note:

- No data types other than **mstring**, or a null terminated array of **char** (i.e., **char ***) may be used as subscripts. Numeric data types (**int**, **short**, **long**, **float**, **double**, etc.) may not be used as global array subscripts.
- In any given global array reference, all the indices must all be of the same data type (**mstring** or **char ***)

mstring is a data type (class) whose behavior is similar to the basic typeless string data type used in Mumps.

Objects of **mstring** are stored internally as strings and may contain text, integers and floating point values.

Addition, multiplication, subtraction, division, modulo, and concatenation may be performed directly on **mstring** objects (see details below). Many of the following examples use **mstring** objects.

9.6 Global Arrays as C++ Trees and Matrices

9.6.1 Traditional Matrix

Global arrays may be viewed either as sparse multi-dimensional matrices or as tree structured hierarchies.

As matrices, data may be stored at fully subscripted matrix elements but also at other levels in the manner used by many programming languages. For example, given a three dimensional matrix *mat1*, you could initialize it as shown in Figure 38.

```
#include <mumpsc /libmpscpp.h>  
  
global mat1("mat1");  
  
int main() {  
    mstring i, j, k;
```

```

        for (i = 0; i < 100; i++)
            for (j = 0; j < 100; j++)
                for (k = 0; k < 100; k++) {
                    mat1(i, j, k) = 0;
                }

GlobalClose;
return 0;
}

```

Figure 38 Global Array as a Matrix

Alternatively, the above can be performed with **int** but the numeric indices must be converted to **mstring** before use. See Figure 39

```

#include <mumpsc /libmpscpp.h>

global mat1("mat1");

int main() {
    int i, j, k;
    for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
            for (k = 0; k < 100; k++) {
                mat1(mcv1(i), mcv1(j), mcv1(k)) = 0;
            }

GlobalClose;
return 0;
}

```

Figure 39 Global Array as Matrix with Numeric Subscripts

In this example, all the elements of a three dimensional matrix of 100 rows, 100 columns and 100 planes are initialized to zero. The function *mcvt()* converts from **int** to **mstring**.

In the view expressed by the code above, the matrix is a traditional three dimensional structure with data stored at each fully indexed position or node.

9.6.2 Alternative Matrix View

Unlike other programming languages, however, there are additional nodes of the matrix which could have been initialized as indicated by Figure 40.

```

#include <mumpsc /libmpscpp.h>

global mat1("mat1");

int main() {
    mstring i, j, k;
    for (i = 0; i < 100; i++) {
        mat1(i) = i;
        for (j = 0; j < 100; j++) {
            mat1(i, j)=j;
            for (k = 0; k < 100; k++) {
                mat1(i, j, k) = 0;
            }
        }
    }

    return 0;
}

```

In effect, this means that **mat1** can also be a single dimensional vector, a two dimensional matrix and a three dimensional matrix simultaneously.

Furthermore, not all elements of a matrix need exist. That is, the matrix can be sparse as shown in Figure 41.

```
#include <mumpsc/libmumpscpp.h>

global mat1("mat1");

int main() {
  mstring i, j, k;
  for (i = 0; i < 100; i = i + 10)
    for (j = 0; j < 100; j = j + 10) {
      for (k = 0; k < 100; k = k + 10) {
        mat2(i, j, k) = 0;
      }
    }
  return 0;
}
```

Figure 41 Global Array as Sparse Matrix

In the above, only index values 0, 10, 20, 30, 40, 50, 60, 70, 80, and 90 are used to create each of the dimensions of the array and only those elements of the matrix are created. The omitted elements do not exist.

For example, if you are running a drug protocol on a number of patients and are dosing with medications M1, M2, M3, ... on patients P1, P2, P3, ... and collecting observations on days D1, D2, D3, ... you could create a three dimensional matrix named *protocol* in which each plane consisted of the observations for each patient on each medication for a given day as shown in Figure 42.

D1						D2						D3						D4					
	M1	M2	M3	M4	M5		M1	M2	M3	M4	M5		M1	M2	M3	M4	M5		M1	M2	M3	M4	M5
P1						P1						P1						P1		X			
P2						P2						P2						P2					
P3						P3						P3						P3					

Figure 42 Tabular View of Tree

You could refer to patient P1, medication M2 on day D4 with the reference:

```
protocol("P1", "M2", "D4")="X";
```

9.6.3 Tree View

Alternatively, you can view the same data base as a tree structure with patient id at the root, followed by medication, followed by day of study as shown in Figure 43.

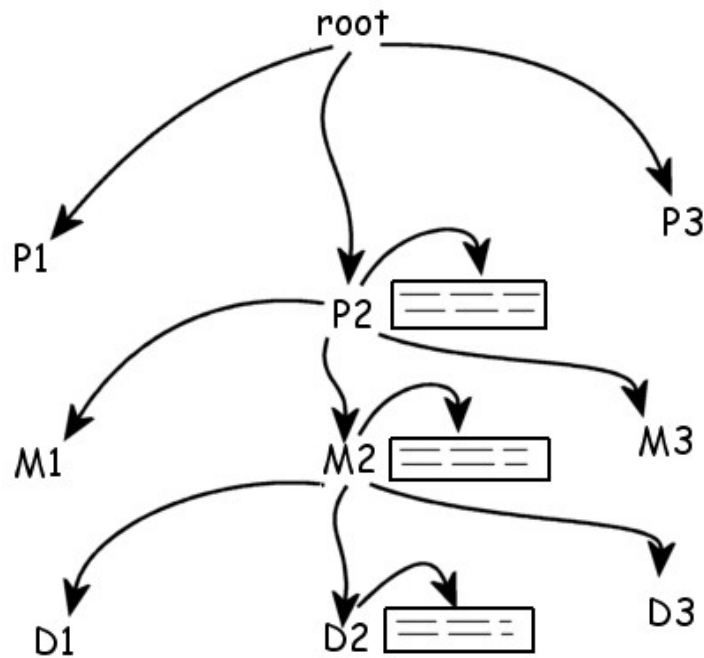


Figure 43 Global Array Tree

Note that at each node in the tree, a data box may appear containing information about the node. Addressing a node is accomplished by giving its path description such as:

```
protocol("P2", "M2", D2)
```

9.7 Accessing Global Arrays

Note: prior to exiting a program that accessed globals arrays, you should execute the *GlobalClose* macro to shut down the global array facility. This flushes the system buffers to disk and insures that the file system is properly closed. Failure to do this may result in data base errors. This appears in your program as:

```
GlobalClose;
```

You may assign global array elements to variables of type **mstring** using the assignment operator (=).

You may assign values of type **int**, **float**, **double**, **mstring**, **string** and **char *** to global array elements using the assignment operator (=).

When global array references are passed to function, no more than one instance of the same **global** object should be used in the argument list. Each **global** object maintains a private static string which contains the most recent value fetched from the data base. When a **global** object is passed to a function, its this string value is effectively passed. This means that, in a function reference where two references to the same **global** object are passed, even though they have differing indices, the value passed will be the value for the second instance of the **global**. This restriction only applies where there are two or more instances of the same **global**.

If you use a reference to a **global** without a parenthesized list following the name of the **global**, the reference will be to the most recent referenced **global**. Effectively, this is similar to the "naked indicator" from Mumps.

9.8 Global Array Indices

Internally, the indices of global arrays are always stored as character strings. If you initialize a global array with a loop, you must insure that the indices are represented as either values of type **mstring** or null

terminated arrays of type **char**. Indices to globals may be either **char*** or **mstring** but MUST all be of the same type (i.e. all **char *** or all **mstring**). For example:

```
mstring A, B, C;

for (A = 0; A < 1000; A++)
    for (B = 0; B < 1000; B++)
        for (C = 0; C < 1000; C++) {
            array1(A, B, C) = "0";
        }
```

The above initializes an array of 1 billion elements to zero.

9.9 Navigating Globals

There are several builtin functions used to navigate the globals. The two most important are the *Data()* function and the *Order()* function. The *Data()* function tells you if a node exists and if it has descendants and the *Order()* function gives you the next higher (or lower) index at a given level in the global array tree.

The *Data()* function returns an integer which indicates whether the global array node is defined:

1. 0 if the global array node is undefined;
2. 1 if it is defined and has no descendants;
3. 10 if it is defined but has no value stored at the node (but does have descendants);
4. 11 it is defined and has descendants.

A global is defined if data has been stored at it. A "10" is returned for a node at which nothing has been stored but the node has descendants. For example, assuming the global array has only the contents created in the example in Figure 44.

```
global array1("array1");

int result;

array1("1","11") = "foo"
array1("1","11","21") = "bar"

result = array1("1").Data() ;           // yields 10
result = array1("1","11").Data();       // yields 11
result = array1("1","11","21").Data();  // yields 1
```

Figure 44 Navigating Global Arrays - Data()

The other major navigation function is the *Order()* function. This gives you, for a given global array index, the next ascending or descending value for the last index. If the parameter to *Order()* is 1 or missing, the next ascending index is returned. If the parameter is -1, the next descending index is returned. To get the first (or last if the parameter is -1) value of an index, start with a null (empty) string. See Figure 45.

```
mstring x, null;
global array1("array1");

array1("100") = "a";           // initialize the array with three entries
array1("200") = "b";
array1("300") = "c";

null = "";
```

```

x = array1(null).Order();    // get the first value of the first index: 100
x = array1(x).Order();      // get the second value of the first index: 200
x = array1(x).Order();      // get the third value of the first index: 300
x = array1(x).Order();      // no more indices - returns empty string
x = array1(null).Order(-1);  // get the last value of the first index: 300
x = array1(x).Order(-1);    // get the second value of the first index: 200
x = array1(x).Order(-1);    // get the first value of the first index: 100
x = array1(x).Order(-1);    // no more indices - returns empty string

for ( x = array1(null).Order(); x != null; x = array1(x).Order())
    cout << endl;          // writes 100 200 300 on separate lines

for ( x = array1(null).Order(-1); x != null; x = array1(x).Order(-1))
    cout << endl;          // writes 300 200 100 on separate lines

for ( x = 10; x < 100; x = x + 10) array1("200" , x) = x;

for ( x = array1("200", null).Order(); x != null; x = array1("200", x).Order())
    cout << endl;          // writes 10 20 30 ... 90 on separate lines

```

Figure 45 Navigating Global Arrays - Order()

Each call to *Order()* gives the next value of the last index. The numeric parameter indicates if the direction is ascending (1) or descending (-1). If omitted, 1 is assumed. To get the first index, the empty string is supplied and the function returns the first index of the global array. For subsequent calls, it returns the next ascendant index value until there are no more indices. Then it returns the empty string.

In the following example, we build a global array vector from an input file consisting of keywords with one keyword per line, keep a count of each time the keyword is used, and, at the end, print an alphabetized list of the keywords followed by the number of times each occurs, do as shown in Figure 46.

```

#include <mumpsc/libmumpscpp.h>
global key("key");

int main() {

mstring word, null;
long i;

null = "";

while (1) {
    if ( ! word.ReadLine(cin)) break;
    if (key(word).Data())          // is word in vector?
        key(word)++;              // yes, increment count
    else key(word) = 1;            // not in vector - add
}

word = null;

while ((word = key(word).Order(1)) != null) // next word

cout << word << " " << key(word) << endl; // print word and count

return EXIT_SUCCESS;
}

```

Figure 46 Global Array Navigation Example

In the above, each line is read into the variable *word* until the end of file is reached. Each word is tested with the *Data()* function of the global array to determine if *word* exists in the *key* vector. The *Data()* returns zero if the element does not exist, non-zero if it does. In the case where the word is in the *key* global array vector, the value stored in the vector for the word is extracted into the variable *i*, incremented and stored back into the vector. If the *word* does not exist in the vector, it is added and its initial count is set to one.

When all the words have been read and stored into the vector, the program sequences through the word entries and prints the words and the total number of times each one was present in the input file. Since global arrays are stored in ascending key order, the display of words will be alphabetic.

Similarly, given a global array of patient lab data organized hierarchically first by patient id, then by lab test, then by date, we can print a table of patient id's, labs, dates and results as shown in Figure 47.

```
#include <mumpsc/libmpscpp.h>

global Labs("labs");

int main() {

mstring null, ptid, lab_test, date, rslt;

null = "";

// create dummy example data base

Labs("1000", "hct", "July 12, 2003") = "45";
Labs("1000", "hct", "July 13, 2003") = "46";
Labs("1000", "hct", "July 14, 2003") = "47";
Labs("1000", "hct", "July 15, 2003") = "48";
Labs("1000", "hgb", "July 12, 2003") = "15";
Labs("1000", "hgb", "July 15, 2003") = "14";
Labs("1001", "hct", "July 12, 2003") = "35";
Labs("1001", "hct", "July 13, 2003") = "36";
Labs("1001", "hct", "July 14, 2003") = "37";
Labs("1001", "hct", "July 15, 2003") = "38";
Labs("1001", "hgb", "July 13, 2003") = "15";
Labs("1001", "hgb", "July 14, 2003") = "15";
Labs("1002", "hct", "Sept 12, 2003") = "35";
Labs("1002", "hct", "Sept 13, 2003") = "36";
Labs("1002", "hct", "Sept 14, 2003") = "37";
Labs("1002", "hct", "Sept 15, 2003") = "38";
Labs("1002", "hgb", "Sept 13, 2003") = "15";
Labs("1002", "hgb", "Sept 14, 2003") = "15";

ptid = null;

while ( (ptid = Labs(ptid).Order(1)) != null) {
    lab_test = null;
    while ( (lab_test = Labs(ptid,lab_test).Order(1)) != null) {
        date = null;
        while ( (date = Labs(ptid,lab_test,date).Order(1)) != null) {
            cout << ptid << " " << lab_test << " " << date;
            cout << " " << Labs(ptid,lab_test,date) << endl;
        }
    }
}

GlobalClose;
```

```

    return 1;
}

```

Output

```

1000 hct July 12, 2003 45
1000 hct July 13, 2003 46
1000 hct July 14, 2003 47
1000 hct July 15, 2003 48
1000 hgb July 12, 2003 15
1000 hgb July 15, 2003 14
1001 hct July 12, 2003 35
1001 hct July 13, 2003 36
1001 hct July 14, 2003 37
1001 hct July 15, 2003 38
1001 hgb July 13, 2003 15
1001 hgb July 14, 2003 15

```

Figure 47 Hierarchical Global Array Example

The example in Figure 47 begins with an empty string for patient id *ptid*. This is used at the outer loop level to cycle through all the patient ids. At the first nexted loop, the program cycles through all the lab test names (*lab_test*) then at the innermost level, it cycles through all the dates (*date*). The resulting table is of the form:

9.10 Locking the Data Base

There are several functions for locking portions of the data base. Following legacy convention, a lock does not prevent access to an element but merely flags the element as locked. Locking views a global array as a tree structure. If an element is locked, its descendants are locked. An attempt to lock a locked element of an element that has a locked parent or a locked descendant will fail. The primary locking functions are *\$lock()*, *Lock()* and *UnLock()*:

```

if ($lock(gbl(a, b, c)) cout << "locked" << endl;
if (gbl(a, b, c).Lock()) cout << "locked" << endl;
gbl(a, b, c).UnLock();

```

The *\$lock()* and *Lock()* functions test to see if the node can be locked and locks it if possible. It returns *true* (1) if successful and false (0) otherwise (*\$test* is set accordingly). A node can be locked if it itself is not locked, if it has no descendants that are locked and if it is not the descendant of a locked node. The *UnLock()* function releases a lock on a node.

Additionally, there are functions to release all locks for the current process and all locks for all processes:

```

CleanLocks();      // release all locks for this process only
CleanAllLocks();   // release all locks for all processes

```

9.11 Invoking the Mumps Interpreter

The full facilities of the Mumps interpreter can be invoked from C++ programs. The interpreter reads, parses and executes commands presented to it at run time. It may also read and execute text files containing Mumps programs. The interpreter is invoked by means of the *Xecute()* macro and *xecute()* functions:

```

int Xecute("command")
int xecute(mstring command)
int xecute(string command)
int xecute(char * command)

```

These functions and macro invoke the Mumps interpreter and execute the text replacing "command". They return 1 if successful, 0 otherwise. With *Xecute()*, if the mumps command contains quotes or other special symbols, they will be automatically prefixed with backslashes (e.g., quote becomes `\`).

```
Xecute("set i='test')");  
Xecute("fors i=$order(^a(i)) quit:i=''' set sum=sum+^a(i)");
```

Details on the Mumps Language are contained in the file *compiler.html* in the *mumpsc/doc* subdirectory of the Mumps Compiler distribution. See also: `mtring::Eval()` for expression interpretation.

9.12 Miscellaneous Functions and System Variables

9.12.1 GTK / Glade functions

The following functions may be used with the GTK / Glade programming facility:

9.12.1.1 void mdh_tree_level_add(GtkTreeStore *tree, int depth, char * col1 [, char *col2 ...]);

Add the value in *col1* to the tree at level *depth* and populate the remaining columns of this row with *col2*, *col3*, ... to a limit of five columns.

9.12.1.2 int mdh_dialog_new_with_buttons(GtkWindow *win, char * text)

Open a modal popup dialog box with the options "Yes" and "No". The contents of *text* will be displayed. Returns 0 if *no* is clicked and 1 if *yes* is clicked. The value -1 is returned if the box is dismissed without selection.

9.12.1.3 int mdh_toggle_button_get_active(GtkToggleButton *b)

Returns 1 if the button is active; 0 otherwise.

9.12.1.4 char * mdh_entry_get_text(GtkEntry *e, char * txt)

Returns the text contents of the specified entry box. The return pointer points to the string pointed to by *txt*. The user is responsible for providing a character array pointed to by *txt* large enough to contain the text retrieved.

9.12.1.5 void mdh_toggle_button_set_active(GtkToggleButton *b, int v)

The named toggle button will be set to active if the value of *v* is non-zero; inactive otherwise. Triggers a toggle signal.

9.12.1.6 void mdh_entry_set_text(GtkEntry *e, char * txt)

Sets the contents of the named entry box. Triggers a entry changed signal.

9.12.1.7 void mdh_text_buffer_set_text(GtkTextBuffer *t, char * txt)

Sets the contents of the named text buffer.

9.12.1.8 void mdh_label_set_text(GtkLabel *l, char * txt)

Sets the contents of the named label.

9.12.1.9 void mdh_widget_hide(GtkWidget *w)

Hides the named widget.

9.12.1.10 void mdh_widget_show(GtkWidget *w)

Displays the named widget.

9.12.1.11 `char * mdh_tree_selection_get_selected(GtkTreeSelection *t, int col, char *txt)`

Returns the value in column 1 of the named tree.

9.12.1.12 `void mdh_tree_store_clear(GtkTreeStore *t)`

Clears the named tree store.

9.12.1.13 `double mdh_spin_button_get_value(GtkSpinButton *s)`

Returns the value in the named spin button.

9.12.1.14 `void mdh_spin_button_set_value(GtkSpinButton *s, double v)`

Sets the value of the named spin button.

9.12.1.15 `$z~mdh~toggle~button~get~active(ToggleButtonReference)`

Returns 0 if the button is inactive, 1 if active

9.12.2 Boyer-Moore-Gosper Functions

`int bmg_fullsearch(mstring search_string, mstring buffer_base)`

Returns the number of non-overlapping instances of "search_string" in "buffer_base". See Figure 48.

```
#include <mumpsc/libmpscpp.h>

int main() {

    mstring a = "now is the time for all good men to come to the aid of the
    party";
    mstring b = "to";
    cout << bmg_fullsearch(b, a) << endl;
    return EXIT_SUCCESS;
}
```

yields:

2

Figure 48 Boyer-Moore Example

These functions are publically available from:

`ftp://ftp.uu.net/usenet/comp.sources.unix/volume5/bmgsubs.Z`

and are believed to be contributed source and are unrestricted with respect to use and redistribution, and, that most, if not all, the code was written by employee(s) of the United States and thus in the public domain. The distribution contains, in part, the following notes:

Here are routines to perform fast string searches using the Boyer-Moore-Gosper algorithm; they can be used in any Unix program (and should be portable to non-Unix systems). You can search either a file or a buffer in memory.

The code is mostly due to James A. Woods (jaw@ames-aurora.arpa) although I have modified it heavily, so all bugs are my fault. The original code is from his sped-up version of egrep, recently posted on mod.sources and available via anonymous FTP from ames-aurora.arpa as pub/egrep.one and pub/egrep.two. That code handles regular expressions; mine does not.

These have only been tested on 4.2BSD Vax systems.

-Jeff Mogul

mogul@navajo.stanford.edu
decwrl!glacier!navajo!mogul
BMGSUBS(3L)

BMGSUBS(3L)

NAME

(bmgsbubs) bmg_setup, bmg_search, bmg_fsearch - Boyer-Moore-Gosper string search routines

SYNOPSIS

```
bmg_setup(search_string, case_fold_flag)
char *search_string;
int case_fold_flag;

bmg_fsearch(file_des, action_func)
int file_des;
int (*action_func)();

bmg_search(buffer_base, buffer_length, action_func)
char *buffer_base;
int buffer_length;
int (*action_func)();
```

DESCRIPTION

These routines perform fast searches for strings, using the Boyer-Moore-Gosper algorithm. No meta-characters (such as '*' or '.') are interpreted, and the search string cannot contain newlines.

Bmg_setup must be called as the first step in performing a search. The search_string parameter is the string to be searched for. Case_fold_flag should be false (zero) if characters should match exactly, and true (non-zero) if case should be ignored when checking for matches.

Once a search string has been specified using bmg_setup, one or more searches for that string may be performed.

Bmg_fsearch searches a file, open for reading on file descriptor file_des (this is not a stdio file.) For each line that contains the search string, bmg_fsearch will call the action_func function specified by the caller as action_func(matching_line, byte_offset). The matching_line parameter is a (char *) pointer to a temporary copy of the line; byte_offset is the offset from the beginning of the file to the first occurrence of the search string in that line. Action_func should return true (non-zero) if the search should continue, or false (zero) if the search should terminate at this point.

Bmg_search is like bmg_fsearch, except that instead of searching a file, it searches the buffer pointed to by buffer_base; buffer_length specifies the number of bytes in the buffer. The byte_offset parameter to action_func gives the offset from the beginning of the buffer.

If the user merely wants the matching lines printed on the standard output, the action_func parameter to bmg_fsearch or bmg_search can be NULL.

AUTHOR

Jeffrey Mogul (Stanford University), based on code written by James A. Woods (NASA Ames)

BUGS

Might be nice to have a version of this that handles regular expressions.

There are large, but finite, limits on the length of both pattern strings and text lines. When these limits are exceeded, all bets are off.

The string pointer passed to `action_func` points to a temporary copy of the matching line, and must be copied elsewhere before `action_func` returns.

`Bmg_search` does not permanently modify the buffer in any way, but during its execution (and therefore when `action_func` is called), the last byte of the buffer may be temporarily changed.

The Boyer-Moore algorithm cannot find lines that do not contain a given pattern (like `"grep -v"`) or count lines (`"grep -n"`). Although it is fast even for short search strings, it gets faster as the search string length increases.

16 May 1986

BMGSUBS(3L)

9.12.3 `cvt()`

```
char *cvt(long i)
char *cvt(double i)
char *cvt(float i)
char *cvt(int i)
```

These functions return a null terminated varying length character string containing in printable version of the argument. The functions contain short static character arrays and, consequently, are not threadsafe.

9.12.4 `xecute()` and `command()`

`command()` is a macro that takes a quoted string constant argument. The macro surrounds the string with an extra set of quotes and processes any embedded quotes to backslash-quote. It then invokes a function (`__command__()`) which strips the extra surrounding quotes. The net effect of this is that you can pass a quoted string containing quotes without the need for "leaning toothpick" notation. Example:

```
xecute(command("for i=1:1:10 "test ",i,!"));
strcpy(target, command("for i=1:1:10 write "test ",i,!"))
```

The argument must be a character string constant.

9.12.5 `ErrorMessage()`

```
void ErrorMessage(char * message, int line_number)
```

This function (written in C and part of the underlying legacy library) will print an error message, close the global array files and terminate the program. The integer "line_number" will be printed with the message. The pre-processor predefined macro `"__LINE__"` can be used here. Example:

```
ErrorMessage("Cannot locate patient", __LINE__);
```

9.12.5.1 Error Exceptions

The toolkit generates (throws) exceptions for certain conditions. For example, when you access global arrays with the toolkit, the accesses may result in the thrown error exceptions:

1. *ConversionException.*
2. *GlobalNotFoundException*
3. *MumpsSymbolTableException.*
4. *NumericRangeException.*

The first can occur in any context that attempts to retrieve data from a global array where none exists. The second occurs if you attempt to convert the contents of a global to a numeric type where the contents of the global are not valid data for the conversion.

If uncaught, both exceptions will result in program termination.

The following are the exceptions thrown by the toolkit:

1. `ConversionException()` - usually occurs when you attempt to store a value from a global array into a numeric variable but the string in the global is not a valid number.
2. `GlobalNotFoundException()` - thrown by an attempt to reference non-existent global array data.
3. `MumpsSymbolTableException()` - thrown by an attempt to fetch the value of a non-existent variable from the Mumps runtime symbol table.
4. `NumericRangeException()` - thrown by attempts to divide by zero or using arguments with values less than or equal to zero to log functions.

See Figure 49.

```
#include <mumpsc/libmumpscpp.h>

global a("a");

int main() {
    long i;
    a().Kill();
    mstring A;
    a("1") = "now is the time";
    try {
        i = a("1");
    }

    catch ( ConversionException ce) {
        cout << ce.what() << endl;
    }
    try {
        i = a("22");
    }
    catch (GlobalNotFoundException nf) {
        cout << nf.what() << endl;
    }

    try {
        A=SymGet("abc");
    }
    catch (MumpsSymbolTableException st) {
        cout << nf.what() << endl;
    }

    return 0;
}
```

Figure 49 Exceptions Examples

9.12.6 HitRatio()

double HitRatio(void)

Calculates the native global array processor cache hit ratio since the beginning of the program or the last call to *HitRatio()* The native global array file processor, as opposed to the Berkeley Data Base, keeps track of how many file I/O requests are satisfied from data already in the file system's cache. This function gives the percentage of cache hits. It only works with the native global array processor.

9.12.7 Hashing functions

char * hash(**char *** str)
long lhash(**char *** str)

hash() returns either a null terminated character string up to 10 characters in length containing a numeric hash code of the string passed as an argument. The argument may be up to *STR_MAX* characters in length. *lhash()* returns an **unsigned long** value of the hash value.

9.12.8 Dump Global Array Database

```
void Dump(char * filename)
void Dump(mstring filename)
void Dump(string filename)

void Restore(char * filename)
void Restore(mstring filename)
void Restore(string filename)
```

The global array data base is dumped (written in its entirety) to filename or read and restored from filename (null terminated array of chars). Both operations must not be done from the same program.

9.12.9 Stream Output Function ostream

```
friend ostream & operator << (ostream&, global)
```

A global array may participate in stream output. For example:

```
gbl("A", "B", "C") << "test test test";
cout << gbl("A", "B", "C") << endl;
```

The above will print "test test test" (without quotes) followed by the newline character. Alternatively:

```
cout << gbl("A", "B", "C").Get() << endl;
```

will do the same thing (the Get() function returns "char *").

9.12.10 Smith-Waterman Alignment Function

```
int sw(mstring s, mstring t, [int show_aligns=0, int show_mat=0, int gap=-1, int
mismatch=-1, int match=2])

int sw(string s, string t, [int show_aligns=0, int show_mat=0, int gap=-1, int
mismatch=-1, int match=2])

int sw(char *s, char *t, [int show_aligns=0, int show_mat=0, int gap=-1, int
mismatch=-1, int match=2])
```

Calculate the Smith-Waterman Alignment between strings "s" and "t". Result returned is the highest alignment score achieved. Parameters other than the first two are optional. If only some of the optional parameters are supplied, only trailing parameters may be omitted, as per C/C++ rules.

If you compare very long strings (>100,000 character), you may exceed stack space. This can be increased under Linux with the command:

```
ulimit -s unlimited
```

(Other options are ulimit -a and ulimit -aH to show limits).

If "show_aligns" is zero, no printout of alternative alignments is produced (default). If "show_aligns" is not zero, a summary of the alternative alignments will be printed. If "show_mat" is zero, intermediate matrices will not be printed (default). The gap and mismatch penalties are -1 and the match reward is +2. The parameters "gap", "mismatch" and "match" are the gap and mismatch penalties (negative integers) and the match reward (a positive integer). These values default to -1, -1 and 2 respectively. If insufficient memory is available, a segmentation violation will be raised.]

The first character of each sequence string MUST be blank.

See Figure 50.

```
#include <mumpsc/libmpscpp.h>

int main() {
    char s[] = " now is the time for all good men to come to the aid of the party";
    char t[] = " time for good men";

    int i = sw(s, t, 1, 0, -1, -1, 3);

    return 0;
}

results in:

S-W Alignments for:
64 now is the time for all good men to come to the aid of the party
22 time for good men

    29 men 32
      :::
    19 men 22
score=12

    29 - men 32
      :::
    18 men 22
score=11

    23 l good-- men 32
      :::: ::::
    11 good men 22
score=24

    22 ll good-- men 32
      :::: ::::
    11 - good men 22
score=23

    16 for all good-- men 32
      :::: :::: ::::
     6 for -- good men 22
score=37

    12 time- for all good-- men 32
      :::: :::: :::: ::::
     1 time for -- good men 22
score=48
```

Figure 50 Smith-Waterman Example

9.12.11 Stop list functions: StopINIT(), StopLookup()

```
void StopInit(mstring file)
void StopInit(string file)
void StopInit(char * file)

int StopLookup(mstring word)
int StopLookup(string word)
int StopLookup(char * word)
```

StopInit() reads the sorted file "file" of stoplist words into the stoplist container (one word per line). *StopLookup()* returns 0 if "word" is not found and 1 if "word" is found in the stoplist.

9.12.12 int \$test

Returns integer 1 or 0 indicating the success or failure of certain previous commands. Some, but not all, commands set "\$test".

9.12.13 Xecute()

```
int Xecute(char * command)
int Xecute(mstring command)
int Xecute(string command)
int Xecute(char * command)
```

These functions invoke the Mumps interpreter which executes *command*. Returns 1 if successful, 0 otherwise.

The macro *Xecute()* is a special case. It is used with character string constants. It will pre-process a character string constant *command* and insert the backslash escape character prior to any embedded quotes thus permitting more normal appearing text (see similar macro *command()*).

Examples:

```
mstring c;
Xecute("for i=$Order(^a(i)) q:i="" s sum=sum+^a(i)");

c = "for i=1:1:10 write i,!";
xecute(c);

c = command("for i=1:1:10 write \"ans=\",i,!");
xecute(c);
```

9.12.14 Zseek(), Ztell()

```
bool Zseek(FILE *file, offset)
bool Ztell(FILE *file, offset)
```

These functions are used in connection with direct access files opened with FILE pointers (see: *fopen()*). They are compatible with 64 bit file pointer systems. *Zseek()* positions the file designated by *file* to the offset specified in *offset*, a positive integer contained in a variable of type **mstring** or **global**.

Ztell() places the current file offset in the file designated by *file* into the **mstring** or global variable represented by *offset*.

Both functions return 'true' if successful. Ordinarily, file offsets will be obtained by *Ztell()* and these will be stored in a data base. These values will be subsequently used in connection with *Zseek()* to reposition the file to the point it was at when the *Ztell()* was performed. After re-positioning, the next input or output operation on the file will occur at the point designated by *offset*. All offsets are relative to the start of the file.

10 Class mstring

The **mstring** class provides Mumps-like strings that can be used to in C++ programs. They treat variable in a manner similar to that of native Mumps strings.

mstring objects are based on C++ **string** strings on which arithmetic and other operations may be performed. The **mstring** includes overloads for many operators as well provides many Mumps-like functions.

A complete list of the overloaded **mstring** operators is in Section

10.1 mstring Functions

10.1.1 Ascii Function

```
int mstring::Ascii()  
int mstring::Ascii(int start)  
int mstring::Ascii(int start)
```

Returns the numeric value of an ASCII character. If no "start" is specified, the numeric values of the first character of invoking mstring is used. If "start" is specified, the numeric value of "start"th character of nvoking is chosen. If the empty string is given, -1 is returned.

```
mstring a;  
a="ABC";  
a.Ascii() yields 65  
a.Ascii(1) yields 65  
a.Ascii(2) yields 66
```

10.1.2 begins Function

```
int mstring::begins(mstring pattern)
```

Returns an integer which is the starting point in the string of pattern or -1 if the pattern is not found. Throws: PatternException if the pattern is in error.

10.1.3 c_str Function

```
char * mstring::c_str()
```

Returns a pointer to a null terminated char array containing the contents of the invoking mstring object.

10.1.4 decorate Function

```
int mstring::decorate(mstring pattern, mstring prefix, mstring suffix)
```

Attempts to locates *pattern* in the invoking mstring and inserts *prefix* immediately to the left of the string that matched the pattern and inserts *suffix* immediately to the right of the found pattern. Returns 1 if the pattern was found and the insertions were made, -1 if the pattern was not found, and less than -1 for other errors (see PCRE documentation concerning pcre_exec() return codes). Throws: PatternException().

10.1.5 EncodeHTML Function

```
char * mstring EncodeHTML(char * arg)  
mstring EncodeHTML(mstring arg)
```

Encodes the argument string according to HTML rules and returns the result. Alphabetics and numbers are unchanged. Blanks become plus signs and all other characters replaced by "%xx" where "xx" is the

hexadecimal value of the character in the ASCII collating sequence. The function is used mainly in connection with parameters passed with URL's which may not contain blanks or special characters. the code in *cgi.h* is used to decode these strings. Example:

```
#include <mumps /libmumps.h>
int main() {
    char x[]="now is =()$.& the time";
    cout << EncodeHTML(x) << endl;
    return EXIT_SUCCESS;
}
```

Yields

```
now+is+%3D%28%29%24%2E%26+the+time
```

10.1.6 ends Function

```
int mstring::ends(mstring pattern)
```

Returns an integer giving the character position (relative to zero) immediately following the string that matched pattern. Returns -1 if the string did not match. Throws: PatternException.

10.1.7 Eval Function

```
mstring mstring::Eval()
```

Evaluates the mumps expression of the invoking mstring object and returns the result in an mstring. If an error occurs, an InterpreterException is thrown. The invoking mstring object may contain a valid mumps expression involving calling program mstring variables.

10.1.8 Extract Function

```
mstring mstring::Extract(int=1, int=-1)
```

Returns an mstring containing a substring of the first argument. The substring begins at the position noted by the second operand. If the third operand is omitted, the substring consists only of the "start" character of invoking source string. If the third argument is present, the substring begins at position "start" and ends at position "end". If no argument is given, the function returns the first character of the string. If "end" specifies a position beyond the end of source string, the substring ends at the end of source string. String position counting begins at one (not zero).

10.1.9 Find Function

```
int mstring::Find(const char *, int=1)
int mstring::Find(mstring, int=1)
```

Find() searches the first argument for an occurrence of the second argument. If one is found, the value returned is one greater than the end position of the second argument in the first argument. If "start" is specified, the search begins at position "start" in argument 1. If the second argument is not found, the value returned is 0. String position counting begins at position one.

```
mstring x;
x="ABCDEF";
x.Extract(2) yields "B"
x.Extract(3,5) yields "CDE"
```

10.1.10 Horolog Function

```
mstring Horolog()
```

Returns an **mstring** of the form "x,y" where x is the number of days since December 31, 1984 and y is the number of seconds since midnight.

10.1.11 Justify Function

```
mstring mstring::Justify(int,int=-1)
```

Justify() right justifies the invoking mstring in an mstring field whose length is given by the first argument. If the second argument is present and a positive integer, the invoking mstring is right justified in a field whose length is given by the first argument with "precision" decimal places. The two argument form imposes a numeric interpretation upon the first argument.

```
x="39";  
x.Justify(3) yields " 39"  
  
x="TEST";  
x.Justify(7) yields " TEST"  
  
x="39";  
x.Justify(4,1) yields "39.0"
```

10.1.12 Length Function

```
int mstring::Length()  
int mstring::Length(mstring pattern_string)  
int mstring::Length(char * pattern_string)
```

The function returns the string length of the invoking mstring.

10.1.13 mcvf Function

```
mstring mcvf(arg)
```

Converts the arg to **mstring**. Arg may be int, char *, float long or double.

10.1.14 Pattern Function

```
int mstring::Pattern(mstring &)  
int mstring::Pattern(const char *)
```

Evaluates the invoking source string according to the pattern_string and returns 0 (does not match) or 1 (does match). *Pattern_string* rules are as shown below but you must remember to place a backslash before quotes in the pattern string (as per usual C++ rules). The pattern match function is used to determine if a string conforms to a certain pattern. Pattern match operations are converted to Perl Compatible Regular Expressions and are executed by functions in the PCRE library which must be present. You may access the PCRE directly, using Perl expression format with the "*perl_pm(string, pattern, 1, svPtr)*" function discussed in Appendix D. The basic Mumps pattern codes are shown in Figure 51.

The Mumps pattern codes are:

- A for the entire upper and lower case alphabet.
- C for the 33 control characters.
- E for any of the 128 ASCII characters.
- L for the 26 lower case letters.
- N for the numerics
- P for the 33 punctuation characters.
- U for the 26 upper case characters.
- A literal string.

Figure 51 Mumps Pattern Codes

A pattern code is made up of one or more of the those shown in Figure 51, each preceded by a count specifier. The count specifier indicates how many of the named item must be present. Alternatively, an indefinite specifier - a decimal point - may be used to indicate any count (including zero). For example:

```
mstring A;  
A="123-45-6789";  
if (A.Pattern(command("3N1"- "2N1"- "4N"))) cout << "OK" << endl;  
A="JONES, J. L.";  
if (A.Pattern(command(".A1", ".A") )) cout << "OK" << endl;
```

Full pattern matching syntax, including support for alternation, are supported as described in Appendix D of the Compiler manual. The macro "command()" will handle the required backslash escape characters required before quote marks.

10.1.15 Perl Function

```
int Perl(mstring string, mstring regex)  
int Perl(mstring string, char * regex)
```

The regular expression in the null terminated character array or **mstring** given by *regex* is applied to the **mstring** *string*. If the pattern match succeeds, true (1) is returned, false (0) otherwise and *\$test* is set accordingly. This macro also sets variables in the run-time symbol table. See *SymGet()* and *SymPut()* for details on accessing the symbol table. See Appendix D for examples of using this function.

10.1.16 Piece Function

```
mstring mstring::Piece(const char *, int, int=-1)  
mstring mstring::Piece(mstring &, int, int=-1)
```

The *Piece()* function returns a substring of the invoking mstring delimited by the instances of the first argument. The substring returned in the two argument case is that substring of the invoking mstring that lies between the "start" minus one and "start" occurrence of the first argument. In the three argument form, the string returned is that substring of the invoking mstring delimited by the "start" minus one instance of the first argument and the end'th instance of the first argument. If only two arguments are given, end is assumed to be start. For example:

```
x="aaa.bbb.ccc.eee.fff";  
cout << x.Piece(".",1) << endl; // writes aaa  
cout << x.Piece(".",2) << endl; // writes bbb  
cout << x.Piece(".",5) << endl; // writes fff  
cout << x.Piece(".",4,5) << endl; // writes eee.fff
```

Global arrays may be used in any argument position but only one instance of the same global may appear (see note in [Accessing global arrays](#)) section.

10.1.17 ReadLine Function

```
bool mstring::ReadLine(FILE *)  
bool mstring::ReadLine(istream &)
```

The next line from the file designated by "unit" is read into the invoking object of mstring. Carriage-returns and line-feeds are removed. The maximum length line that can be read is STR_MAX-1. Returns 'true' if the operation succeeded, 'false' otherwise or if end of file.

10.1.18 replace Function

int mstring::replace(mstring pattern, mstring replacement)

Replaces the string matching pattern with replacement. Returns 1 if successful, 0 if there was no match and less than -1 on error (See PCRE documentation for `pcre_exec()`). Throws: `PatternException`.

10.1.19 ScanAlnum Function

mstring ScanAlnum(FILE *, int min=3, int max=25)

mstring ScanAlnum(istream, int min=3, int max=25)

Returns the next token from the input file with all punctuation removed. Returns empty string on end of file. If min and/or max are provided, only words whose length are less than min and greater than max are discarded. The default values for these parameters are 3 and 25, respectively. Use `stdin` for file to scan standard input.

10.1.20 shred Function

mstring Shred(mstring str, int size)

The `Shred()` function shreds the input string *str* into fragments of length *size* upon successive calls. The function returns a string of length zero when there are no more fragments of length *size* remaining (thus, short fragments at the end of a string are not returned). `Shred()` copies the input string to an internal buffer upon the first call. Subsequent calls retrieve from this buffer. When the buffer is consumed, the function will copy the contents of the next string submitted to the buffer. Figure 52 contains an example.

```
#include <mumpsc/libmpscpp.h>

int main() {

    char x[] = "abcdefghijklmnopqrstuvwxy";
    char *p;

    while(1) {
        p = Shred(x, 5);
        if (strlen(p) == 0) break;
        cout << p << endl;
    }
    return 0;
}

yields:

abcde
fghij
klmno
pqrst
uvwxy
```

Figure 52 Shred Function

10.1.21 ShredQuery Function

mstring ShredQuery(mstring str, int size)

The `ShredQuery()` function shreds *size* shifted copies of the input string *str* into fragments of length *size* upon successive calls. That is, the function first returns all the *size* fragments of the string in the same manner as `Shred()`. However, it then shifts the starting point of the input string to the right by one and

returns all the *size* length fragments relative to the shifted starting point. It repeats this process a total of *size* times.

The function returns a string of length zero when there are no more fragments of length *size* remaining (thus, short fragments at the end of a string are not returned). *ShredQuery()* initially copies the input string to an internal buffer upon the first call. Subsequent calls retrieve from this buffer. When the buffer is consumed, the function will copy the contents of the next string submitted to the buffer. See Figure 53.

```
#include <mumpsc/libmumpscpp.h>

int main() {

    char x[] = "abcdefghijklmnopqrstuvwxy";
    char *p;

    while(1) {
        p = ShredQuery(x, 5);
        if (strlen(p) == 0) break;
        cout << p << endl;
    }
    return 0;
}
```

Yields:

abcde
fghij
klmno
pqrst
uvwxy

bcdef
ghijk
lmnop
qrstu

cdefg
hijkl
mnopq
rstuv

defgh
ijklm
nopqr
stuvw

efghi
jklmn
opqrs
tuvwx

Figure 53 ShredQuery

10.1.22 Stem Function

mstring stem(**mstring** & word)

Returns the original word or the English linguistic root stem of the word, if one can be found.

10.1.23 Synonym Functions: SymInit(), SYN()

```
int SymInit(mstring filename)
int SymInit(string filename)
int SymInit(char * filename)

mstring SYN(mstring word)
string SYN(string word)
char * SYN(char * word)
```

SymInit() opens and reads a synonym file and returns the number of lines read. The maximum number of synonyms permitted is determined by "SYNMAX" in *libmpscpp.h* (default is 20,000). Each line of the synonym file consists of multiple words, in lower case, separated from one another by a single blank. The first word is the root alias and the remaining words are alternative synonyms. The function *SYN()* looks up a word. If the word is an alternative synonym, the root alias is returned. If not, the original word is returned.

10.1.24 SymGet SymPut Functions

```
mstring SymGet(mstring name)
mstring SymGet(char * name)
mstring SymGet(global name)
mstring SymPut(name, value)
```

These functions retrieve and store values from/to the run-time symbol table. In all, *name* is a string containing the name of the variable and *value* is the value to be stored. The *SymPut()* functions return true if successful. A *MumpsSymbolTableException* exception is raised if *SymGet()* fails. If *SymPut()* fails, the program terminates (out of memory). For *SymPut()*, 'name' and 'value' may be any combination of **mstring**, **global** or null terminated **char**.

10.1.25 s_str Function

```
string mstring::s_str()
```

Returns a **string** copy of the contents of the invoking **mstring** object.

10.1.26 Translate

```
mstring global::Translate(mstring)
mstring global::Translate(mstring, mstring)
```

If only one **mstring** argument is given, characters appearing in the argument **mstring** are removed from the invoking **global**.

If two argument **mstrings** appear and the first and second argument **mstring** are of the same length, characters from the invoking **global** that appear in the first argument **mstring** are replaced by their counterparts from the second argument **mstring**.

If the first argument **mstring** is longer than the second argument **mstring**, the characters from the first argument **mstring** which have no counterpart in the second argument **mstring** are removed.

A "counterpart" is a character equally offset in the second argument **mstring** to the character in the first argument **mstring**.

10.1.27 Token Function

```
mstring Token()
mstring TokenInit(mstring)
```

Token() returns the next word token from the input string. Initially a line of text is passed to *TokenInit()*. For each subsequent call to *Token()*, the next lexical token from the original string is returned. Upper case letters are converted to lower case letters. When there are no more words, the empty string is returned. After the the empty string is returned (or when initially called), the function will accept and store a new line of text.

10.1.28 Translate Function

```
mstring mstring::Translate(mstring)
mstring mstring::Translate(mstring, mstring)
```

If only one mstring argument is given, characters appearing in the argument mstring are removed from the invoking mstring.

If two argument mstrings appear and the first and second argument mstring are of the same length, characters from the invoking mstring that appear in the first argument mstring are replaced by their counterparts from the second argument mstring.

If the first argument mstring is longer than the second argument mstring, the characters from the first argument mstring which have no counterpart in the second argument mstring are removed.

A "counterpart" is a character equally offset in the second argument mstring to the character in the first argument mstring.

10.1.29 Basic mstring Examples

Figure 54 gives some examples of the data type **mstring**. In the Mumps language there is one basic data type: string. All operations, including arithmetic calculations result in string values.

The **mstring** data type imitates the string data type in Mumps. It can be used as a traditional string or in mathematics.

In Figure 54 we see the **mstring** variables *a*, *b*, and *c* being used as traditional strings with the MDH concatenation operator (*||*) to form the string *hello world*. The **mstring** variable *a* is then used as a numeric counter to print zero through nine. It is then used to as a numeric index to a global array (*x(a)*) and finally in several expressions accessing the global array.

```
#include <mumpsc/libmpscpp.h>

global x("x");

int main() {
    mstring a, b, c;

    a = "hello ";
    b = "world";

    cout << (a || b) << endl;           // concatenation
                                        // prints "hello world"
    for (a = 0; a < 10; a++)
        cout << a << endl;             // prints 0 thru 9

    for (a = 0; a < 10; a++)
        x(a) = a;                       // sets global array elements

    a = "";

    while (1) {
```

```

        a = x(a).Order(1);
        if (a == "") break;
        cout << a << endl;          // prints 0 thru 9
    }

    cout << x(a).Data() << endl;    // prints 1

    c = "123 elm street";
    c = c + 1;
    cout << c << endl;            // prints 124

    return EXIT_SUCCESS;
}

```

Figure 54 mstring Examples

Note: the code "**(a || b)**" in the cout expression is parenthesized. If not parenthesized, the C++ compiler precedence will result in an error since the precedence of << is greater than ||.

10.2 Assignment from Other Data Types

Variables of type **mstring** may be assigned values from variables or constants of types **char ***, **string**, **global**, **mstring**, **float**, **int**, **long**, or **double** as shown in Figure 55.

```

#include <mumpsc/libmpscpp.h>

int main() {

    // examples of assignment to mstring

    mstring x;

    x = 10;          cout << x << endl;
    x = 10.99;       cout << x << endl;
    x = "test";      cout << x << endl;

    string a1="abcdef";
    float  a2=99.9;
    double a3=99.8;
    int    a4=99;
    short  a5=98;
    char   a6[]="abcdef";
    global a7("a7"); a7("1")=99;

    x = a1;          cout << x << endl;
    x = a2;          cout << x << endl;
    x = a3;          cout << x << endl;
    x = a4;          cout << x << endl;
    x = a5;          cout << x << endl;
    x = a6;          cout << x << endl;
    x = a7("1");     cout << x << endl;

    GlobalClose;

    return EXIT_SUCCESS;

}

```

which writes:

```

10
10.99
test
abcdef
99.9
99.8
99
98
abcdef
99

```

Figure 55 mstring Assignment Examples

10.3 Arithmetic Operations on mstring

Figure 56 gives examples of arithmetic operations of **mstring**.

```

#include <mumpsc /libmpscpp.h>

int main() {

// examples mstring operators.

    mstring x;
    mstring y;
    mstring z;

    y = 1;

    x = 10;

    cout << "expect 11 " << x + 1 << endl;
    cout << "expect 9 " << x - 1 << endl;
    cout << "expect 20 " << x * 2 << endl;
    cout << "expect 5 " << x / 2 << endl;
    cout << "expect 1 " << x % 3 << endl;
    cout << "expect 11 " << x + y << endl;

    cout << "-----\n";

    x = 10; x = x + 1;      cout << "expect 11 " << x << endl;
    x = 10; x = x - 1;      cout << "expect 9 " << x << endl;
    x = 10; x = x * 2;      cout << "expect 20 " << x << endl;
    x = 10; x = x / 2;      cout << "expect 5 " << x << endl;
    x = 10; x = x % 3;      cout << "expect 1 " << x << endl;
    x = 10; x = x + y;      cout << "expect 11 " << x << endl;
    x = 10; x = y + x;      cout << "expect 11 " << x << endl;

    cout << "-----\n";

    x = 10; x += 1;      cout << "expect 11 " << x << endl;
    x = 10; x -= 1;      cout << "expect 9 " << x << endl;
    x = 10; x *= 2;      cout << "expect 20 " << x << endl;
    x = 10; x /= 2;      cout << "expect 5 " << x << endl;
    x = 10; x %= 3;      cout << "expect 1 " << x << endl;

    cout << "-----\n";

    x=10; x += y;      cout << "expect 11 " << x << endl;
    x=10; x -= y;      cout << "expect 9 " << x << endl;
    x=10; x *= y;      cout << "expect 10 " << x << endl;

```

```

x=10; x /= y;      cout << "expect 10 " << x << endl;
x=10; x %= y;      cout << "expect  0 " << x << endl;

cout << "-----\n";

x = 10; x = 1 + x + y;  cout << "expect 12 " << x << endl;
x = 10; x = 1 - x + y;  cout << "expect - 8 " << x << endl;
x = 10; x = 1 * x + y;  cout << "expect 11 " << x << endl;
x = 10; x = 1 / x + y;  cout << "expect 1.1 " << x << endl;

cout << "-----\n";

x = 10; x = 1 + ( x + y ); cout << "expect 12 " << x << endl;
x = 10; x = (x + y) + ( x + y ); cout << "expect 22 " << x << endl;

x = 10; cout << "expect 11 " << ++x ;
      cout << " expect 11 " << x << endl;
x = 10; cout << "expect 10 " << x++ ;
      cout << " expect 11 " << x << endl;
x = 10; cout << "expect  9 " << --x ;
      cout << " expect  9 " << x << endl;
x = 10; cout << "expect 10 " << x-- ;
      cout << " expect  9 " << x << endl;

cout << "-----\n";

x = 10; cout << "expect yes "; if ( x == 10 ) cout << "yes\n";
x = 10; cout << "expect yes "; if ( x >= 10 ) cout << "yes\n";
x = 10; cout << "expect yes "; if ( x <= 10 ) cout << "yes\n";
x = 10; cout << "expect yes "; if ( x >= 9 ) cout << "yes\n";
x = 10; cout << "expect yes "; if ( x > 9 ) cout << "yes\n";

x = 10; cout << "expect no ";
if ( x != 10 ) cout << "yes\n"; else cout << "no\n";

x = 10; cout << "expect no ";
if ( x > 10 ) cout << "yes\n"; else cout << "no\n";

x = 10; cout << "expect no ";
if ( x < 10 ) cout << "yes\n"; else cout << "no\n";

x = 10; cout << "expect no ";
if ( x <= 9 ) cout << "yes\n"; else cout << "no\n";

x = 10; cout << "expect no ";
if ( x < 9 ) cout << "yes\n"; else cout << "no\n";

cout << "-----\n";

x = "test message";
cout << "expect \"test message\" " << x << endl;

cin >> x;
cout << "expect what you typed " << x << endl;

x = "test ";
y = "message";

z = x || y;
cout << "expect \"test message\" " << z << endl;

x = x || x || x;
cout << "expect \"test test test\" " << x << endl;

```

```

x = "test";
z = y = (x || " test");
cout << "expect \"test test test test\" " << y << " " << z << endl;

x = "test";
z = y = x = x || " test";
cout << "expect \"test test test test\" " << y << " " << z << endl;
cout << "expect \"test test\" " << x << endl;

GlobalClose;

return EXIT_SUCCESS;

}

```

Figure 56 mstring Arithmetic Operations

10.4 Miscellaneous mstring Rules

1. Objects of **mstring** may be not initialized in declaration statements.
2. Objects of type **mstring** may participate in add(+, +=), subtract(-, -=), multiply(*, *=), divide(/, /=), modulo (% , %=) (integers values only) pre/post increment/decrement (++/--), and concatenation (| |) operations. The mode of the operation will depend on the mode of the other operand.
3. Objects of type **mstring** may participate in relational expressions >, >=, <, <=. The mode of comparison will depend on the mode of the other operand.
4. Objects of type **mstring** may participate in equality expressions == and !=. The mode of the comparison will depend on the mode of the other operand.
5. Objects of type **mstring** may participate in input and output stream operations >> and <<.
6. Objects of type **mstring** may not be assigned directly to ASCII null terminated string (**char ***) or **string**.
7. Objects of type **mstring** may be declared as arrays or allocated/freed by the **new/delete** operators. Only numeric subscripts permitted at this time.
8. If an object of type **mstring** is to be used in connection with the interpreter, it must be declared with a string giving its name in the run time symbol table. For example:

```
mstring x("x");
```

If this is done, variables in the C++ program are linked to variables of the same name in the Mumps interpreter. That is, values from variables in the C++ program are known by the same name to interpreted programs invoked by the C++ program. Changes made to these variables in the interpreter are changes to the variables in the C++ program. Variable names selected must be compatible with the interpreter's naming conventions.

11 Overloaded mstring Operators

The operator overload for **mstring** are shown in Figure 106. Additional overloads may be added in time.

11.1 Addition

<pre>mstring operator+(int); mstring operator+(long); mstring operator+(double); mstring operator+(float); mstring operator+(mstring); mstring operator+(string); mstring operator+(const char *); mstring operator+(char *); mstring operator+(global); mstring operator+=(mstring); mstring operator+=(int); mstring operator+=(long); mstring operator+=(double); mstring operator+=(float); mstring operator+=(string); mstring operator+=(const char *); mstring operator+=(global);</pre>	<pre>friend mstring operator+(int,mstring); friend mstring operator+(long,mstring); friend mstring operator+(double,mstring); friend mstring operator+(float,mstring); friend mstring operator+(string,mstring); friend mstring operator+(global,mstring); friend mstring operator+(const char *,mstring); friend mstring operator+(char *,mstring);</pre>
--	--

11.2 Subtraction

<pre>mstring operator-(int); mstring operator-(long); mstring operator-(double); mstring operator-(float); mstring operator-(mstring); mstring operator-(string); mstring operator-(const char *); mstring operator-(char *); mstring operator-(global); mstring operator-=(mstring); mstring operator-=(int); mstring operator-=(long); mstring operator-=(double); mstring operator-=(float); mstring operator-=(string); mstring operator-=(const char *); mstring operator-=(global);</pre>	<pre>mstring operator-(int); mstring operator-(long); mstring operator-(double); mstring operator-(float); mstring operator-(mstring); mstring operator-(string); mstring operator-(const char *); mstring operator-(char *); mstring operator-(global);</pre>
--	--

11.3 Multiplication

<pre>mstring operator*(int); mstring operator*(long); mstring operator*(double); mstring operator*(float); mstring operator*(mstring); mstring operator*(string); mstring operator*(global); mstring operator*(const char *); mstring operator*=(mstring); mstring operator*=(int);</pre>	<pre>friend mstring operator*(int,mstring); friend mstring operator*(long,mstring); friend mstring operator*(double,mstring); friend mstring operator*(float,mstring); friend mstring operator*(string,mstring); friend mstring operator*(global,mstring); friend mstring operator*(const char *,mstring);</pre>
--	--

<pre>mstring operator*=(long); mstring operator*=(double); mstring operator*=(float); mstring operator*=(string); mstring operator*=(const char *); mstring operator*=(global);</pre>	
---	--

11.4 Division

<pre>mstring operator/(int); mstring operator/(long); mstring operator/(double); mstring operator/(float); mstring operator/(mstring); mstring operator/(string); mstring operator/(global); mstring operator/(const char *); mstring operator/=(mstring); mstring operator/=(int); mstring operator/=(long); mstring operator/=(double); mstring operator/=(float); mstring operator/=(string); mstring operator/=(const char *); mstring operator/=(global);</pre>	<pre>friend mstring operator/(int,mstring); friend mstring operator/(long,mstring); friend mstring operator/(double,mstring); friend mstring operator/(float,mstring); friend mstring operator/(string,mstring); friend mstring operator/(global,mstring); friend mstring operator/(const char *,mstring);</pre>
---	--

11.5 Increment/Decrement

<pre>mstring operator++(); mstring operator--(); mstring operator++(int); mstring operator--(int);</pre>	
--	--

11.6 Unary Operations

<pre>mstring operator!(); // unary mstring operator+(); // unary mstring operator-(); // unary</pre>	
--	--

11.7 Modulo

<pre>mstring operator%(long); mstring operator%(int); mstring operator%(mstring); mstring operator%(string); mstring operator%(const char *); mstring operator%(global); mstring operator%=(mstring); mstring operator%=(int); mstring operator%=(long); mstring operator%=(string); mstring operator%=(const char *); mstring operator%=(global);</pre>	<pre>friend mstring operator%(int,mstring); friend mstring operator%(long,mstring); friend mstring operator%(string,mstring); friend mstring operator%(global,mstring); friend mstring operator%(const char *,mstring);</pre>
---	---

11.8 Concatenation

<pre>mstring operator (mstring); mstring operator (string); mstring operator (global);</pre>	<pre>friend mstring operator (string, mstring);</pre>
---	--

<pre> mstring operator (const char *); mstring operator (int); mstring operator (long); mstring operator (float); mstring operator (double); mstring operator&(mstring); mstring operator&(char *); mstring operator&(global); mstring operator&(string); mstring operator&(int); mstring operator&(long); mstring operator&(double); </pre>	<pre> friend mstring operator (global, mstring); friend mstring operator (const char *, mstring); friend mstring operator (int, mstring); friend mstring operator (long, mstring); friend mstring operator (float, mstring); friend mstring operator (double, mstring); </pre>
--	--

11.8.1 Relational

<pre> bool operator==(int); bool operator==(long); bool operator==(double); bool operator==(float); bool operator==(const char *); bool operator==(char *); bool operator==(string); bool operator==(global); bool operator==(mstring); bool operator!=(int); bool operator!=(long); bool operator!=(double); bool operator!=(float); bool operator!=(char *); bool operator!=(const char *); bool operator!=(string); bool operator!=(global); bool operator!=(mstring); bool operator<(int); bool operator<(long); bool operator<(double); bool operator<(float); bool operator<(char *); bool operator<(const char *); bool operator<(string); bool operator<(global); bool operator<(mstring); bool operator>(int); bool operator>(long); bool operator>(double); bool operator>(float); bool operator>(char *); bool operator>(const char *); bool operator>(string); bool operator>(global); bool operator>(mstring); bool operator>=(int); bool operator>=(long); bool operator>=(double); bool operator>=(float); bool operator>=(char *); </pre>	
--	--

<pre>bool operator>=(const char *); bool operator>=(string); bool operator>=(global); bool operator>=(mstring); bool operator<=(int); bool operator<=(long); bool operator<=(double); bool operator<=(float); bool operator<=(char *); bool operator<=(const char *); bool operator<=(string); bool operator<=(global); bool operator<=(mstring);</pre>	
---	--

Figure 57 mstring Operator Overloads

12 Class Global

Global arrays are represented as members of **class global** and the contents of a **global** reference may be operated on by several overloaded operators and functions.

12.1 Assignment Operators on globals

Assignments to global arrays may be accomplished the assignment operator (=).

When you access a global array, the access may result in the thrown error exceptions *GlobalNotFoundException* and/or *ConversionException*. The first can occur in any context that attempts to retrieve data from a global array where none exists. The second occurs if you attempt to convert the contents of a global to a numeric type where the contents of the global are not valid data for the conversion.

If uncaught, both exceptions will result in program termination. Both exceptions may be caught, however, with code such as shown in Figure 58.

```
#include <mumpsc /libmpscpp.h>
global a("a");

int main() {
    long i;
    a.Kill();

    a("1") = "now is the time";

    cout << "expect error message" << endl;

    try {
        i = a("1");
    }

    catch ( ConversionException ce) {
        cout << ce.what() << endl;
    }

    cout << "expect error message" << endl;

    try {
        i = a("22");
    }

    catch (GlobalNotFoundException nf) {
        cout << nf.what() << endl;
    }

    GlobalClose;

    return 0;
}
```

Figure 58 Exceptions

You may assign data of the following types directly to global arrays: **char ***, **int**, **string**, **mstring**, **double**, **global**, **unsigned int**, **float**, **short**, **unsigned short**, **long**, and **unsigned long**.

You may assign global arrays directly to variables of the following types: **int**, **mstring**, **double**, **global**, **unsigned int**, **float**, **short**, **unsigned short**, **long**, and **unsigned long**.

12.2 Arithmetic Operators on globals

The operations of add, subtract, multiply, divide, pre/post increment and pre/post decrement are defined (overloaded) for global variables. The operations are defined for **mstring, short, unsigned short, int, unsigned int, long, unsigned long, float** and **double**. Note: the contents of the global array node must be compatible with the dominant data type of the operation. If the contents of a global are not compatible with the operation (example, incrementing a string of text), the value of the global will be interpreted as zero. See Figure 59 for examples.

```
#include <mumpsc /libmcpp.h>

global gbl("gbl");

int main () {

    int i, j = 10;
    string a = "10", b = "20", c = "30";

    gbl.Kill();

    gbl(a, b, c) = 10;

    i = gbl(a, b, c) + 20;
    cout << "expect 20 " << i << endl; // prints 30

    i = 20 + gbl(a, b, c);
    cout << "expect 30 " << i << endl; // prints 30

    i = gbl(a, b, c) / j;
    cout << "expect 1 " << i << endl; //prints 1

    i = gbl(a, b, c) * 2;
    cout << "expect 20 " << i << endl; // prints 20

    gbl(a, b, c) ++;
    cout << "expect 11 " << gbl(a, b, c) << endl; // prints 11

    gbl(a, b, c) --;
    cout << "expect 10 " << gbl(a, b, c) << endl; // prints 10

    i = ++ gbl(a, b, c);
    cout << "expect 11 11 " << i << " " << gbl(a, b, c) << endl; // prints
    11

    i = gbl(a, b, c) ++;
    cout << "expect 11 12 " << i << " " << gbl(a, b, c) << endl; // prints
    11 12

    gbl(a, b, c) += 10;
    cout << "expect 22 " << gbl(a, b, c) << endl; // prints 22

    gbl(a, b, c) -= 10;
    cout << "expect 12 " << gbl(a, b, c) << endl; // prints 12

    gbl(a, b, c) *= 2;
    cout << "expect 24 " << gbl(a, b, c) << endl; //prints 24

    gbl(a, b, c) /= 2;
    cout << "expect 12 " << gbl(a, b, c) << endl; // prints 12
```

```
GlobalClose;
return 0;
}
```

Figure 59 Example Global Array Arithmetic

12.3 Accessing the Value Stored in a Global Array Element

```
int global::Int();
double global::Double();
mstring global::Mstring();
char * global::Char(char * buf, int max);
```

The functions return the content of the invoking global array object converted to the named data type.

The **Char()** function is passed the address of a character array. The null-terminated character string contents of the global array element will be placed in the character array and the address of the array returned.

The *max* argument for **Char()** limits the length of the string returned.

If the global array element does not exist, the *GlobalNotFoundException* exception is thrown. If there is an error in converting the contents of the global to the named data type, a *ConversionException* is thrown. See Figure 60 for examples.

```
#include <mumpsc/libmpscpp.h>

global t("t");

int main() {

    int a;
    float b;
    mstring c;
    mstring x;
    char d[100];

    t.Kill();

    x = 50; t(x) = 99;

    a = t(x).Int();
    cout << "expect 99 " << a << endl;

    b = t(x).Double();
    cout << "expect 99 " << b << endl;

    c = t(x).Mstring();
    cout << "expect 99 " << c << endl;

    t(x).Char(d,100);
    cout << "expect 99 " << d << endl;

    GlobalClose;
}
```

Figure 60 Accessing Global Array Data Example

12.4 Direct Btree Access

Programmers may access the btree directly through the builtin **BTREE** macro. A number of examples can be found in *mumpsc/doc/examples/btree* in the distribution.

To access the btree directly from a C++ program:

You must first install the Mumps compiler and MDH. Include at the beginning of your program. You can now access the btree directly with the **BTREE** macro (see description below). Note: any keys you store in the btree co-exist with Mumps/MDH keys. In rare cases, these can interfere with one another if a key you store lies in the range of a global array key set.

For example, the following program stores **NBR_ITERATIONS** (defined in *btree.h* which is included by *libmpscpp.h* usually with the value 100,000) of keys and data into the btree and then retrieves them (this "btest1.cpp" from *mumpsc/doc/examples/btree.cpp*). See the other examples and the documentation below for further details. See Figure 61.

```
/*#+++++
*#+ Mumps Compiler Run-Time Support Functions
*#+ Copyright (c) 2001, 2002, 2003, 2004 by Kevin C. O'Kane
*#+ okane@cs.uni.edu
*#+
*#+ This library is free software; you can redistribute it and/or
*#+ modify it under the terms of the GNU Lesser General Public
*#+ License as published by the Free Software Foundation; either
*#+ version 2.1 of the License, or (at your option) any later version.
*#+
*#+ This library is distributed in the hope that it will be useful,
*#+ but WITHOUT ANY WARRANTY; without even the implied warranty of
*#+ MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
*#+ Lesser General Public License for more details.
*#+
*#+ You should have received a copy of the GNU Lesser General Public
*#+ License along with this library; if not, write to the Free Software
*#+ Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*#+
*#+ http://www.cs.uni.edu/~okane
*#+
*#+++++
*#+
*#+ Some of this code was originally written in Fortran
*#+ which will explain the odd array and label usage,
*#+ especially arrays beginning at index 1.
*#+
*#+++++

#include <mumpsc /libmpscpp.h>

int main() {

    long i,j;

    unsigned char key[1024],data[1024];

    printf("Store sequentially ascending keys");

    for (i = 0; i < NBR_ITERATIONS; i++) {

        sprintf( (char *) key, "key %ld", i);
        sprintf( (char *) data,"%ld%c", i, 0);
```

```

        if (!BTREE(STORE, key, data)) {
            printf("error\n");
            return 1;
        }

        if (i%60000L == 0) { printf("\n %ld ",i); fflush(stdout); }

        if (i%1000 == 0) { putchar('.'); fflush(stdout); }
    }

    printf("\nretrieve");

    for (i = 0; i < NBR_ITERATIONS; i++) {

        sprintf( (char *) key, "key %ld", i);

        if (!BTREE(RETRIEVE, key, data)) {
            printf("error 1\n");
            return 1;
        }

        sscanf( (char *) data, "%ld", &j);

        if (j!=i) {
            printf("error 2\n");
            printf("%d != %d\n", i, j);
            return 1;
        }

        if (i%60000L == 0) { printf("\n %ld ",i); fflush(stdout); }
        if (i%1000 == 0) { putchar('.'); fflush(stdout); }
    }

    printf("\nlooks good!\n");

    strcpy( (char *) key, "");
    strcpy( (char *) data, "");

    BTREE(CLOSE, key, data);
    return 1;
}

```

Figure 61 BTREE Example

12.5 Functions on class global

12.5.1 Data()

int global::Data()

The function *Data()* returns an integer which indicates whether the global array node is defined. The value returned is 0 if the global array node is undefined, 1 if it is defined and has no descendants; 10 if it is defined but has no value stored at the node (but does have descendants); and 11 if it is defined and has descendants.

If a global array with no indices is passed to these functions, a value of "10" will be returned if the array exists and "0" if the array does not exist. For example:

Given:

```

global gbl("gbl");
global non("non");
gbl("1", "11") = "foo";
gbl("1", "11", "21") = "bar";

```

Then:

```

gbl("1").Data() // 10 - node exists, has no data, has children
gbl("1", "11").Data() // 11 - node exists, has data and has children
gbl("1", "11", "21").Data() // 1 - nodes exists, has data, no children

```

12.5.2 TreePrint()

```
void global::TreePrint([int indt [, const char indtchr]]);
```

The invoking object is printed as an indented tree. If one argument is present (*indt*), it is the amount of indentation. If the second argument is present (*indtchr*) it is the character used in the indentation. The default indentation character is blank and the default amount of indentation is one. See Figures 62 and 63 for examples.

```

#include <mumpsc/libmpscpp.h>

global d("d");

int main() {
    mstring a,b,c;

    for (int i = 1; i < 6; i++)
        for (int j = 1; j < 6; j++)
            for (int k = 1; k < 6; k++) {
                a = mcvT(i);
                b = mcvT(j);
                c = mcvT(k);
                d(a) = rand() % 100;
                d(a,b) = rand() % 100;
                d(a,b,c) = rand() % 100;
            }

    d().TreePrint(1, '.');

    GlobalClose;

    return 0;
}

```

Figure 62 TreePrint

Yields

1=82	2=68	3=72	4=66	5=79
..1=59	..1=54	..1=28	..1=48	..1=72
..1=77	..1=64	..1=96	..1=39	..1=76
..2=35	..2=87	..2=45	..2=69	..2=7
..3=49	..3=78	..3=21	..3=64	..3=79
..4=27	..4=3	..4=88	..4=55	..4=12
..5=63	..5=99	..5=41	..5=11	..5=59
.2=67	.2=78	.2=59	.2=30	.2=21
..1=26	..1=76	..1=0	..1=99	..1=10
..2=11	..2=12	..2=24	..2=68	..2=6
..3=29	..3=94	..3=56	..3=11	..3=72
..4=62	..4=70	..4=27	..4=1	..4=19
..5=35	..5=67	..5=36	..5=78	..5=4
.3=19	.3=44	.3=93	.3=62	.3=69
..1=22	..1=2	..1=37	..1=36	..1=40
..2=67	..2=52	..2=7	..2=22	..2=28
..3=11	..3=80	..3=58	..3=16	..3=84
..4=73	..4=65	..4=37	..4=24	..4=24
..5=84	..5=19	..5=18	..5=24	..5=96
.4=96	.4=53	.4=4	.4=94	.4=98
..1=24	..1=31	..1=11	..1=52	..1=84
..2=13	..2=71	..2=76	..2=50	..2=72
..3=80	..3=9	..3=63	..3=73	..3=85
..4=62	..4=56	..4=6	..4=30	..4=40
..5=81	..5=86	..5=18	..5=60	..5=13
.5=45	.5=8	.5=25	.5=84	.5=69
..1=84	..1=83	..1=69	..1=81	..1=24
..2=5	..2=28	..2=96	..2=59	..2=81
..3=13	..3=29	..3=70	..3=68	..3=32
..4=95	..4=70	..4=99	..4=26	..4=4
..5=14	..5=15	..5=44	..5=40	..5=73

Figure 63 TreePrint Output

12.5.3 UnLock

```
int global::UnLock()
```

UnLock() removes a lock from the designated node.

12.5.4 Count

```
long global::Count()
```

Returns the number of data bearing nodes beneath the given global array reference. See Figure 64 for example.

```
#include <mumpsc/libmumpscpp.h>

global A("A");

int main() {
    mstring i, j;
    for (i = 1; i < 11; i++)
        for (j = 1; j < 11; j++) {
            A(i,j) = 5;
        }
    cout << "Full count: " << A().Count() << endl;
    cout << "A row count: " << A("5").Count() << endl;
    return EXIT_SUCCESS;
}
```

```
}
```

Yields

```
Full count: 100
A row count: 10
```

Figure 64 Count Example

12.5.5 GlobalGet(), GlobalData(), GlobalSet()

```
mstring GlobalGet (mstring global_ref)
char * GlobalGet (char * global_ref)

mstring GlobalOrder (mstring global_ref, int direction)
char * GlobalOrder (char * global_ref, int direction)

int GlobalData (mstring global_ref)
int GlobalData (char * global_ref)

int GlobalSet (mstring global_ref, mstring source)
int GlobalSet (mstring global_ref, char * source)
int GlobalSet (char * global_ref, mstring source)
```

These functions use the interpreter. These functions are used to permit run time construction and access to global arrays. In both cases *global_ref* is a string containing a global array reference. This string can be dynamically constructed at run time or may be read from a file or another global. Note: as this facility uses the interpreter, global array references must be preceded by the circumflex character (^).

In the case of the *GlobalGet()* functions, the string global array reference is interpreted and the value stored at the reference returned. If the reference is invalid or no data is stored, the value returned is the empty string and *\$test* is set to false (zero). If a value is found, *\$test* is set to true and the value is returned.

GlobalOrder() gives the next or prior value of the last index of the global array reference depending upon if *direction* is 1 (next) or -1 (prior). *\$test* is set to 0 in the event of an error and 1 if there is no error. See *Order()*.

GlobalData() returns a number indicating if the node exists and has descendants (see *Data()*). *\$test* is set to 0 if there is an error, 1 otherwise.

In the case of the *GlobalSet()* functions, the second argument is a string of data to be stored at the global array reference. The runtime routines will interpret the *global_ref* and assign the *source* to it. The value returned is one if successful (*\$test* is set to 1), zero if not successful (*\$test* set to 0). Examples:

```
mstring a,b;
a = "^x(\"1\")";
b = "test string";
if (GlobalSet(a,b) != 0) cout << "error\n";
```

These functions can be used to allow a program to create a text string global array reference and then use the string to address the global. Note that the *target* must contain either quoted literals or variables previously instantiated to the interpreter environment (see *\$SymSet()* and *SymGet()*).

Generally speaking, these functions will be only used for dynamically constructed global array references. Most access to globals will be by overloaded shift or assignment operators.

12.5.6 double HitRatio(void)

Calculates the native global array processor cache hit ratio since the beginning of the program or the last call to *HitRatio()*. The native global array file processor, as opposed to the Berkeley Data Base, keeps track

of how many file I/O requests are satisfied from data already in the file system's cache. This function gives the percentage of cache hits. It only works with the native global array processor.

12.5.7 Kill

```
void global::Kill()
```

This function deletes a node and all its descendants. Examples:

```
gbl().Kill();           // kill entire global array "gbl"  
gbl(a,b,c).Kill();     // kill stated node and all descendants
```

12.5.8 Length

```
int mstring::Length()  
int mstring::Length(char * pattern_string)  
int mstring::Length(mstring pattern_string)
```

The function returns the string length of the invoking **mstring**. For example:

```
x="ABC";  
cout << x.Length() << endl;  // writes 3  
  
x = "abcabcabcabc";  
cout << x.Length("abc") << endl;  // writes 5
```

If an argument is given, the function returns the number of non-overlapping occurrences of "pattern_string" in the source string plus 1.

12.5.9 Max

```
double global::Max()
```

Returns the maximum numeric value of the data bearing nodes beneath the given reference. Non-numeric values are treated as zeros. See Figure 65 for example.

```
#include <mumpsc/libmpscpp.h>  
global A("A");  
  
int main() {  
    mstring i, j;  
    for (i = 1; i < 11; i++)  
        for (j = 1; j < 11; j++) {  
            A(i, j) = rand()%1000;  
        }  
    cout << "Max value of all:      " << A().Max() << endl;  
    cout << "Max value of row 10:  " << A("10").Max() << endl;  
    return EXIT_SUCCESS;  
}
```

Yields:

```
Max value of all:      996  
Max value of row 10:  932
```

Figure 65 Max Example

12.5.10 Merge

```
int global::Merge(global)
```

Copies the first **global** and its descendants to the second **global**. The Merge() function copies from one array to another. Examples:

```

Xecute("for i=1:1:9 for j=1:1:9 set ^a(i,j)=i+j");
c().Merge(a());          // copies all of ^a to ^c

Xecute("for i=100:1:109 s ^b(i)=i");

b("103").Merge(a("3")); // copies ^a(3) to ^b(103) and children of
                        // ^a(3) to be children of ^b(103)

d("").Merge(a("3"));    // creates ^d=^a(3); ^d(1)=^a(3,1),...

```

12.5.11 Min

double global::Min()

Returns the minimum numeric value of the data bearing nodes beneath the given reference. Non-numeric values are treated as zeros. Example:

```

#include <mumpsc /libmpsc.cpp.h>
global A("A");

int main() {
    mstring i, j;
    for (i = 1; i < 11; i++)
        for (j = 1; j < 11; j++) {
            A(i,j) = rand() % 1000;
        }
    cout << "Min value of all:      " << A().Min() << endl;
    cout << "Min value of row 10:  " << A("10").Min() << endl;
    return EXIT_SUCCESS;
}

```

Yields:

```

Min value of all:      11
Min value of row 10:  12

```

12.5.12 Multiply

void global::Multiply(global B,global C)

The invoking global is multiplied by *B* and the result is place in *C*. The number of columns of *A* must equal the number of rows of *B*. The resulting matrix *C* will have "n" rows and "m" columns where "n" is the number of rows of "A" and "m" is the number of columns of "B".

In all cases *C* will be deleted before the operation commences. The data stored at each node must be numeric. All calculations are performed in **double** precision arithmetic. Each matrix must be two dimensional. See Figure 66.

```

#include <mumpsc/libmpsc.cpp.h>
#include <mumpsc/libmpsrdbms.h>

global d("d");
global e("e");
global f("f");

int main() {

    d("1", "1") = 2;
    d("1", "2") = 3;
    d("2", "1") = 1;
    d("2", "2") = -1;
}

```

```

d("3", "2") = 0;
d("3", "2") = 4;

e("1", "1") = 5;
e("1", "2") = -2;
e("1", "3") = 4;
e("1", "4") = 7;
e("2", "1") = -6;
e("2", "2") = 1;
e("2", "3") = -3;
e("2", "4") = 0;

d().Multiply(e(),f());
PRINT("f","1");

return EXIT_SUCCESS;
}

```

Yields:

```

^f(1,1)=-8
^f(1,2)=-1
^f(1,3)=-1
^f(1,4)=14
^f(2,1)=11
^f(2,2)=-3
^f(2,3)=7
^f(2,4)=7
^f(3,1)=-24
^f(3,2)=4
^f(3,3)=-12
^f(3,4)=0

```

Figure 66 Multiply Example

12.5.13 Name

mstring global::Name()

Returns a null terminated pointer to array of characters containing of the **global** reference with all variables and expressions in the indices evaluated. See Figure 67.

```

#include <mumpsc/libmpscpp.h>
global a("a");

int main() {
    mstring b = "1", c = "2", d = "3";
    cout << a(b, c, d, c + d).Name() << endl;
    return EXIT_SUCCESS;
}

```

Yields:

```

a("1", "2", "3", "23")

```

Figure 67 Name Example

12.5.14 Order

mstring global::Order([int direction])

The *Order()* function gives the next ascending or descending value of the last index in a global array reference. The direction, ascending or descending, is given by either the name of the function or an integer

"direction" which is either 1 - next ascending index, or -1 - next descending index. If 'direction' is omitted, ascending is assumed. See Figure 68.

given:

```
global test("test");
test("1") = "";
test("1", "10") = "";
test("1", "20") = "";
test("5", "1") = "";
test("5", "5") = "";
```

Then Order() will return the following values:

```
test().Order(1)           yields "1"
test("1", "").Order(1)    yields "10"
test("1", "10").Order(1)  yields 20
test("1", "20").Order(1)  yields "" (empty string)
test("1").Order(1)        yields "5"
test("5", "").Order(1)    yields "1"
test("5", "1").Order(1)   yields "2"
test("5", "2").Order(1)   yields "" (empty string)
test("5").Order(1)        yields "" (empty string)
```

Similarly, a direction code of -1 will reverse the process:

```
test().Order(-1)          yields 5
test("5").Order(-1)       yields "1"
test("1").Order(-1)       yields "" (empty string)
```

Figure 68 Order Example

Use the empty string ("") to get the initial value of an index. When there are no further values, the empty string is returned.

Note: all keys are stored in ASCII character collating order. This means that numeric keys are sorted alphabetically rather than numerically.

12.5.15 Avg

double global::Avg()

Returns the average of the values of data bearing nodes beneath the given global array reference.

Example:

```
#include <mumpsc/libmpscpp.h>

global A("A");

int main() {
    mstring i,j;

    A.Kill();

    for (i = 0; i < 1000; i++)
        for (j = 1; j < 10; j++) {
            A(i, j) = j;
        }
}
```

```

cout << A("100").Avg() << endl; // average of nodes below A("100")
cout << A().Avg() << endl;      // average of all nodes

GlobalClose;

return 0;
}

```

Figure 69 Avg Example

The above prints 5.5 - the average value of numeric data bearing nodes beneath A("100"). If there are non-numeric data elements, they are treated as a zero values and contribute to the result.

The global array object must be specified with indices (*i.e.*, a parenthesized list must follow the name of the **global** array object. An empty list means the entire array.

12.5.16 Locks

```

void CleanLocks(void)
void CleanAllLocks(void)

```

"CleanLocks()" removes all locks for the current process. "CleanAllLocks()" removes all locks for all processes for which the current directory is the default directory. Locks are implemented by entries in a file named "Mumps.Locks" created and maintained in the current directory. This file must be read/write enabled for the current process. You may also delete all locks by removing this file. Locks are discussed elsewhere but, in brief, they are used to signal ownership of a portion of a global array. When a lock has been applied to a node, no other process may lock this node, any descendant node or any parent node. Locking does not actually prevent access, it merely marks a resource as locked.

```

int global::Lock()

```

Creates a lock on the named node. If successful, "\$test" will be true (1), false (0) otherwise. Returns a 1 if the lock succeeds and a 0 otherwise.

The "Lock()" function marks a portion of the data base for exclusive access for an individual user. The "UnLock()" frees prior locks (see below). The locks are stored in a file named "Mumps.Locks" which is opened for exclusive access by the locking/unlocking job. The contents of the file may be deleted to remove all locks. A lock does not actually prevent access to a global but merely marks it as locked. If another task attempts to place a lock on a locked node, the descendant of a locked node or a direct parent of a locked node, the lock attempt will fail. Examples:

```

if (gbl(a, b, c).Lock()) { ..... } // locks gbl(a, b, c) and all children;
if ($lock(gbl(a, b, c))) { ..... }

```

12.5.17 GlobalClose

This macro closes the global array files. The global arrays must be closed on exit or they will be corrupt. The macro causes the file system to flush all its buffers and cache and close the file system. Normally, a "GlobalClose" is executed automatically when your program ends except if your program is terminated by SIGKILL or SIGSTOP (which cannot be trapped). If your program is using a large memory based cache (cache's can be 1 GB or more, on some systems), there may be a noticeable delay in file system shutdown due to the time required to write the cache to disk.

12.5.18 Query functions

```

mstring Query(mstring ref)
mstring Query(char * ref)

```

```

int Qlength(mstring ref)
int Qlength(char * ref)

mstring Qsubscript(mstring ref, mstring index)
mstring Qsubscript(mstring ref, int index)
mstring Qsubscript(char * ref, int index)

```

Query() returns an **mstring** containing the next global array reference in the data base or the empty string.

Qlength() returns the number of subscripts in the global array reference.

Qsubscript() returns the index'th subscript of a global array reference.

Each of these functions operates on a text representation of a global array reference. See also the *Name()* function. The following example makes use of the MeSH subject headings (National Library of Medicine). The MeSH global array was constructed with statements such shown in Figure 70.

```

set ^mesh("A01")="Body Regions"
set ^mesh("A01","047")="Abdomen"
set ^mesh("A01","047","025")="Abdominal Cavity"
set ^mesh("A01","047","025","600")="Peritoneum"
set ^mesh("A01","047","025","600","225")="Douglas' Pouch"
set ^mesh("A01","047","025","600","451")="Mesentery"
set ^mesh("A01","047","025","600","451","535")="Mesocolon"
set ^mesh("A01","047","025","600","573")="Omentum"
set ^mesh("A01","047","025","600","678")="Peritoneal Cavity"
set ^mesh("A01","047","025","750")="Retroperitoneal Space"
set ^mesh("A01","047","050")="Abdominal Wall"
set ^mesh("A01","047","365")="Groin"
set ^mesh("A01","047","412")="Inguinal Canal"
set ^mesh("A01","047","849")="Umbilicus"
set ^mesh("A01","176")="Back"
set ^mesh("A01","176","519")="Lumbosacral Region"
set ^mesh("A01","176","780")="Sacrococcygeal Region"
set ^mesh("A01","236")="Breast"
set ^mesh("A01","236","500")="Nipples"
set ^mesh("A01","378")="Extremities"
set ^mesh("A01","378","100")="Amputation Stumps"
set ^mesh("A01","378","610")="Lower Extremity"
set ^mesh("A01","378","610","100")="Buttocks"
set ^mesh("A01","378","610","250")="Foot"
set ^mesh("A01","378","610","250","149")="Ankle"
set ^mesh("A01","378","610","250","300")="Forefoot, Human"
set ^mesh("A01","378","610","250","300","480")="Metatarsus"
set ^mesh("A01","378","610","250","300","792")="Toes"
set ^mesh("A01","378","610","250","300","792","380")="Hallux"
set ^mesh("A01","378","610","250","510")="Heel"
set ^mesh("A01","378","610","400")="Hip"
set ^mesh("A01","378","610","450")="Knee"
set ^mesh("A01","378","610","500")="Leg"
set ^mesh("A01","378","610","750")="Thigh"
set ^mesh("A01","378","800")="Upper Extremity"
set ^mesh("A01","378","800","075")="Arm"
set ^mesh("A01","378","800","090")="Axilla"
set ^mesh("A01","378","800","420")="Elbow"
set ^mesh("A01","378","800","585")="Forearm"
set ^mesh("A01","378","800","667")="Hand"
set ^mesh("A01","378","800","667","430")="Fingers"
set ^mesh("A01","378","800","667","430","705")="Thumb"
set ^mesh("A01","378","800","667","715")="Wrist"

```



```
set ^mesh("A01","378","800","750")="Shoulder"
```

Figure 70 MeSH Headings¹³

The MeSH headings can be printed as shown in Figure 71.

```
#include <mumpsc/libmpscpp.h>

//      CompiledMtree1.cpp Feb 28, 2007

int main() {

    global mesh("mesh");
    mstring x;
    int i,j;

    x=Query("^mesh(0)");
    while (1) {
        x=Query(x);
        if (x=="") break;
        if (x.Piece("(",1)!="^mesh") break;
        i=length(x);
        for (j=0; j<i; j++) cout << "    ";
        cout << Qsubscript(x,i) << " " << x.Eval() << endl;
    }
    return 0;
}
```

which yields:

```
047 Abdomen
    025 Abdominal Cavity
        600 Peritoneum
            225 Douglas' Pouch
            451 Mesentery
            535 Mesocolon
            573 Omentum
            678 Peritoneal Cavity
        750 Retroperitoneal Space
    050 Abdominal Wall
    365 Groin
    412 Inguinal Canal
    849 Umbilicus
176 Back
    519 Lumbosacral Region
    780 Sacrococcygeal Region
236 Breast
    500 Nipples
378 Extremities
    100 Amputation Stumps
    610 Lower Extremity
        100 Buttocks
        250 Foot
```

¹³ The MeSH (Medical Subject Headings) is a controlled vocabulary hierarchical indexing and classification system developed by the National Library of Medicine (NLM). The MeSH codes are used to code medical records and literature as part of an ongoing research project at the NLM. The examples make use of the 2003 MeSH Tree Hierarchy. Newer versions, essentially similar to these, are available from NLM. Note: *for clinical purposes, the copy of the MeSH hierarchy used here is out of date and should not be employed for clinical decision making. It is used here purely as an example to illustrate a hierarchical index.* The 2003 MeSH file contains approximately 40,000 entries. Each line consists of text along with hierarchical codes describing the subject heading.

```

149 Ankle
300 Forefoot, Human
480 Metatarsus
792 Toes
380 Hallux
510 Heel
400 Hip
450 Knee
500 Leg
750 Thigh
800 Upper Extremity
075 Arm
090 Axilla
420 Elbow
585 Forearm
667 Hand
430 Fingers
705 Thumb
715 Wrist
750 Shoulder

```

Figure 71 Query Functions Example

12.5.19 Similarity functions

```

double Sim1(global A, global B)
double Cosine(global A, global B)
double Jaccard(global A, global B)
double Dice(global A, global B)

```

The global arrays referenced by the invoking object and the passed object are compared and a similarity value is computed. The functions compute the similarities of the data bearing nodes beneath the global array references.

These are some commonly used similarity metrics. (see Salton, G; and McGill, M, *Introduction to Modern Information Retrieval*, McGraw Hill, 1983). See Figures 72 through 75.

```

#include <mumpsc/libmpscpp.h>

global A("A");
global B("B");

int main() {

    A("1","1","1") = 1;
    A("1","1","2") = 1;
    A("1","1","3") = 1;
    A("1","1","5") = 1;

    B("1","1","1") = 1;
    B("1","1","2") = 1;
    B("1","1","4") = 1;
    B("1","1","6") = 1;

    cout << Sim1(A("1","1"), B("1","1")) << endl;

    GlobalClose;

    return 0;
}

```

The above prints 2 since there are two nodes in common below the "1,1" levels.

Alternatively:

```
#include <mumpsc/libmpscpp.h>

global A("A");
global B("B");

int main() {

    A("1","1","1") = 2;
    A("1","1","2") = 1;
    A("1","1","3") = 1;
    A("1","1","5") = 1;

    B("1","1","1") = 2;
    B("1","1","2") = 1;
    B("1","1","4") = 1;
    B("1","1","6") = 1;

    cout << Sim1(A("1","1"), B("1","1")) << endl;

    GlobalClose;

    return 0;
}
```

The above prints 5 since there are two nodes in common below the "1,1" levels but one of the set of nodes in common have a stored value of 2. ($2*2+1*1$)

Figure 72 Sim1 Example

```
#include <mumpsc /libmpscpp.h>

global A("A");
global B("B");

int main() {

    A("1") = 3;
    A("2") = 2;
    A("3") = 1;
    A("4") = 0;
    A("5") = 0;
    A("6") = 0;
    A("7") = 1;
    A("8") = 1;

    B("1") = 1;
    B("2") = 1;
    B("3") = 1;
    B("4") = 0;
    B("5") = 0;
    B("6") = 1;
    B("7") = 0;
    B("8") = 0;

    cout << Jaccard(A(), B()) << endl;
```

```
GlobalClose;

return 0;
}
```

prints 1

Figure 73 Jaccard Example

```
#include <mumpsc/libmpscpp.h>

global A("A");
global B("B");

int main() {

    A("1") = 3;
    A("2") = 2;
    A("3") = 1;
    A("4") = 0;
    A("5") = 0;
    A("6") = 0;
    A("7") = 1;
    A("8") = 1;

    B("1") = 1;
    B("2") = 1;
    B("3") = 1;
    B("4") = 0;
    B("5") = 0;
    B("6") = 1;
    B("7") = 0;
    B("8") = 0;

    cout << Dice(A(), B()) << endl;

    GlobalClose;

    return 0;
}
```

prints 1

Figure 74 Dice Example

```
#include <mumpsc/libmpscpp.h>

global A("A");
global B("B");

int main() {

    A("1") = 3;
    A("2") = 2;
    A("3") = 1;
    A("4") = 0;
    A("5") = 0;
    A("6") = 0;
    A("7") = 1;
    A("8") = 1;
```

```

B("1") = 1;
B("2") = 1;
B("3") = 1;
B("4") = 0;
B("5") = 0;
B("6") = 1;
B("7") = 0;
B("8") = 0;

cout << Cosine(A(), B()) << endl;

GlobalClose;

return 0;
}

```

prints 0.75

Figure 75 Cosine Example

12.5.20 Transpose

void global::Transpose(global out)

The invoking object is transposed and the result is placed in *out*. Any prior contents of the array *out* are deleted before the operation commences. See Figure 76.

```

#include <mumpsc/libmpscpp.h>
#include <mumpsc/libmpsrdhms.h>

global d("d");
global f("f");

int main() {

    d("1","1")=2;
    d("1","2")=3;
    d("2","1")=4;
    d("2","2")=0;

    d().Transpose(f()); // transpose d() placing result in f()

    cout << f("1","1") << " " f("1","2") << endl;
    cout << f("2","1") << " " f("2","2") << endl;

    GlobalClose;

    return EXIT_SUCCESS;
}

```

Yields:

```

2 4
3 0

```

Figure 76 Transpose Example

12.5.21 Centroid

void global::Centroid(global B)

A centroid vector *B* is calculated for the invoking two dimensional global array. The centroid vector is the average value for each for each column of the matrix. Any previous contents of the global array named

to receive the centroid vector are lost. The invoking global array (*A*) must contain at least two dimensions. See Figure 77.

```
#include <mumpsc/libmumpscpp.h>

global A("A");
global B("B");

int main() {
    mstring i,j;
    for (i=0; i<10; i++)
        for (j=1; j<10; j++) {
            A(i,j) = 5;
        }

    A().Centroid(B());
    mstring a="";

    while (1) {
        a=B(a).Order(1);
        if (a=="") break;
        cout << a << " --> " << B(a) << endl;
    }

    return 0;
}
```

Yields:

```
1 --> 5
2 --> 5
3 --> 5
4 --> 5
5 --> 5
6 --> 5
7 --> 5
8 --> 5
9 --> 5
```

Figure 77 Centroid Example

The above yields a vector giving the average value of each named column of the matrix "A" (5 in this case since each column is initialized with 5).

12.5.22 Correlation Functions

```
void global::TermCorrelate(global B)
void global::DocCorrelate(global B, mstring fcname, double threshold)
```

These functions build document indexing correlation matrices. The invoking **global** is assumed to be a two dimensional document-term matrix whose rows are documents and whose columns represent the occurrence of terms in the documents (either weights or frequencies).

TermCorrelate() builds a square term-term correlation matrix in *B* from the invoking document-term matrix.

DocCorrelate() builds a square document-document correlation matrix from the invoking document-term matrix. The name of the function to be used in calculating the document-document similarity is given

in *fcn* and may be *Cosine*, *Jaccard*, *Dice*, or *Sim1*. The minimum correlation threshold is given in *threshold* which defaults to 0.80 if omitted.

```
#include <mumpsc/libmumpscpp.h>

global A("A");
global B("B");

int main() {
    long i,j;

    A("1", "computer") = 5;
    A("1", "data") = 2;
    A("1", "program") = 6;
    A("1", "disk") = 3;
    A("1", "laptop") = 7;
    A("1", "monitor") = 1;

    A("2", "computer") = 5;
    A("2", "printer") = 2;
    A("2", "program") = 6;
    A("2", "memory") = 3;
    A("2", "laptop") = 7;
    A("2", "language") = 1;

    A("3", "computer") = 5;
    A("3", "printer") = 2;
    A("3", "disk") = 6;
    A("3", "memory") = 3;
    A("3", "laptop") = 7;
    A("3", "USB") = 1;

    A().TermCorrelate(B());

    mstring a;
    mstring b;

    a="";

    while (1) {
        a=B(a).Order();
        if (a == "") break;
        cout << a << endl;
        b="";

        while (1) {
            b=B(a, b).Order(1);
            if (b == "") break;
            cout <<"      " << b << "(" << B(a, b) << ")" << endl;
        }

    }
    return 0;
}

Yields:
USB
  computer(1)
  disk(1)
  laptop(1)
  memory(1)
  printer(1)
computer
```

```
USB(1)
data(1)
disk(2)
language(1)
laptop(3)
memory(2)
monitor(1)
printer(2)
program(2)
data
  computer(1)
  disk(1)
  laptop(1)
  monitor(1)
  program(1)
disk
  USB(1)
  computer(2)
  data(1)
  laptop(2)
  memory(1)
  monitor(1)
  printer(1)
  program(1)
language
  computer(1)
  laptop(1)
  memory(1)
  printer(1)
  program(1)
laptop
  USB(1)
  computer(3)
  data(1)
  disk(2)
  language(1)
  memory(2)
  monitor(1)
  printer(2)
  program(2)
memory
  USB(1)
  computer(2)
  disk(1)
  language(1)
  laptop(2)
  printer(2)
  program(1)
monitor
  computer(1)
  data(1)
  disk(1)
  laptop(1)
  program(1)
printer
  USB(1)
  computer(2)
  disk(1)
  language(1)
  laptop(2)
  memory(2)
  program(1)
```



```

program
  computer(2)
  data(1)
  disk(1)
  language(1)
  laptop(2)
  memory(1)
  monitor(1)
  printer(1)

```

Figure 78 TermCorrelate Example

The example in Figure 78 gives the number of co-occurrences of each word with each other word. For example, the words "computer" and "memory" co-occur in two vectors (2 nd 3) while the words "laptop" and "computer" co-occur in all three vectors. If each vector is thought of as a document, the strength of the co-occurrences between words is a measure of similarity for indexing purposes.

```

#include <mumpsc/libmumpscpp.h>

global A("A");
global B("B");

int main() {
    long i,j;

    A("1","computer")=5;
    A("1","data")=2;
    A("1","program")=6;
    A("1","disk")=3;
    A("1","laptop")=7;
    A("1","monitor")=1;

    A("2","computer")=5;
    A("2","printer")=2;
    A("2","program")=6;
    A("2","memory")=3;
    A("2","laptop")=7;
    A("2","language")=1;

    A("3","computer")=5;
    A("3","printer")=2;
    A("3","disk")=6;
    A("3","memory")=3;
    A("3","laptop")=7;
    A("3","USB")=1;

    A().DocCorrelate(B(),"Cosine",.5);

    mstring a;
    mstring b;

    a=""

    while (1) {
        a=B(a).Order(1);
        if (a == "") break;
        cout << a << endl;
        b = "";
        while (1) {
            b = B(a, b).Order(1);
            if (b == "") break;

```

```

        cout << "    " << b << "(" << B(a, b) << ")" << endl;
    }
    }
    return 0;
}

```

Yields

```

1
    2 0.887096774193548
    3 0.741935483870968
2
    1 0.887096774193548
    3 0.701612903225806
3
    1 0.741935483870968
    2 0.701612903225806

```

Figure 79 DocCorrelate Example

The example in program in Figure 79 calculates the similarities between the document vectors according to the Cosine method.

12.5.23 IDF

```
void global::IDF(double DocCount)
```

The *IDF()* function calculates for the global array vector provided the *inverse document frequency* weight of each term. The vector should be indexed by words and have stored the number of documents in which each word occurs. The document count will be replaced by the calculated IDF value. The IDF is $\log_2(\text{DocCount}/W_n)+1$ where W_n is the number of documents in which a term appears (the document frequency). The value *DocCount* is the total number of documents present in the collection. See Figure 80.

```

#include <mumpsc/libmpscpp.h>

global a("a");

int main() {
    kill(a());
    a("now") = 2;
    a("is") = 5;
    a("the") = 6;
    a("time") = 3;
    a().IDF(4);
    a().TreePrint();
    return 0;
}

yields:

is=0.678072
now=2.000000
the=0.415037
time=1.415037

```

Figure 80 IDF Example

12.5.24 Sum

```
double global::Sum()
```

The global array nodes beneath the referenced global array are summed. Non numeric quantities are treated as zero. See Figure 81.

```
#include <mumpsc/libmpscpp.h>

global A("A");

int main() {
mstring i, j;

for (i = 1; i < 11; i++)
    for (j = 1; j < 11; j++) {
        A(i, j) = 5;
    }
cout << "Full sum: " << A().Sum() << endl;
cout << "A row sum: " << A("5").Sum() << endl;

GlobalClose;

return EXIT_SUCCESS;
}

Yields

Full sum: 500
A row sum: 50
```

Figure 81 Sum Example

12.5.25 Btree

int BTREE(int code, unsigned char * key, unsigned char * data)

BTREE() is a macro permitting direct access to the underlying btree system. The first argument, "code" is an integer indicating the operation to be performed (see below). The second argument is the key to be stored consisting of a null-terminated array printable ASCII characters. The length of the key should be no greater than one quarter of the btree block size whose default value is 8192 (i.e., max key length is about 2048 bytes in the default case). The third argument is the data to be stored with the key. It is a null-terminated string of printable ASCII characters not greater than the system defined limit STR_MAX (defaults to 4096). An empty string is interpreted as no data to be stored. Note that the second and third arguments must be **unsigned char ***. The macro returns an integer indicating success. It may also alter "key" or "data" to return values or for other purposes. The contents of "key" and "data" are not preserved across in invocation of **BTREE()** Examples of using **BTREE()** are given in *mumpsc/doc/examples/btree*.

Permitted btree operations:

1. STORE - store a key and data value in the btree; returns zero if successful, non-zero otherwise:

```
unsigned char key[] = "test key";
unsigned char data[] = "test data";
if ( BTREE(STORE, key, data) == 0 ) cout << "stored" << endl;
else cout << "not stored" << endl;
```

2. RETRIEVE - retrieve data stored with a key; returns zero if successful, non-zero otherwise:

```
unsigned char key[] = "test key";
```

```

unsigned char data[STR_MAX];

if (BTREE(RETRIEVE, key, data) == 0)
    cout << "retrieved: " << data << endl;
else cout << "not retrieved." << endl;

```

3. CLOSE - close the btree data base; returns zero:

```

unsigned char key[] = "";
unsigned char data[] = "";
BTREE(CLOSE, key, data);

```

4. XNEXT/PREVIOUS - retrieve next ascending/descending key; returns one. Value of second and third arguments become the value of the next ascending/descending key. An initial value of the empty string for the second argument will retrieve the first/last key and the value of the second argument becomes the empty string when there are no more ascending/descending values. An initial value of the empty string for the second argument will retrieve the first/last key.

```

unsigned char key[] = "";
unsigned char data[STR_MAX];

printf("\nbegin retrieve...\n");
while(1) { // retrieve keys in ascending order
    i=BTREE(XNEXT, key, data);
    if (strlen( (char *) data) == 0) break;
    cout << key << endl;
}

```

13 Overloaded global Operators

Figure 82 shows the current list of operator overloads for class **global**. Additional overloads will be added in time.

13.1 Assignment

```
global & operator=(const char *);
global & operator=(int);
global & operator=(double);
global & operator=(string);
global & operator=(global);
global & operator=(unsigned int);
global & operator=(float);
global & operator=(short);
global & operator=(unsigned short);
global & operator=(long);
global & operator=(unsigned long);
global & operator=(mstring);
```

13.2 Addition

<pre>int operator+(int); unsigned int operator+(unsigned int); long operator+(long); unsigned long operator+(unsigned long); short operator+(short); float operator+(float); unsigned short operator+(unsigned short); double operator+(double); double operator+(global); int operator+=(int); unsigned int operator+=(unsigned int); short operator+=(short); unsigned short operator+=(unsigned short); long operator+=(long); unsigned long operator+=(unsigned long); float operator+=(float); double operator+=(double);</pre>	<pre>friend int operator+(int,global); friend unsigned int operator+(unsigned int,global); friend unsigned long operator+(unsigned long,global); friend long operator+(long,global); friend short operator+(short,global); friend unsigned short operator+(unsigned short,global); friend float operator+(float,global); friend double operator+(double,global); friend int operator+=(int &,global); friend unsigned int operator+=(unsigned int,global); friend short operator+=(short,global); friend unsigned short operator+=(unsigned short,global); friend long operator+=(long,global); friend unsigned long operator+=(unsigned long,global); friend float operator+=(float,global); friend double operator+=(double,global);</pre>
---	---

13.3 Subtraction

<pre>int operator-(int); unsigned int operator-(unsigned int); long operator-(long); unsigned long operator-(unsigned long); short operator-(short); float operator-(float); double operator-(double); double operator-(global); unsigned short operator-(unsigned short);</pre>	<pre>friend int operator-(int,global); friend unsigned int operator-(unsigned int,global); friend unsigned long operator-(unsigned long,global); friend short operator-(short,global); friend long operator-(long,global); friend float operator-(float,global); friend double operator-(double,global); friend unsigned short operator-(unsigned short,global);</pre>
--	--

<pre>int operator-=(int); unsigned int operator-=(unsigned int); short operator-=(short); unsigned short operator-=(unsigned short); long operator-=(long); unsigned long operator-=(unsigned long); float operator-=(float); double operator-=(double);</pre>	<pre>friend int operator-=(int &,global); friend unsigned int operator-=(unsigned int,global); friend short operator-=(short,global); friend unsigned short operator-=(unsigned short,global); friend long operator-=(long,global); friend unsigned long operator-=(unsigned long,global); friend float operator-=(float,global); friend double operator-=(double,global);</pre>
--	--

13.4 Multiplication

<pre>int operator*(int); unsigned int operator*(unsigned int); long operator*(long); unsigned long operator*(unsigned long); short operator*(short); float operator*(float); double operator*(double); double operator*(global); unsigned short operator*(unsigned short); int operator*=(int); unsigned int operator*=(unsigned int); short operator*=(short); unsigned short operator*=(unsigned short); long operator*=(long); unsigned long operator*=(unsigned long); float operator*=(float); double operator*=(double);</pre>	<pre>friend int operator*(int,global); friend unsigned int operator*(unsigned int,global); friend long operator*(long,global); friend unsigned long operator*(unsigned long,global); friend short operator*(short,global); friend float operator*(float,global); friend double operator*(double,global); friend unsigned short operator*(unsigned short,global); friend int operator*=(int &,global); friend unsigned int operator*=(unsigned int,global); friend short operator*=(short,global); friend unsigned short operator*=(unsigned short,global); friend long operator*=(long,global); friend unsigned long operator*=(unsigned long,global); friend float operator*=(float,global); friend double operator*=(double,global);</pre>
---	---

13.5 Division

<pre>int operator/(int); unsigned int operator/(unsigned int); long operator/(long); unsigned long operator/(unsigned long); short operator/(short); unsigned short operator/(unsigned short); float operator/(float); double operator/(double); double operator/(global); int operator/=(int); unsigned int operator/=(unsigned int); short operator/=(short); unsigned short operator/=(unsigned short); long operator/=(long); unsigned long operator/=(unsigned long);</pre>	<pre>friend int operator/(int,global); friend unsigned int operator/(unsigned int,global); friend long operator/(long,global); friend unsigned long operator/(unsigned long,global); friend short operator/(short,global); friend unsigned short operator/(unsigned short,global); friend float operator/(float,global); friend double operator/(double,global); friend int operator/=(int &,global); friend unsigned int operator/=(unsigned int,global); friend short operator/=(short,global); friend unsigned short operator/=(unsigned short,global);</pre>
---	---

float operator/=(float); double operator/=(double);	friend long operator/=(long,global); friend unsigned long operator/=(unsigned long,global); friend float operator/=(float,global); friend double operator/=(double,global);
--	---

13.6 Increment/Decrement

double operator++(); double operator--(); double operator++(int); double operator--(int);	
--	--

13.7 Unary

mstring operator+() ; // unary plus mstring operator-() ; // unary minus	
---	--

13.8 Relational

int operator>(global); int operator>(int); int operator>(unsigned int); int operator>(long); int operator>(unsigned long); int operator>(short); int operator>(unsigned short); int operator>(float); int operator>(double); int operator>(char *); int operator>(string);	friend int operator>(int,global); friend int operator>(unsigned int,global); friend int operator>(long,global); friend int operator>(unsigned long,global); friend int operator>(short,global); friend int operator>(unsigned short,global); friend int operator>(float,global); friend int operator>(double,global); friend int operator>(char *,global); friend int operator>(string,global);
int operator<(global); int operator<(int); int operator<(unsigned int); int operator<(long); int operator<(unsigned long); int operator<(short); int operator<(unsigned short); int operator<(float); int operator<(double); int operator<(char *); int operator<(string); int operator<(mstring);	friend int operator<(int,global); friend int operator<(unsigned int,global); friend int operator<(long,global); friend int operator<(unsigned long,global); friend int operator<(short,global); friend int operator<(unsigned short,global); friend int operator<(float,global); friend int operator<(double,global); friend int operator<(char *,global); friend int operator<(string,global); friend int operator<(mstring,global);
int operator<=(global); int operator<=(int); int operator<=(unsigned int); int operator<=(long); int operator<=(unsigned long); int operator<=(short); int operator<=(unsigned short); int operator<=(float); int operator<=(double); int operator<=(char *); int operator<=(string);	friend int operator<=(int,global); friend int operator<=(unsigned int,global); friend int operator<=(long,global); friend int operator<=(unsigned long,global); friend int operator<=(short,global); friend int operator<=(unsigned short,global); friend int operator<=(float,global); friend int operator<=(double,global); friend int operator<=(char *,global); friend int operator<=(string,global);

<pre> int operator>=(global); int operator>=(int); int operator>=(unsigned int); int operator>=(long); int operator>=(unsigned long); int operator>=(short); int operator>=(unsigned short); int operator>=(float); int operator>=(double); int operator>=(char *); int operator>=(string); int operator==(global); int operator==(int); int operator==(unsigned int); int operator==(long); int operator==(unsigned long); int operator==(short); int operator==(unsigned short); int operator==(float); int operator==(double); int operator==(char *); int operator==(string); int operator!=(global); int operator!=(int); int operator!=(unsigned int); int operator!=(long); int operator!=(unsigned long); int operator!=(short); int operator!=(unsigned short); int operator!=(float); int operator!=(double); int operator!=(char *); int operator!=(string); </pre>	<pre> friend int operator>=(int,global); friend int operator>=(unsigned int,global); friend int operator>=(long,global); friend int operator>=(unsigned long,global); friend int operator>=(short,global); friend int operator>=(unsigned short,global); friend int operator>=(float,global); friend int operator>=(double,global); friend int operator>=(char *,global); friend int operator>=(string,global); friend int operator==(int,global); friend int operator==(unsigned int,global); friend int operator==(long,global); friend int operator==(unsigned long,global); friend int operator==(short,global); friend int operator==(unsigned short,global); friend int operator==(float,global); friend int operator==(double,global); friend int operator==(char *,global); friend int operator==(string,global); friend int operator!=(int,global); friend int operator!=(unsigned int,global); friend int operator!=(long,global); friend int operator!=(unsigned long,global); friend int operator!=(short,global); friend int operator!=(unsigned short,global); friend int operator!=(float,global); friend int operator!=(double,global); friend int operator!=(char *,global); friend int operator!=(string,global); </pre>
---	--

13.9 Casts

<pre> operator char*() ; operator int(); operator unsigned int(); operator short(); operator unsigned short(); operator long(); operator unsigned long(); operator float(); operator double(); operator mstring(); </pre>	
---	--

Figure 82 Global Array Operator Overloads

14 GTK Desktop GUI Apps

Several simplified GTK functions are included. These will allow you to create desktop GUI applications. These are functions that control GTK widgets in a graphical application.

14.1 Glade GUI Design Tool

The open source program *Glade* allows the user to design the layout of a desktop GUI app by dragging and dropping GUI widgets (buttons, text boxes, etc.) onto a canvas. Figure 83 gives an example that includes several widget types.

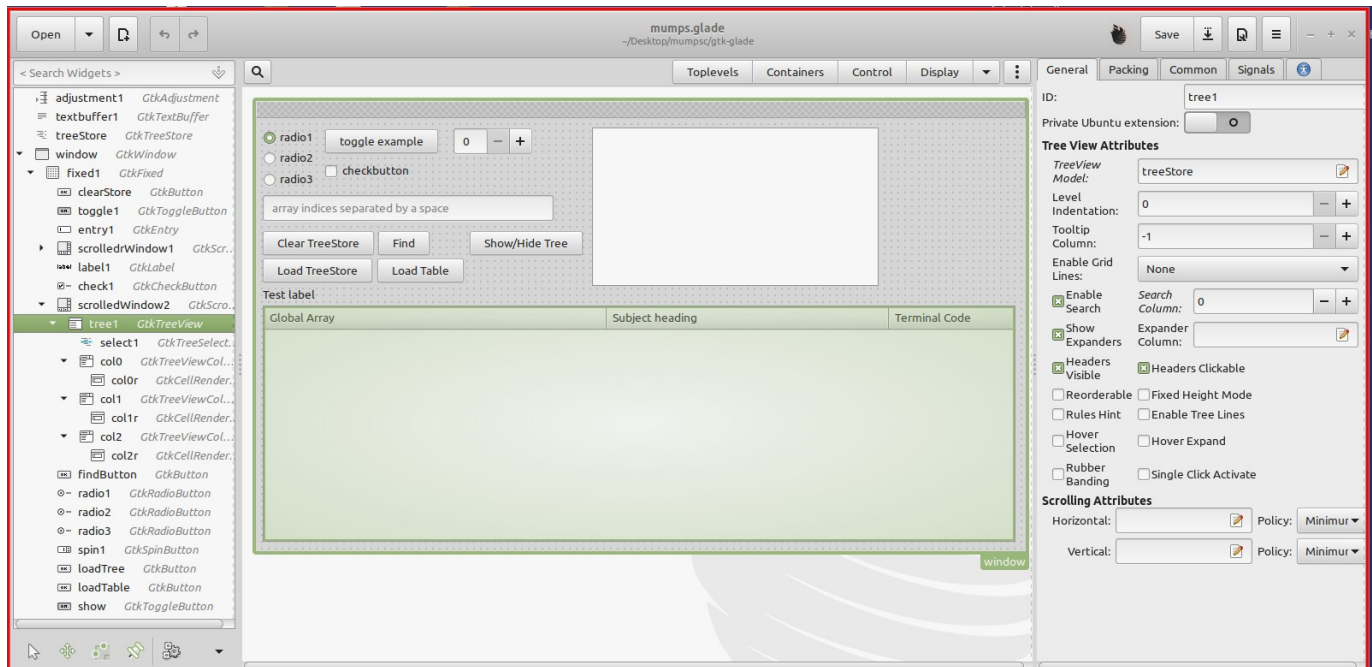


Figure 83 Glade Canvas

When you save a Glade canvas it appears in your directory as a file with the *.glade* extension. This is an XML file giving the details on your design.

Included with the Mumps distribution in the directory *gtk-glade* is a script file named *appBuild.script* and a Mumps program named *extractWidgets.mps*. The script file:

1. runs the Mumps file which reads the file *.glade* file from above and builds several files;
2. compiles (using the Mumps compiler) the file *gtk.mps* which includes the files from the previous step and creates an executable named *gtk* which will render the GUI application on the screen.

Among the files created by *extractWidgets.mps* are several files containing Mumps programs to service the actions to be performed by interacting with the on-screen GUI. There will be a file for each signal defined for each widget. The files will have names of the form:

on.widgetName.clicked.mps

where *widgetName* is the name of the widget as given in the *ID* field in the glade app and *clicked* is a signal established for that widget. The file will be invoked if the action associated with the signal is detected (for example, a button is clicked).

14.2 GTK Example

14.2.1 Glade Design Tool

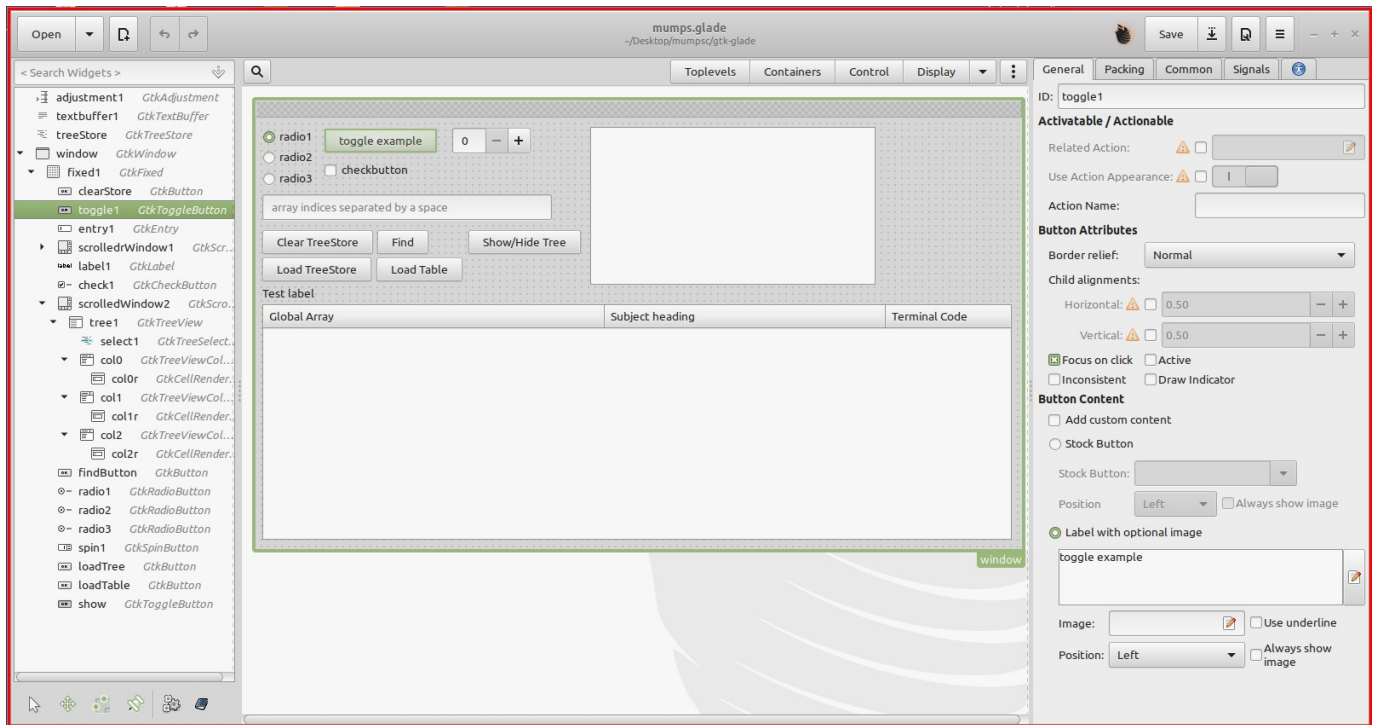


Figure 84 Toggle Button Screen 1

In Figure 84 you see the a Glade layout page. The center panel is the layout for the on-screen app that is being built. The various entities (widgets) have been dragged and dropped into their positions from widgets available in dropdown menus shown at the top named Toplevel, Containers, Control, and Display.

The leftmost panel contains the user assigned names (IDs) of the widgets along with an indication of their data types.

Some widgets are nested within others according to the display hierarchy. This, the GtkToggleButton named toggle1 is contained within the GtkFixed container named fixed1 which in turn is contained within the GtkWindow named window.

The rightmost panel contains tabs which show options for a selected widget. In this case, the selected widget is the toggle1 button which is highlighted in green in upper left of center panel and also as a row in panel one.

As can be seen in panels 1 and 3, the ID of the widget is toggle1 (user assigned), The widget is a GtkToggleButton (as seen in panel 1).

The text displayed in the button is set in panel 3 under *Label with Optional Image*. No image is assigned in this case.

Except for assigning the ID name of the widget and entering the text to appear in the button, the remainder of the options are defaults which are suitable for most ordinary applications.

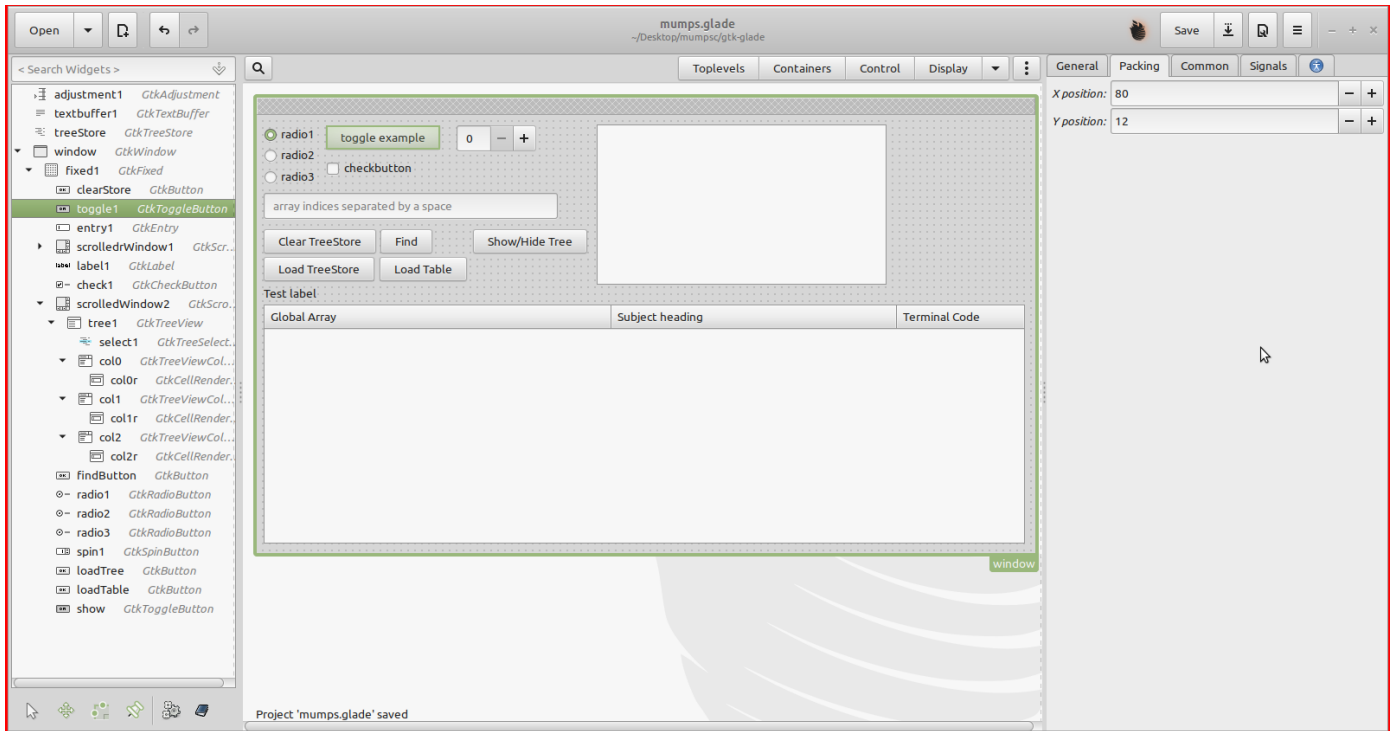


Figure 85 Toggle Button Screen 2

In Figure 85 the second tab of panel 3 has been selected. This panel determines the location of the widget within the window. Changing these numbers moves the widget accordingly.

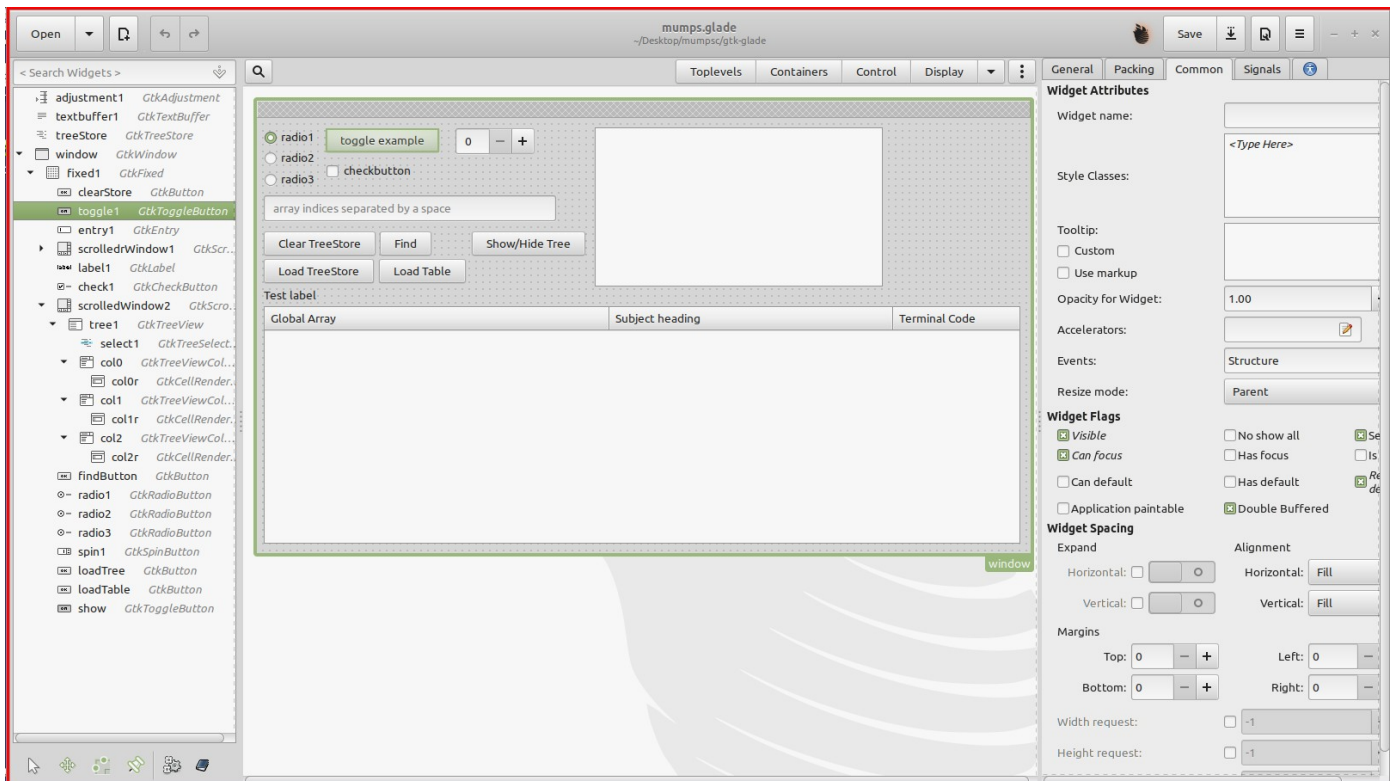


Figure 86 Toggle Button Screen 3

In the third tab of panel 3 are many adjustments all of which are defaults except for the height and width settings. These determine the size of the button. The height and width request boxes have been unchecked which causes the button to be sized to fit the contained text.

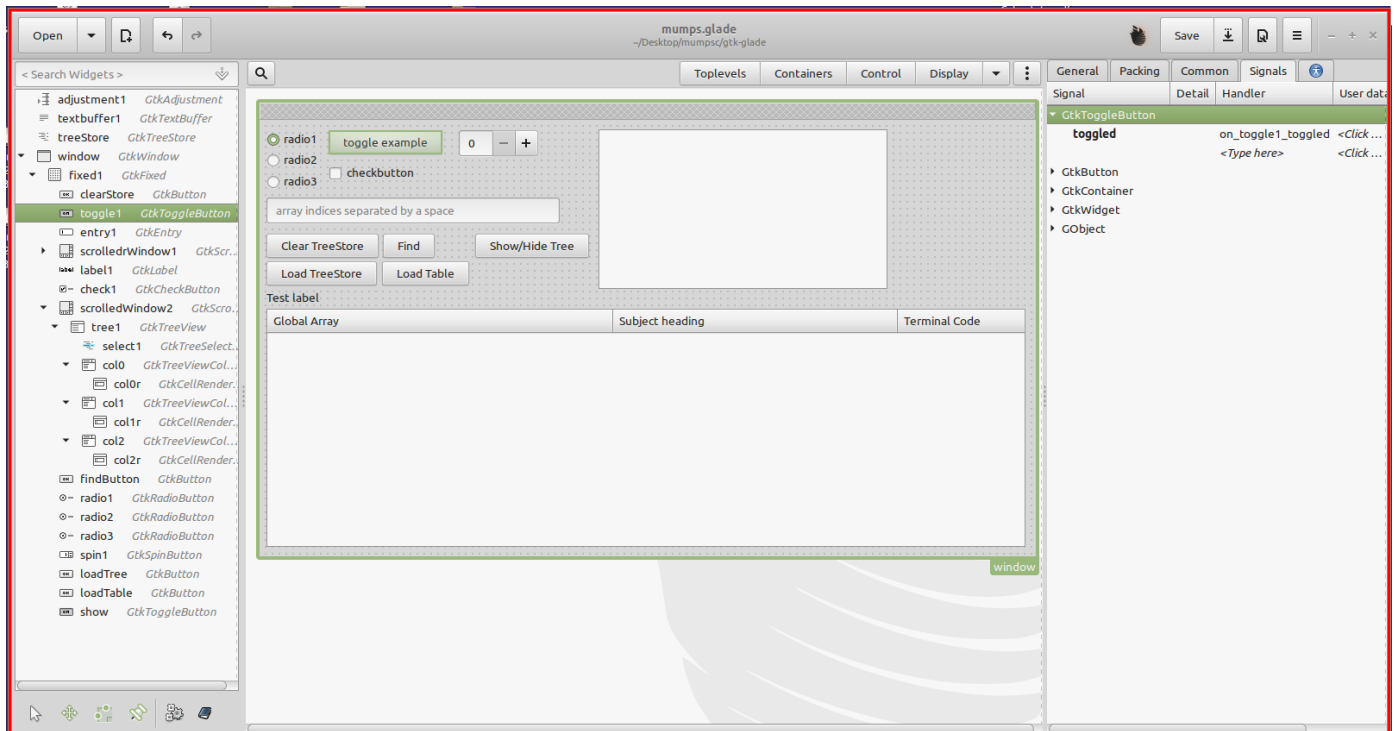


Figure 87 Toggle Button Screen 4

In Figure 87 we see the last tab of panel 3. This is the panel where you select the signals to be emitted for actions on the widget. Since this is a toggle button, the primary action is to click the button using the left button on your mouse. This action can emit a toggled signal.

If you want your program to process this signal, you enter the name of the routine to be called should the signal emit. In this case, the function named `on_toggle1_toggled` will be called if the button is clicked. The GTK GUI manager will cause the button to appear depressed or not depressed after successive clicks. Your function can determine the state of the button by using a system function.

When you save a Glade layout, it is saved as an XML file with the extension `.glade`.

14.2.2 Building A Mumps App from The Glade XML File

The disk representation of a Glade design is a XML file. For purposes of building a Mumps program from this file, the file needs to be named `mumps.glade`.

In the above we highlighted the togg1 toggle button. The Glade XML for that button looks like:

```
<child>
  <object class="GtkToggleButton" id="toggle1">
    <property name="label" translatable="yes">toggle example</property>
    <property name="visible">True</property>
    <property name="can_focus">True</property>
    <property name="receives_default">True</property>
    <signal name="toggled" handler="on_toggle1_toggled" swapped="no"/>
  </object>
</packing>
  <property name="x">80</property>
  <property name="y">12</property>
```

```

    </packing>
</child>

```

The above is a fragment of the larger Glade file which is 299 lines in length. The XML tells us that the name of the widget (*toggle1*), its data type (*GtkToggleButton*), its label contents (*toggle example*), any signals it emits (*toggled*) and the name of the signal handlers (*on_toggle1_toggled*). It also gives the location of the button on the app window and other information concerning its appearance and performance.

The distro program *extractWidgets.mps* reads the XML file and generates files that are used to compile and service an application. These are:

14.2.2.1 *gtk1.h*

This file contains C declarations for all the widgets defined in the XML file. It also includes the relevant GTK header files. In the case of the *toggle* one widget, the line:

```
GtkToggleButton *toggle1;
```

appears, among others.

14.2.2.2 *gtk2.h*

This file contains code that will invoke a Mumps signal handler (see below) for each signal emitted for a widget. In the case of the *toggle1* widget, this code looks like:

```
toggle1=GTK_TOGGLE_BUTTON(gtk_builder_get_object(builder,"toggle1"));
{ char tmp[128]; sprintf(tmp,"%p", toggle1);
  SymPut("toggle1",tmp); fprintf(f," set toggle1=\"%s\"\\n",tmp); }
```

The above code fragment which will be compiled into the base program *gtk.mps* builds the internal data structure and screen representation associated with the widget by means of *gtk_builder_object()*. This function reads the *mumps.glade* XML file information for the parameter *toggle1*. The function returns a pointer to the object which is stored in the *GtkToggleButton* pointer *toggle1* (the names of the widgets and the internal pointers as usually the same, both are *toggle1* in this case).

The string value of the pointer is stored in the Mumps symbol table (*SymPut()*) and a string containing the Mumps command or the form: *set toggle1=0x123456* is written to the file *gtk4.mps*.

14.2.2.3 *gtk3.h*

This file contains the basic signal handlers (written in C) which are used to invoke the corresponding Mumps programs which will actually handle the signal. The code for the *toggle1* widget looks like:

```
extern "C" void on_toggle1_toggled(GtkWidget *w)
{struct MSV * Ptr = AllocSV(); char tmp[512];
 sprintf(tmp,"set widget=\"%p\" g ^on.toggle1.toggled.mps",w);
 Interpret((const char *) tmp, Ptr); free(Ptr);}
```

This fragment establishes the signal handler (*on_toggle1_toggled()*), creates an instance of the Mumps state vector (*MSV *Ptr*), creates a string consisting of Mumps *set* and *goto* (*g commands*) with the string value of the widget *w* as the right hand side of the *set* command.

The subject of the *goto* command is a file named *^on.toggle1.clicked.mps* which will contain the Mumps code to process the signal.

Next, it then invokes the mumps interpreter (*Interpret()*) which executes the commands in *tmp*.

The first line specifies that the calling conventions for this function will follow C language rules. This is because the Mumps interpreter is actually a collection of C++ programs and the basic GTK library is written in C.

14.2.2.4 gtk4.h

This file is created when the actual application is run. It writes, for each widget, a Mumps set command that establishes the address of the data structure for the widget. In the case of the `toggle1` example, this looks like:

```
set toggle1="0x55ab6337e230"
```

When the Mumps signal handler is invoked, the file containing this information will be run by the signal handler thus giving the signal handler the memory references of all widgets in the application.

14.2.2.5 gtk.mps

This is the main routine that is compiled by the Mumps compiler. It will start the GTK GUI system. It looks like:

```
# Jan 30, 2022
+ #include "gtk1.h"
    zmain
+ #include "gtk2.h"
    do ^gtk4.h
+ gtk_main();
    write "Goodbye!",!
    zexit
+ #include "gtk3.h"
```

The lines that begin with a plus sign are passed directly to the C++ compiler. The function `gtk_main()` passes control to the GTK runtime routines. Return is only made upon program termination.

The first `#include` brings in the global widget declarations (in C++). The second `#include` incorporates all the builder calls which create the widgets on the screen and their associated data structured. The third `#include` brings in the C++ signal handlers for all signals used by the widgets.

14.2.2.6 on.toggle1.toggled.mps

The actual Mumps signal handler created by *extractWidgets.mps*, named *on.toggle1.toggled.mps* looks like:

```
#!/usr/bin/mumps

#      Mumps GTK Signal Handler

do ^gtk4.h
write "on.toggle1.toggled.mps", " ", widget,!
write $z~mdh~toggle~button~get~active(toggle1),!
```

The function `$z~mdh~toggle~button~get~active(toggle1)` returns 0 or 1 depending if the button is not depressed or depressed. In this case of the function, it's Mumps reference (`toggle1`) was used but the variable `widget` is also present which contains a pointer to the data structure of the widget (`toggle1` in this case) which emitted the signal.

You're on your own from here.

15 Pattern Matching

15.1 Mumps 95 Pattern Matching

Author: Matthew Lockner

Mumps 95 compliant pattern matching (the '?' operator) is implemented in this compiler/interpreter as given by the following grammar:

```
pattern      ::= {pattern_atom}
pattern_atom ::= count pattern_element
count        ::= int | '.' | '.' int | int '.' | int '.' int
pattern_element ::= pattern_code {pattern_code} | string | alternation
pattern_code  ::= 'A' | 'C' | 'E' | 'L' | 'N' | 'P' | 'U'
alternation   ::= '(' pattern_atom {',' pattern_atom} ')'
```

The largest difference between the current and previous standard is the introduction of the alternation construct, an extension that works as in other popular regular expressions implementations. It allows for one of many possible pattern fragments to match a given portion of subject text.

A string literal must be quoted. Also note that alternations are only allowed to contain pattern atoms and not full patterns; while this is a possible shortcoming, it is in accordance with the standard. It is a trivial matter to extend alternations to the ability to contain full patterns, and this may be implemented upon sufficient demand.

Pattern matching is supported by the Perl-Compatible Regular Expressions library (PCRE). Mumps patterns are translated via a recursive-descent parser in the Mumps library into a form consistent with Perl regular expressions, where PCRE then does the actual work of matching. Internally, much of this translation is simple character-level transliteration (substituting '|' for the comma in alternation lists, for example). Pattern code sequences are supported using the POSIX character classes supported in PCRE and are mostly intuitive, with the possible exception of 'E', which is substituted with `[[:print][:cntrl:]]`. Currently, this construct should cover the ASCII 7-bit character set (lower ASCII).

Due to the heavy string-handling requirements of the pattern translation process, this module uses a separate set of string-handling functions built on top of the C standard string functions, using no dynamic memory allocation and fixed-length buffers for all operations whose length is given by the constant `STR_MAX` in *sysparms.h*. If an operation overflows during the execution of a Mumps compiled binary, a diagnostic is output to *stderr* and the program terminates. If such termination occurs too frequently, simply increase the value of `STR_MAX`.

15.2 Using Perl Regular Expressions

Author: Matthew Lockner

In addition to Mumps 95 pattern matching using the '?' operator, it is also possible to perform pattern matching against Perl regular expressions via the *perlmatch* function. Support for this functionality is provided by the Perl-Compatible Regular Expressions library (PCRE), which supports a majority of the functionality found in Perl's regular expression engine.

The *perlmatch* function works in a somewhat similar fashion to the '?' operator. It is provided with a subject string and a Perl pattern against which to match the subject. The result of the function is boolean and may be used in boolean expression contexts such as the "If" statement.

Some subtleties that differ significantly from Mumps pattern matching should be noted:

1. A Mumps match expects that the pattern will match against the entire subject string, in that successful matching implies that no characters are left unmatched even if the pattern matched

against an initial segment of the subject string. Using *perlmatch*, it is sufficient that the entire Perl pattern matches an initial segment of the subject string to return a successful match.

2. The *perlmatch* function has the side effect of creating variables in the local symbol table to hold *backreferences*, the equivalent concept of \$1, \$2, \$3, ... in Perl. Up to nine backreferences are currently supported, and can be accessed through the same naming scheme as Perl (\$1 through \$9). These variables remain defined up to a subsequent call to *perlmatch*, at which point they are replaced by the backreferences captured from that invocation. Undefined backreferences are cleared between invocations; that is, if a match operation captured five backreferences, then \$6 through \$9 will contain the null string.

15.3 Examples

This program asks the user to input a telephone number. If the data entered looks like a valid telephone number, it extracts and prints the area code portion using a backreference; otherwise, it prints a failure message and exits.

```
Write "Please enter a telephone number:",!
Read phonenum

If $$^perlmatch(phonenum,"^(1-)?(\\(?\\d{3}\\)?)?(-| )?\\d{3}-?\\d{4}$") Do
. Write "+++ This looks like a phone number.",!
. Write "The area code is: ",$2,!
Else Do
. Write "--- This didn't look like a phone number.",!
```

The output of several sample runs of the program follows:

```
Please enter a telephone number:
1-123-555-4567
+++ This looks like a phone number.
The area code is: 123
```

```
Please enter a telephone number:
(123)-555-1234
+++ This looks like a phone number.
The area code is: (123)
```

```
Please enter a telephone number:
(123) 555-0987
+++ This looks like a phone number.
The area code is: (123)
```

As in Perl, sections of the regular expression contained in parentheses define what is contained in the backreferences following a match operation. The backreference variables are named in a left-to-right order with respect to the expression, meaning that \$1 is assigned the portion matched against the leftmost parenthesized section of the regular expression, with further references assigned names in increasing order. For a much more in-depth treatment of the subject of Perl regular expressions, refer to the *perlre* manpage distributed with the Perl language (also widely available online).

16 Mumps Language Basics

16.1 Mumps Syntax Warning

Mumps Syntax Warning

Mumps syntax will be discussed in detail below but it is important at this time to point out that standard Mumps code may not contain embedded blanks except within quoted strings.

In Mumps, a blank is a delimiter.

```
set var=3*x+y
set var = 3 * x + y ; blanks not allowed
```

16.2 Variables

Mumps has two types of variables: local and global.

Global variables are stored on disk and continue to exist when the program that created them terminates.

Local variables are stored in memory and disappear when the program that created them terminates.

A Mumps variable name must begin with a letter or percent sign (%) and may be followed by letters, percent signs, or numbers.

Variable names are case sensitive. The underscore (_) and dollar sign (\$) characters are not legal in variable names.

Global variable names are always preceded by a circumflex (^), local variables are not.

The contents of all Mumps variables are stored as varying length character strings. The maximum string length permitted is determined by the implementation but this number is usually at least 512 and often far larger.

In Mumps there are no data declaration statements. Variables are created as needed when a value is assigned to a variable name for the first time.

Values may be assigned to variables by either a set, merge or read command.

Variables may also be created if they appear as arguments to the new command.

Once created, local variables normally persist until the program ends or they are destroyed by a kill command. Global variables persist until destroyed by a kill command.

In its original implementation, Mumps did not have a means to pass parameters to invoked routines. Consequently, to this day, variables are, ordinarily, known to all routines.

Mumps variables are not typed.

The basic data type is string although integer, floating point and logical (true/false) operations can be performed on variables if their contents are appropriate.

The values in a string are, at a minimum, any ASCII character code between 32 to 127 (decimal) inclusive.

In Mumps, some characters outside this range can be generated in write commands with the \$char() function.

Some implementations permit additional character codings for other languages.

Variables receive values by means of the set, merge or read commands.

Array references are formed by adding a parenthesized list of indices to the variable name such as:

```
name("abc",2,3)
```

Indices may evaluate to numbers or strings or both.

Strings constants must be quoted, numeric constants need not be quoted.

Example Variables

```
set %=123 ; a scalar local variable
```

```
set ^x1("ducks")=123 ; ^ducks is a global array
```

```
set fido="123" ; Local variable
```

```
set Fido="dog"; Names are case sensitive
```

```
set x("PI")=3.1414 ; x is a local array reference
```

```
set input_dat=123 ; underscore not permitted
```

```
set $x=123 ; $ sign not permitted
```

```
set 1x=123 ; must begin with a letter or %
```

```
read ^x(100) ; read value into global array element
```

```
read %=123 ; read value into scalar
```

```
read _A ; underscore error
```

16.3 String Constants

String constants are enclosed in double quote marks (").

A double quote mark itself can be inserted into a string by placing two immediately adjacent double quote marks (""") in the string.

The single quote mark (') is the not operator with no special meaning within quoted strings.

The C/C++/Java convention of preceding some special characters by a backslash does not apply in Mumps.

```
"The seas divide and many a tide"
```

```
"123.45" (means the same as 123.45)
```

```
"Bridget O'Shaunessey? You're not making that up?"
```

```
""""The time has come,"" the walrus said."
```

```
"\"the time has come" (mismatched quotes)
```

```
'now is the time' (single quote means NOT)
```

16.4 Numeric Constants

Numbers can be integers or floating point. Quotes are optional.

```
100
1.23
123
1.23
"3.1415"
```

Some implementations permit scientific notation. Each implementation has limits on accuracy and size.

Consult implementation documentation for details.

In this version of Mumps, constants in scientific notation are a special case of strings and must be enclosed in quotes as strings and a numeric operator (such as unary +) is needed to impose a numeric interpretation on the contents:

```
> set i="123E4" set j="100E4"
> write i+j," ",+i," ",+j,!
2.23e+06 1.23e+06 1e+06
In GTM, quotes are not required
GTM>WRITE 8E6
8000000
GTM> WRITE 8E 6
.000008
```

16.5 Mixed Strings & Numeric Constants

Mumps has some peculiar ways of handling strings when they participate in numeric calculations.

If a string begins with a number but ends with trailing non-numeric characters and it is used as an operand in an arithmetic operation, only the leading numeric portion will participate in the operation. The trailing non-numeric portion will be ignored. A string not beginning with a numeric character is interpreted numerically as having the value of zero.

Numeric Interpretation of Strings

```
1+2 is evaluated as 3
"ABC"+2 is evaluated as 2
"1AB"+2 is evaluated as 3
"AA1"+2 is evaluated as 2
"1"+"2" is evaluated as 3
"" is evaluated as 0
+" 12AB" is evaluated as 12
+"123.45e4" is evaluated as 1.2345e+06
```

16.6 Logical Values

Logical values in Mumps are special cases of strings.

A numeric value of zero, any string beginning with a non-numeric character, or a string of length zero is interpreted as false.

Any numeric string value other than zero is interpreted as true.

Logical expressions yield either the digit zero (for false) or one (for true).

The result of any expression can be used as a logical operand.

The **not** operator is the single quote (')

```
1
true
0 false
"" false
"A" false
"99" true
"1A" true
"000" false
" 000" false
"+000" false
"0001" true

'1 false
'0 true
'"" true
'"A" true
'"99" false
'"1A" false
'"000" true
'" 000" true
'" +000" true
'"0001" false
```

16.7 Arrays

Arrays in Mumps come in two varieties: local and global.

Global array names are always prefixed by a circumflex (^) and are stored on disk.

They retain their values when a program terminates and, once set, can be accessed by other programs operating at the same time. They can only be deleted by the kill command.

Local arrays are destroyed when the program creating them terminates or when they are the subject of a kill command.

Local arrays are not accessible to other programs unless the other programs were invoked by the program which created the array.

Arrays (both global and local) are not declared or pre-dimensioned.

Array (both global and local) elements are created by set, merge or read statements.

The indices of an array (both global and local) are specified by a comma separated list of numbers or strings or both surrounded by parentheses.

Arrays (both local and global) are sparse. That is, if you create an element of an array, let us say element 10, it does not mean that Mumps has created any other elements.

In other words, it does not imply that there exist elements 1 through 9. You must explicitly create these if you want them.

Array indices may be positive or negative numbers, character strings, or a combination of both.

Arrays in Mumps may have multiple dimensions limited by the maximum line length (at least 512 characters).

Arrays may be viewed as either matrices or trees.

When viewed as trees, each successive index is part of a path description from the root to an internal or leaf node.

Data may be stored (or not stored) at any node along the path of a tree.

Global array names are prefixed with the circumflex character (^) and local arrays are not.

Local arrays are destroyed when the program ends while global arrays, being disk resident, persist.

```
set a(1,2,3)="text value" ; local array
set ^b(1,2,3)="text value" ; global array
```

```
set a("text string")=100 ; local array
set ^b("text string")=100 ; global array
```

```
set i="testing" set a(i)=1001 ; local array
set i="testing" set ^b(i)=1001 ; global array
```

```
set a("Iowa","Black Hawk County","Cedar Falls")="UNI"
set ^b("Iowa","Black Hawk County","Cedar Falls")="UNI"
set a("Iowa","Black Hawk County","Waterloo")="John Deere"
set ^a("Iowa","Black Hawk County","Waterloo")="John Deere"
```

```
set a[1][2][3]=123 ; brackets not used for array refs
set a(1, 2, 3)=123 ; no embedded blanks
set a[1,2,3]=123 ; brackets again
```

Array Examples

```
set a= 1ST FLEET
set b= BOSTON
set c= FLAG
set ^ship(a,b,c)="CONSTITUTION"
set ^captain(^ship(a,b,c))="JONES"
set ^home(^captain(^ship(a,b,c)))="PORTSMOUTH"
write ^ship(a,b,c) CONSTITUTION
write ^captain("CONSTITUTION") JONES
write ^home("JONES") PORTSMOUTH
write ^home(^captain("CONSTITUTION")) PORTSMOUTH
write ^home(^captain(^ship(a,b,c))) PORTSMOUTH
```

16.8 Hierarchical Data

Mumps was originally written to manage medical records which are often viewed as hierarchically organized. For that reason, the designers needed a convenient means to store data in a tree structure and developed the notion of global arrays as a result.

Arrays As Trees

In Mumps, both global and local arrays can be viewed as trees. When viewed as a tree, each successive index in an array reference is interpreted as part of a path description from the root of the array to a node. Nodes along the path may (or may not) contain data. In the diagram below, the array is named root. Numeric indices have been used for simplicity but string indices are also legal.

Data may be inserted into the tree by assignment statements. Not all nodes need have data.

```
set ^root(1,37)=1
set ^root(1,92,77)=2
set ^root(1,92,177)=3
set ^root(5)=4
set ^root(8,1)=5
set ^root(8,100)=6
set ^root(15)=7
set ^root(32,5)=8
set ^root(32,5,3)=9
set ^root(32,5,8)=10
set ^root(32,123)=11
```

String Indices

```
set ^lab(1234,hct,"05/10/2008",38)=""
set ^lab(1234,hct,"05/12/2008",42)=""
set ^lab(1234,hct,"05/15/2008",35)=""
set ^lab(1234,hct,"05/19/2008",41)=""
```

Mumps permits both numeric and string indices.

Sometimes the indices themselves are the data and nothing is actually stored at the node ("" is the empty string).

That is the case in the above where the last index values are the actual test results. Using the functions \$data() and \$order() it is easy to navigate through nodes at any level of a tree and retrieve the values of the indices.

In the code above, the Hematocrit (hct) results for patient 1234 are stored for several dates. The actual hct results for each observation are the last index value.

Access to Mumps Arrays

Mumps array nodes can be accessed directly if you know all the indices.

Alternatively, you can navigate through an array tree by means of the \$data() and \$order() builtin functions.

The first of these, \$data(), tells you if a node exists, if it has data, and if it has descendants.

The second, \$order(), is used to navigate, at a given level of a tree, from one sibling node to the next node having an alphabetically higher (or lower) index value.

For example, given:

```
set ^a(1)=100,^a(2)=200,^a(3)=300,^a(2,1)=210,^a(4,1)=410
$data(^a(1)) is 1 (node exists, has no descendant)
$data(^a(1,1)) is 0 (node does not exist)
$data(^a(2)) is 11 (node exists, has data, has descendant)
$data(^a(4)) is 10 (node exists, has no data, has descendant)
$order(^a( )) is 1 (first index at level 1)
$order(^a(1)) is 2 (next higher index)
$order(^a(2, )) is 1 (first index at level 2 beneath index 2)
$order(^a(4)) is (no more indices)
```

Building a three dimensional matrix ^mat1:

```

for i=1:1:100 do ; store values only at leaf nodes
. for j=1:1:100 do
.. for k=1:1:100 do
... set ^mat1(i,j,k)=0

```

The matrix `^mat1()` in this example is similar to a traditional three dimensional matrix in C or Fortran. There are 1,000,000 cells each with the value zero. The global array may be thought of as 100 planes of matrices each 100 by 100. Access to each element of the matrix requires three indices (`^mat1(i,j,k)`).

```

for i=1:1:100 do
; store values at all node levels
. set ^mat(i)=i
. for j=1:1:100 do
.. set ^mat(i,j)=j
.. for k=1:1:100 do
... set ^mat1(i,j,k)=k

```

This version of the matrix is best thought of as a tree of depth three. At the first level under the root, there are 100 nodes at each of which is stored a value from 1 to 100, inclusive. For each node at level 1, there are 100 descendant nodes at level two each containing a value from 1 to 100. Finally for each node at level two, there are 100 descendant nodes at level 3 likewise containing values between 1 and 100. In total, there are 100 level one nodes, 10,000 level two nodes, and 1,000,000 level three nodes. The tree contains 1,010,100 values in total (100 + 10,000 + 1,000,000). Access to a node requires one, two or three indices depending on the location of the level of the tree at which the node sought is stored.

```

for i=10:10:100 do ; sparse matrix elements missing
. for j=10:10:100 do
.. for k=10:10:100 do
... set ^mat1(i,j,k)=0

```

Note: for `i=10:10:100` means iterate with `i` beginning at 10 and incrementing by 10 up to and including 100. Thus, I will have the values 10, 20 ... 100.

The `do` without an argument causes the block following the command to be executed. Inner blocks have leading decimal points to indicate their level of depth. In this version of the matrix, the array is also a tree but, unlike the other examples, many elements do not exist.

At level one, there are only ten nodes (10, 20, 30, ... 90, 100). Each level one node has ten descendants and each level two node likewise has ten descendants.

In total, the tree has 10,110 (10 + 100 + 10,000) nodes.

For example, the node `^a(15,15,15)` does not exist. You may, however, create it with something of the form:

```
set ^mat1(15,15,15)=15
```

Now the tree has 10,111 nodes.

16.9 Mumps Commands

A Mumps program consists of a sequence of commands. Most commands have arguments that are to be executed. Some commands, however, do not have arguments.

In most cases, more than one command may appear on a line. Some example common commands that have arguments are set (assignment), for (loop control), if (conditional execution), read (input), write (output), and so forth.

The halt command, on the other hand, does not have an argument (if it does, it becomes the hang command).

Each Mumps command begins with a keyword that may, in most cases, be abbreviated. Most abbreviations are a single letter. Excessively abbreviated Mumps code is compact but difficult to read. The complete list is given below.

16.10 Postconditionals

Mumps commands may optionally be followed by what is known as a postconditional.

Postconditionals are truth-valued expressions that immediately follow a command. There are no spaces between a command and a postconditional. A postconditional is delimited from the command word (or abbreviation) by a colon.

If the postconditional expression is true, the command and its arguments are executed. If the expression is false, the command and all of its arguments are skipped and execution advances to the next command.

The following is an example of a post-conditional applied to the set command:

```
set:a=b i=2
```

The set command argument (assign 2 to variable i) will be executed only if a equals b. Some commands permit individual arguments to be postconditionalized.

One use of postconditionals of Mumps is for loop control. The scope of a for command is the current line only (although the argumentless do and blocks can effectively extend this).

A for loop with current line scope creates a problem if a loop needs to terminate for a condition not established in the loop control expression. If you attempt to use an if command to remedy the problem, because the scope of an if command is also the remainder of the line on which it appears, a similar problem exists.

For example, assume you have a global array named ^a which has an unknown number of nodes.

Assume that each node is indexed sequentially beginning with 1 (1, 2 , 3 , ...).

If you attempt to print out the values in the nodes with the following:

```
for i=1:1 write ^a(i),!
```

You will encounter an error when you attempt to access a node of the array that does not exist.

If you add an if command:

```
for i=1:1 if $data(^a(i)) quit write ^a(i),!
```

you still have a problem (single quote is the NOT operator in Mumps). The expression \$data(^a(i)) is TRUE if the node ^a(i) does NOT exist and false otherwise.

The intent is to quit the loop when there are no more nodes in ^a (note the two blanks after the quit command - since it has no arguments, two blanks are required if there is another command on the line).

However, the if command has scope of the remainder of the line on which it occurs. Thus the write will not execute if a node does exist because the expression in the if will be false (\$data(^a(i)) is true which is then made false by the not operator). Consequently, the remainder of the line is skipped.

Likewise, if the node does not exist (\$data(^a(i)) is false but becomes true because of the not operator), the loop will terminate due to the quit command. Thus, nothing at all will be printed.

An else command will not help since it will still be on the same line as the if and will be ignored if the if expression is false:

```
for i=1:1 if $data(^a(i)) quit else write ^a(i),!
```

Note the two blanks after the else.

If ^a(i) exists, the remainder of the line is skipped (including the else). If ^a(i) does not exist, the loop is terminated.

Hence the postconditional was invented.

```
for i=1:1 quit:$data(^a(i)) write ^a(i),!
```

The quit will execute only when the postconditional expression is true which occurs when there are no more nodes of the array. Problem solved!

16.11 Operator Precedence

Expressions in Mumps are evaluated strictly left-to right without precedence. If you want a different order of evaluation, you must use parentheses. This is true for all Mumps expressions in all Mumps commands.

It is a common source of error, especially in if commands with compound predicates.

For example, `a<10&b>20` really means `((a<10)&b)>20` when you probably wanted `(a<10)&(b>20)` or, equivalently, `a<10&(b>20)`.

16.12 Operators

Assignment: =

Unary Arithmetic: + -

Binary Arithmetic

- + addition
- subtraction
- * multiplication
- / full division
- \ integer division
- # modulo
- ** exponentiation

Arithmetic Relational

- > greater than
- < less than
- '> not greater / less than or equal
- '< not less / greater than or equal

String Binary

- _ concatenate

String relational operators

```
= equal
[ contains left operand contains right
] follows left operand alphabetically follows right operand
? pattern
]] Sorts after
' = not equal
'[ not contains
'] not follows
'? not pattern
']] not sorts after
```

Pattern Match Operator: ? ' ?

A for the entire upper and lower case alphabet.
C for the 33 control characters.
E for any of the 128 ASCII characters.
L for the 26 lower case letters.
N for the numerics
P for the 33 punctuation characters.
U for the 26 upper case characters.
A literal string.

The letters are preceded by a repetition count. A dot means any number. Consult documentation for more detail.

```
set A="123 45 6789"
if A?3N1" "2N1" "4N write "OK" ; writes OK
if A'?3N1" "2N1" "4N write "OK" ; writes nothing
set A="JONES, J. L."
if A?.A1", ".A write "OK" ; writes OK
if A'?A1", ".A write "OK" ; writes nothing
```

Logical operators:

- any non zero numeric value is true
- strings are false except if they have a leading non zero numeric

& and
! or
' not

```
1&1 yields 1
2&1 yields 1
1&0 yields 0
1&(0<1) yields 1
1!1 yields 1
1!0 yields 1
0!0 yields 0
2!0 yields 1
'0 yields 1
'1 yields 0
'99 yields 0
'"" yields 1
```

Indirection Operator

The indirection operator (@) causes the value of the expression to its right to be executed and the result replaces the indirect expression.

```

set a="2+2"
write @a,!

; writes 4

kill ^x
set ^x(1)=99
set ^x(5)=999
set v="^x(y)"
set y=1
set x=$order(@v) ; equivalent to ^x(1)
write x,! ; writes next index of ^x(1): 5
set v1="^x"
set x=$order(@(v1_"("_y_")"))
write x,! ; writes 5

```

16.13 Commands

break
 close
 database

Suspends execution or exits a block (nonstandard extension) release an I/O device

do

execute a program, section of code or block

else

conditional execution based on value of \$test

for

iterative execution of a line or block

goto

transfer of control to a label or program

halt

terminate execution

hang

delay execution for a specified period of time

if

conditional execution of remainder of line

job

Create an independent process

lock

Exclusive access/release named resource

kill

delete a local or global variable
merge
copy arrays
new
create new copies of local variables
open
obtain ownership of a device
quit
end a for loop or exit a block
read
read from a device
set
assign a value to a global or local variable
shell
tcommit
commit a transaction
trestart
roll back / restart a transaction
trollback
Roll back a transaction
tstart
Begin a transaction
use
select which device to read/write
view
Implementation defined
write
write to a device
xecute
dynamically execute strings
z...

implementation defined - all begin with the letter z

16.14 Syntax Rules

A line may begin with a label. If so, the label must begin in column one.

If column one has a semi-colon, the line is a comment.

If a semi-colon appears in a position where a command word could appear, the remainder of the line is a comment.

After a label there must be at least one blank or a <tab> character before the first command.

If there is no label, column one must be a blank or a <tab> character followed by some number of blanks, possibly zero, before the first command.

After most command words or abbreviations there may be an optional postconditional. No blanks or <tab> characters are permitted between the command word and the post-conditional.

If a command has an argument, there must be at least one blank after the command word and its post-conditional, if present, and the argument.

Expressions (both in arguments and post-conditionals) may not contain embedded blanks except within double-quoted strings.

If a command has no argument and it is the final command on a line, it is followed by the new line character.

If a command has an argument and it is the final command on a line, its last argument is followed by a new line character.

If a command has an argument and it is not the last command on a line, it is followed by at least one blank before the next command word.

If a command has no argument, there must be two blanks after the command word if there is another command on the line. If it is the last command on a line, it is followed by the new line character.

16.15 Syntax Extensions

In this version of Mumps:

If a line begins with a pound-sign (#) or two forward slashes (//), the remainder of the line is taken to be a comment (non-standard extension).

If a line begins with a plus-sign (+), the remainder of the line is interpreted to be an inline C/C++ statement (non-standard compiler extension).

After the last argument on a line and at least one blank (two if the command has no arguments), a double-slash (//) causes the remainder of the line to be interpreted as a comment (non-standard extension). A // may begin a line in which case the entire line is a comment.

16.16 Blocks

Originally, all Mumps commands only had line scope. That is, no command extended beyond the line on which it appeared. In later years, however, a block structure facility was added to the language.

This led to the introduction of the argumentless do command.

Originally, all do commands had arguments consisting of either a label, offset, file name or combination of these that addressed a block of code to be invoked as a subroutine.

An argumentless do command also invokes a block of code. The code invoked consists of the lines immediately following the line containing the do command if they are at a line level one greater than the line level of the do. The block ends when the line declines to the line level of the do or below.

Mumps lines are normally at line level one. A higher line level is achieved by preceding the code on a line with one or more periods. A line with one period is at line level two, one with two periods is at level three and so on.

Execution of a Mumps program normally proceeds from one level one line to the next except that if, else, and goto commands may alter flow.

If execution encounters a line at a level greater than the current level, the line is skipped unless it is entered by means of an argumentless do command.

For example:

```
1) set a=1
2) if a=1 do
3) . write "a is 1",! ; block dependent on do
4) write "hello",!
```

The do on line 2, at line level one, if executed, invokes the one line block on line 3 which is at line level two. If the do is not executed, the block consisting of line 3 is skipped and line 4 is executed after line 2.

Code at line levels greater than one should only be entered by means of an argumentless do command. The goto command should not be used to enter or exit blocks of code with line levels greater than one.

```
1) set a=1
2) if a=1 do
3) . write "a is 1",!
4) . set a=a*3
5) else do
6) . write "a is not 1",!
7) . set a=a*4
8) write "a is ",a,!
```

Because a on line 2 has a value of 1, lines 3 & 4 will be executed and lines 6 & 7 will not be executed. Line 8 is executed in either case.

```
1) if a =0 do
2) . write "a is 1",!
3) . set a=a*3
4) . if a>10 do
5) .. write "a is greater than 10",!
6) .. set a=a/2
7) . set a=a+a
```

8) write "a is ",a,!

The block beginning on line 2 is entered if variable a is not zero. Otherwise, execution skips to line 8.

The block beginning at line 5 is entered if variable a is greater than 10. Otherwise line 5 & 6 are skipped and execution resumes at line 7.

Blocks and \$Test

\$test is a builtin system variable that indicates if certain operations succeeded (true) or failed (false).

For example, if a read command fails to read data (end of file, for example), \$test will be false after the read command, otherwise, true.

The open command sets \$test to be true if a file is successfully opened, false otherwise. The if is another command sets \$test based on the result of its predicate.

Both the if and else commands have an argumentless form where execution of the command is determined by the current value in \$test.

An oddity in Mumps is that the else command is not necessarily connected with a preceding if command. The else command can be standalone. If the value in \$test is false, the remainder of the line on which the else appears is executed. If the value is true, execution skips to the next line.

For example:

```
read x
else write "end of file",! halt
```

The else executes based on the value of \$test that was set as the result of the read command.

If the read failed, the code on the line following the else will execute and the program will halt. If the read succeeded, the program will not halt.

However, in practice, an else command is often used with a preceding if command where \$test is set by the if s predicate.

In the case of an argumentless if command, the value of \$test determines if the remainder of the line is executed.

The value of \$test is restored upon exit from a block:

```
1) set a=1,b=2
2) if a=1 do ; $test becomes true
3) . set a=0
4) . if b=3 do ; $test becomes false
5) .. set b=0 ; not executed
6) . else do
7) .. set b=10 ; executed
8) . write $test," ",b,! ; $test is false
9) write $test," ",b,! ; $test is true
```

In the above, line 2 sets \$test to be 1 (the predicate is true). Lines 3 & 4 are executed.

Line 4 sets \$test to be 0 (the predicate is false). Line 5 is not executed.

Line 6 is executed and, since \$test is false, the do is executed and line 7 is executed.

Line 8 is executed with \$test as 0 and b as 10.

Line 9 is executed. \$test is restored to the value it had in line 2 (1). The value of b is 10.

16.17 Quit

A quit command in standard Mumps causes:

- 1) A single-line scope for command to terminate, or
- 2) A subroutine or function invoked by an entry reference to return, or
- 3) A code block to be exited.

A quit command without arguments requires at least two blanks after it if there are more commands on the line. This case only occurs when a quit has a postconditional. If a quit does not have a postconditional, the line it is on is terminated unconditionally.

Using quit to terminate a for command:

```
for i=1:1 quit:i>100 write i," ",i*i,!
```

When i becomes 101, the loop will terminate.

Read and write until no more input (\$test becomes 0):

```
set f=0
for read a quit:$test write a,! ; quit, when executed, ends the loop
```

The for command without arguments (note the two blanks following it) loops forever.

When there is no more data, \$test becomes 0 and the quit executes and the for terminates. If the read succeeds, \$test is 1 and the value read is written and the loop iterates.

Using quit to return from a legacy subroutine

Mumps originally used the do command to invoke a local or remote block of code which, when completed, would return to the line containing the invoking do command. This was similar to the early BASIC gosub statement.

The original Mumps do command did not allow arguments to be passed. The argument for a do command was either a label, a label with offset, a file name, or a combination of all three (some systems used different syntax):

```
do lab1 ; execute beginning at label lab1
do lab1+3 ; execute beginning at the 3rd line following lab1
do ^file.mps ; execute the contents of file.mps
do lab1^file.mps ; execute contents of file.mps beginning at lab1
do lab1+3^file.mps ; execute file.mps beginning at 3rd line from lab1
```

In each case, when the code block thus invoked ended or encountered a quit command, return was made to the invoking do command.

Note: if the invoking do command had additional similar arguments, they would now be executed in sequence.

do top ; using quit as a return from a subroutine

```

...
top set page=page+1
write #,?70,"Page ",page,!
quit ; return to invoking do command

; non standard use of break instead of quit

for do
. read a
. if '$test break ; exits the loop and the block
. write a,!

; loop while elements of array a exist
; display all nodes at level one

    for i=1:1 quit:'$data(a(i)) write a(i),!

    set i=""
    for set i=$order(^a(i)) quit:i="" write ^a(i),!

; nested inner loop quits if an element of array b has the value 99
; outer loop continues to next value of i.

    set x=0
    for i=1:1:10 for j=1:1:10 if b(i,j)=99 set x=x+1 quit

; outer loop terminates when f becomes 1 in the block

    set f=0
    for i=1:1:10 do quit:f=1
. if a(i)=99 set f=1

; The last line of the block is not executed when i>50

    set f=1
    for i=1:1:100 do
. set a(i)=i
. set b(i)=i*2
. if i>50 quit
. set c(i)=i*i

```

Later versions of Mumps permitted functions and subroutines that could pass parameters and return, in the case of functions, values. Thus, the quit command can also have an argument:

```

; returning a value from a function

set i=$$aaa(2)
write i,!
halt
aaa(x)
set x=x*x
quit x

; writes 4

```

16.18 Break

Originally, break was used as an aid in debugging. See documentation for your system to see if it is implemented.

In this version of Mumps, a break command is used to terminate a block (nonstandard). Execution continues at the first statement following the block.

```
; non standard use of break (Open Mumps)
for do
. read a
. if '$test break ; exits the loop and the block
. write a,!
```

16.19 Close

Closes and disconnects one or more I/O units. May be implementation defined. All buffers are written to output files as needed and released. No further I/O may take place to closed unit numbers until a successful open command has been issued on the unit.

The syntax of the arguments to this command varies depending on implementation.

```
close 1,2 ; closes units 1 and 2
```

16.20 Do

Executes a dependent block, or a labeled block of code either in the current program or a program on disk.

```
if a=b do ; executes the dependent block that follows
. set x=1
. write x
```

```
do abc
; executes the code block beginning at abc
... intervening code ...
```

```
abc set x=1
write x
quit ; returns to invoking do
```

```
do ^abc.mps ; invokes the code block in file abc.mps
```

```
do abc(123) ; invokes code at label abc passing an argument
```

16.21 Else

The remainder of the line is executed if \$test is false (0). \$test is builtin system variable which is set by several commands to indicate if they succeeded. No preceding if command is required. Two blanks must follow the command.

```
else write "error",! halt
```

```
else do ; executed if $test is false
. write "error",!
. halt
```

16.22 For

The for command can be iterative with the basic format:

```
for variable=start:increment:limit
```

```
for i=1:1:10 write i,! ; writes 1,2,...9,10
```

```
for i=10: 1:0 write i,! ; writes 10,9,...2,1,0
```

```
for i=1:2:10 write i,! ; writes 1,3,5,...9
```

```
for i=1:1 write i,! ; no upper limit endless
```

The for commands can be nested:

```
for i=1:1:10 write !,i,": " for j=1:1:5 write j," "
```

output:

```
1: 1 2 3 4 5
2: 1 2 3 4 5
3: 1 2 3 4 5
.
.
.
10: 1 2 3 4 5
```

A comma list of values may also be used:

```
for i=1,3,22,99 write i,! ; 1,3,22,99
```

Both modes may be mixed:

```
for i=3,22,99:1:110 write i,! ; 3,22,99,100,...110
```

```
for i=3,22,99:1 write i,! ; 3,22,99,100,...
```

With no arguments, the command becomes loop forever (two blanks required after for):

```
set i=1
for write i,! set i=1+1 quit:i>5 // 1,2,3,4,5
```

Note: two the blanks after for and do

```
set i=1
for do quit:i>5
. write i,!
. set i=i+1
```

writes 1 through 5

```
for i=1:1:10 do
. write i
. if i>5 write ! quit
. write " ",i*i,!
```

output:

```
1 1
2 4
3 9
4 16
5 25
6
7
8
9
10
```

```
for i=1:1:10 do
. write i,": "
. for j=1:1 do quit:j>5
.. write j," "
. write !
```

output:

```
1: 1 2 3 4 5 6
2: 1 2 3 4 5 6
3: 1 2 3 4 5 6
.
.
.
8: 1 2 3 4 5 6
9: 1 2 3 4 5 6
10: 1 2 3 4 5 6
```

16.23 Goto

Transfer of control to a local or remote label. Return is not made.

```
goto abc ; go to label abc in current routine
```

```
goto abc^xyz.mps ; go to label abc in file xyz.mps
```

```
goto abc:i=10,xyz:i=11 ; argument level postconditionals
```

Note: the arguments of both the do and goto commands may be individually postconditionalized as seen above. The commands themselves may be postconditionalized as well.

16.24 Halt

Terminate a program.

```
halt
```

The program terminates. Halt takes no arguments but may be postconditionalized. Two blanks are required after the command if there is another command on the line (meaningful only if the halt is postconditionalized).

16.25 Hang

Pause the program for a fixed number of seconds. Both halt and hang have the same abbreviation (h) but the hang has an argument and the halt does not. Both may be postconditionalized.

```
hang 10 ; pause for 10 seconds
```

16.26 If

```
set i=1,j=2,k=3
if i=1 write "yes",! ; yes
```

```
if i<j write "yes",! ; yes
```

```
if i<j,k>j write "yes",! ; yes
```

```
if i<j&k>j write "yes",! ; does not write
```

```
if i<j&(k>j) write "yes",! ; yes
```

```
if i write "yes",! ; yes
```

```
if 'i write "yes",! ; does not write
```

```

if '(i=0) write "yes",! ; yes

if i=0!(j=2) write "yes",! ; yes

if a>b open 1:"file,old" else write "error",! halt
    ; the else clause never executes

if write "hello world",! ; executes if $test is 1

else write "goodbye world",! ; executes if $test is 0

```

16.27 If and Else

If the keyword `if` is followed by an expression and if expression is true (evaluates to a non-zero number), the remainder of line is executed. If false, next line executed.

```
if a>b open 1:"file,old"
```

If sets `$test`. If the predicate is true, `$test` is 1, 0 otherwise.

An `if` with no arguments executes the remainder of the line if `$test` is true. An `if` with no arguments must be followed by two blanks.

The `else` command is not directly related to the `if` command.

An `else` command executes the remainder of the line if `$test` is false (0). An `else` requires no preceding `if` command. An `else` command following an `if` command on the same line will not normally execute unless an intervening command between the `if` and `else` changes `$test` to false.

16.28 Job

Creates an concurrently executing independent process. Implementation defined.

16.29 Kill

The `kill` command destroys local and global variables.

```
kill i,j,k ; removes i, j and k from the local symbol table
```

```
kill (i,j,k) ; removes all variables except i, j and k
```

```
kill a(1,2) deletes node a(1,2) and any descendants of a(1,2)
```

```
kill ^a ; deletes the entire global array ^a
```

```
kill ^a(1,2) ; deletes ^a(1,2) and any descendants of ^a(1,2)
```

16.30 Lock

Locks for exclusive access a global array node and its descendants.

```
lock ^a(1,2) ; requests ^a(1,2) and descendants for exclusive access
```

Lock may have a timeout which, if the lock is not granted, will terminate the command and report failure/success in `$test`.

Implementations vary. Lock is a voluntary signaling mechanism and does not necessarily prevent access. Consult documentation. See also: transaction processing.

16.31 Merge

Copies one array and its descendants to another.

```
merge ^a(1,2)=^b ; global array ^b and its descendants are copied
                  ; as descendants of ^a(1,2)
```

16.32 New

Creates a new copy of one or more variables pushing previous copies onto the stack. The previous copies will be restored when the block containing the new command ends.

```
if a=b do
. new a,b ; variables local to this block
. set a=10,b=20
. write a,b,!
```

; the previous values and a and b, if any, are restored.
; the versions of a and b from the block are deleted

16.33 Open, Use and Unit Numbers

The format of the open command is implementation dependent. In this Mumps, unit numbers are used. Unit 5 is always open and considered to be the console and is always open for both input and output.

In Linux terms, unit 5 is stdin and stdout. Other unit numbers are available for assignment by the open command.

In this Mumps, the format of the argument to an open command is the unit number followed by a colon followed by a string. The string must contain the file name followed by a comma followed by one of the keywords old, new, or append.

old means that the file exists and is being opened for input.

new means that the file is to be created and is being opened for output.

append means that the file is being opened for output and new data will be appended to existing data.

Open arguments vary widely depending on implementation.

The use command determines which unit will be used for input and output. Only unit 5 may be used for both.

When a unit is selected for use by the use command, that selection remains in effect until changes.

```
open 1:"aaa.dat,old" ; existing file
if '$test write "aaa.dat not found",! halt
```

```
open 2:"bbb.dat,new" ; new means create (or re create)
if '$test write "error writing bbb.dat",! halt
```

```
write "copying ...",!
for do
. use 1
; switch to unit 1
. read rec
; read from unit 1
. if '$test break
. use 2
; switch to unit 2
. write rec,! ; write to unit 2
close 1,2 ; close the open files
use 5 ; revert to console i/o
```

```

write "done",!

set in="aaa.dat,old"
set out="bbb.dat,new"

open 1:in
if '$test write "error on ",in,! halt

open 2:out
if '$test write "error on ",out,! halt

write "copying ...",!
for do
. use 1
. read rec
. if '$test break
. use 2
. write rec,!
close 1,2
use 5
write "done",!

```

16.34 I/O Format Codes

The read and write commands have basic format controls for output intended to be displayed or printed. These codes are embedded among command arguments. While they are mainly used with the write command, the read command permits a written prompt.

```

! - new line (!! means two new lines, etc.)

# - new page

?x - advance to column "x" (newline generated if needed).

```

16.35 Read

The read command reads an entire line into the variable. It may include a prompt.

Reading takes place from the current I/O unit (see \$io). Variables are created if they do not exist.

```

read a ; read a line into a

read a,^b(1),c ; read 3 lines

read !,"Name:",x ; write prompt then read into x

read *a ; read ASCII code of char typed

read a#10 ; read maximum of 10 characters

read a:5 ; read with a 5 second timeout
          ; $test will indicate if anything was read

```

16.36 Set

The set command is the basic assignment statement. Each argument consists of a variable or function reference to the left of the = operator and an expression to the right. The expression is evaluated and assigned to the variable or passed to the function.

```

set a=10,b=20,c=30

```


Expressions of the form:

```
set a=b=c=10
```

are not permitted in Mumps.

16.37 Database Transaction Commands

Some versions of Mumps permit database transaction commands. The implementation of these varies so the following commands may or may not be implemented. Check your implementation's documentation for details.

```
TCommit  
TREstart  
TRollback  
TSTART
```

In this Mumps these are not implemented. However, when used with a Sqlite3 backend store, the full range of SQL transaction controls are available.

16.38 Use

The use command selects the I/O unit to be used by the next read or write command.

This unit remains in effect for subsequent I/O activity until change by another use command.

Implementation of this command will be vendor specific.

At any given time, one I/O unit is in effect. All read and write operations default to the current unit until explicitly changed.

In this version of Mumps, unit 5 is the console and is available for input and output.

```
use 2 ; unit 2 must be open
```

16.39 View

The view command is vendor defined. It is often used for debugging or similar activities. It is not implemented in this Mumps.

16.40 Write

The write command writes text lines to the current I/O unit.

The format codes !, # and ?exp may be used to skip lines (!), skip to the top of a page(#), or indent to a specific column (?exp), respectively.

```
write "hello world",!  
  
set i="hello",j="world" write i," ",j,!  
  
set i="hello",j="world" write i,!,j,!  
  
write 1,?10,2,?20,3,?30,4,!!
```

16.41 Xecute

The xecute command is used to dynamically execute strings as though they were code.

```
set a="set b=10+456 write b"  
xecute a ; 466 is written  
  
set a="set c=""1+1"" write @c"
```

```
xecute a ; 2 is written
```

```
set b="a"
```

```
xecute @b ; 2 is written
```

```
for read x xecute x ; read and xecute input
```

16.42 Z... Commands

Commands that begin with the letter Z are implementation defined and have no standard meaning. The Z commands accepted by this version of Mumps are discussed elsewhere.

16.43 Navigating Arrays

Global (and local) arrays are navigated by means of the \$data() and \$order() functions.

The \$data() function can determine if a node exists, whether it has data assigned to it, and if it has descendants.

The \$order() permits you to move from one sibling to another at a given level of a global array tree.

The function \$data() returns a 0 if the array reference passed as a parameter to it does not exist. It returns 1 if the node exists but has no descendants, 10 if it exists, has no data but has descendants, and 11 if it exists, has data and has descendants.

The \$order() function returns the next higher (or lower) value of the last index in the array reference passed to the function.

Function \$order(), by default, returns indices in ascending collating sequence order unless a second argument of -1 is given. In this case, the indices are presented in descending collating sequence order.

\$order() returns the first value (or last value when a second argument of -1 is given) if the value of the last index of the array reference passed to it is the empty string.

It returns an empty string when there are no more nodes at this level of the tree.

```
kill ^a ; all prior values deleted  
for i=1:1:9 set ^a(i)=1 ; initialize
```

```
write $data(^a(1)) ; writes 1  
write $order(^a("")) ; writes 1  
write $order(^a(1)) ; writes 2  
write $order(^a(9)) ; writes the empty string (nothing)
```

```
set i=5  
for j=1:1:5 set ^a(i,j)=j ; initialize at level 2  
write $data(^a(5)) ; writes 11  
write $data(^a(5,1)) ; writes 1  
write $data(^a(5,15)) ; writes 0  
write $order(^a(5,"")) ; writes 1  
write $order(^a(5,2)) ; writes 3  
set ^a(10)=10  
write $order(^a(1)) ; writes 2  
write $order(^a(10)) ; writes nothing  
set ^a(11,1)=11  
write $data(^a(11)) ; writes 10  
write $data(^a(11,1)) ; writes 1
```

The following writes 1 through 5 (see data initializations above):

```
set j=""
for set j=$order(^a(5,j)) quit:j="" write j,!
```

The following writes one row per line:

```
set i=""
for do
. set i=$order(^a(i))
. if i="" break
. write "row ",i," "
. if $data(^a(i))>1 set j="" do
.. set j=$order(^a(i,j))
.. if j="" break
.. write j," " ; elements of the row on the same line
. write ! ; end of row: write new line
```

16.44 Indirection

Indirection is indicated by means of the unary indirection operator (@) which causes the string expression to its right to be executed as a code expression.

Indirection permits strings created by your program, read from a file, or loaded from a database can be interpretively evaluated and executed at runtime as expressions.

Note: The xecute command permits commands to be executed.
The indirection operator (@) is for expressions.

```
set i=2,x= 2+i
write @x,! ; 4 is written
```

```
set a=123
set b="a"
write @b,! ; 123 is written
```

```
set c="b"
write @@c,! ; 123 is written
```

```
set d="@@c+@@c"
write @d,! ; 246 is written
```

```
write @a+a,! ; 246 is written
```

```
set @("a("_a_")")=789 ; equiv to ^a(a)=789
write ^a(123),! ; 789 is written
```

```
read x write @x ; xecute the input expr as code
```

```
set a="^m1.mps" do @a ; routine m1.mps is executed
```

```
set a="b=123" set @a ; 123 is assigned to variable b
```

16.45 Subroutines

Originally, subroutines were ordinary local blocks of code in the current routine or in files of code on disk. These were (and still can be) invoked by a do command whose argument is a label indicating the first line of the code block or the name of a file or some combination of these.

With this form of subroutine invocation, there are no parameters or return values. However, the full symbol table is accessible to such a subroutine and any changes to a

variable made in a subroutine block are visible upon return. This form of subroutine call is similar to the early BASIC GOSUB statement.

Later versions of Mumps permitted parameters and return values (for functions) including call by value and call by name. These later changes to Mumps also permit the programmer to create variables local to the subroutine (using the new command) which are deleted upon exit.

In most cases, the full symbol table of variables, remains accessible to a subroutine. In all cases, all global variables are available to all routines.

```
; original style of Mumps subroutine invocation
set i=100
write i,! ; writes 100
do block1 ; invoke local code block
write i,! ; writes 200
halt
block1 set i=i+1
quit ; returns to invocation point
```

Invoking a subroutine original style:

```
do lab1 ; call local code block

do ^file1.mps ; call file containing program

do lab2^file1.mps ; call file, entry point lab2
```

Invoking a subroutine with parameters (call by value):

```
do lab2(a,b,c) ; call local label with params

do ^file2.mps(a,b,c) ; call file program with params

do lab2^file2.mps(a,b,c) ; call file with params, at entry point lab2
```

If you pass parameters, they are call by value unless you precede their names with a dot:

```
do lab3(.a,.b,.c) ; call local call by name

do ^file3.mps(.a,.b,.c) ; call file call by name
```

The following subroutine creates a variable (x) that is not destroyed on exit. The variable is accessible after return is made.

```
do two
write "expect 99 1 > ",x," ",$data(x),!
halt
two
set x=99
quit
```

Similar to the original style but the following subroutine uses the new command to create new copy of x which is deleted upon exit from the routine. The variable x is not available after return is made.

```
set y=99
do one
write "expect 99 0 > ",y," ",$data(x),!
halt
```

```

one new x
  set x=100
  write "expect 99 100 > ",y," ",x,!
quit

```

Call by value example. Parameter variable d only exists in the subroutine. The variable and any changes to it are lost on exit. The variable d does not exist after return is made.

```

do three(101)
  write "expect 0 > ",$d(d),! ; d only exists in the subroutine
three(d)
  write "expect 101 > ",d,!
quit

```

Call by name example. Modification of z in the subroutine changes x in the caller. Note the .x in the call. This signifies call by name.

```

kill
set x=33
do four(.x)
write "expect 44 > ",x,!
four(z)
  write "expect 33 > ",z,!
  set z=44
quit

```

Using the new command. Subroutine one creates x and subroutine two uses it. Changes to x in two are seen upon return to one. Variable x is destroyed upon return from subroutine one.

```

set y=99
do one
  write "expect 99 0 > ",y," ",$data(x),!
one new x
  set x=100
  write "expect 99 100 > ",y," ",x,!
  do two
  write "expect 99 99 > ",y," ",x,!
  quit
two set x=99
quit

```

16.46 Functions

A function with returns a value. The calling code variable i is not changed by the subroutine (call by value). The variable i in the function is a temporary copy.

```

set i=100
set x=$$sub(i)
write x," ",i,! ; writes 500 100
halt
sub(i)
  set i=i*5
quit i

```

16.47 Builtin Functions & Variables

Mumps has many builtin functions, called intrinsic functions, and system variables, called intrinsic variables. These handle string manipulation, tree navigation and so on.

Each function and system variable begins with a dollar sign. Some system variables

are read-only while others can be set.

While most functions appear in expressions only and yield a result, some functions may appear on the left hand side of an assignment operator or in read statements.

Uppercase characters indicate abbreviation requirements.

16.47.1 Intrinsic Special Variables

\$Device	Status of current device
\$ECode	List of error codes
\$EStack	Number of stack levels
\$ETrap	Code to execute on error
\$Horolog	days,seconds time stamp
\$Io	Current IO unit
\$Job	Current process ID
\$Key	Read command control code
\$Principal	Principal IO device
\$Quit	Indicates how current process invoked.
\$STack	Current process stack level
\$Storage	Amount of memory available
\$SYstem	System ID
\$Test	Result of prior operation
\$TLevel	Number transactions in process
\$TRestart	Number of restarts on current transaction
\$X	Position of horizontal cursor
\$Y	Position of vertical cursor
\$Z...	Implementer defined

16.47.2 Intrinsic Functions

\$Ascii()	ASCII numeric code of a character
\$Char()	ASCII character from numeric code
\$Data()	Determines variable's definition
\$Extract()	Extract a substring1
\$Find()	Find a substring
\$FNumber()	Format a number
\$Get()	Get default or actual value
\$Justify()	Format a number or string
\$Length()	Determine string length
\$Name()	Evaluate array reference
\$Order()	Find next or previous node
\$Piece()	Extract substring based on pattern1. Function may appear on LHS of assignment or in a read command.
\$QLength()	Number of subscripts in an array reference.
\$QSubscript()	Value of specified subscript
\$Query()	Next array reference
\$Random()	Random number
\$REverse()	String in reverse order
\$Select()	Value of first true argument
\$STack()	Stack information
\$Text()	String containing a line of code
\$TRanslate()	Translate characters in a string
\$View()	Implementation defined
\$Z...()	Implementation defined

16.47.2.1 \$Ascii()

\$Ascii(arg[,pos])
\$Ascii() returns the numeric equivalent of the character argument. If a second argument is given, it is the position of the character in the first argument.
\$A("A")

```
$A("Boston")  
$A("Boston",2)
```

yields 65 the ASCII code for A
yields 66 the ASCII code for B
yields 98 the ASCII code for o

16.47.2.2 \$Char()

```
$Char(nbr[, ...])  
$Char() returns a string of characters corresponding to the ASCII codes given  
as arguments.  
$C(65)  
$C(65,66,67)  
$C(65, 1,66)
```

yields A the ASCII equivalent of 65
yields ABC
yields AB invalid codes are ignored

16.47.2.3 \$Data(var)

\$Data() returns an integer which indicates whether the argument exists, has data, and descendants. The value returned is 0 if var is undefined, 1 if var is defined and has no associated array descendants; 10 if var is defined but has no associated value (but does have descendants); and 11 if var is defined and has descendants. The argument var may be either a local or scalar or global variable.

```
set A(1,11)="foo"  
set A(1,11,21)="bar"  
$data(A(1)) ; yields 10  
$data(A(1,11)) ; yields 11  
$data(A(1,11,21)) ; yields 1  
$data(A(1,11,22)) ; yields 0
```

16.47.2.4 \$Extract(e1,i2[,i3])

\$Extract() returns a substring of the first argument. The substring begins at the position noted by the second operand. Position counting begins at one.

If the third operand is omitted, the substring consists only of the i2'th character of e1.

If the third argument is present, the substring begins at position i2 and ends at position i3.

If only e1 is given, the function returns the first character of the string e1.

If i3 specifies a position beyond the end of e1, the substring ends at the end of e1.

```
$extract("ABC",2) YIELDS "B"  
$extract("ABCDEF",3,5) YIELDS "CDE"
```

16.47.2.5 \$Find(e1,e2[,i3])

\$Find() searches the first argument for an occurrence of the second argument. If one is found, the integer returned is one greater than the end position of the second argument in the first argument.

If i3 is specified, the search begins at position i3 in argument e1.

Position counting begins with one.

If the second argument is not found, the value returned is 0.

```
$find("ABC","B") YIELDS 3
$find("ABCABC","A",3) YIELDS 5
```

16.47.2.6 \$FNumber(a,b[,c])

\$FNumber() is a function used to format numbers using codes based on local currency flags. See your documentation for details.

```
$FN(100,"P")
$FN( 100,"P")
$FN( 100,"T")
$FN(10000,".2")
$FN(100,"+")
```

```
yields 100
yields (100)
yields 100
yields 10,000.00
yields +100
```

16.47.2.7 \$Get(var[,default])

\$get() returns the current value of a variable or a default value if the variable is undefined. If a default value is not specified, the empty string is used.

```
kill x
$get(x,"?")
```

```
yields ?
```

```
set x=123
$get(x,"?")
```

```
yields 123
```

16.47.2.8 \$Justify(str,fld[,dec])

\$Justify() returns a string in which the first argument is right justified in a field whose length is given by the second argument.

In the three argument form, the first argument is right justified in a field whose length is given by the second argument rounded to dec decimal places.

The three argument form imposes a numeric interpretation upon the first argument. If the field length is too small, it will be extended.

```
$justify(39,7) yields " 39"
$justify("test",7) yields " test"
$justify(39,7,1) yields " 39.0"
$justify( test ,7,2) yields 0.00
$justify(123.45,3) yields 123.45
```

16.47.2.9 \$Length(exp[,str])

\$Length() returns the length of the string (in the 1 argument form) or the number of pieces in the string delimited by the second argument.

```
set x="1234 x 5678 x 9999"
$length(x)
yields 18
```

```
$length(x,"x") yields 3 (number parts)
```


16.47.2.10 \$Name(arrayVar[,int])

`$Name()` Returns a string containing an evaluated representation of all or part of the array variable reference. It does not check to see if the array variable exists. If a second argument is given, it specifies the portion of the representation to return: if zero, the name of the array, otherwise the indices 1 through the value of the integer.

If no second argument is provided or if it exceeds the number of indices, the entire representation is returned.

```
set x=10,y=20,z=30,a= abc(x,y,z)
$na(abc(x,y,z)) yields abc("10","20","30")
$na(abc(x,y,z),1) yields abc("10")
$na(abc(x,y,z),2) yields abc("10","20")
$na(abc(10,20,25+5)) yields abc( 10 , 20 , 30 )
$na(@a) yields abc( 10 , 20 , 30 )
abc() need not exist
```

16.47.2.11 \$Order(vn[-1])

The `$Order()` function returns the next ascending or descending index at a given level of an array reference. The function traverses an array from one sibling node to the next in key ascending or descending order. The result returned is the next value of the last index of the global or local array given as the first argument to `$Order()`.

The default traversal is in key ascending order except if the optional second argument is present and evaluates to "-1" in which case the traversal is in descending key order.

If the second argument is present and has a value of "1", the traversal will be in ascending key order which is the default. In Open Mumps, numeric indices are retrieved in ASCII collating sequence order. Other systems may retrieve subscripts in numeric order. Check your documentation.

`$Order()` examples

```
for i=1:1:9 s ^a(i)=i
set ^b(1)=1
set ^b(2)= 1
write "expect (next higher) 1 ", $order(^a("")), !
write "expect (next lower) 9 ", $order(^a(""), 1), !
write "expect 1 ", $order(^a(""), ^b(1)), !
write "expect 9 ", $order(^a(""), ^b(2)), !
set i=0, j=1
write "expect 1 ", $order(^a(""), j), !
write "expect 9 ", $order(^a(""), j), !
write "expect 1 ", $order(^a(""), i+j), !
write "expect 9 ", $order(^a(""), i j), !

set i=""
write "expect 1 2 3 ... 9", !
for set i=$order(^a(i)) quit:i= write i, !

set i=""
write "expect 9 8 7 ... 1", !
for set i=$order(^a(i), 1) quit:i="" write i, !
```

16.47.2.12 \$Piece(str,pat[,i3[,i4]])

The `$Piece()` function returns a substring of the first argument delimited by the instances of the second argument.

The substring returned in the three argument case is that substring of the first argument that lies between the i3 minus one and the i3 occurrence of the second argument.

In the four argument form, the string returned is that substring of the first argument delimited by the i3 minus one instance of the second argument and the i4 instance of the second argument.

If only two arguments are given, i3 is assumed to be 1.

The function may appear on the left hand side of an assignment operator in which case the substring addressed by i3 and i4 (if present) is replaced by the result of the right hand side of the assignment operator.

```
$piece("A.BX.Y",".",2) yields "BX"
$piece("A.BX.Y",".",1) yields "A"
$piece("A.BX.Y",".") yields "A"
$piece("A.BX.Y",".",2,3) yields "BX.Y"
```

```
set x="abc.def.ghi"
set $piece(x,".",2)="xxx" causes x to be "abc.xxx.ghi"
```

16.47.2.13 \$QLength(string)

\$QLength() returns the number of subscripts contained in the in the array reference in the string argument.

```
set x= a(1,2,3)
write $qlength(x)
write $qlength( ^a(i,j) ),!
write $qlength( a ),!
writes 3, 2 and 0
```

16.47.2.14 \$QSubscript(string, int)

The \$QSubscript(string,int) function returns a portion of the array reference given by the first string. If the second integer argument is -1, the environment is returned (if defined in your implementation), if 0, the name of the global array is returned. In Open Mumps, subscripts of arrays are evaluated before being returned.

For values greater than 0, the value returned is that of the associated subscript. If a value exceeds the number of indices, an empty string is returned.

\$QSubscript() Examples

```
set i=10,j=20,k=30
set x= ^a(i,j,k)
write $qsubscript(x,0),!
write $qsubscript(x,1),!
write $qsubscript(x,2),!
write $qsubscript( ^a(i,j,k) ,3),!
writes ^a, 10, 20, and 30 respectively.
```

Note that in this version of Mumps, unlike other versions, the indices of the array are evaluated.

16.47.2.15 \$QQuery(array ref)

The \$QQuery() function returns the next array element in the array space denoted by the string argument.

The argument to \$query() is a global or local array reference (not a string like the other \$q... functions).

The value returned is a string containing the next ascending entry in the array space or, if there are no more, the empty string.

\$Query() Examples

```
set a(1,2,3)=99
set a(1,2,4)=98
set a(1,2,5)=97
set x="a"
set x=$query(@x)
write "expect a(1,2,3) ",x,!
set x=$query(@x)
write "expect a(1,2,4) ",x,!
set x=$query(@x)
write "expect a(1,2,5) ",x,!
write expect a(1,2,3) ,$query(a(1)),!
```

16.47.2.16 \$Random(int)

\$Random() returns a random number between zero and one less than the integer argument. \$random(10) yields a random number between 0 and 9

16.47.2.17 \$Reverse(str)

\$Reverse() returns the string passed as the argument in reverse order.

```
$reverse("abc")
```

yields cba

16.47.2.18 \$Select(texp1:exp1[,...])

\$Select() evaluates each truth valued expression (texp1, ...) and, if true, returns the result of the corresponding expression following the colon (:).

```
set x=10
$select(x=9:"A",x=10:"B",x=11:"C",1: ) yields B
set x=22
$select(x=9:"A",x=10:"B",x=11:"C",1: ) yields
```

16.47.2.19 \$Stack(intexp1[,...])

\$Stack() returns information concerning the Mumps stack environment based on the numeric codes supplied. Consult your documentation for details as the relate to your implementation. Not implemented in this version of Mumps.

16.47.2.20 \$Test(entryRef)

\$Test() returns a string a line of code from the the routine at the location given by the entry reference (label, offset, and/or routine).

Assume the program code:

```
L1 set a=10
  set b=20
  set c=30
; line of comment
```

```
$text(L1) yields "L1 set a=10"
$text(L1+1) yields "  set b=20"
$text(4) yields "; line of comment"
```

16.47.2.21 \$TTranslate(exp1[,exp2[,exp3])

Returns exp1 after dropping or substituting characters.

If the second and third operands are omitted, the original string is returned.

Characters from the first operand are selected if they occur in the second and (1) replaced by the character from the third operand which positionally correspond to the second operand or, (2) dropped if there is no corresponding third operand character (third operand is thus shorter than second).

```
set x="arma virumque cano"
$str(x,"a") yields "rm virumque cno"
```

```
$str(x,"a","A") yields "ArmA virumque cAno"
```

16.47.2.22 \$View()

\$View() is implementation defined.

16.47.2.23 \$Z...()

\$Z...() functions are added by the implementer and are, thus, implementation defined. See your documentation.

16.48 Programming Example

We have a file of text each very lone line of which is represents an abstract from a medical journal. The first token (tokens are delimited by a blank) on each line is a number that refers back to the original journal article. This is followed by the article number (1,2,3...). There then follow an arbitrary number of word tokens each separated from one another by a blank.

The words are the text of the abstract of the original journal article modified as follows:

- 1) All words are in lower case and punctuation, except for hyphens, has been removed.
- 2) Common words (and, or, the ...) have been removed.
- 3) The resulting lane may be very long if the original abstract was long.

There is a Mumps global array named ^Title indexed by document number that contains the original article's title.

Example line from the file:

```
5745 5 platelet affinity serotonin increased alcoholics former alcoholics biological
marker dependence? kinetics serotonin platelet uptake were studied alcoholics former
alcoholics see whether differences found between alcohol-preferring non-preferring
rats could reproduced man three groups patients were studied dependent alcoholics
admission treatment dependent alcoholics after days treatment former dependent
alcoholics abstinent years controls were non-alcoholics matched age sex km serotonin
uptake platelets lower patients from three groups compared controls this phenomenon
could congenital induced the previous excessive intake alcohol believe that this
increased platelet affinity serotonin the absence cirrhosis the liver or depression
could a marker alcohol dependence enabling therapeutic effort be focused these
patients
```

This is article 5. The contents of ^Title(5) are:

Platelet affinity for serotonin is increased in alcoholics and former alcoholics: a biological marker for dependence?

Objective: Write a Mumps program that will permit Boolean searching of words in the file.

For example, if we want to find all the articles containing both serotonin and cirrhosis, we would enter the following to our Mumps program:

serotonin & cirrhosis

The program will scan the file and locate those abstracts containing both words and respond by printing the abstract number and title of the corresponding article.

For example run:

```
$ ./slides.boolean.mps
Enter query terms serotonin & cirrhosis
Mumps expression to be evaluated on the data set:
$f(line,"serotonin")&$f(line,"cirrhosis")
```

5 Platelet affinity for serotonin is increased in alcoholics and former alcoholics

10000 documents searched
(long titles truncated)

Another Example run:

```
$ ./slides.boolean.mps
Enter query terms serotonin & platelet
Mumps expression to be evaluated on the data set:
$f(line,"serotonin")&$f(line,"platelet")
```

5 Platelet affinity for serotonin is increased in alcoholics and former alcoholics
269 Cooperative mediation by serotonin S and thromboxane A prostaglandin H receptor
2724 The effect of ketanserine on blood pressure and platelets during cardiopulmonary
3804 Hypersensitivity of phospholipase C in platelets of spontaneously hypertensive r
8399 Cerebral vasoconstrictor responses to serotonin after dietary treatment of ather

10000 documents searched
(long titles truncated)

The program:

```
1) #!/usr/bin/mumpsR0
2) # Copyright 2016 Kevin C. O'Kane
3) # boolean.mps February 14, 2014
4) # assumes that ^titles(docnbr) exists
5)
6) read "Enter query terms ",query
7)
8) set query=$zlower(query)
9) set exp=""
10)
11) for i=1:1 do
12) . set w=$piece(query," ",i)
13) . if w="" break
14) . if $find("()",w) set exp=exp_w continue
15) . if w="|" set exp=exp_"|" continue
16) . if w="~" set exp=exp_"~" continue
17) . if w="&" set exp=exp_"&" continue
18) . set exp=exp_"$f(line,"""_w""")"
19)
20) write !,"Mumps expression to be evaluated on the data set: ",exp,!!
21)
22) set $noerr=1 // turns off error messages
23) set line=" " set i=@exp // test trial of the expression
```

```

24) if $noerr<0 write "Expression error number ", $noerror,! got to again
25)
26) set M=10000
27)
28) set file="osu.converted,old"
29)
30) open 1:file
31) if '$test write "file error",! halt
32)
33) set i=0
34)
35) for j=1:1:M do
36) . use 1
37) . read line
38) . if '$test break
39) . if @exp do
40) .. set off=$piece(line," ",1)
41) .. set docnbr=$piece(line," ",2)
42) .. use 5
43) .. write docnbr,?10,$e(^title(docnbr),1,80),!
44)
45) use 5
46) write !,M," documents searched",!!
47) halt

```

The details:

The program initially reads in a query in the form of a logical expression involving words, parentheses and symbols separated by blanks. For examples:

```

word1 & word1
( word1 & word2 ) | word3
( word1 | word2 ) & ( word4 | word5 )
( word1 & word2 ) & ~ word3

```

where & means and, | means or and ~ means not. Thus, the last expression above would retrieve titles of those documents that contain both word1 and word2 but not word3.

The program initially converts all query text to lowercase using a built in function. It then extracts each blank delimited token and builds a Mumps logical expression that corresponds to the input. In place of words, the program substitutes an expression of the form:

```
$f(line,"word1")
```

The end result is an executable Mumps expression (exp). For example, the 3rd expression above would translate to:

```
($f(line,"word1")!$f(line,"word2"))&($f(line,"word4")!$f(line,"word5"))
```

The expression in exp is then tested for syntax (a special feature of the Open Mumps Interpreter):

```

set $noerr=1 // turns off error messages
set line=" " set i=@exp // test trial of the expression
if $noerr<0 write "Expression error number ", $noerror,! got to again

```

If the expression parses, the program proceeds to read (up to a limit) lines from the file of abstracts and apply the expression to each line. The builtin variable \$noerr is less than zero if the expression contains a syntax error.

If the expression parses, the program proceeds to read (up to a limit) lines from the file of abstracts and apply the expression (exp) to each line:

```
for j=1:1:M do
. use 1
. read line
. if '$test break
. if @exp do
.. set off=$piece(line," ",1)
.. set docnbr=$piece(line," ",2)
.. use 5
.. write docnbr,?10,$e(^title(docnbr),1,80),!
```

If the expression is true, that is, the words are found (or not found) by \$find() and the and / or / not logic is true, the title corresponding to the abstract number is fetched and printed.

17 Licenses

17.1 GNU Licenses

17.1.1 GNU General Public License

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

<>

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

<>

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program

with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

<>

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues),

conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

<>

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,

REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

<>

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy <name of author>
```

```
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.
```

```
<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into

proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

17.1.2 GNU Free Documentation License

GNU Free Documentation License
Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that

the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent

copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

17.1.3 GNU LESSER GENERAL PUBLIC LICENSE

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any

application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not.

Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on

which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any

such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

Copyright (C)

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

17.2 Perl Compatible Regular Expression Library License

Programs written with the MDH may call upon the Perl Compatible Regular Expression Library. In some cases, this library is distributed with the Mumps Compiler. The PCRE Library is not covered by the GNU GPL/LGPL Licenses but, rather, by the license shown below. The following is the PCRE license:

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

Written by: Philip Hazel

University of Cambridge Computing Service,
Cambridge, England. Phone: +44 1223 334714.

Copyright (c) 1997-2001 University of Cambridge

Permission is granted to anyone to use this software for any purpose on any computer system, and to redistribute it freely, subject to the following restrictions:

1. This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. In practice, this means that if you use PCRE in software which you distribute to others, commercially or otherwise, you must put a sentence like this
Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England.

somewhere reasonably visible in your documentation and in any relevant files or online help data or similar. A reference to the ftp site for the source, that is, to

<ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>
should also be given in the documentation.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software.
4. If PCRE is embedded in any software that is released under the GNU General Purpose Licence (GPL), or Lesser General Purpose Licence (LGPL), then the terms of that licence shall supersede any condition above with which it is incompatible.

The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

End

Alphabetical Index

absolute value.....	36	Exponential base 10.....	37
ACID.....	18, 27	Exponential base 2.....	37
alignment.....	50	exponential format.....	57
arc cosine.....	36	Extended Arithmetic Precision.....	57
Arc sine.....	36	Extended Precision Math.....	20
Arc tangent.....	37	extended precision software.....	19
Array Index Collating Sequence.....	65	Extract Function.....	92
ASCII.....	39	file.....	39
Ascii Function.....	91	File Names.....	65
astyle.....	57	File Names Containing Directory Information.....	65
backreferences.....	47	file pointer.....	40
Base 10 log.....	37	files.....	72
Base 2 log.....	37	Find Function.....	92
basename.....	36	floating point numbers.....	19
Bash Functions.....	36	For Command Extensions.....	61
Begin Transaction.....	27	fractional precision.....	57
BEGIN TRANSACTION;.....	30	ftello.....	41
begins Function.....	91	Function Calls.....	66
blanks.....	38, 40	Global Array Database Overview.....	22
bmg_fullsearch.....	84	Global arrays.....	23
Boyer-Moore-Gosper Function.....	46	GNU GPL/LGPL.....	73
Break and Quit.....	62	GNU MPFR library2.....	20
btree.....	39	GNU multiple precision arithmetic library1.....	20
Btree block size.....	26	Goto.....	57
buffer.....	41	Gregorian.....	38
buffers.....	39	Gregorian date.....	38
build-mumps.script.....	18	GTK Desktop GUI Apps.....	138
c_str Function.....	91	Hardware Based Math.....	19
C++ container.....	53	Horolog Function.....	92
cache hit ratio.....	39	HTML.....	39
cache size.....	25	include.....	13
cache_size.....	31	Inline C++.....	68
casting.....	27	int.....	19
centroid vector.....	43	Integer arithmetic.....	19
cgi-bin.....	39	internal buffer.....	41
Checkout.....	13	Interpreter & Compiler Implementation Notes.....	57
Class mstring.....	91	Interpreting a Mumps Program.....	32
collating sequence.....	65	Inverse Document Frequency score.....	51
command line interpreter.....	32	Jaccard.....	45
Commit.....	27	January 1, 1970.....	37
COMMIT TRANSACTION;.....	30	Job command.....	65
Compiler Error Messages.....	34	journal_mode.....	31
Compiling Large Programs.....	67	Julian date.....	38
Correlation Functions.....	51	Justify Function.....	93
Cosine.....	37, 45	Kill Command.....	61
database.....	72	left justifies.....	40
Date functions.....	37	Legacy-Mumps-Interpreter.....	13
Debian.....	15	Length Functio.....	93
decorate Function.....	91	libmpscpp.h.....	74
Dice.....	45	limit to integer precision.....	58
DocCorrelate().....	126	Linux time.....	38
document vectors.....	45	local_install.....	14
Document-Document matrix.....	53	Lock Command.....	64
Documentation.....	13	logarithm.....	37
dump.....	39	logarithms.....	37
dump file.....	39	Math Functions.....	36
EncodeHTML Function.....	91	maximum number of indices.....	27
ends Function.....	92	mcvf Function.....	93
ength Function.....	93	mdh_dialog_new_with_buttons.....	83
Eval Function.....	92	mdh_entry_get_text.....	83
Examples.....	13	mdh_entry_set_text.....	83
exponent.....	37	mdh_label_set_text.....	83
exponential.....	37	mdh_spin_button_get_value.....	84
Exponential.....	37	mdh_spin_button_set_value.....	84

mdh_text_buffer_set_text.....	83	ScanAlnum Function.....	95
mdh_toggle_button_get_active.....	83	seed.....	41
mdh_toggle_button_set_active.....	83	shell.....	41
mdh_tree_level_add.....	83	Shell Commands.....	71
mdh_tree_selection_get_selected.....	84	shred Function.....	95
mdh_tree_store_clear.....	84	ShredQuery Function.....	95
mdh_widget_show.....	83	Sim1.....	45
mmap.....	31	similarity coefficients.....	45
Modulo Operator.....	57	Similarity Functions.....	45
mstring.....	74	sine.....	37
mstring Example.....	98	Sine function.....	37
mstring Functions.....	91	Smith Waterman.....	50
mstring Operator Overloads.....	106	source code programs.....	32
Multi-Dimensional and Hierarchical Database Toolkit.....	73	SQL.....	27
multiple blanks.....	38	SQL comparisons.....	27
Mumps-Interpreter-Compiler-Library.....	13	Sqlite3.....	27, 30
Mumps-Language-Processors.....	13	Sqlite3 Command Line Interpreter.....	30
mumps-native-single-user-amd64.....	14	Sqlite3 Database Configuration.....	29
Mumps-Projects.....	13	Sqlite3 Performance Tuning.....	29
mumps-sql-db-create.....	29	square root.....	37
mumps-sqlite3-amd64.....	14	Square root.....	37
mumps.sqlite.....	18, 29	stdin.....	42
mumps2c.....	33	Stem Function.....	96
mumpsc.....	33	Stop and Synonym Function.....	53
Naked indicator.....	65	stop words.....	53
National Library of Medicine.....	121	STR_MAX.....	50
Native Database Configuration.....	25	strfmon().....	67
native global arrays.....	72	string alignment.....	50
Natural log.....	37	string replacement.....	48
natural logarithm.....	37	string search.....	46
Navigating Globals.....	79	sum.....	45
New Command.....	58	SymGet SymPut Functions.....	97
NLM.....	121	synchronous_mode.....	31
offset.....	41	synonym.....	54
open.....	31	System Software Requirements.....	15
Other Configure Options.....	20	tangent.....	37
Overloaded global Operators.....	133	Tangent function.....	37
Overloaded mstring Operators.....	103	Term-Term matrix.....	52
padding.....	40	TermCorrelate().....	126
PAGE_SHIFT.....	26	Text Processing Functions.....	45
pattern.....	48	time().....	39
Pattern Function.....	93	Token Function.....	97
pcre.....	16	Translate Function.....	98
Perl.....	47	Tree Structured Medical Record.....	22
Perl Compatible Regular Expression.....	16	ulimit.....	50
Perl Compatible Regular Expression Library License.....	201	Vector and Matrix Functions.....	43
Perl Function.....	94	void mdh_widget_hide.....	83
Piece Function.....	94	with-float_digits.....	19
Power function.....	37	word stem.....	41
pragma.....	18, 30	xecute.....	33
PRAGMA.....	31	zbasename.....	36
precision.....	20	zmain.....	33
radians.....	37	--with-ibuf.....	21
random number generator.....	41	configure prefix.....	21
readline.....	32	20, 21
ReadLine Function.....	94	--with-hardware-math=no.....	20
Relational Database Medical Record.....	23	--with-locale.....	21
replace Function.....	95	--with-.....	29
restore.....	39	--with-block.....	26
Rollback.....	27	--with-cache.....	25
ROLLBACK TRANSACTION;.....	31	--with-data_size.....	29
Rounding.....	58	--with-dbfile.....	29
Running the Mumps CLI Interpreter.....	32	--with-dbname.....	29
s_str Functio.....	97	--with-float-bits.....	20, 21
SAVEPOINT.....	30	--with-float-bits=val.....	57
Scan Functions.....	42	--with-float-digits.....	20, 21, 57

-with-hardware-math.....	19, 21	\$zIndexmax.....	35
-with-index_size.....	29	\$zIndexsize.....	35
-with-int-32.....	19	\$zlog.....	37
-with-long-double.....	19	\$zlog10.....	37
-with-maxglobal.....	25	\$zlog2.....	37
-with-no-inline.....	21	\$zlower.....	39
-with-profile.....	21	\$znative.....	31
-with-strmax.....	21	\$zNative.....	31
-with-terminate-on-error.....	21	\$zNoBlanks(arg).....	40
}.....	132	\$znormal.....	40
#!/usr/bin/mumps.....	32	\$zp.....	40
\$atan.....	37	\$zpad.....	40
\$fnumber().....	57	\$zPerlMatch(.....	47
\$Fnumber() Function.....	67	\$zpow.....	37
\$justify().....	58	\$zProgram.....	35
\$random.....	41	\$zReplace.....	48
\$select().....	67	\$zrestore.....	39
\$Select() Function.....	67	\$zrestore[.....	39
\$test.....	42	\$zseek.....	40, 41
\$z.....	30, 37, 39	\$zShred.....	48
\$z~mdh~dialog~new~with~buttons.....	55	\$zShredQuery.....	48
\$z~mdh~entry~get~text.....	55	\$zsin.....	37
\$z~mdh~entry~set~text.....	55	\$zSmithWaterman.....	50
\$z~mdh~label~set~text.....	55	\$zSqlite.....	30
\$z~mdh~spin~button~get~value.....	55	\$zSqlite("begin transaction").....	30
\$z~mdh~spin~button~set~value.....	55	\$zSqlite("commit transaction").....	30
\$z~mdh~text~buffer~set~text.....	55	\$zSqlite("pragma",option).....	31
\$z~mdh~toggle~button~get~active.....	84	\$zSqlite("rollback",[savepoint]).....	31
\$z~mdh~toggle~button~set~active.....	55	\$zSqlite("savepoint",[savepoint_name]).....	30
\$z~mdh~tree~level~add.....	55	\$zsqlOpen.....	31
\$z~mdh~tree~selection~get~selected.....	55	\$zsqr.....	37
\$z~mdh~tree~store~clear.....	55	\$zsrnd.....	41
\$z~mdh~widget~hide.....	55	\$zstem.....	41
\$z~mdh~widget~show.....	56	\$zStopInit.....	53
\$zabs.....	36	\$zStopLookup.....	53
\$zacos.....	36	\$zSynInit.....	53
\$zasin.....	36	\$zSynLookup.....	53
\$zb.....	38	\$zsystem.....	41
\$zchdir.....	38	\$ztan.....	37
\$zcos.....	37	\$ztell.....	41
\$zCurrentFile.....	38	\$zu.....	41
\$zd1.....	37	\$zwi.....	41
\$zd2.....	38	\$zwn.....	41
\$zd3.....	38	\$zwp.....	41
\$zd4.....	38	\$zws.....	41
\$zd5.....	38	\$zws(string).....	41
\$zd6.....	38	\$zz.....	43
\$zd7.....	38	\$zzAvg.....	43
\$zd8.....	38	\$zzBMGSearch.....	46
\$zDatabase.....	35	\$zzCosine.....	45
\$zDatasize.....	35	\$zzCount.....	44
\$zdate.....	37	\$zzDice.....	45
\$zDBfile.....	35	\$zzDocCorrelate.....	53
\$zDBname.....	35	\$zzInput(var).....	42, 43
\$zdump.....	39	\$zzJaccard.....	45
\$zexp.....	37	\$zzMax.....	44
\$zexp10.....	37	\$zzMin.....	44
\$zexp2(arg).....	37	\$zzMultiply.....	45
\$zfile.....	39	\$zzScan.....	42
\$zfiletest.....	36	\$zzScanAlnum.....	42
\$zflush.....	39	\$zzSim1.....	45
\$zfunctions.....	35	\$zzSoundex(s1).....	50
\$zgetenv(arg).....	39	\$zzSum.....	45
\$zGlobal(string).....	35	\$zzTermCorrelate.....	51
\$zhit.....	39	\$zzTranspose.....	45
\$zhtml.....	39		

