

# OPAL

## Programming Reference Manual Relative to version 600.04

### Contents

<b>Introduction.....</b>	<b>7</b>
What is OPAL?.....	7
<b>Recent Changes .....</b>	<b>10</b>
Aug 2020           Manual Revisions .....	10
Jul 2017 590.03 New.Read modifier:BDRECORDS .....	10
<b>Getting Started.....</b>	<b>11</b>
Learning OPAL.....	11
Basic Elements .....	12
OPAL Program Identifiers in SUPERVISOR .....	13
FLEX Programs .....	13
OPAL attributes .....	13
SUPERVISOR Programs.....	14
FLEX Programs .....	15
Stand-alone OPAL Compiler. ....	15
Supervisor Auto Enter .....	16
<b>OPAL variables .....</b>	<b>18</b>
Overview .....	18
Variable storage - the HEAP .....	18
Scope of Variables .....	19
Historical Variable handling .....	19
Enhanced Variable handling .....	19
Methods .....	20
Properties.....	21
Limits.....	22
Syntax .....	22
Variables .....	22
<string variable> .....	22
<real variable> .....	22
Dynamic OPAL variables .....	22
<dynamic string variable> .....	22
<dynamic real variable> .....	22
OPAL variable properties .....	24
PERM property .....	24
CONFIG property .....	25
GLOBAL property .....	25
Simple variable methods.....	26
<String method> .....	26
<Arithmetic method> .....	26
.ACUM arithmetic method .....	26
.COLLECT string method .....	27

.COPY string method .....	29
.CUT string method .....	30
.DELTA arithmetic method .....	31
.DISTRIBUTE string method .....	32
.FILE .READ string method .....	34
.INCLUDES boolean method .....	37
.INSERT empty string method .....	38
.REVERSE string method .....	39
.SPLIT string method .....	39
.STORE arithmetic or string method .....	42
.SUM arithmetic method .....	43
.WRITE string method .....	43
.UNZIPFILE string method .....	46
.ZIPFILE string method .....	47
Object Variable Methods .....	49
.MX .....	49
.PD .....	49
.PER .....	50
.PRINTS .....	50
.TAPEDB .....	50
.VDBS .....	50
.WHEN .....	51
<b>Expressions .....</b>	<b>53</b>
<OPAL expression> .....	53
Arithmetic Expression .....	53
<arithmetic expression> .....	53
<simple arithmetic expression> .....	53
<arithmetic operator> .....	54
<arithmetic primary> .....	54
<unsigned number> .....	54
<arithmetic attribute> .....	54
<arithmetic function> .....	54
<arithmetic assignment> .....	55
<conditional arithmetic expression> .....	55
<case arithmetic expression> .....	55
Boolean Expression .....	56
<boolean expression> .....	56
<simple boolean expression> .....	56
<boolean primary> .....	56
<boolean operator> .....	56
<arithmetic comparison> .....	57
<relational operator> .....	57
<operator list> .....	57
<string comparison> .....	57
<string relational operator> .....	58
EQW Operator .....	58
ISIN and INCL Operators .....	59
HDIS Operator .....	60
TLIS Operator .....	60
<attribute-mnemonic comparison> .....	61
<boolean case expression> .....	62
<conditional boolean expression> .....	62
Expression lists .....	62
String Expression .....	63
<string primary> .....	63
<string constant> .....	64
<string expression> .....	64
<OPAL string> .....	65
#(.) or the 'Hash-Paren' construct .....	65
<string assignment> .....	66
<conditional string expression> .....	66
<case string expression> .....	67
<b>OPAL Functions .....</b>	<b>68</b>
ABS arithmetic function .....	68
.ACCUM arithmetic method .....	68

ATTINFO String function.....	69
CLEAR string function .....	70
.COLLECT string method.....	70
COMS string function Supervisor/Trim only.....	70
.COPY string method .....	73
COUNT arithmetic function Supervisor/Trim only .....	74
COUNT and contexts .....	74
COUNT in WAIT statements .....	75
COUNT and PER considerations .....	75
COUNT and TAPEDB context.....	76
Count vs Objects.....	76
.CUT string method.....	77
DATES string function.....	78
DATETOTEXT string function .....	78
DAYNAME string function .....	80
DAYS arithmetic function .....	81
DBS string function Supervisor/Trim only .....	82
DECAT string function.....	84
DECIMAL arithmetic function .....	85
.DELTA arithmetic method .....	85
.DISTRIBUTE string method.....	86
DROP string function .....	86
DROPFIELDS string function .....	86
.FILE/.READ string method .....	87
FILEID string function .....	87
FILEIDS arithmetic function .....	88
HEAD string function .....	89
HEX arithmetic function .....	89
HEXSTRING string function .....	90
.INCLUDES boolean method .....	90
INDEXOF arithmetic function.....	91
.INSERT empty string method .....	92
INPUT string function Supervisor/Trim only.....	92
INTEGER arithmetic function.....	93
JULIAN arithmetic function .....	94
KEYIN string function.....	94
LENGTH arithmetic function .....	96
LOWER string function .....	96
MAIL arithmetic function .....	97
<header string>.....	97
MAIL semantics.....	97
MAX arithmetic function .....	99
MEMBER arithmetic function .....	99
<member target> .....	99
<member source>.....	99
<member source>.....	99
<member source>.....	100
MIN arithmetic function .....	101
MONTHNAME string function.....	101
NABS arithmetic function .....	102
NEWDATE arithmetic function .....	103
NUMBER string function.....	104
OBJECTS string function Supervisor/Trim only .....	104
OBJECTS and contexts .....	105

OBJECTS in WAIT statements .....	106
OBJECTS and PER context .....	106
OCTAL string function .....	107
PAD string function .....	107
PING integer function .....	108
POST string function .....	109
REPEAT string function .....	110
RESPOND string function Only valid in Supervisor HTTP context .....	110
.REVERSE method string method .....	111
.SPLIT string method .....	112
SQRT arithmetic function .....	112
.STORE arithmetic or string method .....	112
STRING string function .....	112
STRING8 string function .....	113
.SUM method arithmetic method .....	113
TAG string function .....	114
TAIL string function .....	114
TAKE string function .....	115
TAKEFILEIDS string function .....	115
TDADDRESS string function .....	116
TDPAGE string function .....	117
TIME string function .....	117
TIMETOTEXT string function .....	118
TRANSLATE string function .....	119
TRANSMIT string function .....	122
TRIM string function .....	122
TT string function Supervisor Only .....	123
UNZIP string function .....	125
.UNZIPFILE method string method .....	125
UPPER string function .....	126
USERFN string function .....	126
VALID boolean function .....	127
VIA function primary Supervisor/Tape Library only .....	127
<reference attribute primary> .....	127
OPAL reference attributes .....	130
WFL string function Supervisor Only .....	132
Remote File IO .....	135
.WRITE method string method .....	142
ZIP string function .....	142
.ZIPFILE method string method .....	143
<b>OPAL Attributes .....</b>	<b>144</b>
Accessing OPAL Attribute information .....	146
Attribute Types .....	148
Attribute Parameters .....	149
Attribute Context .....	149
<b>OPAL Reporting .....</b>	<b>152</b>
OPAL Strings .....	152
<OPAL string> .....	152
<string expression> .....	152
<arithmetic expression> .....	153
<mnemonic attribute> .....	153
<boolean expression> .....	154

@ character .....	154
/ character .....	154
<field width> .....	154
Lookup functions .....	157
Character Lookup function .....	158
<b>OPAL Statements .....</b>	<b>160</b>
<OPAL statement List> .....	160
Supervisor OPAL statements .....	161
FLEX OPAL statements .....	161
Common OPAL statements .....	162
Opal statement semantics .....	162
ABORT statement .....	162
CALL statement Supervisor Only .....	163
CALL DO parameters .....	164
CASE statement .....	165
<case body> .....	165
<numbered statement group> .....	166
<case number> .....	166
<string valued statement group> .....	166
<string case> .....	166
<statement list> .....	166
CHECKPOINT statement Supervisor Only .....	167
Compound (BEGIN ... END) statement .....	169
CONTINUE statement Supervisor Only .....	169
DBS Function .....	170
DISPLAY statement .....	170
DO...UNTIL statement .....	171
DUMP statement Supervisor Only .....	172
EXIT statement .....	174
IF statement .....	174
INPUT Function .....	175
KEYIN Function .....	175
LOG statement Supervisor only .....	175
MAIL Function .....	176
ODT statement Supervisor only .....	176
ODT statement command logging .....	177
ON JOBMESAGE DO statement Supervisor Only .....	178
WFL function .....	178
JOB context .....	178
Using the TEST command .....	180
WFL job identity .....	180
PING Function .....	181
PRINT Statement Supervisor Only .....	181
PRINT and Virtual Mail .....	182
QUIT statement Flex Only .....	183
RECORD statement .....	183
Recording to PORT files .....	184
Recording to disk files .....	185
Recording to disk file 0 .....	185
Recording to disk file 11 .....	185
Recording to disk files 50,51,52,53,54,55 (type VAR) .....	186
Causing RECORDER to quit .....	186
RECORD file names for DISK and VAR .....	186
RECORD file names for BYTE .....	187
RESPOND Function .....	188
SHOW statement .....	188
TAPERECORD statement Supervisor Only .....	189
TT Function .....	190
WAIT statement Supervisor Only .....	191
WAIT(<string expression>,OK) .....	191
WAIT(<boolean expression>) .....	191
WAIT(<time delay>) .....	192
WAIT(<string expression>, <task state>) .....	192
WAIT statement Flex Only .....	193

WFL Function Supervisor Only .....	193
WFL statement Flex only .....	193
WHEN/ONCE statement Supervisor Only .....	194
WHILE...DO statement .....	195
<b>Obsolete OPAL variable handling .....</b>	<b>196</b>
ACCUM arithmetic function .....	196
DELTA arithmetic function .....	197
GET arithmetic function .....	198
GETSTR string function .....	199
PUT arithmetic function .....	200
PUTSTR string function .....	201
SPLIT string function .....	202
STORE string function .....	204
SUM arithmetic function .....	206
<b>Optimisation in the OPAL Compiler .....</b>	<b>208</b>
<b>Date Formats .....</b>	<b>209</b>
<b>Date Arithmetic .....</b>	<b>212</b>
<b>Background .....</b>	<b>213</b>
<b>Copyright .....</b>	<b>215</b>

## What is OPAL?

OPAL (an acronym for **OP**erations **Al**gorithmic **L**anguage) is a Problem Oriented Language tailored to operating an A Series System. It is a simple, yet very powerful language based on the two universal languages used in operating A Series computers – Work Flow Language (WFL) and the operating system (ODT) commands. Programmers with experience in ALGOL or WFL will quickly appreciate the similarities.

OPAL works with three types of algorithms – Information Retrieval, Pattern Recognition, and Report Formatting. These three algorithm types match the previously stated function areas of an AI system: receiving input from its environment; determining an action or response; and delivering an output to its environment. OPAL has been richly endowed with powerful capabilities to make each of these algorithms very easy to program.

One line of OPAL very often corresponds to more than 100 lines of ALGOL to achieve the same function. These features make OPAL an unrivalled language for programming Operations Automation and File Directory Maintenance.

These are some of its capabilities:

- Easy syntax similar to WFL, but with extensions.
- Fully general ALGOL-like arithmetic, Boolean, and string expressions.
- IF ... THEN ... ELSE , and CASE expressions, even for strings.
- Many arithmetic functions, such as ABS or MAX.
- All WFL string functions, e.g. HEAD or TAKE.
- Extra string functions for easy Pattern Recognition e.g. ISIN, DECAT.
- For FLEX, in particular, special functions to parse file titles
- Information Retrieval about an A Series system (files, tasks, devices, etc) with two types of key words called Attributes and Mnemonics,
- Powerful TIME and DATE manipulation functions.
- Simplified methods to control datacom terminals and format screens, e.g.
- Infinite domain SET membership functions
- Report generation.

Since OPAL is used in SUPERVISOR, the TRIM Tape Library System, FLEX, and FLEX Inquiry, users have only one common language to learn for all these powerful operations tools.

The differences between OPAL in these products lies in the control structures used and the information available to them (the Attributes). Each program has a unique context that determines the Attributes available within the scope of the program.

The various contexts available to Supervisor can be shown by the command TT HELP CONTEXTS. Valid contexts as of version 540.62, are shown below:

Context	Description
AFTER	Supervisor schedule information
COMPLETED	Completed tasks or jobs
CUSTOM	Allows access to user defined contexts
DATABASE	LOG database open and close records
DMS	LOG database FREEZE and RESUME
DBACCESS	DMSII access log entries
DEFINE	Defined Opal programs
EI	Establish identity events
FILECLOSE	LOG file close records
FILEOPEN	LOG file open records
FILESTATUS	LOG file status records
HTTP	Requests from web browsers.
JOB	Limited job-tracking using ON JOBMESAGE
JOBQUEUE (SQ)	Capture of queued WFL job information
JOBREJECT	LOG tracking of rejected jobs
LOG	Generic SUMLOG records
LOGBOJ	LOG beginning-of-job records
LOGEOJ	LOG end-of-job records
LOGHTTP	LOG HTTP request
LOGOFF	LOG session log-on records
LOGON	LOG session log-off records
LOGPS	LOG PrintS event records
MAIL	Metalogic Mail program
MCSSECURITY	LOG MCS security records
METALOG	Metalogic log files
MSG	System & program display messages
MX	Task information
NAPLOG	NAPLOG events from the NAP system
OPERATOR	System commands from the ODT
PD	File information
PER	Peripheral information
PRINTS	Print System request information
SECURITY	System security violations



Context	Description
SESSIONS	COMS SESSIONS
SHOWOPEN	Open Files
SL	SLed Libraries
SMTP	Input from mail programs
SQ	System queue access
STATIONS	Datacom stations
SYSTEM	Global system information
TAPEDB	Tapes logged in the Metalogic TRIM system
TAPELABEL	Tape creation
USER	Usercode attributes from USERDATAFILE
VDBS	Database statistics and audit info
VL	Volume Library directory analyser
VOLUME	Volume status changes of tapes, packs, CDs
WHEN	Supervisor 'when' slots
WINDOWS	COMS Windows open and close actions

The number of contexts in the Supervisor environment has increased significantly over the past few releases with the arrival of the LOG contexts implementation. This range of types, including BOJ, EOJ, FILEOPEN and SECURITY, has attribute subsets that map directly onto individual log records in the SYSTEM/SUMLOG. It is likely that this range of contexts will continue to increase.

HELP CONTEXT will list all the the current context with any applicable sub contexts.

HELP ATT = : <context> will list all of the available attributes for <context>.

NAPLOG and the NSL attribute subsets within the SYSTEM context are special contexts that are for use only on Unisys NAP (Network Applications Platform) systems. For general information on these contexts, please refer to the **Metalogic Supervisor Reference** manual.

DIRECTORY is the only context permitted to the FLEX package. Supervisor's equivalent of DIRECTORY is the PD context which is much enhanced for Supervisor customers who also have a Flex license.

If a Custom attribute name is the same as an OPAL keyword or a SYSTEM attribute name, it will not be recognised. Previously each Custom Library provided a prefix to allow a synonym, to avoid this problem. Now, an underscore may be used as a prefix for an attribute name, and OPAL will strip it off before passing the attribute name to the Custom Library.

# Recent Changes

This section of the manual highlights important new features available in the OPAL compiler and run-time 'machine', in reverse-chronological order. The information will be regularly updated every time the content changes.

## Aug 2020 Manual Revisions

Updated syntax and semantics for the Coms function.

Updated semantics for the Split function and ,Split method

## Jul 2017 590.03 New.Read modifier:BDRECORDS

A new modifier, BDRECORDS, has been added to the .Read method.

`$S:=$$FL.READ(0,10,bdrecords)`

will read the first 10 printer backup records into the variable \$S. Note that a printer backup record will almost certainly contain multiple lines.

```
Ex. TT DEFINE + ODTSEQUENCE BOB_READRANGE(MSG) :  
  $F1:=Upper(Trim(text)) ;  
  Show(#recs:=$F1.PD(Lastrecord)+1) ; #next:=0 ;  
  Do Begin  
    #RD:=Min(#Recs,2) ;  
    $S1:=$$FL.READ(#next,#RD,bdrecords) ;  
    WHILE $S1 NEQ EMPTY DO  
      BEGIN  
        $LINE :=$S1.SPLIT(/) ;  
        IF "2" ISIN $LINE THEN  
          $PAGE.INSERT( #("FOUND:" , #LN.SUM(1) 3 , , TAKE($LINE,20)) , / ) ;  
      END ;  
      #Next.Sum(#rd) ;  
  End Until #Recs.Sum(-#rd) Leq 0 ; SHOW($PAGE) ;
```

The above ODTs will process a printer backup file with the title passed as a parameter. Any line in the file containing the digit 2 will be added to the variable \$page. This mechanism is needed to process very large printer backup files when the total size is too large to be held in a string variable.

# Getting Started

## Learning OPAL

This manual is intended primarily for reference, and so it is organised to reflect the structure of the language from the bottom up – not necessarily the best way for a newcomer to learn it. For users unfamiliar with OPAL, the recommended starting point is Unisys A Series Work Flow Language (WFL) because a great effort has gone into making OPAL as consistent with WFL as possible. Once the basics of WFL have been acquired, mastering OPAL is a relatively simple process of learning its extensions beyond WFL.

WFL can be learnt from Unisys training courses, the **WFL Primer** published by Gregory Publications, or the Unisys **WFL Reference Manual**. OPAL extensions are best learnt on a METALOGIC Training Course for SUPERVISOR or FLEX; by reading METALOGIC's **FLEX Inquiry** and **TRIM** manuals; or by reading example OPAL programs such as those distributed with SUPERVISOR or FLEX.

A recommended starting point for novices is the **METALOGIC SUPERVISOR Reference Manual**. This contains a useful tutorial section in **Getting Started** with many detailed and practical examples of OPAL programs. Naturally, these are all specific to SUPERVISOR but they do give a good grounding in the features available in OPAL.

This OPAL manual starts with a discussion of the basic elements of OPAL programs: Constants, Comments, and Basic Symbols. It goes on to describe how these can be juxtaposed in expressions: String, Boolean, and Arithmetic. It fully describes the specific Functions and the Attributes, which can be used in Expressions. These elements can then be further combined into OPAL Strings.

Throughout, examples are given for both FLEX and SUPERVISOR and expanded comments are given where applicable. These simple examples can be tried out, and many are useful tools in their own right.

Finally, for readers with access to SUPERVISOR or the TRIM Tape Library System, the syntax of ODTSequence statements is described.

The syntax of OPAL is defined in Unisys standard railroad diagrams. For brevity in some cases, Backus-Naur Form (BNF) has been used.

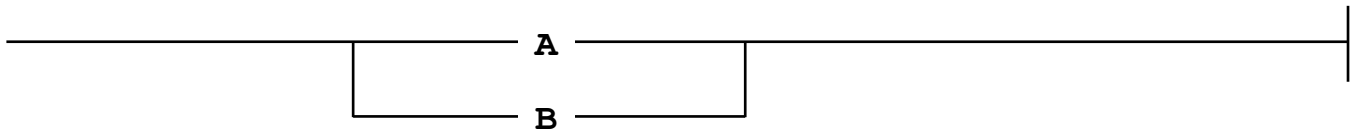
Thus the syntax described as:

```
<item> ::= A | B
```

Can be read as

```
<item> is defined as A or B.
```

In railroad this is:



Railroad diagrams are described more fully in all of the Unisys language reference manuals (such as ALGOL or WFL).

## Basic Elements

At the level of basic elements, OPAL has been modelled on WFL and is essentially identical. OPAL uses the EBCDIC character set and has the same constants. OPAL, unlike WFL, does include the lower case letters, but treats them no differently to the upper case ones. Thus in OPAL BEGIN, Begin, and BeGiN are identical.

**NOTE:** both WFL and OPAL differ from ALGOL in their string constants, using the simpler method of expressing an embedded quote by using two quote symbols together. Opal also allows the single quote ' as a string delimiter.

Example

```
"RUN *SYSTEM/DUMPALL (" "TEACH" ") "  
or  
'RUN *SYSTEM/DUMPALL ("TEACH") '
```

evaluates to:

```
RUN *SYSTEM/DUMPALL ("TEACH")
```

There are a great many identifiers known to the OPAL Language. These fall into three categories:

- Basic Symbols, such as BEGIN, IF, etc.
- Function Functions, such as MAX, HEAD, etc.
- Attributes, such as IOTIME, UNITNO, PDTITLE, etc.

Basic Symbols are largely identical to those in WFL or ALGOL. The few differences will be explained in the appropriate section on the relevant expression type involved. Notable are:

- 'curly' braces { and }
- CASE (not in WFL)
- string relational operators ISIN, INCL, HDIS, TLIS, and EQW

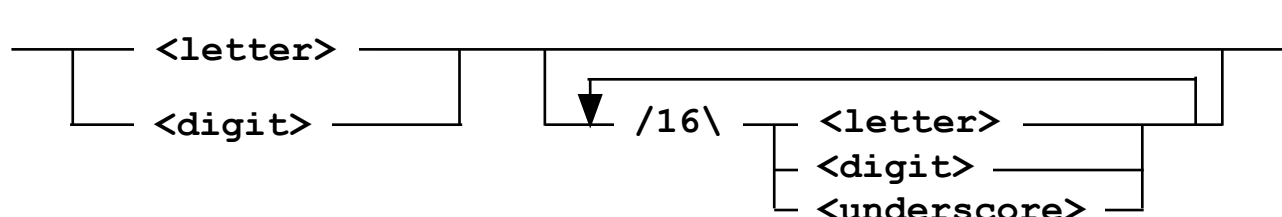
OPAL programs are input as single screen transmissions or from a JOBSYMBOL file. OPAL is record oriented, even if the source of a program is a full screen

transmission. Each logical line is 80-characters long. Reserved words and symbols cannot be split across line boundaries. Strings may be continued across line boundaries. The ALGOL/WFL <escape remark>, i.e. a trailing % character, should be used to denote comments.

## OPAL Program Identifiers in SUPERVISOR

In SUPERVISOR, OPAL programs are uniquely identified with a name (similar to a procedure name in ALGOL) up to 17 characters long that can include letters, digits or underscore. OPAL identifiers are similar to file identifiers in WFL. They can start with a letter or digit, and can have embedded underscore characters "\_" – e.g. ANY\_OLD\_ID\_1. This identifier is then stored as the name of the program and is used to list or print it.

Program Identifier



Examples

```
DEFINE + SITUATION EX_ABEND (COMPLETED) :
    EOJTYPE NEQ NORMALEOTV

DEFINE + ODTSEQUENCE EX_AUTORECOVERY:
    ODT("MIXL = 10");
    ODT("TT DO EX_ATTENDED");
```

## FLEX Programs

FLEX does not have a requirement for program identifiers in the same way as SUPERVISOR. Instead, OPAL programs are maintained as normal, external files and the normal Unisys file naming conventions apply.

## OPAL attributes

OPAL Attributes are analogous to File and Task Attributes in WFL and will be a familiar concept to anyone with knowledge of that language (the OPAL Attribute for the FILEKIND of a file is FILEKIND), but in OPAL there are Attributes for items of operational data. For example, the Attribute that gives the current amount of save core on the machine is called SAVECORE, or the reason for the last halt-load can be interrogated through HLREASON.

Just as File Attributes describe some characteristic of a file, [OPAL Attributes](#) describe some characteristic of a file, task, unit, completed entry, tape label, or the current state of the system. In many cases, the name of an OPAL Attribute is the same as the name of a File or Task Attribute. Task Attributes are a way of manipulating task related values; where an OPAL Attribute has the same name, it returns the same value.

Attributes return information in a variety of different formats: string, integer, real and mnemonic values. OPAL is very flexible at handling these various formats and has the capability to coerce, automatically, an attribute value into another OPAL format. For example, the uses of integer or real attributes inside an <OPAL string> are automatically represented as strings, without conversion by the user.

Certain timestamp OPAL attributes returned the day as an integer-valued Julian date of the form YYDDD-70000. Although the date functions, like DAYS, could recognise and process this format, its usage is not user-friendly. With the new header formats on MCP 4.2, many more timestamp attributes are now available. All the current attributes with the YYDDD-70000 format have been replaced by newer alternatives:

<b>PDTSDAY</b>	<b>replaces</b>	<b>PDTIMESTAMPDAY</b>
<b>TSDAY</b>	<b>replaces</b>	<b>TIMESTAMPDAY</b>
<b>CATALOGTSDAY</b>	<b>replaces</b>	<b>CATALOGTIMESTAMPDAY</b>
<b>CATUSEDAY</b>	<b>replaces</b>	<b>CATALOGUSEDAY</b>

Note that the above existing timestamp attributes are scheduled for de-implementation.

The Supervisor HELP ATTR command allows use of an optional OPAL context filter that enables selective searches within an attribute subset. The context must follow the attribute pattern, preceded by a ":". For example,

```
TT HELP ATTR =:TAPEDB
TT HELP ATTR =CPU=:SYSTEM
```

The supplied context must be valid from the current list of legal Supervisor contexts e.g. PER, MX, LOG, FILECLOSE, TAPEDB etc. The Flex HELP ATTR command does not have or need access to this capability.

## SUPERVISOR Programs

In SUPERVISOR, the DEFine command is used to create and maintain OPAL programs. The command has many variations and each is described in detail in the **METALOGIC SUPERVISOR Reference Manual**.

The information contained in the DEFine command is used to create the OPAL program, name it and control the execution of the OPAL compiler.

Briefly, DEFine can:

- create OPAL Programs or MEMOs

- interrogate the OPAL Directory held within SUPERVISOR's SCHEDULE
- list OPAL Programs or MEMOs
- modify OPAL Programs or MEMOs
- delete OPAL Programs or MEMOs

## FLEX Programs

In FLEX, two commands are used to create and maintain OPAL programs. These are SELECT and REPORT. SELECT is equivalent to a SUPERVISOR SITUATION while REPORT is equivalent to a SUPERVISOR DISPlay. Both commands are described in detail in the **METALOGIC FLEX Reference Manual**.

## Stand-alone OPAL Compiler.

As an aid to OPAL program development for SUPERVISOR users, it is possible to run the OPAL Compiler as a stand-alone program through CANDE (just like ALGOL or WFL). In this way, OPAL programs can be developed and compiled for syntax prior to testing within the SUPERVISOR or FLEX environment. Once a workfile has been compiled clean, it can be ENTERed through SUPERVISOR for testing.

Only JOBSYMBOL files can be used with the OPAL compiler.

Example

```
G ODTs/TCPIP_STATUS
#WORKFILE ODTs/TCPIP_STATUS: JOB, 8 RECORDS, SAVED
PA
```

```
NEXT+ .....1.....2.....3.....4.....5.....6.....7...
00000100  DEFINE + ODTSEQUENCE TCPIP_STATUS:
00000200Show([FF]);
00000300SHOW(KEYIN("NW TCPIP TCPIPHOSTNAME"),/,
00000400 KEYIN("NW TCPIP STATUS"),/,KEYIN("NW TCPIP TCPIPIDENTITY"),/,/
00000500 KeXin("NW CNS"),/,
00000600 Keyin("NA RES STATUS"),/,KeyIn("NA RES SERVER"),/,/,
00000700 Keyin("NW NAMESERVICE"),/,/,Keyin("NW BNA"))
00000800\
#DISPLAY COMPLETE
```

```
C WITH OPAL
#UPDATING
#COMPILING 43482
#?
** Compiling ODTs TCPIP_STATUS **
KeXin("NW CNS"),/,
00000500 Right parenthesis expected - KEXIN
KeXin("NW CNS"),/,
00000500 Unrecognised construct - (
Compile failed: Error count 2
#SNTX
#ET=0.3 PT=0.1 IO=0.0
```



Since OPAL does not generate native Clearpath code, the workfile object produced by a CANDE compile is made unavailable. This will result in a "LOST WORKFILE" message in CANDE whenever the source file is saved.

Example:

```
?ON CANDE
G ODTs/TCPIP
#WORKFILE ODTs/TCPIP: JOB, 8 RECORDS, SAVED
```

```
c with opal
#UPDATING
#COMPILING 43484
#?
  ** Compiling ODTs TCPIP_STATUS **
Compiled Ok.. 6 lines, 213 bytes code, 5 sectors
#ET=0.3 PT=0.0 IO=0.0
SA
#ERROR: LOST WORKFILE: (BOB) CANDE/CODE1870 ON DEV
#
```

Please note that the source file is still saved correctly.

From SUPERVISOR window:

```
enter from (BOB)odts/tcpip_status
RB(43490):Rebuild:Processed, 8 records, 1 command
RB(43490):Rebuild:1 Define Found, 1 Compiled
ENTER COMPLETED
```

## Supervisor Auto Enter

The standalone permits the automatic compilation of all SUPERVISOR defines in a source file directly into the SCHEDULE file. This is implemented by calling an external procedure in SUPERVISOR which now freezes as a BYTITLE library. Previously, after editing and test compilation, the source file would have to be manually ENTERed from a SUPERVISOR window, MARC or ODT.

To auto-enter DEFINES into the SCHEDULE, the compiler must be executed with the title of its CODE file set to values other than "CODE." or "CANDE/CODEnnnn." and have Compiler Target set to 255, or have the title of the code file set to "AUTO/ENTER".

The code file title must be AUTO/ENTER, not OBJECT/AUTO/ENTER.



From CANDE, this would be achieved by:

```
C WITH OPAL AS $AUTO/ENTER
#UPDATING
#COMPILING 1381
#?
  ** Compiling ODTS BOB **
Compiled Ok.. 5 lines, 59 bytes code, 4 sectors
  ** Compiling ODTS BOB9 **
Compiled Ok.. 5 lines, 59 bytes code, 4 sectors
2 DEFINE(S) processed, 0 DEFINE(S) failed, 0 Error(s) found, 0 Warning(s)
  -- Entering DEFINES into live SCHEDULE --
RB(1420):Rebuild:Processed, 14 records, 2 commands
RB(1420):Rebuild:2 Defines Found, 2 Compiled
#ET=0.9 PT=0.1 IO=0.1
```

Or

```
C WITH OPAL AS $X;COMPILER TARGET=255
#COMPILING 1418
#?
  ** Compiling ODTS BOB **
Compiled Ok.. 5 lines, 59 bytes code, 4 sectors
  ** Compiling ODTS BOB9 **
Compiled Ok.. 5 lines, 59 bytes code, 4 sectors
2 DEFINE(S) processed, 0 DEFINE(S) failed, 0 Error(s) found, 0 Warning(s)
  -- Entering DEFINES into live SCHEDULE --
RB(1420):Rebuild:Processed, 14 records, 2 commands
RB(1420):Rebuild:2 Defines Found, 2 Compiled
#ET=0.9 PT=0.1 IO=0.1
```

By specifying the object file using 'AS \$AUTO/ENTER' and after a successful compile, the SYSTEM/OPAL compiler will initiate an ENTER command to compile the DEFINES in the source file into the live SCHEDULE file.

Only users that are marked as privileged (PU) or SYSTEMUSER may perform the ENTER from an OPAL compile. Other users will receive a 'not allowed' error.

## Overview

The OPAL machine is that component of both Supervisor and Flex which executes a compiled OPAL code string. The machine operates within an environment which applies to an individual OPAL program, for example, an individual Supervisor WHEN slot or a FLEX Inquiry session. As with other programming languages, OPAL variables allow the user to store temporary data within these individual environments.

A maximum number of 1000 variable identifiers are permitted in each environment. These variables are of two types: strings and decimal (both integer and real values are allowed with full precision) and OPAL permits the same variable identifier to be used to reference both string and decimal values.

OPAL variable name identifiers have similar characteristics to program identifiers. A variable identifier can be constructed from a simple string constant to form a "normal" variable or by the run-time evaluation of a non-constant, string expression. The latter are referred to as "complex" variables.

For both normal and complex variables, the maximum length of a variable name is 17-characters. For a normal variable identifier, the OPAL compiler will give an error if the string literal exceeds this limit.

For complex variables the OPAL machine will, if necessary, automatically truncate the variable name to 17 characters at the time the expression is evaluated.

## Variable storage - the HEAP

OPAL's mechanism for storing and retrieving variable information relies on an area of storage known internally as the HEAP. Each time a new variable name is encountered it is added to the HEAP that is extended until the current limit of 1000 variables is reached.

For "normal" variables, if all 1000 variables slots are occupied, attempts to assign a new variable name will abnormally terminate the WHEN or FLEX session with the following error:

```
HEAP VARIABLE LIMIT EXCEEDED (>1000)
```

For "complex" variables, once the heap limit is reached the new variable will be added to the HEAP but space is provided for it by deleting the **least recently accessed** complex variable. At least one existing complex variable must already exist in the HEAP for this to happen.

Two variants of the MYSELF Attribute return information about the HEAP. MYSELF(HEAPIDS) returns the number of slots in the HEAP currently in use. MYSELF(MAXHEAP) returns the maximum number of slots in the HEAP. Therefore,

when MYSELF(HEAPIDS) = MYSELF(MAXHEAP) then any additional variable will result in an old variable being deleted.

## Scope of Variables

In SUPERVISOR, there is an unique list of variables belonging to each WHEN slot so values stored by any of the variable functions within a SITUation can be retrieved by a linked ODTSequence or DISPlay, and vice versa. Similarly, if an ODTSequence or DISPlay is invoked from another ODTSequence using the **CALL** statement, the called code has access to the same environment as its caller and, therefore, can share the same variables.

In FLEX, each program has a unique list so that each FLEX INQUIRY session has its own variables and they can be referenced anywhere in a SELECT or REPORT. Similarly, every RULE for a user can reference its variables but not the variables of any other user.

## Historical Variable handling

Historically, the GET and PUT functions were used to store and retrieve real or integer values from a variable whilst the GETSTR and PUTSTR functions were used for handling strings. The STORE function was used to update both decimal and string variables - the OPAL machine determining which variable type is to be acted upon by the value to be stored. Whilst these functions are still available and are fully described later in this document recent versions of Opal provide an enhanced, more straight forward, means of accessing variables as described in [Obsolete Variable Handling](#).

## Enhanced Variable handling

The latest versions of Opal implement an enhanced syntax for handling variables where String and numeric variable names are prefixed by a special character to denote their type. String variables are prefixed with \$ while numeric variables are prefixed by #. The variable name can be made up of up to 17 alphanumeric characters together with the '\_' (underscore) character.

Typical numeric variable names might be:

```
#Segs, #MYREAL, #123_abc
```

Typical string variable names could be:

```
$123_abc, $MYSTRING, $mymsg
```

Internally, variable names are always treated as if they were in upper case.

Examples:

```
#Segs = #segs = #SEGS  
and  
$123_abc = $123_aBc = $123_ABC
```

In Supervisor, assignment statements are allowed and are implemented using the '**:=**' assignment operator. The following example shows the current accumulated processor time for a task being assigned to the numeric variable **#PROC** and the use of the stored value in the comparison.

Example:

```
IF #PROC:= ACCUMPROCTIME GTR 100 THEN ...
```

This previously could have been coded as:

```
IF PUT("PROC", ACCUMPROCTIME) GTR 100 THEN
```

The following illustrates the assignment of a value to the string variable **\$STR**:

```
$STR:="a,b,c";
```

Which might previously have been coded as:

```
PUTSTR("STR", "a,b,c") or  
STORE("STR", "a,b,c")
```

## Methods

With the implementation of 'enhanced' variables, allowing direct assignment of values using the '**:=**' operator, several older OPAL functions have now been replaced by 'methods'. As in other object-oriented languages, a method is signified by using '.' to separate the variable name and the action to be performed on that variable.

Several OPAL functions have been replaced by methods, such as ACCUM, DELTA, FILE, SPLIT, STORE and SUM. Each method usually requires a parameter to store or update the value in a variable.

For example:

```
#A.DELTA(25)  
#CPU.ACCUM(PROCTIME)  
$LIST.SPLIT  
$STR.STORE("MCP IS "&MCP)
```

Other variable methods include MX, PER, TAPEDB, PRINTS and VDBS. This allows the interrogation of an object determined by the value in the variable and the type specified by the method name.

For example:

#### **`$A.MX (NAME)`**

This would return the name of the mix entry whose mix number is held (as a string) in `$A`.

#### **`#A.PRINTS (TOTALLINES)`**

This would return the total number of lines in the print request whose request number is held as an integer value in `#A`.

#### **`#A.PER (# (KIND , , LABEL) )`**

This returns the unit kind and name of the peripheral whose unit number is held in `#A`.

#### **`$A.TAPEDB (TITLE)`**

Lastly, this example of a TAPEDB method would return the title of the tape whose serial number is held in `$A`. The TAPEDB method always requires a string value.

These methods give a more concise and meaningful alternative to using VIA to extract attribute information for objects other than the current object.

All methods may be used as statements.

For example:

```
#A.Sum(1) ;  
$LIST.SPLIT ;  
$STR.Cut ("ID=")
```

## **Properties**

The enhanced variable access allows access to global, permanent and configuration variables by using the GLOBAL, PERM or CONFIG 'properties'. A property of a variable is denoted by using the '.' character after the variable name (similar to other object-oriented programming implementations). Both properties are valid for string variables but only PERM is permitted for numeric variables.

Some examples using these properties are shown below:

```
$LIST.PERM:="TEST"  
$SUP_DESTINATION.CONFIG  
#SEGS.PERM  
$SUPINFO.GLOBAL
```

Permanent, global and configuration variables are discussed in further detail in a later section of this manual.

## Limits

A maximum number of 1000 variable identifiers are permitted in each environment. These variables are of two types: strings and decimal (both integer and real values are allowed with full precision) and OPAL permits the same variable identifier to be used to reference both string and decimal values.

Up to 1,999,999 characters can be stored in a single OPAL string variable.

OPAL decimal variables have the same scope as normal Unisys single-precision real variables. Integer values range –549755813887 to +549755813887. Values beyond these limits are held as the usual, single-precision, floating-point values.

Please see [Obsolete Variable Handling](#) for more information on the older variables implementation.

## Syntax

### Variables

<string variable>

\_\_\_\_\_ \$<identifier> \_\_\_\_\_|

<real variable>

\_\_\_\_\_ #<identifier> \_\_\_\_\_|

### Dynamic OPAL variables

Dynamic variables in the older implementation allowed another OPAL string variable to be used as a key e.g.

```
GETSTR (GETSTR ("A" ) )
```

means use the string value held in variable A as a key for the outer GETSTR function. This is sometimes useful in WHENs or EVALs, for example, to use a mixnumber or unit number as a key to hold information about that mix entry or unit. The above capability has now been introduced for the new variable mechanisms.

<dynamic string variable>

— \$\$<identifier> \_\_\_\_\_|

<dynamic real variable>

— #<identifier> \_\_\_\_\_|

In both cases, the prefixing of an additional '\$' before the usual variable name, means use the value held in the string <var name> as the key for loading the requested value.

Dynamic string example:

```
$A:="TESTVAR";  
$$A:= "THIS IS REALLY TESTVAR"; %string is actually in $TESTVAR
```

`$$A` is therefore identical to `$TESTVAR` or `GETSTR("TESTVAR")`.

Dynamic real example where MixNo=1234:

```
$MX:=#("CPU",MIXNO);  
#$MX:= CPUTIME; % Stores into variable CPU1234
```

```
#$MX          is the same as  
GET("CPU1234") or #CPU1234 (if MIXNO=1234)
```

It is not possible to use a string expression with a \$\$ or #\$ operation.

Note that, as with the older variable mechanism, using an empty string as a key for a dynamic variable is not permitted.

Example error message:

```
FUNCTION VARIABLE KEY CANNOT BE NULL STRING
```

OPAL automatically up-cases any lower case characters into the loaded key, as all \$ and # variable conventions expect upper case key names.

Example:

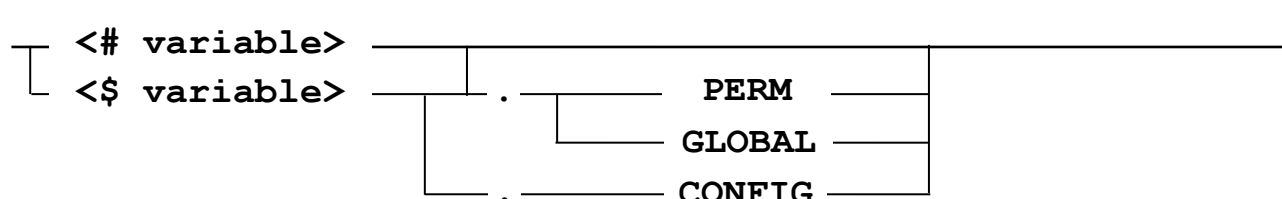
```
DEFINE + ODTSEQUENCE GET_CONFIG(MSG):  
    %% Using CONFIG property allows retrieval of Magus  
    %% configuration variables.  
    $PARAM:=TRIM(TEXT);  
    SHOW("Config ", $PARAM, " = ", $$PARAM.CONFIG);
```

The MAGUS configuration variables SUP\_UCODE and SUP\_SCHEDULEFAM hold the current SUPERVISOR usercode and the TT USE FAMILY... FOR SCHEDULE assignments.

These values can then be retrieved using:

```
TT DO GET_CONFIG SUP_UCODE  
Config SUP_UCODE = SUPERVISOR  
  
TT DO GET_CONFIG SUP_SCHEDULEFAM  
Config SUP_UCODE = SCHEDULEFAM
```

For use in later diagrams we define:

$$\langle \$ \text{ variable} \rangle ::= \langle \text{string variable} \rangle \mid \langle \text{dynamic string variable} \rangle$$


METALOGIC/MAGUS/CONFIGURATIONDATA

Both SUPERVISOR and FLEX can use a third type of property, GLOBAL, that allows data, held in variables, to be shared between OPAL programs.

To store permanent values in the CONFIGURATIONDATA file:

```
#MYVAR.PERM:= 100
$MCPNAME.PERM:= MCP
```

To retrieve the values later from any OPAL program :

```
#A:= #MYVAR.PERM
$MCPNAME.PERM
```

The assignment of an EMPTY string to an existing permanent variable will remove it from the CONFIGURATIONDATA file; similarly, a permanent arithmetic value may be removed by assigning the value 0.



Example:

```
$ABC.PERM:=EMPTY  
#MYPERMVAR:=0
```

## CONFIG property

The CONFIG property allows the retrieval of the string assigned to the installation configuration variable denoted by \$<identifier>. Usually, these Metalogic environment variables would only be accessible using the Metalogic INSTALL utility. Only string configuration variables may be retrieved since Metalogic software does not use numeric values to reflect its operating environment.

For example:

```
$SUP_USERCODE.CONFIG    might return "SUPERVISOR"  
$SUP_PRIORITY.CONFIG    might return "85"
```

As with using PERM, the obsolete STORE, PUTSTR and GETSTR functions can also use the CONFIG property.

Example:

```
GETSTR("SUP_USERCODE",CONFIG)  
GETSTR("SUP_PRIORITY",CONFIG)
```

It should be noted that GET and PUT cannot use the CONFIG modifier as any non-string valued environment variables will always return numeric string values (e.g. as with SUP\_PRIORITY in the above example).

**Although values may be assigned, using the CONFIG property, to existing Metalogic configuration settings, this mechanism is not recommended. Changing the contents of these environment variables should be managed using the INSTALL utility CONFIG menu or the recognised command syntax available in the software (e.g. TT USE for SUPERVISOR, DEFAULTS in Flex).**

## GLOBAL property

The GLOBAL property allows the 'sharing' of values between multiple OPAL programs run under SUPERVISOR or FLEX/LIBRARY. The implementation permits shared variables within each application without the I/O overhead incurred by using PERM variables.

For example:

```
$SUP_TEST.GLOBAL:="TEST STRING";  
#SUP_COUNT:= #CNT+1;
```

In Supervisor, GLOBAL variables are shared between all WHEN slots and their values are retained only for the duration of the current invocation of Supervisor. They are not retained after a Halt Load or Supervisor restart.

In Flex the variables are shared by all runs of Flex Inquiry or the FAMILYMANAGER utility and retained only for the duration of the life of METALOGIC/FLEX/LIBRARY.

GLOBAL variables cannot be shared between Flex and Supervisor. GLOBAL can also be used in the old GET,GETSTR, PUT, PUTSTR and STORE functions.

Example:


<code>\$STR.GLOBAL</code>	is equivalent to	<code>GETSTR("STR",GLOBAL)</code>
<code>\$STR.GLOBAL:="A"</code>	is equivalent to	<code>STORE("STR",GLOBAL)</code>

## Simple variable methods


With the implementation of 'enhanced' variables, allowing direct assignment of values using the ' := ' operator, several older OPAL functions have now been replaced by 'methods'. As in other object-oriented languages, a method is signified by using '.' to separate the variable name and the action to be performed on that variable.

The following two types of methods exist, with the following general usage:

### <String method>

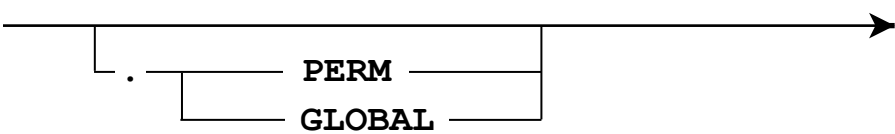

— `<$ variable>.<method>` 

### <Arithmetic method>

— `<# variable>.<method> ( <Arithmetic expression> )` 

`<variable>.<method>` may be used in expressions or as a stand alone statement, in which case the result normally returned is discarded.

## .ACCUM arithmetic method

— `<# variable>`   


The ACCUM method adds the new arithmetic value from the `<arithmetic expression>` parameter to the current value held in the OPAL variable specified by `<# variable>`. The method will return the value in `<arithmetic expression>` as a result.

The ACCUM method is very similar to SUM, except the previous value in `<# variable>` is returned as a result to the caller instead of the new, calculated value.

## SUPERVISOR Example

```
DEFINE + ODTSEQUENCE ACCUM:
```

```
#TOTL:=17;
```

```
#DIFF:=12;
```

```
SHOW("ACCUM = ", #TOTL.ACCUM(#DIFF))
```

```
SHOW("TOTAL = ", #TOTL);
```

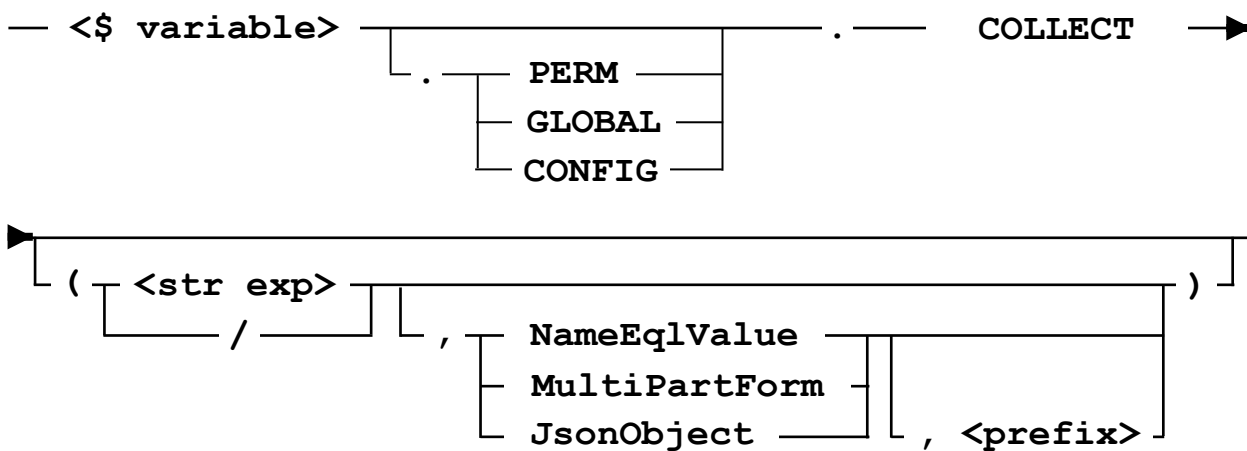
```
TT DO ACCUM
```

```
ACCUM = 12
```

```
TOTAL = 29
```

### .COLLECT

string method



The collect method complements the Distribute method. It is used to collect the names and values of string variables into the target string variable. If no parameters are passed it collects all used variables and stores them in the form of a coma separated list with each element of the form `<name> =<value>`.

Example:

```
$A:="ONE";
```

```
$B:="TWO";
```

```
$LIST.Collect;  %$List = A=ONE,B=TWO
```

The optional <str exp> or / defines an alternate delimiter to comma, / indicating new line.

If the delimiter has been specified it is possible to define the name/value mapping to use. The MultiPartForm list format may be used with an HTML Multi Part Form and is in the format "<name>"CRLF<value>". The JSONOBJECT list format may be used to collect variables into a JSON object in the form {"name":"value",...} or {} if there are none.

If both delimiter and name/value mapping are defined it is possible to define a prefix to be used when searching for string variables. The variable name stored will not include the prefix.

A prefix of "S" will find \$S1 but not \$S. Variables must have a name at least one character longer than the prefix to be included.

If any variable has a value which contains the specified delimiter it will not be added to the list. The names of any such variables are returned as a comma separated list in the value of the method.

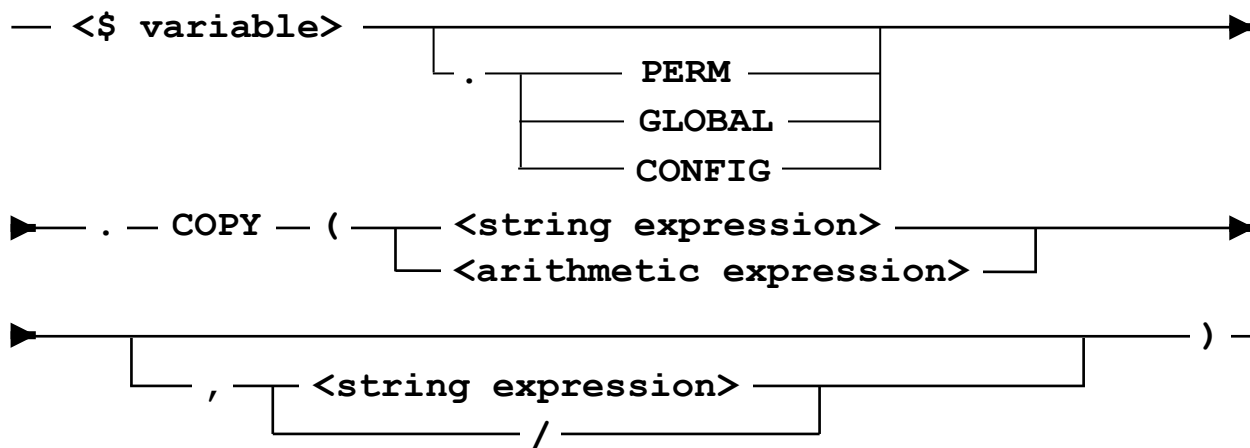
Example:

```
$NS_A:="FIRST" ;  
$NS_B:="SECOND" ;  
$NS_C:="A&B" ;  
$Rejected:=$Collection.Collect("&",NameEqValue,"NS_") ;  
%$Rejected will hold $NS_C  
%$Collection will hold A=FIRST&B=SECOND
```

This example finds all local string variables with a namespace prefix of NS\_ and creates a list of <name>=<value> elements, separated by an &, and stores them into \$Collection.

\$NS\_C contains an &, so it is added to the comma separated list of names returned in \$Rejected.

## **.COPY** string method



The .COPY method allows the contents of a string variable to be searched for a user-provided target or specified entry; the source should be a list of entities delimited by a known delimiter, the default is assumed as comma (.). Wild cards may be used with the target string to facilitate the search.

The pattern to be searched for is specified in the first string expression. The delimiter to use to divide the string , if not comma, is specified in the second string expression. The special delimiter / allows the end of line character to be used as a delimiter.

If an arithmetic expression is provided as the first parameter then the element of the list specified by that value is returned.

If the target string is found in the source, the target is returned with delimiters removed. The string contents of the variable are left unchanged.

For example:

```
$SOURCE:="THIS,IS,AN,EXAMPLE,LIST";
$Z:=$SOURCE.Copy("E=");
```

After the COPY

```
$Z holds "EXAMPLE"
$SOURCE holds is unchanged "THIS,IS,AN,EXAMPLE,LIST"
```

Using an arithmetic expression:

```
$SOURCE:="THIS,IS,AN,EXAMPLE,LIST";
$Z:=$SOURCE.Copy(2);
```

After the COPY

```
$Z holds "IS"
$SOURCE holds is unchanged "THIS,IS,AN,EXAMPLE,LIST"
```

Similarly, if the delimiter string was "\_":

```
$SOURCE:="THIS_IS_AN_EXAMPLE_LIST";
$Z:= $SOURCE.COPY("LIST","_");
```

After the COPY

```
$Z holds "LIST"  
$SOURCE is unchanged "THIS_IS_AN_EXAMPLE_LIST".
```

Again with an arithmetic expression:

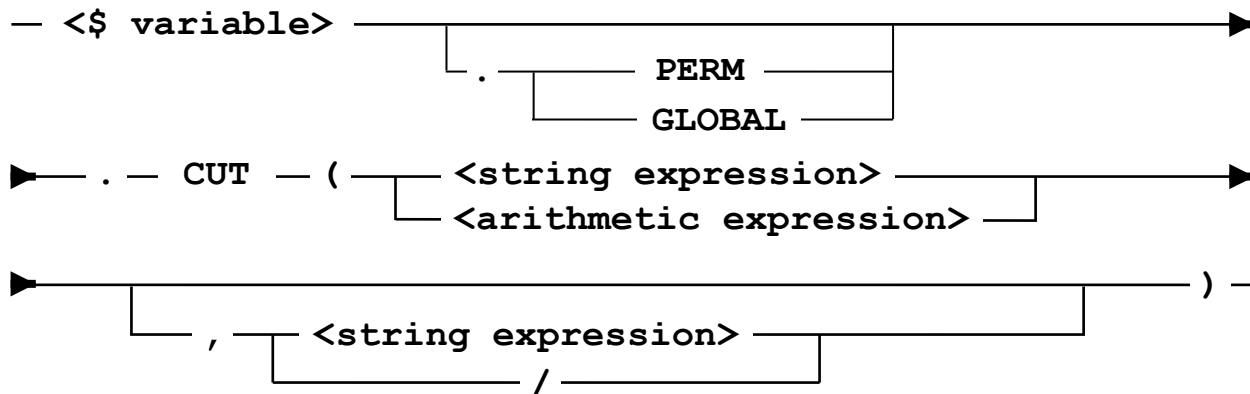
```
$SOURCE:="THIS_IS_AN_EXAMPLE_LIST";  
$Z:= $SOURCE.COPY(4,"_");
```

After the COPY

```
$Z holds "EXAMPLE"  
$SOURCE is unchanged "THIS_IS_AN_EXAMPLE_LIST"
```

If no match is found, an empty string will be returned.

## **.CUT** string method



The `.CUT` method allows the contents of a string variable to be searched for a user-provided target or a specified entry; the source should be a list of entities delimited by a known delimiter, the default is assumed as comma (`,`). Wild cards may be used with the target string to facilitate the search. If an arithmetic expression is provided as the first parameter, then that value determines the element of the list selected.

The pattern to be searched for is specified in the first string expression. The delimiter to use to divide the string, if not comma, is specified in the second string expression. The special delimiter `/` allows the end of line character to be used as a delimiter.

If the target string is found in the source, the target is returned with delimiters removed. The string contents of the variable is then adjusted to remove the search string and any associated delimiters.

For example:

```
$SOURCE:="THIS,IS,AN,EXAMPLE,LIST";  
$Z:=$SOURCE.Cut("E=");
```

## After the CUT

**\$Z holds "EXAMPLE"**

\$SOURCE holds the new string "THIS,IS,AN,LIST"

## Using an arithmetic expression

```
$SOURCE:="THIS,IS,AN,EXAMPLE,LIST";
```

```
$Z := $SOURCE.Cut(4);
```

## After the CUT

**\$Z holds "EXAMPLE"**

\$SOURCE holds the new string "THIS,IS,AN,LIST"

Similarly, if the delimiter string was "\_":

```
$Source:="THIS IS AN EXAMPLE LIST";
```

```
$Z := $Source.Cut("LIST", " ");
```

## After the CUT

**\$Z holds "LIST"**

```
$SOURCE has "THIS IS AN EXAMPLE".
```

## Using an arithmetic expression

```
$SOURCE:="THIS IS AN EXAMPLE LIST";
```

```
$Z := $SOURCE.Cut(4, " " );
```

## After the CUT

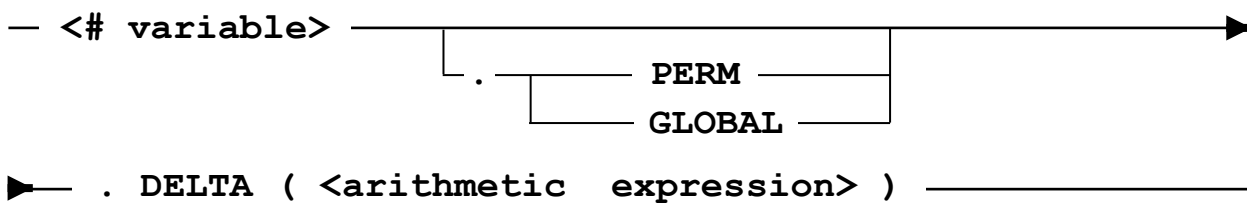
**\$Z holds "EXAMPLE"**

`$SOURCE` holds the new string "THIS IS AN LIST"

If no match is found, an empty string will be returned.

**.DELTA**

## arithmetic method



The DELTA method assists with the "polling" of OPAL variables or system information that is changing over time. DELTA stores the value given by the <arithmetic expression> parameter into the variable named in #<identifier>, similar to a STORE or assignment operation. However, DELTA returns, as a function result, the difference between the new value and the prior value held by #<identifier>.

This is useful when dealing with some of the attributes in SUPERVISOR that are totals, often time-based, since the last halt-load.

Example attributes are RFERRORS (PER context), MCPCLOCKS (SYSTEM), ACCUMPROCTIME (MX).

Example

```
#HLTime.Delta(MCPClocks)
#CPUTime.Global.Delta(AccumProcTime)
```

If no prior value exists, it is assumed to be zero, i.e. the result is equal to the value of the <arithmetic expression>.

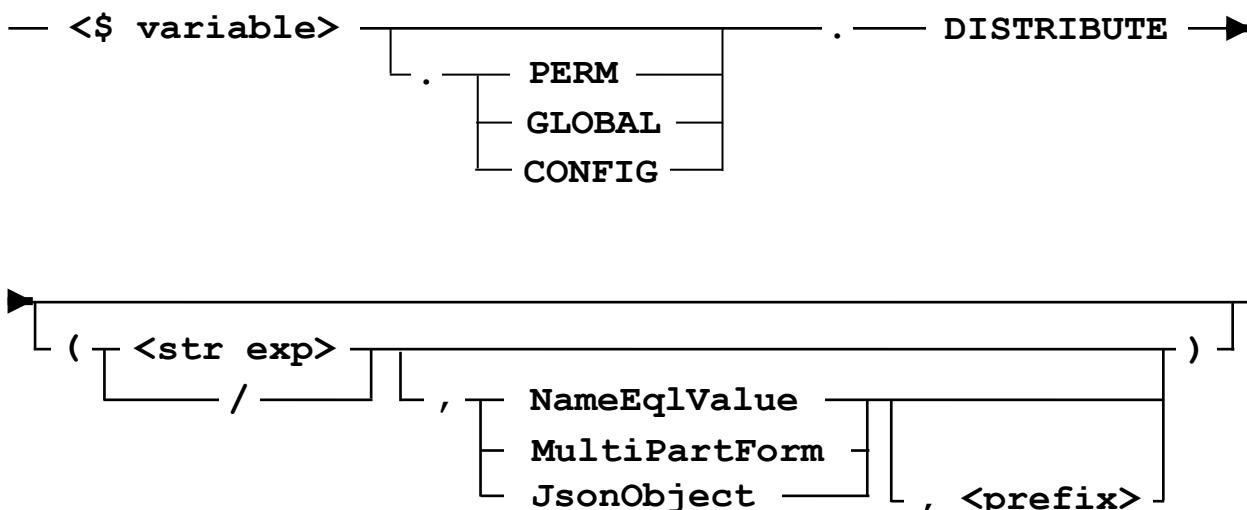
A common problem occurs when an attribute appears in the <arithmetic expression> which is a changing total since the last halt-load. When first used, the initial value of such an attribute may be very large; this behaviour can be protected by performing the DELTA method only if the value in the variable is non-zero.

```
If #MyVar = 0 Then
    #MyVar:= AccumProcTime
Else
    If #MyVar.Delta(AccumProcTime) > 60 Then
        Display(MIXNO, " EXCEEDED 60 SECONDS SINCE LAST CHECK")
```

SUPERVISOR Example

```
DEFINE + ODTSEQUENCE DELTA:
    #Totl:=17;
    #Del:= 22;
    Show("DELTA = ", #Totl.Delta(#Del));
    Show("TOTAL = ", #Totl);
TT DO DELTA
TOTAL = 22
DELTA = 5
```

## **.DISTRIBUTE** string method





The DISTRIBUTE method of a string variable may be used to generate local variables from a string containing name/value pairs.

A local variable, set to the corresponding value, is created for each name/value pair found in the list. The local variable is always stored in upper case.

Any characters which are not distributed as local variables are returned as the method value.

The default form expects a list syntax of <name1>=<value1>, <n2>=<v2>...

Example

```
$r:="S1=V1,S2=V2";  
$l:=$r.distribute;  
would create $S1 with value "V1" $S2 with value "V2"
```

The optional <str exp> or / defines an alternate delimiter to comma, / indicating new line.

Example

```
$r.Global:="X1=abc;x3=pqr";  
$l:=$r.Global.distribute(";");  
would create $X1 with value "abc" $X3 with value "pqr"
```

If the delimiter has been specified it is possible to define the name/value mapping to use. The MultiPartForm list format may be used with an HTML Multi Part Form and is in the format "<name>"CRLF<value>". The JSONOBJECT format is used to distribute a JSON object in the form {"name":"value",...}

If both delimiter and name/value mapping are defined it is possible to define a prefix to be used when creating string variables.

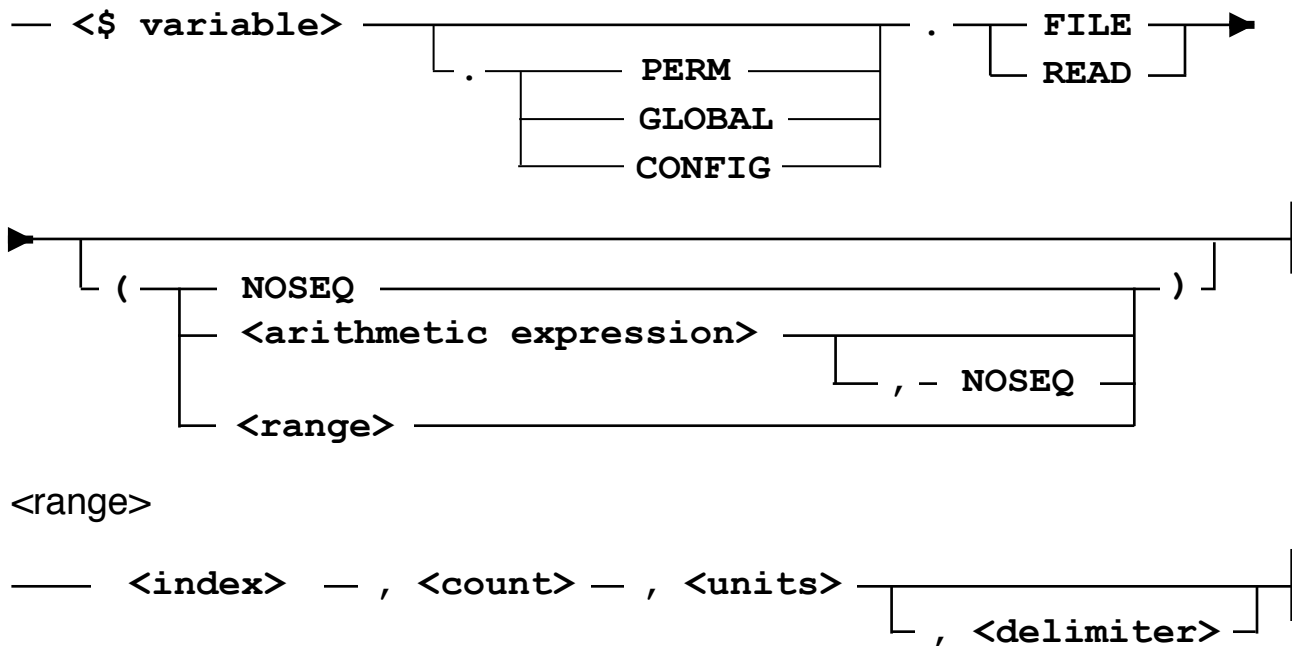
Example

```
$List:="A=FIRST,B=SECOND,ABC";  
$Reject:=$List.Distribute(",","NameEqValue","NS_");  
%Reject will hold ABC since it is not a valid name/value pair  
%$NS_A will hold FIRST  
%$NS_B will hold SECOND
```

## **.FILE**

### **.READ**

#### string method



The **FILE** (**READ** is a synonym) method allows the contents of files of specific filekinds. Any stream or symbolic (i.e. JOBSYMBOL, ALGOL, SEQDATA, DATA etc) plus Keyedio and BackupPrinter files may be read. Data are read from the file title represented by the OPAL variable `$<identifier>`. When reading a symbolic file, sequence numbers mark IDs and trailing spaces are deleted if the **NOSEQ** option is specified. Each record from a symbol file is delimited by the CR (/) character. The file content, up to a maximum of 1,999,999 characters, is returned to the caller.

If the **READ** function detects an **INVALIDTRANSLATIONRSLT** when opening the file, it sets **DEPENDENTINTMODE** and tries again.

If the optional `<arithmetic expression>` is supplied then only that number of characters will be read.

For example:

```

$Content:= $Temp.File;
or
$Content:=$Temp.Read(100);
  
```

The above construct instigates a search for a file called `*TEMP` on the callers default family substitution.

If the file is not available or is not a stream or symbolic file, the following errors will be returned to the caller:

```

Error: No File *T
Error: File *T Invalid. Avail=[110]
Error: Not Byte Stream or Symbolfiledef
  
```

The above method has limited use because most files have usercodes and ON family specifications. It is not possible to specify such file titles within an OPAL variable name, because of OPAL identifier restrictions; in these cases, dynamic variables should be used.

SUPERVISOR example:

```
$FileName:= "(DEV)STREAM/OUTPUTFILE ON WORKPACK";  
$Read:= $$FileName.File;
```

The use of READ operations from SUPERVISOR scripts will automatically prefix the file title with a '\*' if the filename does not start with '\*' or '('.

For example:

```
$File:="SOURCE/LIVE ON DEV";  
$Text:= $$File.Read  
or  
$Sys_File.Perm:="*SYS/OPTIONS ON DISK";  
$Text:=$$Sys_File.Perm.Read
```

The above causes the SUPERVISOR script to look for a file called \*SOURCE/LIVE regardless of the calling usercode. However, this behaviour is not desirable in FLEX scripts which invariably are run under a usercode, causing the file to be not found or a non-usercoded version to be read instead.

To address this, FLEX scripts behave differently by not prefixing an unqualified title with '\*', allowing the READ to use the local usercode file first, if available.

.READ can also read files stored in UNISYS WRAP containers by appending an IN part to the title.

Example.

```
$File:='*SOURCE/LIVE IN "MYFILES.CON" ON DEV';  
$Read:=$$File.Read
```

The expected use of reading KEYEDIO files is to read the header record of a file.

Ex

If MYKEYEDIO had a header record with the following description

```
01 HDR  
03 FILLER          PIC X(15) .  
03 SEQNO          PIC 9(5) COMP.  
03 RATE           PIC 99V9999 COMP.
```

Then the following code could be used to get these values.

```
$FILE:="*MYKEYEDIO ON LIVE";  
$BUF:=$$FILE.READ(0,1,RECORDS);  
$HEXREC:=HEXSTRING(DROP($BUF,15)); %Skip over filler and convert hex
```

```
$SEQNO:=$HEXREC.SPLIT(5);
```

```
$RATE:=$HEXREC.SPLIT(2) &"." &$HEXREC.SPLIT(4);
```

If the <range> option is specified then the parameters are as follows:

<index>	the index at which to start the read, in <units>.
<count>	the number of <units> to read.
<units>	RECORDS BDRECORDS or BYTES
<delimiter>	Optional <string> delimiter or / to separate multiple RECORDS

If BYTES is specified for a file which is Blocked, the File is assumed to be a ByteStream consisting of the concatenated Records, where each Record is MAXRECSIZE in Bytes long.

If BDRECORDS is specified the specified number of records are read from a printer back file. Note that a printer backup record will almost certainly contain multiple lines. This mechanism is needed to process very large printer backup files when the total size is too large to be held in a string variable.

```
$S:=$$FL.READ(0,10,bdrecords)
```

**will read the first 10 printer backup records into the variable \$S**

```
TT DEFINE + ODTSEQUENCE BD_READRANGE(MSG):
```

```
%reads the backup file passed as a parameter
```

```
%Reporting any line containing the digit 2
```

```
$F1:=Upper(Trim(text));
```

```
Show(#recs:=$F1.PD(Lastrecord)+1); #next:=0
```

```
Do Begin
```

```
  #Rd:=Min(#Recs,10);
```

```
  $S1:=$$FL.READ(#next,#RD,bdrecords);
```

```
  WHILE $S1 NEQ EMPTY DO
```

```
  BEGIN
```

```
    $LINE :=$S1.SPLIT(/);
```

```
    IF "2" ISIN $LINE THEN
```

```
      $PAGE.INSERT("#FOUND:",#LN.SUM(1) 3,,TAKE($LINE,20)),/);
```

```
  END;
```

```
  #Next.Sum(#Rd);
```

```
End Until #Recs.Sum(-#rd) Leq 0 ;
```

```
SHOW($PAGE);
```

The read method may read from files on a share on a PC. This is done by specifying a UNC title for the file.

A UNC title is of the form:

```
*UNC/<host name or address>/<share name>/<path to file>  
Ex. $ID:='*UNC/PCSERVER/PUBLICSHARE/Myfolder/"myfile.txt";  
SHOW($$ID.READ);
```

The usercode and password used to access a server is controlled by the MAKECREDENTIALS Utility. It must be run from an ODT or Supervisor window. The MAKECREDENTIALS utility provides a method of encrypting, credentials, and creates a credentials file for each system for which the user has remote access.

MAKECREDENTIALS is a simple command mode utility that expects the following login credentials for a network host: <host> <username> <password> <user domain> (optional) These four pieces of information correspond to the IOHSTRING keywords as follows: SERVER = <host> CREDENTIALS = <username>/<password> USERDOMAIN = <user domain>

### Examples

The following example shows how the MAKECREDENTIALS utility is for a host of MYSERVER (10.0.0.10), a username of OPS, and a password of frog:

```
RUN *SYSTEM/NXSERVICES/MAKECREDENTIALS("MYSERVER OPS "&"&"frog"&"")
```

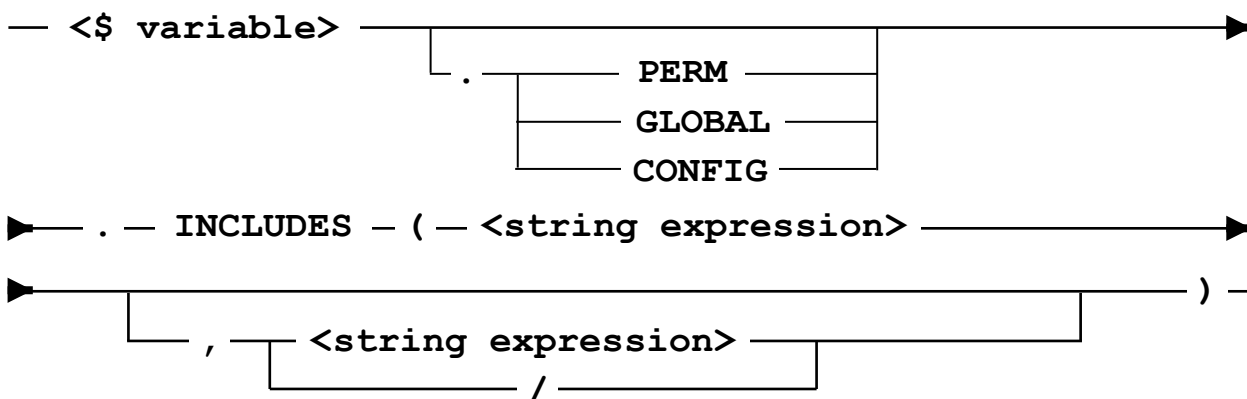
or if the IP address is to be used:

```
RUN *SYSTEM/NXSERVICES/MAKECREDENTIALS("10.0.0.10 OPS "&"&"frog"&"")
```

Note that if the password on the PC is not all uppercase then the password must be quoted.

## .INCLUDES

boolean method



The .INCLUDES method allows the contents of a string variable to be searched for a user-provided target; the source should be a list of entities delimited by a known delimiter, the default is assumed as comma (,). Wild cards may be used with the target string to facilitate the search.

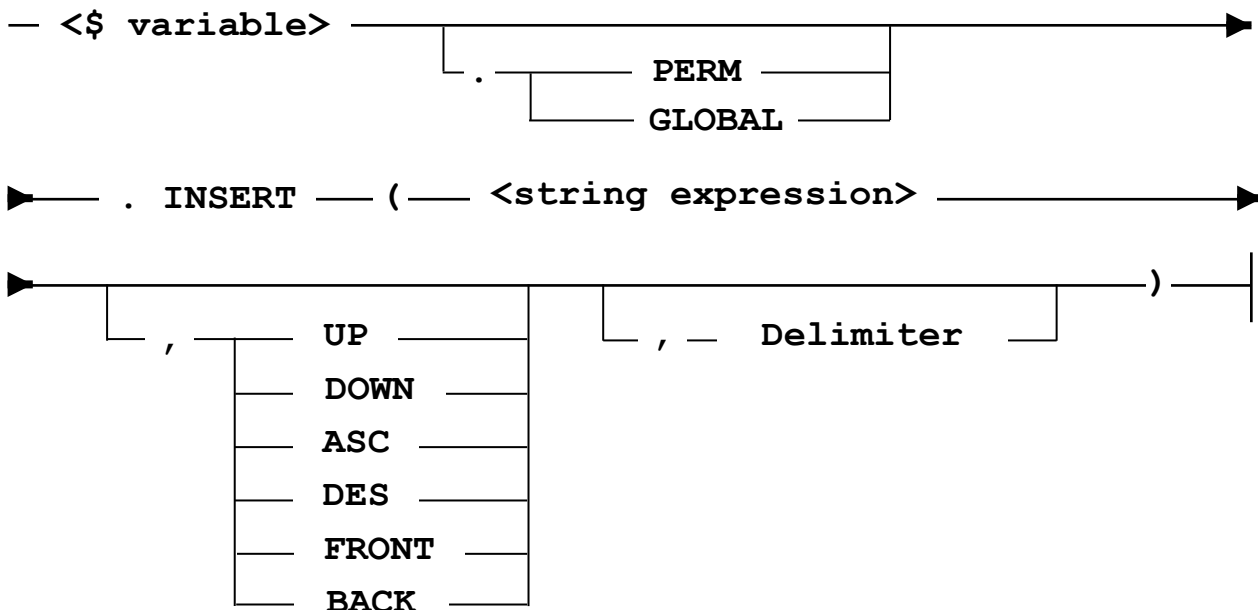
The pattern to be searched for is specified in the first string expression. The delimiter to use to divide the string , if not comma, is specified in the second string expression. The literal '/' allows the end of line character to be used as a delimiter.

TRUE is returned if the target string is found in the source.

Example:

```
If $Source.Includes("IS") Then
If $$Source.Config.Includes("=DEV=",/) Then
```

## **.INSERT** empty string method



The optional delimiter is a string expression. If omitted "," is assumed.

If the optional modifier is omitted then BACK is assumed.

The string expression is added to the String variable. If the modifier is BACK it is added to the end with <delimiter> as a separator. If the modifier is FROM it is added to the beginning with <delimiter> as a separator.

If the modifier is UP or ASC (Ascending) then it is inserted before the first item in the list which is greater than it.

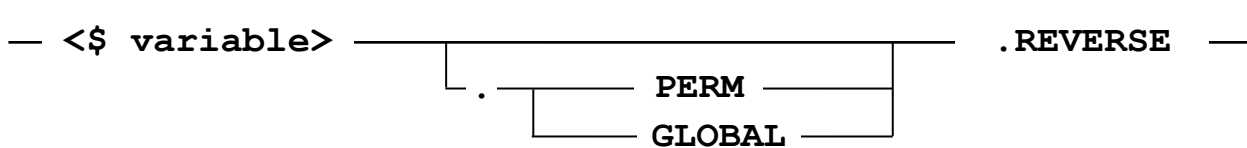
If the modifier is DOWN or DES (Descending) then it is inserted before the first item in the list which is less than it.

Note that if a sorted list is desired, care should be taken to only use .Insert , with an appropriate modifier, to maintain it.

## Examples.

```
$L1.Insert("A");$L1.Insert("C");$L1.Insert("B");  
    would leave $L1="A,C,B"  
$L2.Insert("A",Front);$L2.Insert("C",Front);$L2.Insert("B",Front);  
    would leave $L2="B,C,A"  
$L3.Insert("A",UP);$L3.Insert("C",UP);$L3.Insert("B",UP);  
    would leave $L3="A,B,C"  
$L4.Insert("A",DOWN,":=");$L4.Insert("C",DOWN,":=");  
$L3.Insert("B",DOWN,":=");  
    would leave $L3="C:=B:=A"
```

## **.REVERSE** string method

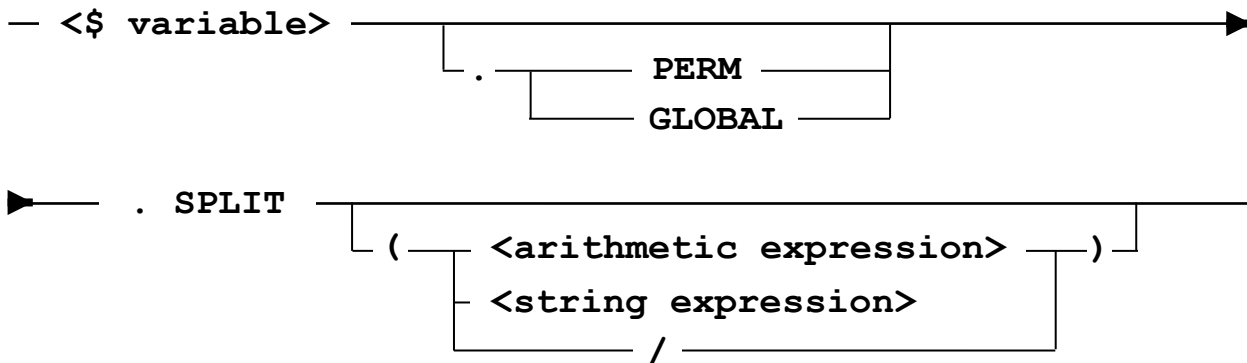


The reverse method returns the contents of the variable in reverse order.

Ex:

```
$S:="ABCEDF";  
$R:=$S.Reverse; %$R=FDECBA"
```

## **.SPLIT** string method



**SPLIT** is a string method that may be used to parse entity "lists" held in an OPAL string variable. Such a list might be a series of mix or unit numbers, each entity delimited by a recurring character such as a comma, slash, etc.:

```
"1234,1236,8999,9000"
```

The **SPLIT** method only operates on a string variable; the name is specified by the term `$ variable`. The bracketed, optional parameter is a 'target' and may be a string expression, real expression or the literal `'/'`. If the parameter is absent, then a default target of `'` (comma) is assumed.

Semantically, **SPLIT** returns the string of all characters held in the \$ variable, up to but not including the first occurrence of the target. The string stored in the \$ variable is then updated to include all characters following the first occurrence of the target but NOT including the delimiter. Multiple delimiters and leading delimiters are skipped over.

Alternatively, if the <arithmetic expression> parameter is used, **SPLIT** will return the appropriate number of characters from the head of the string held in the variable specified by the first parameter. At the same time, these characters are removed from the string and the resultant string is stored back into the variable.

**SPLIT** allows the replacement of multiple **DECAT**, **TAKE**, **DROP**, **HEAD** and **TAKE** functions into a much simpler representation.

For example:

```
$First:= $AB.Split
```

is similar to the following two actions:

```
$First:= Decat($AB," ",4) )  
$AB:=Decat($AB," ",1)
```

In the following examples, assume that the OPAL string variable **\$VAR** holds the string value:

```
$Var:=",1234,,5678,HIT,4567,9123"
```

Now using the **SPLIT** method on the string held in **\$VAR**, assuming comma as the default delimiter:

```
$Var.Split           returns   "1234"
```

The variable **\$VAR** now holds the truncated string:

```
"5678,HIT,4567,9123"
```

Similarly, using **SPLIT** again with a target of "HIT":

```
$Var.Split("HIT")      will return   "5678,"
```

and **\$VAR** will now hold

```
",4567,9123"
```

Using <arithmetic expression> as parameter to **SPLIT**, consider the following case:

```
$TEST.Perm:= "abcdefghijkl"
```

To store the first 5 characters of the string in the \$TEST variable into the variable \$TOP and place the truncated string result back into \$TEST:

```
$Top:=$Test.Perm.Split(5)           will place  "abcde"  in $TOP
```



One of the various equivalents might be

```
$Top:= Take($Test.Perm,5);  
$Test.Perm:= Drop($Test.Perm,5);
```

Also, **SPLIT** allows the special character, /, as an optional parameter. Using a slash is equivalent to using the hash-paren expression # (/) or the lookup string #[CR] .

**SUPERVISOR** Example:

The NW Hosts command gives a response like:

```
nw hosts  
LOCAL HOST NAME = RED (1,1,1,2)  
KNOWN REMOTE HOSTS ARE:  
  BLUE          ACCESS  +      (1,1,1,1)  
                      FLEXIBLE -      HOST GROUP = BLUE  
                      (ACTIVE)      READY,  
  GREEN          ACCESS  +      (1,1,1,3)  
                      FLEXIBLE -      HOST GROUP = GREEN  
                      (ACTIVE)      READY,
```

The following ODTs will display a list of all valid hosts.

```
TT DEFINE + ODTSEQUENCE HOSTS:  
$Rslt:=Keyin("NW HOSTS");  
$Tmp:=$Rslt.split(/)&$Rslt.Split(/); %discard headings  
$Hosts:=HOSTNAME;  
While $Line:=$Rslt.Split(/) Neq Empty Do  
If Not $Line HdIs "    " Then  
    $Hosts:="&","&Trim(Decat($Line,"    ",4));  
Show($Hosts)
```

When used within a **WHILE...DO** block, the **SPLIT** function is ideal for simple manipulation of mix, unit or serial number lists as returned by the **OBJECTS** function.

For example, the ODTSequence below uses one **OBJECTS** call to return all waiting entries and **SHOW** a single line of information for each:

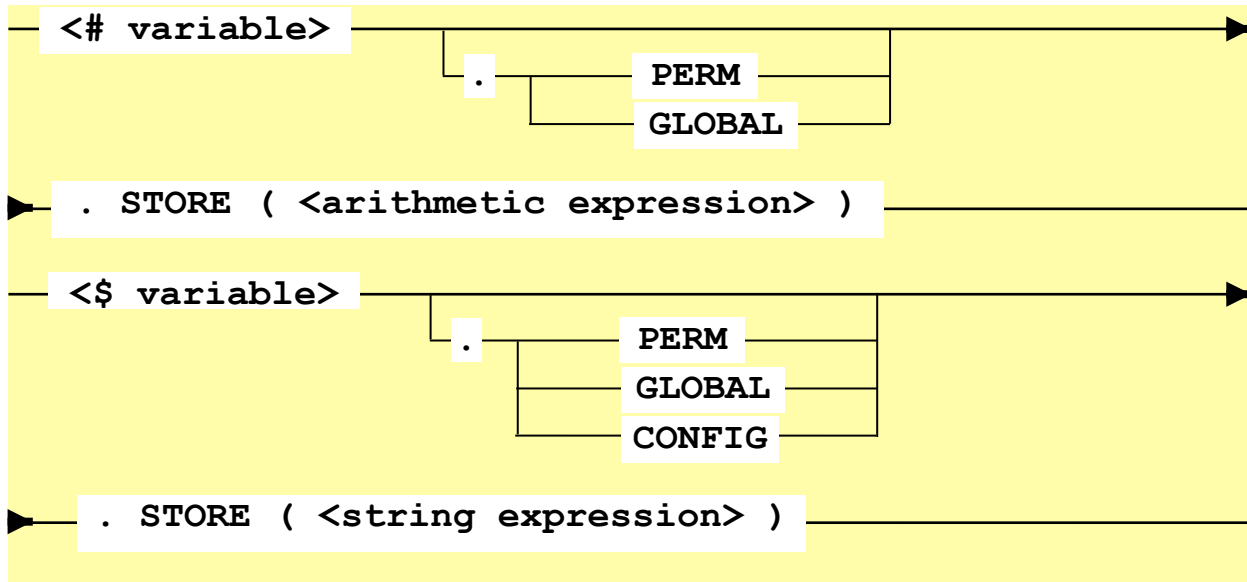
```
TT DEFINE + ODTs SPLIT_TEST:  
    $Mix:=Objects(MX=WAITING: TRUE);  
    While $Item:= $Mix.Split NEQ Empty Do  
        Show($Item.Mx(#(MixNumber 4,, Name 40,,  
                        TIME(StateTime))));  
  
TT DO SPLIT_TEST  
1299 (PROD)OBJECT/WAITER          05:12:00  
1305 *SYSTEM/DUMPALL             01:03:11  
1408 *SYSTEM/MYMCS               04:12:02
```

The following code would process each line of a file:

```
$Title:= `*SUPERVISOR/RECORD/TEST/"BOB.TXT" ON DEV` ;  
$Read:= $$Title.File;    %Dynamic variable  
While $Read NEQ Empty Do  
    Show($Read.Split(/));
```

## **.STORE**

arithmetic or string method



The **STORE** method is identical to its equivalent, older **STORE** function that permitted a string or arithmetic variable to be updated without returning the value stored, like the older **PUTSTR** and **PUT** functions. Like the **STORE** function, a **STORE** method will always return an EMPTY string in any expression.

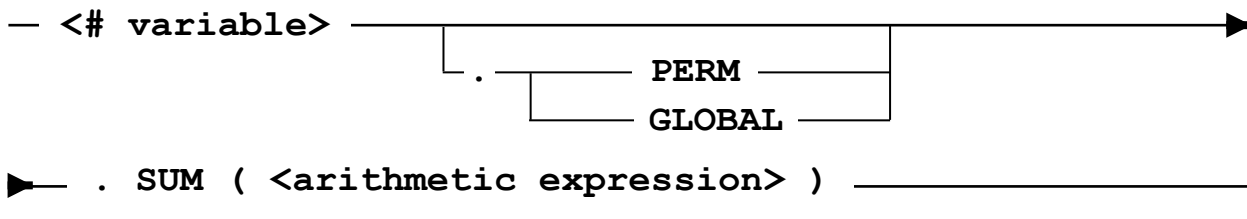
Although its usage may be limited with the introduction of variable assignments, there are occasions in the building of OPAL expressions where this known behaviour may be useful.

A **STORE** method assignment will not compile correctly if used as a stand-alone statement in an ODTSequence.

### SUPERVISOR Examples

```
#A.Store(22)  
$Str.Global.Store(MCP)  
If $A:=MCP&($B.Store("TEST")) = MCP Then  
    Show("$B.STORE RETURNED AN EMPTY STRING")
```

## **.SUM** arithmetic method



The **SUM** method adds the arithmetic value from the <arithmetic expression> parameter to the current value held in the OPAL variable specified by #<identifier>. The method will return the new calculated value in as a result.

The **ACCUM** method is very similar to **SUM** but returns the value being added (i.e. the <arithmetic expression>) instead of the new value.

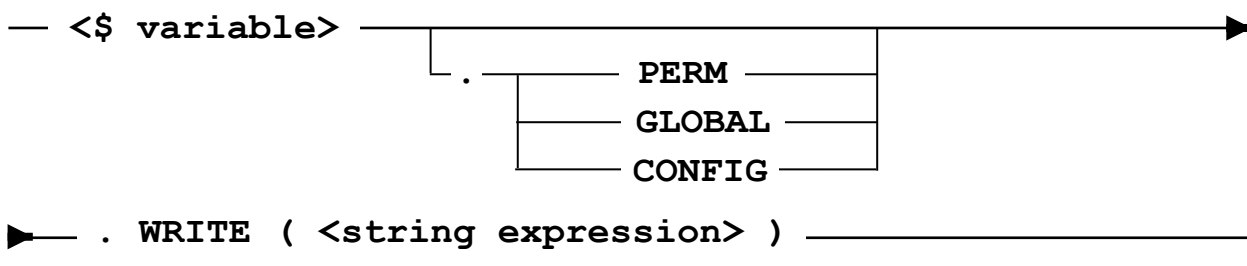
### SUPERVISOR Examples

```
DEFINE + ODTSEQUENCE DOIT:
  While #Loop.Sum(1) NEQ 10 Do
  Begin
    ODT('STARTJOB JOB/BATCH(" ', #LOOP, '"')');
  End;
```

### FLEX Example

```
SELECT FILEKIND=ALGOLCODE AND #Segs.Global.Sum(SEGMENTS) > -1
REP FOOT "TOTAL SEGMENTS FOR ALL FILES SCANNED=",#SEGS.Global
```

## **.WRITE** string method



Both SUPERVISOR and FLEX now have the ability to write symbolic or DATA files from any OPAL script. A new string variable method called WRITE (similar to READ or FILE) has now been implemented to provide this feature. Like the READ method, WRITE is only effective when used with a dynamic variable reference.

(i.e. using the \$\$ variant):

```
$Res:= $$Var.Write($TEXT)
```

In the above example, the variable \$VAR must hold a list of valid file attributes, including title, to determine the type of file being written. This attribute list is identical to that used by any file equation used in a WFL job. The data to be written to the file is held in the variable \$TEXT or this may be any string expression.

A more detailed example:

```
$Var:= "TITLE=*OUTPUT/MESSAGES, FILEKIND=JOBSYMBOL";  
$Text:= KeyIn("MSG");  
$Res:= $$Var.Write($TEXT);  
If $Res NEQ "OK" Then  
    Display("WRITE METHOD FAILED: ", $Res);
```

Here, WRITE will create a file called \*OUTPUT/MESSAGES with a FILEKIND of JOBSYMBOL and default blocking attributes associated with the FILEKIND. The above script will write the contents of the KEYIN call (i.e the response to the MSG command) and will respect end of line terminators in \$TEXT, creating new records where appropriate.

To create a byte stream file, the attribute list in \$VAR requires a change:

```
$VAR:="TITLE=*STREAM/FILE, FILEKIND=DATA, " &  
    "FILESTRUCTURE=STREAM";
```

If FILESTRUCTURE=STREAM is specified, other blocking attributes will be ignored. If the WRITE operation is successful, the method returns the string "OK" or "OK UPDATE" to the caller; any other value indicates that the file could not be written or the attribute list in \$VAR was incorrect.

Previously the EXTMODE is set to ASCII when writing a STREAM file if the EXTMODE has not been label-equated.

Any DATA or symbolic file can be written. If FILEKIND=DATA is used the caller may provide his own BLOCKSIZE/MAXRECSIZE specifications; any MAXRECSIZE or BLOCKSIZE specifications for all other FILEKINDS will be ignored. At the current time, it is not possible to create a DATA file with a MAXRECSIZE greater than 255 characters.

WRITE automatically permits data to be appended to an existing file but only if the file has been created by a previous .WRITE operation from SUPERVISOR or FLEX. Files created by the WRITE method have a special user-defined disk file attribute which marks them as Metalogic origin. If a WRITE operation is attempted to an eligible existing physical file, the WRITE will ONLY be permitted if this user disk file attribute is present and set to a preset value.

Any attempts to WRITE to any other existing file will fail with the result:

```
"Error: Not allowed:Ineligible resident file"
```

If a physical file is being appended by a WRITE operation, OPAL will use the file's blocking and structure characteristics at the time the file is opened. Any MAXRECVSIZE, BLOCKSIZE or FILESTRUCTURE attributes included in the attribute list in \$VAR will be ignored. "OK UPDATE" for a successful append.

Normal security restrictions and family substitution will apply in files created by the WRITE method. Because SUPERVISOR is privileged WRITE files can be created

under any usercode and subsequently appended. Note that if no usercode or \* is assigned to a TITLE assignment for SUPERVISOR WRITE operations, any files will be created under the USE USER FOR SUPERVISOR usercode.

No special file-creation privileges apply to the normal FLEX user.

Because of OPAL variable limits, a maximum of 1,999,999 characters may be written by any single WRITE operation.

The formatting of the output may be changed by assigning specific FORMID values to the file. If the file specification passed to WRITE includes a FORMID assignment then WRITE will check for the following string values:

RECORD	Data is written 'as is' to the file without any checks by WRITE for line termination (CR) in the Opal string.
LOG	As with RECORD but it is assumed that the caller is creating an Extract of one or more SUMLOGs. This data will have been created from a SUPERVISOR EVAL of a LOG context script. LOG file may also be appended by EVAL. Both LOGANALYZER and SUPERVISOR's EVAL command will be able to access these extract files. A new LOG context attribute, LOGBINARY, is now available to capture the raw log record. The file is created with a FORMID set to 'APPLICATION/X-MCP-LOG'.
METAFLAT	As with LOG but it is assumed that the caller is creating an extract of disk file directory. The intent is that the Extract file will be available for access by the FILES command using OBJECT/FLEX. The file is created with a FORMID set to 'APPLICATION/X-MCP-HEADER'.

The following example OPALs show how to create a log extract file:

```
DEFINE + ODTSEQUENCE WRITELOG2 (LOG) :  
  If Not LASTEVAL Then  
    $REC:= LOGBINARY & $REC  
  Else  
    Begin  
      $LOG:='TITLE=*EXTRACT/LOGA ON DEV,FORMID="LOG",NEWFILE=TRUE';  
      Show ($$LOG.WRITE ($REC) );  
    End;
```

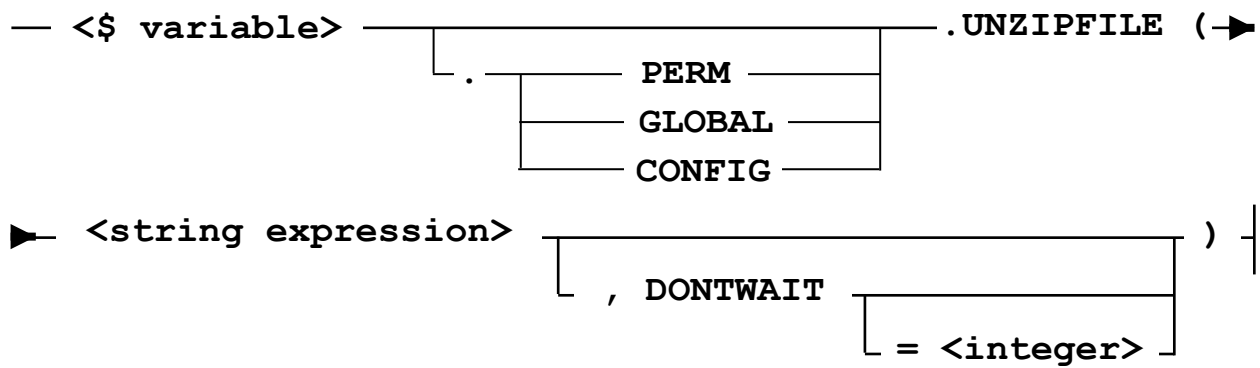
To extract all operator commands from all SUMLOGs over the past 3 days:

```
EVAL (OPERATOR:TRUE) [@BACK 3 DAYS FOR 3 DAYS] DO WRITELOG
```

Note that EVAL returns log entries in reverse chronological order so each new log entry is prefixed to the \$REC string: this mechanism is not intended to write large extract files.

## .UNZIPFILE

### string method



UNZIPFILE handles GZip, ZLib and MZip compression - the latter is Metalogic's own compression format and is intended currently for use on MCP systems.

UNZIPFILE is a method similar to `.READ` in that it uses dynamic variable references to access a physical file.

If the calls fails, the method returns the error message and the `MZIPLASTERROR` attribute is set to `MZIPERROR`. Each error message is prefixed by "Err".

The `<string expression>` is used to specify the title of the file to be unzipped.

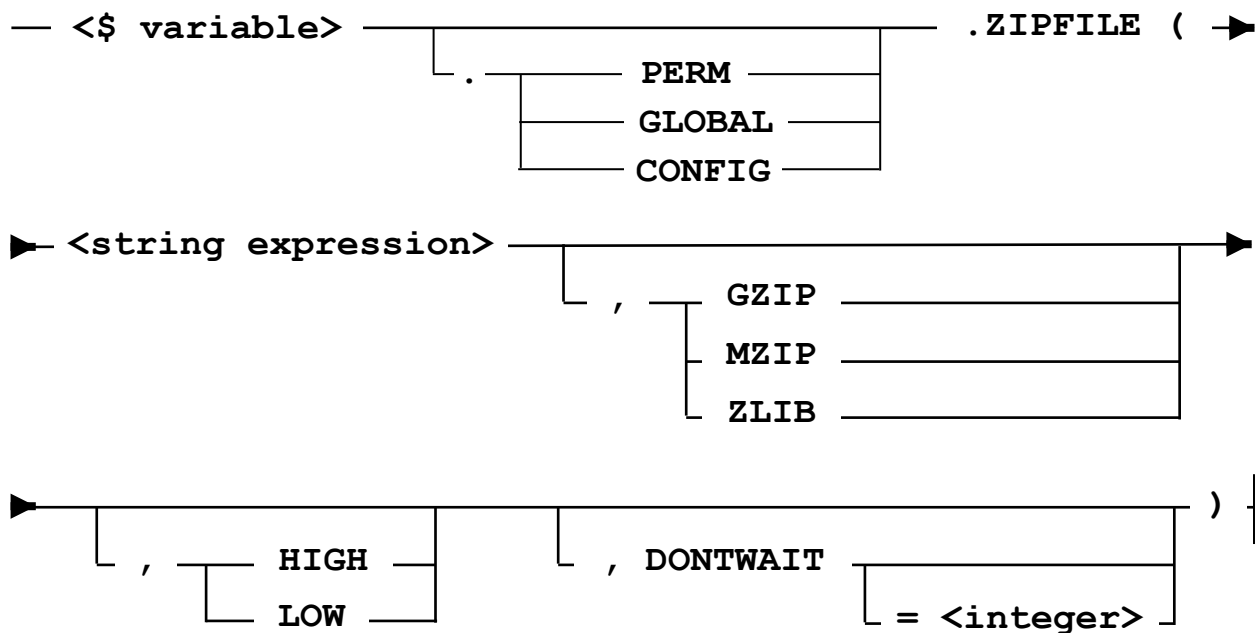
The optional `<Dontwait>` specification is only allowed when used in Supervisor. It allows the caller to track the progress of a zip/unzip operation from OPAL. A new Supervisor statement, called `ON PROGRESS`, is available which is regularly called by the zip routines to return updated progress information. Currently, only one progress attribute is available - `MYSELF(PROGRESS)`. The default refresh time for `DONTWAIT` is 1 second. Other times in seconds may be specified using the `= <INTEGER>` syntax.

Here is an example:

```
$SOURCE:="(META)BIG/FILE/GZ ON DEV";
$ZIP:="(META)BIG/FILE ON DEV";
$RES:= $$ZIP.UNZIPFILE($SOURCE,DONTWAIT);
If MZIPLASTERROR Neq MZIPOK Then
    Show("MZIP error ->", $RES);
Else
    ON PROGRESS DO
    Begin
        SHOW("UnZip progress ", MYSELF(PROGRESS), " %");
    End;
```

## .ZIPFILE

### string method



.ZIPFILE handles GZip, ZLib and MZip compression - the latter is Metalogic's own compression format and is intended currently for use on MCP systems.

In general, GZip is used for interchange of files between MCP and non-MCP systems. It offers the best compression, but requires more CPU to do it and to calculate the CRC-32 Checksum. MZip is most useful for interchange of files between MCP systems. It is optimised for speed on the MCP Architecture and it retains the MCP File Attribute information. ZLIB is useful for interchange of byte stream data, such as between a Browser and an HTTP Server. Although Gzip is also valid in an HTTP ACCEPT-ENCODING, ZLib (compress) uses less CPU for the Adler-32 Checksum.

If none of GZIP, MZIP or ZLIB are specified then GZIP is used.

GZIP will only compress FILESTRUCTURE=STREAM files.

ZIPFILE is a method similar to .READ in that it uses dynamic variable references to access a physical file.

The `<string expression>` is used to specify the title of the file to be unzipped.

The optional `<Dontwait>` specification is only allowed when used in Supervisor. It allows the caller to track the progress of a zip/unzip operation from OPAL. A new Supervisor statement, called ON PROGRESS, is available which is regularly called by the zip routines to return updated progress information. Currently, only one progress attribute is available - MYSELF(PROGRESS). The default refresh time for DONTWAIT is 1 second. Other times in seconds may be specified using the `= <INTEGER>` syntax.

To create a zip file:

```
$ZIP:=" (META)BIG/FILE/GZ ON DEV" ;
$SOURCE:=" (META)BIG/FILE ON DEV" ;
$RES:= $$ZIP.ZIPFILE($SOURCE,MZIP,HIGH) ;
If MZIPLASTERROR Neq MZIPOK Then
    Show("MZIP error ->", $RES) ;
```

Here the calling OPAL will wait until the ZIPFILE operation is complete and the slot is marked as 'WAIT ZIP' if interrogated using an EV ? or SLOT command.

If the call fails, the \$RES variable is assigned the error message and the MZIPLASTERROR attribute is set to MZIPERROR. Each error message is prefixed by "Err".

A compression model identified by HIGH or LOW may be specified. HIGH compression yields smaller archives but consumes more CPU; LOW compression is the default.

The optional <Dontwait> specification allows the caller to track the progress of a zip/unzip operation from OPAL. A new statement, called ON PROGRESS, is available which is regularly called by the zip routines to return updated progress information. Currently, only one progress attribute is available - MYSELF(PROGRESS).

Here is an example:

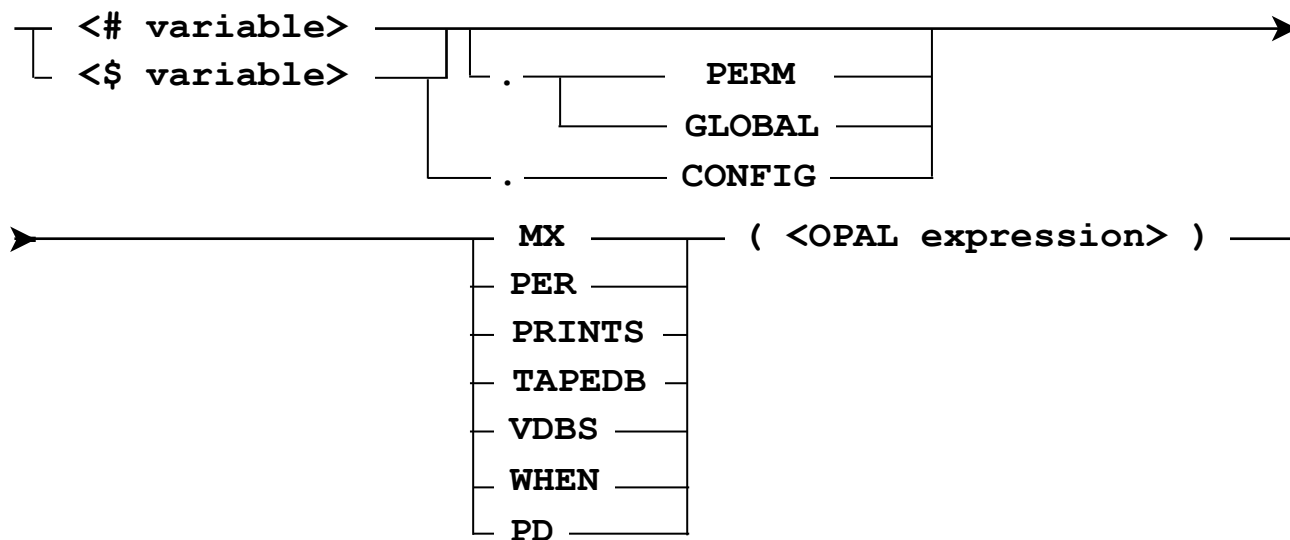
```
$ZIP:=" (META)BIG/FILE/GZ ON DEV" ;
$SOURCE:=" (META)BIG/FILE ON DEV" ;
$RES:= $$ZIP.ZIPFILE($SOURCE,MZIP,HIGH,DONTWAIT) ;
If MZIPLASTERROR Neq MZIPOK Then
    Show("MZIP error ->", $RES) ;
Else
    ON PROGRESS DO
        Begin
            SHOW("Zip progress ",MYSELF(PROGRESS)," %") ;
        End;
```

By default, SUPERVISOR triggers ON PROGRESS every second but this may be overridden by assigning an integer value to DONTWAIT:

```
$RES:= $$ZIP.ZIPFILE($SOURCE,MZIP,HIGH,DONTWAIT=5) ;
```



## Object Variable Methods



Object variable methods allow retrieval of object attribute information external to the current object environment. The object is determined by the value held in the OPAL variable and the type is specified by the method name.

For each of these methods the object is identified by an integer value held as an integer in <# variable> or as a string in <\$ variable>. The <OPAL expression> can use attributes belonging to SYSTEM or the context identified by the method name.

The [OBJECTS](#) function is often used to build a list of objects which can then be further interrogated using the appropriate Object Variable method.

**.MX**

The MX method expects a valid mix number in the variable.

For example:

```
#A:=12345;  
$Name:= #A.Mx(NAME)
```

```
$M:="12346";
$Text:= $M.Mx(#(UserCode,,Priority,,SourceStation))
```

**.PD**

The PD method (only valid with string variables) expects a valid file title as a parameter and allows access to any attributes in the PD context.

For Example:

```

DEFINE + ODTSEQUENCE PDTEST:
    $FILE:= "*SYSTEM/FILEDATA ON DISK";
    SHOW("INFO = ", $FILE.PD(#(RELEASEID, , CREATIONDATE)))

```

## **.PER**

The PER method expects a valid unit number in the variable.

For example:

```
#U:=51;  
Show("LABEL = ",#U.Per(Label)  
  
$U:="51";  
$UInfo:= $U.Per(#(UnitNo,,Label))
```

## **.PRINTS**

The PRINTS expects a valid print request number in the variable.

For example:

```
#P:=12345  
$Name:= #P.Prints(Task)  
  
$PS:="12346";  
$Text:= $PS.Prints(#(RequestNo,,UserCode,,JobName))
```

## **.TAPEDB**

The TAPEDB method expects a valid tape serial number in the variable.

Since a tape serial number is always a string, the TAPEDB property will automatically convert an integer value into a leading zero string e.g. the value 1 is converted to "000001". For example:

```
#Tape:=1;  
Show(#Tape.TapeDB(Title)  
  
$Serial:="51";  
$Info:= $Serial.TapeDB(#(Density,,Title))
```

## **.VDBS**

The VDBS method expects the mix number of a Database in the variable.

The VDBS context allows retrieval of database attribute information, using the Visible DBS interface, such as OVERLAYRATE and ALLOWEDCORE.

Example:

```
#DBS:=1234;  
Show("Allowedcore =", ,#DBS.VDBS(ALLOWEDCORE))
```

## .WHEN

The WHEN method expects a valid Supervisor When slot number in the variable.

Example:

```
$Slots:=Objects(When:User="BOB");  
While $Slot:=$Slots.Split Neq Empty DO  
    Show($Slot.WHEN(#(SituName," + ",ODTSName)))
```

Defining the following ODTS would allow TT DO META\_SLOTRATE to give a display of all of the active slots sorted by CPU rate.

Example

```
TT DEFINE + ODTSEQUENCE META_SLOTRATE:  
$Slots:=Objects(When:OdtsName Neq WhenId(Action)); %ignore ourself  
While $Slot:=$Slots.Split Neq Empty Do  
BEGIN  
    $Item:=$Slot.WHEN(#(String8(  
        #Rate:=Integer(CpuTime*100000/Elapsed)/1000,6),,  
        If SituType > 0 Then #(SituType,,Situname,,) Else Empty,  
        OdtsType,,OdtsName));  
    #Done:=0;  
    $New:=Empty;  
    While #Done=0 And $Lst Neq Empty Do  
    Begin  
        $Tmp:=$Lst.Split(/)&#(/);  
        If Decimal(Take($Tmp,6)) > #Rate Then  
            $New:=$New&$Tmp  
        Else  
            Begin  
                $Lst:=$New&$Item&#(/)&$Tmp&$Lst;  
                #Done:=1;  
            End;  
        End;  
    End;  
    If #Done=0 Then $Lst:=$New&$Item&#(/);  
End;  
Show("Active slots sorted by % Cpu rate",/ /,$Lst)
```

An example of its use.

```
TT DO Meta_SlotRate
Active slots sorted by % Cpu rate
0.077 DO REC_STATUS
  0.02 DO REC_PKSPACE
0.015 WHEN REC_MSG DO REC_MSG
0.014 WHEN REC_W DO REC_W
0.013 WHEN REC_C DO REC_C
0.012 WHEN REC_BOJ DO REC_BOJ
0.008 WHEN REC_GOING DO REC_GOING
0.005 DO MENU_FAMCHK
0.003 WHEN HTTP_KEEPALIVE DO HTTP_KEEPALIVE
0.003 WHEN TCPIPATTACH DO TCPIPATTACH
0.003 WHEN META_MSGMON DO META_MSGMON
0.002 WHEN TPDB_SILOW DO TPDB_SILOW
0.002 WHEN REC_SEC DO REC_SEC
0.002 WHEN XREF DO XREF
0.002 WHEN META_W DO META_W
0.001 WHEN REC_MCSSEC DO REC_MCSSEC
0.001 WHEN TPDB_AUTOCD DO TPDB_AUTOCD
0.001 WHEN META_MAKELIVE DO META_MAKELIVE
0.001 WHEN BOB_WEBSECURITY DO BOB_WEBSECURITY
0.001 WHEN META_SLOTCHECK DO META_SLOTCHECK
0.001 WHEN BOB_CLEANUP DO BOB_CLEANUP
```

# Expressions

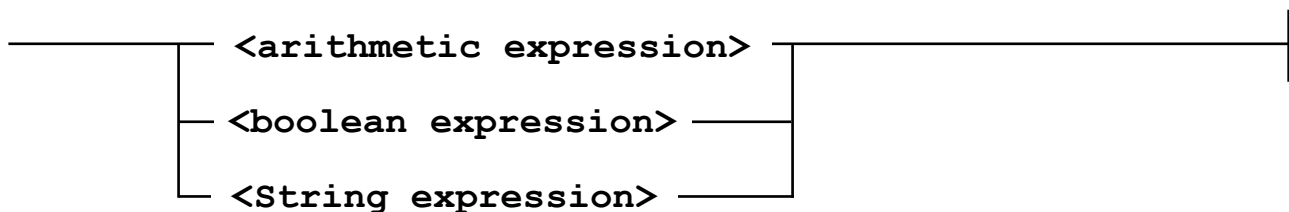
An expression is a meaningful combination of basic elements which on evaluation yields a result. The result may be of type Boolean, Arithmetic (Integer or Real), or String. The syntax for expressions in OPAL is closely based on that in Unisys Work Flow Language as defined in the **A Series WFL Reference Manual**.

Great care has been taken to ensure that programming in OPAL is natural for anyone well versed in WFL programming. Each of the different types of expression will be discussed separately, with emphasis on the differences between OPAL and WFL.

The layout of the sections in this chapter is first to describe the syntax formally using railroad diagrams, then to expand on the elements that are unique to OPAL in the semantics. Any syntax element that is not defined is the same as the entity of the same name in the WFL manual.

The general syntax for an OPAL expression is shown below:

## <OPAL expression>

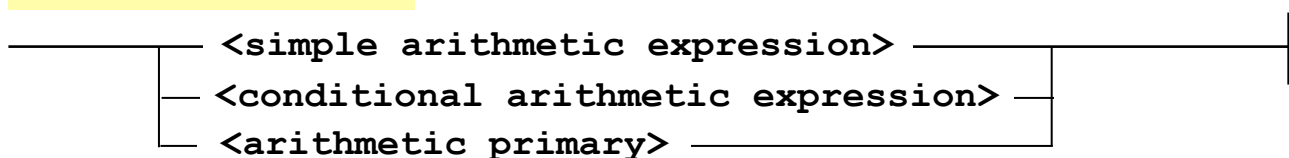


These expression types will be discussed in the remainder of this chapter. There are many similarities between the Metalogic and Unisys implementations of expressions in the languages such as ALGOL and WFL but OPAL has several interesting variations.

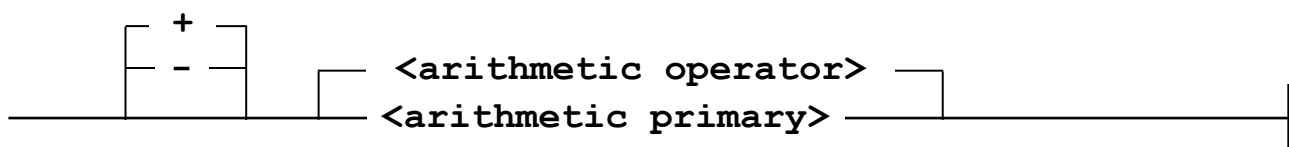
## Arithmetic Expression

Arithmetic expressions perform specified operations on designated arithmetic primaries to return numerical values.

### <arithmetic expression>



### <simple arithmetic expression>



### <arithmetic operator>

	+	
	-	
	*	
	/	
	MOD	
	DIV	
	**	

The operators +, -, \*, / and \*\* have the usual mathematical meaning of addition, subtraction, multiplication division and exponentiation. No two operators can be adjacent and implied multiplication is not allowed.

### <arithmetic primary>

	<unsigned number>	
	<arithmetic attribute>	
	<arithmetic function>	
	<arithmetic assignment>	

### <unsigned number>

	<integer>	
	<simple real>	

Real and integer constant values may only be represented as whole or decimal values e.g. 99, 1.2345

### <arithmetic attribute>

<arithmetic attribute> is defined in [OPAL Attributes](#).

**Some Supervisor examples of <arithmetic attribute>:**

**CPUS**

**LOGMAJOR**

**ACCUMPROCTIME**

### <arithmetic function>

<arithmetic function> is defined in [Opal Functions](#).

Some Supervisor examples of <arithmetic function>:

**ABS**

**COUNT**

**MAIL**

**SQRT**

## <arithmetic assignment>

— #<identifier> — := — <arithmetic expression> —————|

An Opal identifier may not be more than 17 characters in length and can only include alphanumeric characters or '\_'.

Opal <arithmetic variables> are discussed at length in [Opal Variables](#).

Examples of <arithmetic primary> usage

In Supervisor:

```
1.23
#PROC
#Cnt:=#Cnt+1-CPUS
Count (Mx:Name="METALOGIC/JAMPACK" And ElapsedTime>300)
Wait (Max (Min (60,Decimal (Text) ),20) )
(CUAvail/4096)/20
```

## <conditional arithmetic expression>

———— IF     <boolean expression> —————>

➤———— THEN     <arithmetic expression> —————>

➤———— ELSE     <arithmetic expression> —————|

<boolean expression> is covered in more detail in the next section.

Some Supervisor examples:

```
If ProcTime+IOTime < 1 Then 0
                        Else WSIntegral/(ProcTime+IOTime)
#A:=If #B>0 Then #B Div #C Else 0
```

## <case arithmetic expression>

— CASE <arithmetic expression> OF —————>

➤———— (     <arithmetic expression>     ) —————|

This provides a means of selecting one of many alternative expressions by a number derived from the value of the first expression from 0 through N-1, where N is the number of <arithmetic expressions>s in the list.

Note that no ELSE support is provided; if a CASE index exceeds the entries in the list, the calling OPAL script will be terminated.

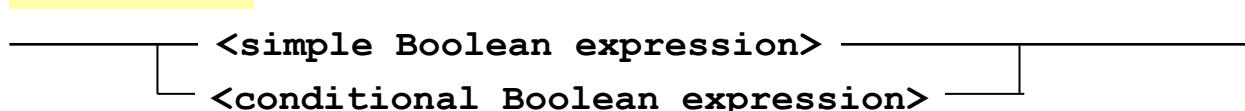
Supervisor examples:

```
#MonthLetters:=Case Month Of (8,8,5,5,3,4,4,7,9,7,8,8)
#Var:=Case #A+2 Of (#B+23,#C,11,0)
```

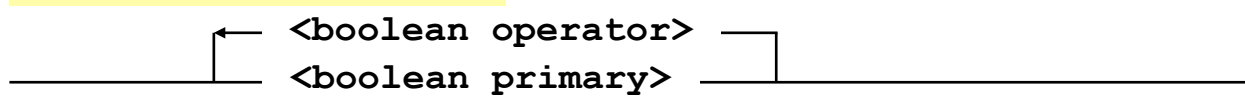
## Boolean Expression

Boolean expressions return logical values (TRUE, FALSE) by applying the specified operations to designated Boolean primaries.

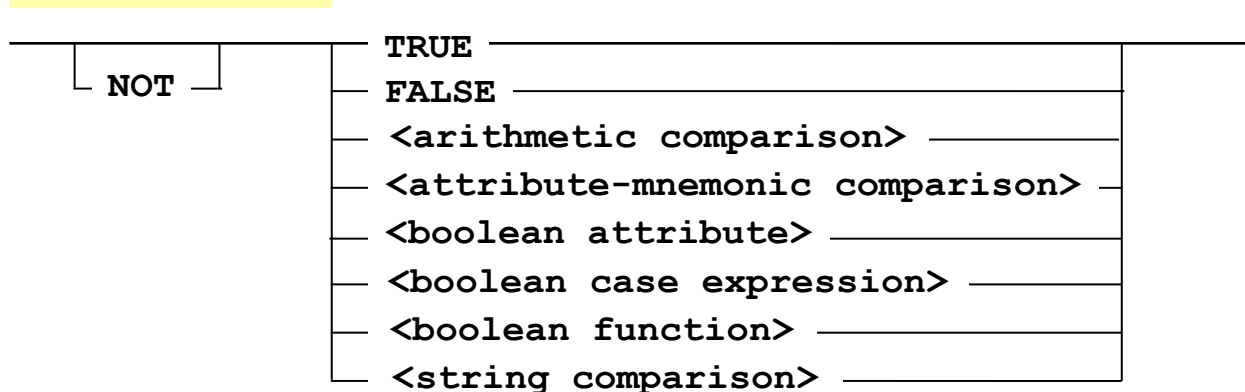
<boolean expression>



<simple boolean expression>



<boolean primary>



<boolean operator>



The order of evaluation of Boolean expressions is identical to that in WFL. The Boolean operations OR, AND and IMP are evaluated like the NEWP conditional Boolean operations COR, CAND, and CIMP. For example,

A AND B	is evaluated as	IF A THEN B	ELSE FALSE
A OR B	is evaluated as	IF A THEN TRUE	ELSE B
A IMP B	is evaluated as	IF A THEN B	ELSE TRUE

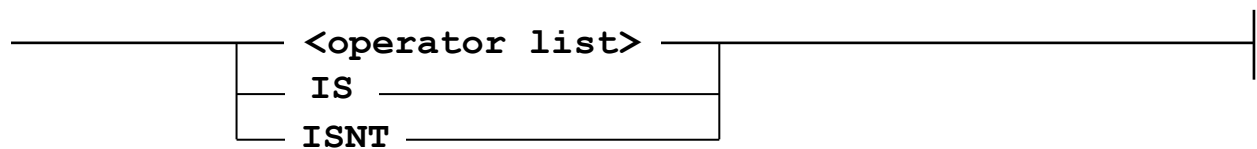
<boolean attribute>::= <attribute primary>



<arithmetic comparison>

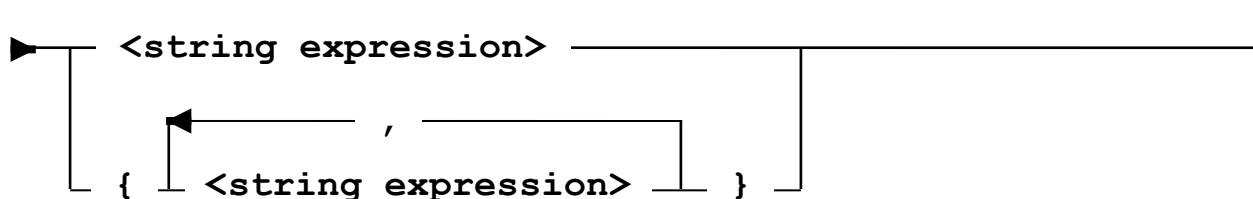
► `<arithmetic expression>`

<relational operator>



LEQ
<=
LSS
<
EQL
=
NEQ
GTR
>
GEQ
>=

- `<string expression>` — `<string relational operator>`  $\rightarrow$



Page 57

<string relational operator>

<relational operator>	
EQW	
HDIS	
ISIN	
INCL	
TLIS	

## EQW Operator

**EQW** is a string relational operator that uses wildcards. It has the form:

— <string expression> — **EQW** — <pattern list> —

<pattern list> ::= <string expression> | {<string expression list> }

A <pattern list> contains one or more string expressions, which are evaluated to a wildcard pattern.

The wildcard characters are:

Wildcard	Action
"="	match any string, including an empty string
"?"	match any single character
"&"	match any alphanumeric character ( upper and lower case letters and digits)
"@"	match any upper or lower case alphabetic character
"#"	match any numeric digit
"~"	matches any string not containing "/"
"\"	Escape character use before any other wild card to use the literal value of the character not its wildcard action.  Ex. =A\\=B= would match any string containing A=B

All other characters, graphic or not, will be matched exactly.

## SUPERVISOR Examples

```
"PDSEGMENTS" EQW "=SEG=" => TRUE
"FLEX93125B/file0A" EQW "?=####@/file&&" => TRUE
"CREATIONDATE" EQW {"=DAY=", "=DATE="} => TRUE
"SYSTEM/COMS" EQW "=TEM~COMS" => FALSE
```

## FLEX Examples

```
Select FileId(Title,0) EQW "META="
Select Title EQW "*SYSTEM/=SUPPORT"
```

## ISIN and INCL Operators

Many long and complex string expressions in WFL are really to determine whether one string appears in another. The **ISIN** operator simplifies this process. It returns TRUE if the evaluated string in the left-side operand appears anywhere as a substring in the evaluated right-side operand. For example:

```
"CDE" ISIN "ABCDEF" - is TRUE
"ECD" ISIN "ABCDEF" - is FALSE
```

Although it is an extension on the syntax of WFL, **ISIN** does appear in the syntax of Unisys ERGO and SMFII Program Products with exactly the same definition as in OPAL except for the ability in OPAL to use **ISIN** with lists.

Since a list may not appear on the left hand side of an operator, the **INCL** (short for INCLUDES) operator is provided. **INCL** is related to **ISIN** by the equivalence

```
A IsIn B    is identical to    B Incl A
```

For example:

```
Name Incl {"*OLD/", "*LOG/", "*TEST/", "*DASDL/"}
```

Note the use of curly braces { and } to allow the construction of lists. **NISI** (**ISIN** reversed) is an older, non-preferred synonym for **INCL**.

## SUPERVISOR Examples

```
DEF + SITU NAME_CHECK(MX) :
    Name Incl {"/TEST/", "/DEMO/"}

DEF + SITU RSVP_CHECK(MX=W) :
    "SECTORS REQUIRED" IsIn RSVP
```

## FLEX Example

```
Select Title Incl {"ABC", "DEF", "PQR"}
```

## HDIS Operator

Most uses of the TAKE function in WFL are to see if the first few characters of a string are equal to another string. TAKE is one of the few places in WFL where it is necessary to count up the number of characters in a literal. The HDIS ("head is") operator is a simpler and more efficient way to express this.

For example, if the SUPERVISOR MSG attribute **TEXT** contains "**WXY**" and we write the following SITUation:

```
DEFINE + SITUATION HDIS (MSG) :  
    Text HdIs "WX"
```

Then the SITUation will return TRUE.

Using TAKE as in:

```
Take (Text,1) = "A" OR Take (Text,3) = "BCD"
```

is the same as

```
Text HdIs {"A", "BCD"}
```

## TLIS Operator

Using TAKE to look at the end of a string is even uglier. The TLIS ("tail is") operator corresponds to the HDIS operator but matches the end of the string. As above, if the SUPERVISOR MSG attribute TEXT contains "WXY" then the following SITUation returns TRUE:

```
DEFINE + SITUATION TLIS (MSG) :  
    Text TlIs "XY"
```

Using TAKE as in this FLEX program:

```
SELECT Take (Title,Length (Title)-2) = "FS" Or  
       Take (Title,Length (Title)-4) = "DECS"
```

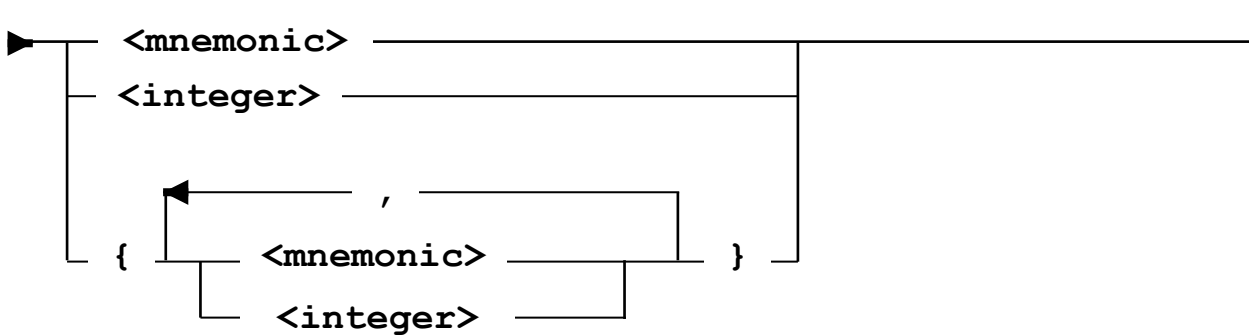
is the same as

```
SELECT Title TlIs {"FS", "DECS"}
```

(TITLE is a FLEX attribute indicating the name of the file currently being scanned.)

## <attribute-mnemonic comparison>

– <attribute primary> — <relational operator> —————→



Each OPAL <attribute primary> can have its own set of mnemonic values which only apply to that attribute, though it should be noted that not all attributes have mnemonics. This set of mnemonics is shown in the response to a TT HELP ATTR <attribute> command from Supervisor.

Many OPAL task and file attributes which have identical functionality to their A-Series equivalents, such as FILEKIND, EOJREASON and KIND, will automatically use the same set of mnemonics that is provided by the MCP to the normal compilers. Usually, the Supervisor HELP text for these attributes will include the following reference.

e.g. for the Density attribute:

```
HELP ATT DENSITY:PER
---- HELP ATTRIBUTES ----
DENSITY      (PER) Returns INTEGER mnemonic
              Mnemonic values : As for File attribute DENSITY
                  BPI800(0) FMTLT05(1) BPI1600(3) BPI6250(4) BPI38000(5) BPI1250(6)
                  BPI11000(7) FMT36TRK(8) FMTDDS2(9) FMTQIC1000(10) FMTDDS3(11) FMTST9840C(12)
                  FMTDLT3(13) FMTDLT6(14) FMTDLT10(15) FMTDLT20(16) FMTDLT35(17) FMTST9840(18)
                  FMTDDS(19) FMTDDS4(20) FMTAIT(21) FMTAIT2(22) FMTDLT40(23) FMTST9940(24)
                  FMTLT0(25) FMTDLT110(26) FMTLT04(27) FMTDLT160(28) FMTDDS5(29) FMTDLT300(30)
                  FMTLT03(31) FMTT10KA(32) FMTT10KB(33) FMTST9840D(34).
              Semantics : DENSITY returns a mnemonic value for the density associated with
                  the unit. Applies only to tapes.
```

For the DENSITY attribute shown above, valid mnemonic values would be BPI11000, FMT36TRK, FMTDDS2, FMTQIC1000( etc.

The OPAL compiler will also allow attributes that normally return mnemonic values to be optionally compared with integer values as well. In a <mnemonic list>, both integer and mnemonic values are permitted.

So, the following relational expressions are allowed:

```
EXCEPTIONREASON NEQ 75
STOPREASON GTR 21
STACKSTATE IS {FROZEN, WAITING}
FILEKIND ISNT DCALGOLCODE
KIND EQL TAPE
KIND NEQ {DISK, CD, 17, 45}
```

**<boolean case expression>**

— CASE — <arithmetic expression> — OF —————>

▶ — ( — ————> , ————> <boolean expression> ————> ) —————>

The < Boolean case expression> provides a convenient means of selecting one of many alternative expressions. The <boolean expression> to be evaluated is selected as follows: first, the <arithmetic expression> is evaluated; this value is then used as an index into the <boolean expression>s within the parentheses.

The expressions are numbered sequentially from 0 through N-1, where N is the number of <boolean expressions>s in the list. The expression selected by the index is then evaluated and its value becomes the value of the <case boolean primary>. If the value of the index lies outside the range 0 through N-1, the OPAL program is discontinued with the message

**BAD INDEX TO CASE EXPRESSION**

Example:

**If Case #A OF (TRUE, #B=1, FALSE, #C=0) Then ....**

**<conditional boolean expression>**

— IF <boolean expression> ———— THEN —————>

▶ — <boolean expression> — ELSE — <boolean expression> ————>

The <conditional boolean expression> has no scope limitations and is ideally suited for usage in a Supervisor SITUation program or FLEX SELECT, since these program types always evaluate to a Boolean value.

SUPERVISOR Example:

```
DEFINE + SITUATION PER_CHECK(PER=MT) :  
  If AvailableToGroup Then  
    (Scratch And Not InUse)  
  Else  
    False
```

## Expression lists

It is quite common in OPAL to test an attribute to see if it is one of a series of values. This would be done in WFL or ALGOL by writing expressions of the form:

**A = 10 OR A = 20 OR A = 30**

OPAL allows convenient shorthand for such condition checks:

```
#A = {10,20,30}
```

This is not only faster to write but is also faster to execute because the left hand side (**A**) need only be evaluated once.

If the **NEQ** operator is used with lists:

```
#A NEQ {10,20,30}      is equivalent to  
#A NEQ 10 AND #A NEQ 20 AND #A NEQ 30
```

This facility also applies to string list comparisons e.g.

```
$Str = {"ABC","DEF","GHI"}      is equivalent to  
$Str = "ABC" OR $Str = "DEF" OR $Str = "GHI"
```

## String Expression

OPAL can generate strings from many different types of functions, attributes and expressions. These strings can also be readily combined from constants, arithmetic and/or condition expressions. This chapter briefly describes the types of strings that can be used in OPAL which are described in more detail elsewhere in this manual.

OPAL strings/expressions can be a maximum of 1,999,999 characters in length. Both SUPERVISOR and FLEX are capable of handling strings of such magnitude.

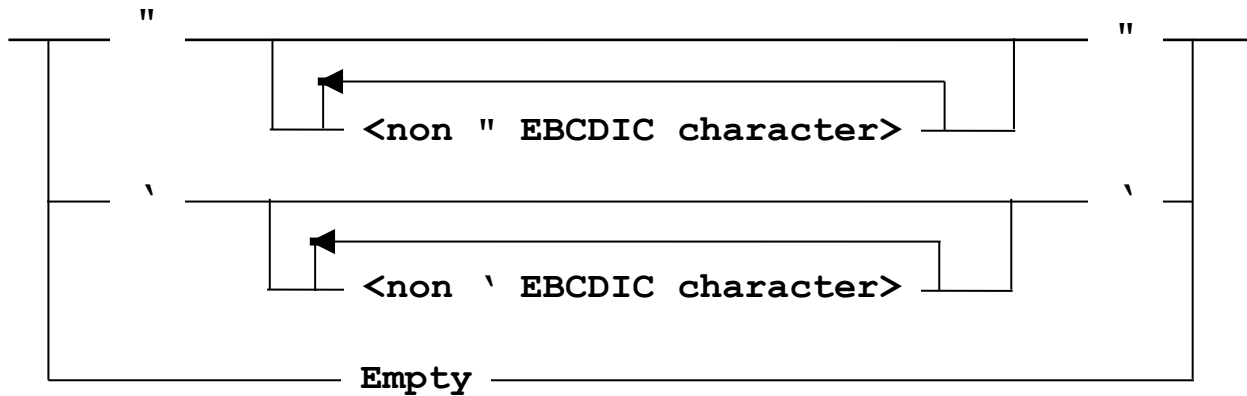
OPAL is record oriented, with each logical line 80 characters long. While reserved words and symbols cannot be split across line boundaries, string constants can be continued across line boundaries.

<string primary>

—	<string constant>	—
—	<string attribute>	—
—	<string function>	—
—	<string variable>	—
—	<string expression>	—
—	<conditional string expression>	—
—	<case string expression>	—
—	#(<Opal String>)	—
—	<string assignment>	—
—	( <string expression> . )	—

A <string primary> can exist in many forms, from a simple quoted string to a complex combination of all of the above. The following pages will show the format of each primary type.

## <string constant>



Some simple examples:

```
"ABCDEF"  
EMPTY  
"ABC WITH A "" IN THE MIDDLE"  
'ABC WITH A " IN THE MIDDLE'
```

Note that, similar to WFL, a double quote (""") is required to insert a single quote into a string literal. Unlike WFL a single quote character ' may be used as a string delimiter.

For example, in OPAL

```
"RUN *SYSTEM/DUMPALL ("TEACH") "  
Or  
'RUN *SYSTEM/DUMPALL ("TEACH") '
```

evaluates to:

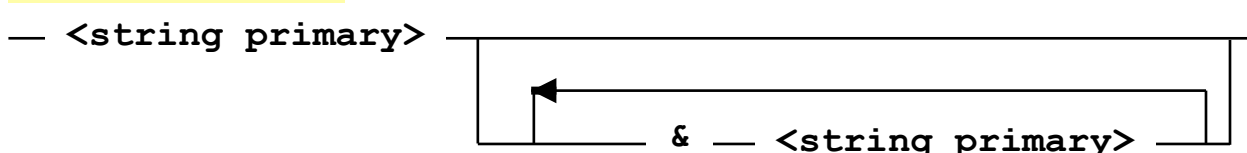
```
RUN *SYSTEM/DUMPALL ("TEACH")
```

The reserved word EMPTY can be used in place of the empty string "".

<string attribute> is explored in [OPAL Attributes](#).

<string function> is explored in [Opal Functions](#).

## <string expression>

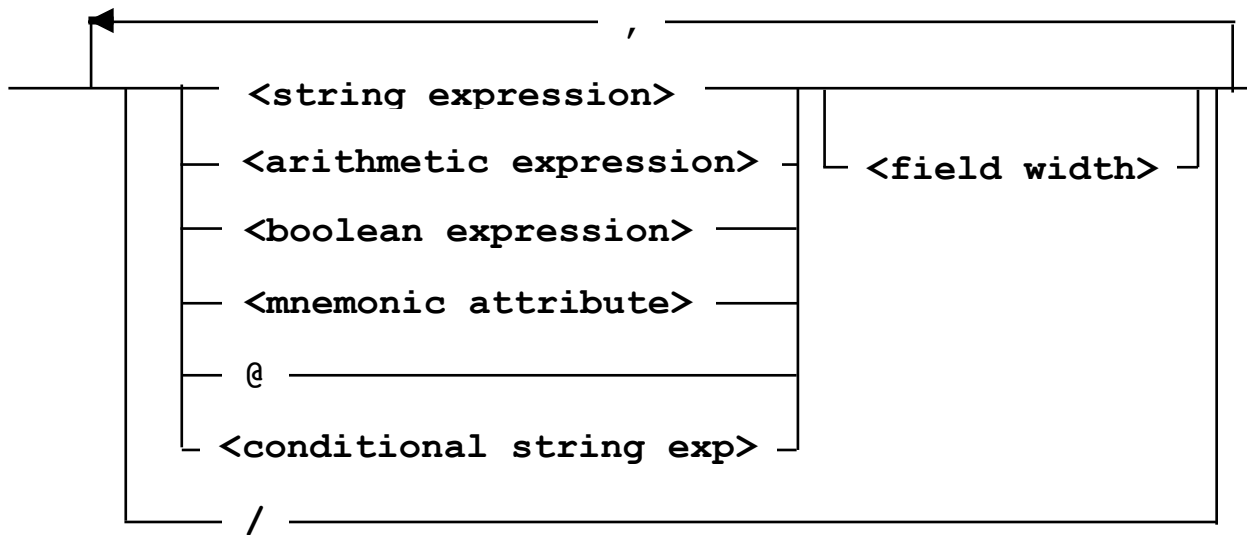


Examples:

```
"MCP NAME IS" & MCP  
$TEST & " Returns the contents of the string TEST"  
"Mixing Strings " & $A & " and Reals " & String(#Z,*)
```



<OPAL string>



An <OPAL string> is very versatile, allowing coercion of boolean and arithmetic expressions into strings. They may only be used in simple expressions – usage in conditional expressions is not possible unless the Hash-Paren function is used. A comma may be used as a concatenation symbol; inside an OPAL string, each concatenation operation can occur between different types – unlike the & operator which only works with string expressions.

The '/' character signifies a new line i.e. carriage return-line feed. The '@' character has special significance and represents the attributes UNITNUMBER and MIXNUMBER if the PER and MX contexts respectively.

The semantics of <OPAL string> and <lookup functions> are dealt with in more detail in [OPAL Reporting](#).

### #(..) or the 'Hash-Paren' construct

The OPAL Hash-Paren construct allows the use of <OPAL string> where only a <string expression> could previously be used.

An example is shown below:

```
$Var := ( MCP & $A & "/" & $MyStr )
```

The Hash-Paren construct, invoked using the syntax #(<OPAL string>), can change the above into a more readable expression.

Example:

```
$Var := # ( MCP, $A, "/", $MyStr )
```

Consider the handling of conditional strings in Opal DISPlays: the following is a typical example, the use of the concatenation operator '&' and the requirement that the components used in the <string expression> must be all string entities, make this expression complex and difficult to read.

```
Label,/,If ExtraEntry Then
    "TOTAL"&STRING(#T,6)&"#[CR]DONE" Else ""
```

Alternatively, using Hash-Paren can help to clarify the expression. Inside the # ( ) construct, normal <OPAL strings> may be used allowing commas as string concatenators , field lengths and ' / ' to indicate new lines.

```
Label,/,If ExtraEntry Then #("TOTAL ",#T 6,/,,"DONE")
      Else ""
```

All the usual features of OPAL strings can be used inside the # ( ) construct. This includes <lookup functions> which will be dynamically evaluated by the OPAL machine where appropriate. For example:

```
$Marc:= "#[MX=MARC]"      will still return #[MX=MARC]
```

but

```
$Marc:=#("#[MX=MARC]")    will return e.g. 1234,1235
```

In SUPERVISOR, this can be very useful instead of having to resort to COUNT (MX: ... or COUNT (PER=PK: ... to get pack or mix information.

#### <string assignment>

—— \$<identifier> — := — <string expression> —————|

Both <case string expression>s and <conditional string expression>s are available in OPAL. This is unusual because they are not available in either WFL or ALGOL.

#### <conditional string expression>

———— IF <boolean Expression> —————>  
➤ ——— THEN <string expression> —————>  
➤ ——— ELSE <string expression> —————|

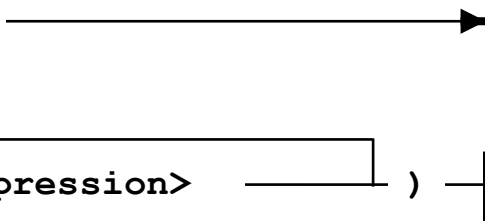
#### SUPERVISOR Example

```
DEFINE + DISPLAY UTIL(PER) :
  If RFErrors > 0 Then
    #("ERRORS FOR PK", UnitNo, " = ", RFErrors)
  Else
    Empty
```

#### FLEX Example

```
REPORT Title 50, SecurityClass,
  If SecurityClass = {GUARDED, CONTROLLED}
  Then "BY " & SecurityGuard
  Else Empty
```

<case string expression>

– CASE <arithmetic expression> OF 

The <case string expression> provides a convenient means of selecting one of many alternative expressions. The <string expression> to be evaluated is selected as follows: first, the <arithmetic expression> is evaluated; this value is then used as an index into the <string expression> list within the parentheses.

The expressions are numbered sequentially from 0 through N–1, where N is the number of <string expressions>s in the list. The expression selected by the index is then evaluated and its value becomes the value of the <string primary>.

If the value of the index lies outside the range 0 through N–1, the OPAL program is discontinued with the message:

**BAD INDEX TO CASE EXPRESSION**

Note that the <string expression>s need not be the same length in the clauses of **IF** or **CASE**, and that the **MEMBER** and **INDEXOF** functions are often useful in generating an integer index for use with **CASE**.

## SUPERVISOR Examples

**DEFINE + DISPLAY CAPITALS:**

Case #A Of ("LONDON", "PARIS", "ROME")

**DEFINE + DISP TP\_DENSITY (PER) :**

Label,

Case Member (DENSITY, BPI38000, BPI6250,  
BPI1250, BPI1600, ELSE) OF  
("38000", "6250", "1250", "1600", "\*UNKNOWN\*")

## FLEX Example

**REPORT Title, Case #TYPE Of ("FLAT", "CODE",  
"DBDATA", "UNKNOWN")**

# OPAL Functions

OPAL Functions are similar to those in WFL. They have a type, such as REAL or STRING, and an expected parameter structure. Unlike WFL, the functions of OPAL are numerous and powerful. They are listed here in alphabetical order. Variable methods are easy to mistake for functions so they are also listed here with links to their definition.

However, in OPAL there is a strict distinction between constructs which manipulate data in the language environment, which are known as **Functions**, and constructs which give information about the environment outside the language, known as **Attributes**.

Thus, in WFL **SYSTEM** returns information about the machine hardware and is a Function, while in OPAL it is an Attribute. Attributes are discussed in detail in [OPAL Attributes](#).

## ABS

[arithmetic function](#)

— **ABS** — ( — <arithmetic expression> — ) —————|

As in ALGOL, ABS returns the value of <arithmetic expression> as a positive real number.

Example

If a SUPERVISOR **DEF + DISP** or a FLEX **REPORT** contained the following statements:

```
#Diff:=-6
Abs (#Diff)
```

The above **ABS** call would return the value 6.

**See also:**

[NABS](#)

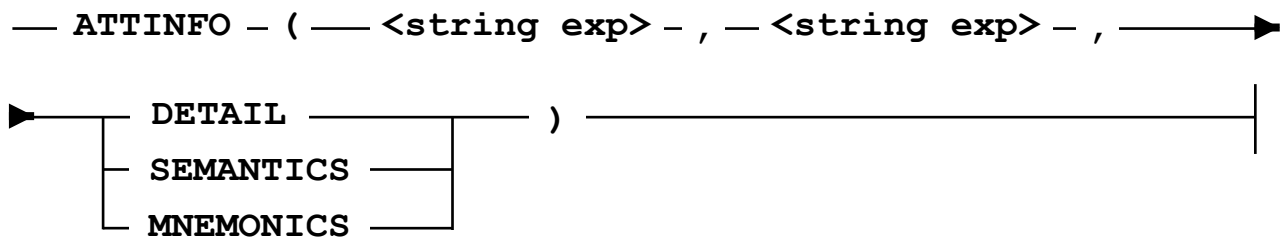
## .ACCUM

[arithmetic method](#)

[Definition](#)

# ATTINFO

## String function



The ATTINFO function returns selected information about an attribute.

The first parameter specifies the name of the attribute.

The second parameter defines the Context to which the Attribute belongs.

The third parameter selects the type of information to be returned.

If DETAIL is specified, then a list of name=value; elements are returned, which may be dereferenced using the \$.Distribute method.

These names are defined,

- AtLink- Returns the name of a <context> if the value of the attribute can be used as the Cardinal Attribute of the linked context, otherwise EMPTY.
- AtKind - If the attribute has \$AtLink, "DO" is returned if the attribute is a link to another object, otherwise "EVAL" is returned if the attribute is a link to a list of objects.
- AtProxy- If the attribute has \$AtLink, a "1" is returned if the attribute is a proxy attribute and represents other information for the web interface.
- AtArgs - Returns the number of parameters.
- AtType - Returns the user type of the attribute (eg. INTEGER).
- AtValue- Returns the format of the attribute value (eg. SECONDS).
- AtErr - If an error occurs, the reason for the error, otherwise it is empty.

If there is an error, AtErr contains the reason, otherwise all names are returned, so that a \$.Distribute will either assign a value, or reset the corresponding variable.

Examples:

```
AttInfo("LibUserList","Mix",DETAIL) returns
  AtLink=MIX;AtKind=EVAL;AtProxy=;AtArgs=0;
  AtType=STRING;AtValue=STRING;AtErr=;
AttInfo("UseTime","PD",DETAIL) returns
  AtLink=;AtKind=;AtProxy=;AtArgs=0;
  AtType=REAL;AtValue=SECONDS;AtErr=;
```

If SEMANTICS is specified, then the function returns the Semantics for the attribute, with new lines indicated by <br/>.

Example:

```
AttInfo("UseTime","PD",SEMANTICS) returns  
ACCESSTIME returns the time of day when the file  
entry was last accessed.
```

If MNEMONICS is specified the the list of Mnemonics which are valid for this attribute is returned.

Example:

```
ATTINFO("KIND","PER",MNEMONICS)  
returns  
AtMnem=ODT,LAN,DISK,PACK,VSID,TAPE,PUNCH,TAPE7,TAPE9,READER,  
TAPEPE,PRINTER,UNKNOWN SCSI,HOSTCONTROL,HYPERCHANNEL,  
KANJIPRINTER,IMAGEPRINTER,CD,DC,VC,ARP;
```

## CLEAR

string function

```
—— CLEAR ————  
      |  
      | <string expression> |  
      |
```

By default the CLEAR function clears the local variable heap ,deleting all local variables. It returns an empty string.

If the optional string parameter is supplied then only string variables with that prefix will be cleared.

Note:

CLEAR("S") will clear all variables with a prefix of S but not \$S itself.  
Config,Global and Perm variables are untouched.

## .COLLECT

string method

Definition

## COMS

string function

Supervisor/Trim only

```
—— COMS —— ( <string expression> ) ————
```

The COMS function allows any Supervisor OPAL program to pass control or utility commands to the Unisys COMS Transaction server, returning any generated response to the caller. Previously, the only way to process responses to a COMS command using Supervisor, was to issue a <mixnumber>SM command to the COMS/INPUT process and start a MSG context WHEN to detect the display responses.

Some system setups require a usercode to execute the Coms function.

If none is provided a non critical security violation may be returned. This may be avoided by running the ODTS with a 'FOR <USER> clause.

If the first word in the <string expression> is STATS then the function will return statistical information for WINDOWS and PROGRAMS. For Direct Windows and REMOTE FILE Windows the information returned is for the associated Program. For MCS windows the information is for the Window.

Cumulative information is maintained for subsequent calls from the same slot.

COMS("STATS") return basic data, COMS("STATS FULL") more complete data. The data are returned in a comma separated list for each Program/Window with the Program/Window name as the First item. Each list is separated by a new line.

The items returned are as follows with an X indicating if the value is returned for STATS or STATS FULL.

Field	"STATS"	"STATS FULL"
Name	X	X
Type		X
Trans	X	X
TotTrans		X
QDepth	X	X
LastResp		X
RespTime	X	X
TotResp		X

The Fields above return the following:

Name	Program name for Direct or Remote windows, Window name for MCS windows.
Type	DIRECT,MCS or REMOTE
Trans	Number of transactions since last call in this slot, blank if first call
TotTrans	Number of transactions since Coms started.
Qdepth	Blank for non Direct windows. QDepth at time of call
LastResp	Blank for non Direct windows. Last response time in milli seconds. Blank if no transaction in this period.

Name	Program name for Direct or Remote windows, Window name for MCS windows.
RespTime	Blank for non Direct windows. Total response time in milli seconds since the last call in this slot. Blank if first call.
TotResp	Blank for non Direct windows. Total response time in milli seconds since Coms started.

We are working on example ODTsequences to collect Coms stats in a .CSV file which could be opened by EXCELL. If you are interested in monitoring COMS in this way please contact METALOGIC for an example.

Examples:

```
Coms ("STATUS PROGRAM HOTLINE")
Coms ("STATUS STATION META01")
Coms ("ENABLE WINDOW STATISTICS")
```

COMS returns a response very similar to that of KEYIN; each line of the response returned to the caller is separated by a carriage return.

As with KEYIN, the COMS function is actually handled by the Supervisor process GRINDER and any active WHEN waiting for GRINDER to complete a COMS request on its behalf will be seen in a WHEN? response as:

**WAITING COMS**

The COMS function invokes a well-documented entrypoint in the Unisys COMSSUPPORT library called PA\_REQUEST. The range of control and utility commands that are supported is documented in the Unisys Transaction Server for Clearpath MCP Programming guide, Section 10 Program Agent Facility;.

Please note that not all of these commands may work on earlier varieties of MCP levels 46.1 and 47.1.

On MCP 48.1 and later, the following example COMS commands are supported:

```
ATTACH, CLEAR, DATABASE, DISABLE, ENABLE, QUIT, READY,
STATUS, CREATE, DELETE, LOAD, INQUIRE
```



## SUPERVISOR examples:

```
TT DO (Show(Coms("STATUS PROGRAM MARC")))
Program status for MARC
Codefile:  MARC
Queue depth: 0, Response time: 0.06 secs, Messages processed:
35
Input Queue memory in use for this Program: 907 words
This program is currently ENABLED.
Associated window is MARC (ENABLED).
The following copies are active:
    Copy 2: Mix 300: idle
                Message from station  for dialog ,
                Task Queue Depth = 0
    Copy 1: Mix 299: processing for 1:51:51.5118643641
                Message from station IP101 for dialog MARC/1,
                Task Queue Depth = 0
```

```
TT DO (Show(Coms("DISABLE PROGRAM STATISTICS")))
The following were processed:
    Program STATISTICS Disabled
```

Because of the COMS functions' similarity to KEYIN, each line of output is delimited by carriage return and can be readily parsed using the SPLIT method. e.g.:

```
TT DEFINE + ODTs COMS_PARSER:
    $Res:= Coms("STATUS PROGRAM MARC");
    While $Line:= $Res.Split(/) NEQ Empty Do
        Show("LINE #", #L:=#L+1, $Line)
```

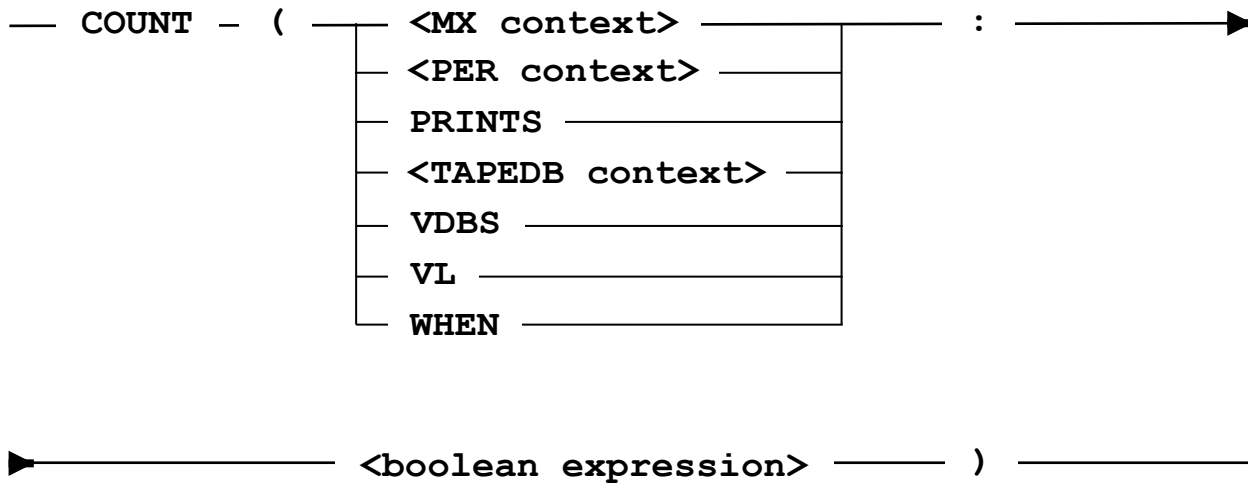
## .COPY

[string method](#)

[Definition](#)

# COUNT

arithmetic function  
Supervisor/Trim only



COUNT is available in the SUPERVISOR and the TRIM modules only and provides a way to count the number of occurrences of a SITUation. OPAL evaluates the **<boolean expression>** as a SITUation for each mix number, unit number, print request or tape serial number, as described by the **<context>** type, and maintains a tally of the number of times TRUE was returned.

The MX, PER and TAPEDB contexts may take optional sub-contexts, as in SUPERVISOR SITUation DEFINES, that allow the search to be filtered. The VL, VDBS and PRINTS contexts do not allow sub-context usage.

Some example expressions using COUNT:

```
Count(MX:Name EQW "=SUPERVISOR=")
Count(VL:VolumeName HdIs "FLEX")
Count(PRINTS:User="META")
Count(PER:Kind=PACK And Label HdIs "DEV")
```

A COUNT within the body of another COUNT is not allowed. If this were allowed, a multiplicative overhead would occur. Such recursion can always be avoided.

## COUNT and contexts

As with SITUations of context PER,MX and TAPEDB, the **COUNT** function may use sub-contexts to restrict the search to specific areas of that context.

Note that a TAPEDB **COUNT**, without a sub-context, will scan the entire METATAPELIB database, so using TAPEDB subcontexts if possible will be more efficient.

Some sub-context examples:

```
Count(MX=LIBS, WAITING: True)
Count(PER=MT-: AvailableToGroup)
Count(TAPEDB=BYNAME: SerialNo HdIs "Z")
```

COUNT using the PRINTS context causes SUPERVISOR to scan the current PRINTS queue (as seen by a PS SHOW ALL). Using the VL context causes the system Volume Library to be scanned, examining all volumed pack and tape entries (only applicable on cataloging systems). The VDBS context searches for active DMSII databases and can be used to examine database attributes normally only available via SM STATUS.

For a list of valid sub-contexts, the Supervisor command '**HELP CONTEXT**' from any Supervisor window will show this information about each context.

It should be noted that evaluation of a **COUNT** function is relatively expensive, TAPEDB in particular, and especially if an unqualified context is used.

## COUNT in WAIT statements

If a WAIT statement occurs within an ODTSequence of context PER or MX, and the WAIT contains a COUNT, SUPERVISOR will give a syntax error unless the body of WAIT is a Boolean expression of the form:

```
Wait( Count(.....) <relational op> <arithmetic expression> )  
Wait(Count(PER=MT:Scratch) GTR 0)
```

This restriction is imposed to simplify the analysis of just what the ODTSequence must be woken up to evaluate. The count will be re-evaluated every time there is a change to the context it is counting. In the PER=MT example above this would be for every PER:MT change.

The constructs allowing a program to wait for a program to enter a given mix state (see [WAIT](#) statement) usually provide clearer and more efficient ways of avoiding COUNT in WAIT statements. Where they do not, a good technique is to generate an EVALUATE command or a ONCE command within an ODTSequence.

## COUNT and PER considerations

The COUNT(PER=<unit type>.... syntax allows a minus sign after the PER sub-context such that all units on the system, belonging to that device type, are returned regardless of status.

For example:

```
Count (PER=PK:True)
```

will look at all on-line, visible packs.

```
Count (PER=PK- :True)
```

will examine all packs on the system even if they are not READY, FREED, etc.

Please note that if multiple sub-contexts are used (for example, PER=PK,LP-) then a minus sign following any of the device types in the list will cause the COUNT to act as if a minus had followed all of the device types.

The special case COUNT(PER-: has not been considered here since a normal COUNT(PER: already looks at all units, regardless of status.

## COUNT and TAPEDB context

A TAPEDB COUNT operation can be a very time-consuming process if the number of tapes in the METATAPELIB database is very large. If the TAPEDB context is used without a sub-context, a linear scan of the whole database will be performed.

To prevent Supervisor from being held up whilst the scan is taking place, a separate task is invoked to handle database access.

This process is called:

### METALOGIC/SUPERVISOR/EVREADER

The process will remain in the mix until the COUNT has terminated or has been manually aborted.

The EVREADER task is also invoked for similar OBJECTS calls and EVALS of LOG and TAPEDB OPAL scripts.

An OPAL program executing such a COUNT statement will appear in the response to a WHEN ? interrogation command.

As follows:

```
ev ? ex_tape=
----- SUPERVISOR WHEN STATUS (Limit = 40, Active = 27, Queued = 0) -----
W 233*46775          DO  EX_TAPESEARCH      (WAIT TAPEDB,COUNT)          23
                                TIMES:CPU=00:00:00,IO=00:00:00,ET=00:00:00
                                201 evals in COUNT
```

If the above ODTs were to be manually terminated, the COUNT operation would be aborted and the EVREADER would terminate normally.

## Count vs Objects

In older versions of OPAL Count was often used, with contrived expressions, to store information in parallel with the count.

Example:

```
DEFINE + ODTs COUNTER:
  If Count(MX:NAME="METALOGIC/JAMPAK" And
    ElapsedTime > 300 And
    #Mix.Store(MixNo)=Empty ) > 0 Then
  Begin
    Display("JAMPAK ACTIVE FOR MORE THAN 5 MINUTES");
    Odt(#Mix, "AX QUIT");
  END;
```

Note the use of #MIX.STORE(...) which stores the mix number of the JAMPAK task (if found) in the variable #MIX. This allows the use of the ODT statement later on to QUIT the active JAMPAK. We can take advantage of the conditional nature of the AND operator in OPAL to ensure that if the first test fails then none of the following expressions will be evaluated.

The OBJECTS function largely negates such a contrived use of the count function:

```
DEFINE + ODTs COUNTER:
  If $Mix:=Objects(MX:NAME="METALOGIC/JAMPAK" And
    ElapsedTime > 300) NEQ Empty Then
  Begin
    Display("JAMPAK ACTIVE FOR MORE THAN 5 MINUTES");
    Odt($Mix, "AX QUIT");
  END;
```

**See also:**

[OBJECTS function](#)

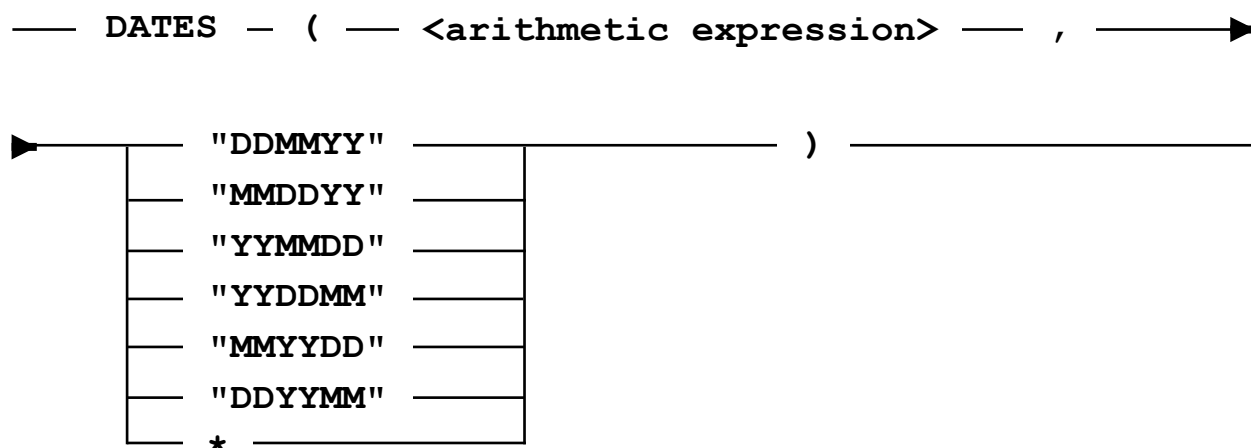
## **.CUT**

**string method**

[Definition](#)

# DATES

## string function



The DATES function converts Julian dates to strings in the Gregorian format. The function requires two parameters: the first must be a valid Julian date, the second is a quoted string that determines the format of the result. For example, if the second parameter has the value "DDMMYY", the result is an eight character string, with slash (/) as the delimiter, representing the English style Gregorian date corresponding to the Julian date in the first parameter.

Note: the [DATETOTEXT](#) function is now the preferred method for the conversion of Julian dates to Gregorian format.

### SUPERVISOR Example

```
DEFINE + DISPLAY CALCDATE:  
    Dates(NewDate(Today, 30), "DDMMYY")
```

Note the usage of NEWDATE to add 30 to today's Julian date to protect date additions that may traverse a year boundary.

### FLEX Example

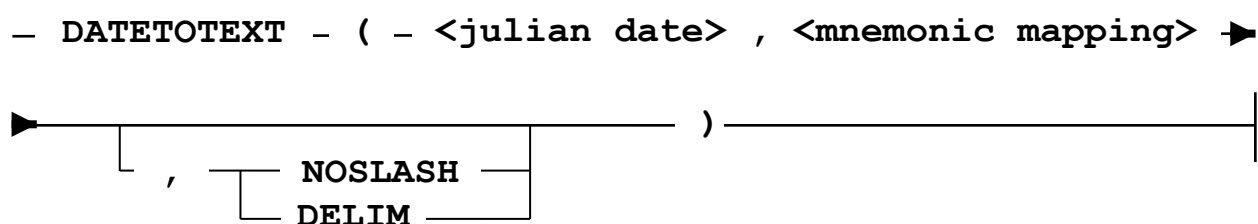
```
REPORT Dates(CreateDay, "YYDDMM")
```

### See also:

[Date Formats](#)

# DATETOTEXT

## string function



The DATETOTEXT function converts Julian dates to strings in the Gregorian format. It takes a minimum of two parameters: the first must be a valid Julian date, the second determines the format of the result.

For example, if the second parameter has the value DDMMYY, the result is an eight character string, with slash (/) characters as the delimiters, representing the English style Gregorian date corresponding to the Julian date in the first parameter. An optional third parameter may be specified to provide further output formatting options as detailed below.

The usage of DATETOTEXT in OPAL programs is much preferred over the use of the less flexible DATES function. The variety offered in <mnemonic mapping> allows a Julian date to be displayed in many formats.

The <mnemonic mapping> may be any one of those listed below while the <julian date> may be in a 5 or 7 digit format.

In the examples shown it is assumed that the date is 3 December 2002.

Mnemonic	Result	Example
DDMMYY	returns dd/mm/yy	03/12/02
DDMMYYYY	returns dd/mm/yyyy	03/12/2002
DDMONYY	returns ddmonyy	03DEC02
DDMONYYYY	returns ddmonyyyy	03DEC2002
DDMONTHYY	returns ddmonth/yy	03December02
DDMONTHYYYY	returns ddmonthyyyy	03December2002
MMDDYY	returns mm/dd/yy	12/03/02
MMDDYYYY	returns mm/dd/yyyy	12/03/2002
MONDDYY	returns mon/dd/yy	DEC/03/02
MONDDYYYY	returns mon/dd/yyyy	DEC/03/2002
MONTHDDYY	returns monthddyy	December0302
MONTHDDYYYY	returns monthddyyyy	December032002
YYDDD	returns yyddd	02337
YYYYDDD	returns yyyyddd	2002337
YYMMDD	returns yy/mm/dd	02/12/03
YYMONDD	returns yymondd	02DEC03
YYMONTHDD	returns yymonthdd	02December03
YYYYMMDD	returns yyyy/mm/dd	2002/12/03
YYYYMONDD	returns yyyymondd	2002DEC03
YYYYMONTHDD	returns yyyymonthdd	2002December03
DEFAULT	returns mm/dd/yyyy if USDATES is set else returns dd/mm/ yyyy	12/03/2002 or 03/12/2002

USDATES is a Supervisor option set using the SO command and a Flex option set using the Defaults command. The option determines the default form of dates displayed by Supervisor or Flex. (dd/mm/yy if the option is reset or mm/dd/yy if the option is set).

If the optional NOSLASH parameter is specified then the dates are returned without the slash (/) character. For example, on a site with the option USDATES reset:

```
DateToText(Today, Default, NoSlash)
```

might return

```
03122002
```

The optional **DELIM** parameter allows certain date formats to be returned with each elements separated by a space . This **only** applies to mnemonic mapping that have 'MON' or 'MONTH' in the name.

For example::

```
DateToText(Today, YYYYMONTHDD, Delim)
```

might return

```
2003 02 05
```

It is not possible to use the NOSLASH or DELIM modifier without a preceding <mnemonic mapping>. The following OPAL expression would generate a syntax error:

```
DateToText(TODAY, NoSlash) ;
```

If the Julian date passed as parameter does not contain the century, then years < 70 will be treated as being year 2000 relative.

For example:

```
69001 = 2069001 = 1st January 2069
```

```
70001 = 1970001 = 1st January 1970
```

**See also:**

[Date Formats](#)

[Date Arithmetic](#)

## DAYNAME

string function

— DAYNAME — ( — <arithmetic expression> — ) —————|

DAYNAME returns as a string the name of the day of the week indicated by the <arithmetic expression>. The <arithmetic expression> may either be a day number or a Julian date. If a



day number is specified then Day one is considered to be SUNDAY. If a negative or zero parameter is passed to DAYNAME, an error message is generated.

To maintain compatibility with earlier Opal code, where DAYNAME was designed only to interpret a day number, the parameter value is treated as follows:

A parameter in the range 1 to 70000 will be treated as a Day Number using the formula  $(D-1) \text{ MOD } 7 + 1$  where D is the parameter supplied.

A parameter in the range 70001 to 135365 will be treated as a '1900 relative' date.

A parameter in the range 1900001 to 2035365 is assumed to be a full Julian date.

All other values will result in a run-time error.

Assuming today's date to be 3 December 2007, DAYNAME could be invoked using either a parameter value of 107337 or 2007337 and would return 'MONDAY'.

#### SUPERVISOR Example

```
TT DEF + DISPLAY DATES:
    "Day = ", DayName(DayInWeek) , ,
    "Month = ", MonthName(MonthInYear)
```

#### FLEX Example

```
REP HEAD "REPORT FOR ", DayName(DayInWeek)
```

See also:

[MONTHNAME](#)

[Date Formats](#)

[Date Arithmetic](#)

## DAYS

arithmetic function

– DAYS — ( — <arithmetic expression> — , —————>

————— <arithmetic expression> ————— ) — |

DAYS gives the number of days in between two Julian dates, with due regard to any intervening leap years. The two parameters must be valid Julian dates.

The DAYS function can be used to determine which of two dates passed is the greater. If the – (negative) sign precedes one of the parameters then the difference

is calculated by subtracting that date from the other. Normally the smallest date is subtracted from the largest.

### Examples

```
Days (108001, -2008002) = -1
Days (-108001, 2008002) = 1
Days (108001, 2008002) = 1
```

### SUPERVISOR Example

```
TT DEFINE + DISPLAY DAY(MESSAGE) :
    #DT.Store(Decimal(Trim(Text))),
    #DIFF.Store(Days(#DT, -TODAY)),
    If #DIFF GEQ 0 Then
        DayName( ((DayInWeek+ (#DIFF MOD 7)-1) MOD 7)+1)
    Else
        DayName( ((7+DayInWeek- (ABS(#DIFF) MOD 7)-1) MOD 7)+1)
```

The above is a useful utility that will display the DAYNAME of any Julian date, past or present, when compared with TODAY. This shows the complexity required before the DayName function was enhanced to accept Julian dates.

A simplified version would be:

```
TT DEFINE + DISPLAY DAY(MESSAGE) :
    DayName(Decimal(Trim(Text)))
```

### See also:

[Date Formats](#)

[Date Arithmetic](#)

## DBS

string function

Supervisor/Trim only

```
- DBS ( <arithmetic expression> , <string expression> )
```

The DBS function allows OPAL programs to retrieve database information via the DMSII visible DBMS interface (normally via SM commands to an active database).

DBS requires two parameters: the first must be the mix number of a valid DMSII database and the second should be a valid VDBMS database command.

For example:

```
$Rslt:=DBS(12345,"STATUS")
DBS(MixNo,"ALLOWEDCORE=40000")
$Rslt:=DBS(#MyMixNo,"STATUS STRUCTURE *")
```

Each of the above commands will return any generated response back to the caller for subsequent analysis.

Like the COMS function, DBS will return any response with each line terminated by a '/', allowing each line to be easily parsed in an OPAL routine.

Because the DBS function causes Supervisor's GRINDER task to temporarily link to the database's DMSUPPORT library, there is some overhead associated with its usage though this is kept to a minimum.

The DBS function can return very large responses up to the maximum string length of 1,999,999 characters permitted by OPAL.

SUPERVISOR Examples:

```
TT DO (Show(DBS(12345,"STATUS"))
ACCESSROUTINES 48.189.8185
PRINT STATISTICS = ON
AUDIT PROCESSOR TIMES OFF
POPULATIONINCR IS AVAILABLE FOR THIS DB
POPULATIONWARN IS NOT AVAILABLE FOR THIS DB
AUDIT FILE# = 1,BLOCK SERIAL# = 2174 (33534/149999=22%)
AUDIT BLOCKSIZE = 900, AREASIZE = 1500, AREAS = 100
CURRENT AUDIT BUFFERS = 11, CF AUDIT BUFFERS = AUTOMATIC
CURRENT AUDIT SECTIONS = 1, CF AUDIT SECTIONS = DEFAULT
SYNCPOINT = 1, CONTROLPOINT = 1
CONTROL POINT AGING AFTER AUDIT SWITCHES = FORCE
CORE TOTAL:ALLOWED=5000,IN USE=4746,OLAYRATE=          0%
RESIDENT TOTAL:ALLOWED=5000,IN USE=0
OVERLAYGOAL =          2 % ALLOWEDCORE / MINUTE
SYNC WAIT IS 0 SECONDS
FORCED OVERLAYS = 282
OPEN COUNTS: INQUIRY =2, UPDATE = 1
PRIMARY AUDIT COPY SCRATCHPOOL NAME = NOT SET
THE DO WILL BE DONE
```

For example, to get the ALLOWEDCORE value for a database using the STATUS command:

```
TT DEFINE + ODTSEQUENCE GETAC(MX) :
  $Reply:=DBS(MIXNO,"STATUS") ;
  $Core:= Decat(Decat($Reply,"ALLOWED=",1) , " , 4) ;
  Show(MixNo , ,Name , , "CORE=" , $Core)
TT DO GETAC 12345
12345 (TAPELIB)METATAPELIB4 CORE=5000
```

The VDBS method simplifies this to

```
TT DEFINE + ODTSEQUENCE GETAC (MX) :  
    #Mix:=MixNo;  
    Show (MixNo, , Name, , "CORE=", #Mix.VDBS (AllowedCore) )  
TT DO GETAC 12345  
12345 (TAPELIB)METATAPELIB4 CORE=5000
```

If a DBS call has not responded within 5 seconds it is assumed to be blocked and will return an error. This 5 second time limit may be altered by setting a config variable SUP\_DBSTIMEOUT. This may be set using an inline Opal. Ex. / (\$SUP\_DBSTIMEOUT.CONFIG:="10")

## DECAT

### string function

– DECAT — ( — <String expression> — , —————>

▶— <string expression> ————— <selector> ————— ) —|

<selector> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Programming OPAL often involves complex string manipulations. DECAT helps to simplify such expressions, which could be very difficult if restricted to those available in WFL (HEAD, TAIL, TAKE and DROP).

DECAT requires three parameters — the source string (S), the target string (T) and a numeric constant (N), which must be between 0 and 7 i.e. DECAT(S,T,N). An important advantage of DECAT is that the target string (T) may be of any length.

The effect of DECAT is to divide S into three virtual strings, which are used to build the new string — the substring prior to T, T itself (null if not present in S), and the substring following T. The exact composition of the new string depends on N as follows:

N	Binary String	Result
0	000'	empty string (degenerate case)
1	001'	following substring
2	'010'	target string
3	'011'	target & following substring
4	'100'	prior substring
5	'101'	prior & following substring
6	'110'	prior substring & target string
7	'111'	source string (degenerate case)

Note that the binary string reflects the resultant string.

## Examples

```
DECAT ("AAAXXBBB" , "XX" , 0) = ""
DECAT ("AAAXXBBB" , "XX" , 1) = "BBB"
DECAT ("AAAXXBBB" , "XX" , 2) = "XX"
DECAT ("AAAXXBBB" , "XX" , 3) = "XXBBB"
DECAT ("AAAXXBBB" , "XX" , 4) = "AAA"
DECAT ("AAAXXBBB" , "XX" , 5) = "AAABBB"
DECAT ("AAAXXBBB" , "XX" , 6) = "AAAXX"
DECAT ("AAAXXBBB" , "XX" , 7) = "AAAXXBBB"
DECAT ("AAAXXBBB" , "YY" , 1) = ""
DECAT ("AAAXXBBB" , "YY" , 2) = ""
DECAT ("AAAXXBBB" , "YY" , 4) = "AAAXXBBB"
```

### SUPERVISOR Example of nested DECAT

If the MESSAGE attribute TEXT has the following value:

```
"DISPLAY: MISSING FILE SALARYFILE REQUIRED"
```

```
DEFINE + DISPLAY SHOWFILE (MSG) :
    $F:=Decat (Decat (TEXT, "MISSING FILE ",1) , " REQUIRED", 4)
```

When invoked, DISPLAY SHOWFILE would output the string "SALARYFILE" as well as storing it into the string variable \$F.

## DECIMAL

### arithmetic function

— **DECIMAL** — ( — <string expression> — ) ————|

As in WFL and ALGOL, **DECIMAL** converts a number in a string to its corresponding arithmetic value.

### SUPERVISOR Example

```
DEFINE + DISPLAY LONG_WINDED (MSG) :
    Number (Decimal (Text))
```

If the following were entered to invoke the display:

```
TT / LONG_WINDED 1234
"ONE THOUSAND TWO HUNDRED AND THIRTY FOUR"
```

## .DELTA

### arithmetic method

#### Definition

# .DISTRIBUTE

## string method

### Definition

## DROP

### string function

– DROP – ( — <string expression> — , —————>  
————— <arithmetic expression> — ) ———|

The DROP function works as in WFL, except no run time errors occur if the value of <arithmetic expression> is greater than the length of the string.

If the arithmetic expression returns a negative value then the characters are dropped from the end of the string instead of the beginning. If the absolute value of the arithmetic expression is greater than the length of the string then the empty string is returned.

Example

```
Drop("ABCDEF",2) returns "CDEF"  
Drop("ABCDEF",-2) returns "ABCD"
```

SUPERVISOR Example

```
DEFINE + DISPLAY DROPIT(MSG) :  
  If Text HdIs "DISPLAY:" Then  
    Drop(TEXT, 8)      % get rid of "DISPLAY:"  
  Else  
    Text
```

FLEX Example

```
REPORT IF Title HdIs "*DOWNLOAD/" Then  
  Drop(Title, 10) % drop unwanted prefix  
Else  
  Title
```

## DROFILEIDS

### string function

– DROFILEIDS – ( – <string expression> — , —————>  
————— <arithmetic expression> — ) ———|

The DROPFILEIDS function is similar to the DROP function but treats the string expression as a file title and returns the remainder of the title after dropping the requested number of levels ,as specified by the arithmetic expression, from the start.

If the string does not contain a valid title then an empty string is returned.

If the number of levels requested is zero then the usercode and on part are returned..

If a negative number of levels is requested then the levels are dropped from the end not the start.

If the number of levels requested is greater than the number in the title then only the Usercode and ON part, if present will be returned.

If the file title is usercoded or prefixed with \* then the usercode or \* will be returned with the requested levels.

If the file title has an ON part then it will be returned with the requested levels.

Examples

```
$S:="A/B/C"
DropFileIds($s,2) returns C
DropFileIds($S,-2) returns A
DropFileIds($S,10) returns the empty string
$X:=" (BOB)A/B/C ON DEV"
DropFileIds($X,1) returns (BOB)B/C ON DEV
DropFileIds($X,-1) returns (BOB)A/B ON DEV
DropFileIds($X,0) returns (BOB) ON DEV
DropFileIds($X,5) returns the empty string
$Bad:="a/b/c"
DropFileIds($bad,1) returns the empty string
$Good:="'a"/B/C'
DropFileIds($Good,2) returns C
```

## .FILE/.READ

string method

[Definition](#)

## FILEID

string function

- FILEID - ( — <string expression> — , —————>

————— <arithmetic expression> — ) ————|

FILEID returns a level of a file title. The <string expression> in the first parameter must be a valid file title.

The <arithmetic expression> selects which particular level is required, its value must range between 0 and 14. Zero corresponds to the usercode (or "\*" for a system file). Higher values give the subsequent identifiers in the title. If the integer value exceeds the number of names in the file title, the result returned by FILEID will be a null string.

For example:

If the title is \*SYSTEM/PL/I

```
FileID(TITLE,0) has the value "*"
FileID(TITLE,1) has the value "SYSTEM"
FileID(TITLE,2) has the value "PL"
FileID(TITLE,3) has the value "I"
FileID(TITLE,4) has the value "" (null string)
```

SUPERVISOR Example

```
DEFINE + ODTS FILE_CHK(MX) :
  If FileID(Name,1) = "SYSTEM" Then
    $Type:="TASK IS SYSTEM"
  Else
    $Type:="NORMAL TASK";
```

FLEX Example

```
SELECT FileID(NAME,1) = "SYSTEM"
```

## FILEIDS

arithmetic function

— FILEIDS — ( — <string expression> — ) —————|

FILEIDS returns is closely related to FILEID (and often used in conjunction with it). It returns the number of levels in the title. The parameter must be a valid title. The usercode or family name are not considered as levels, thus

(MYUSER)X/Y ON MYPACK and \*X/Y

both have FILEIDS = 2.

SUPERVISOR Example

```
DEFINE + DISPLAY GETVERSION:
  "VERSION = ", FileID(WM, FileIDs(WM))
```

This DISPlay shows the last level of the running MCP.



```
"*SYSTEM/ASD/AMLIP/MCP/40088"
```

## FLEX Example

This would select files whose name began with "BD" and ended with "TASKFILE".

## string function

HEAD is identical to the same function in WFL, except no run time errors occur if the <string expression> has zero length.

```

TT DEFINE + DISPLAY EX_WM:
    "MCP: ", MCP, " ",
    MCPMark 1, MCPlevel 1, ".", MCPCycle 3, ".", /,
    "COMPILED: ", Drop(Tail(MCPTimeStamp, Not ","),1), " @ ",
    Head(MCPTimeStamp, Not ",")

```

## arithmetic function

The OPAL and WFL versions of HEX are identical. Both convert a hexadecimal string to an arithmetic value. For example:

## SUPERVISOR Example

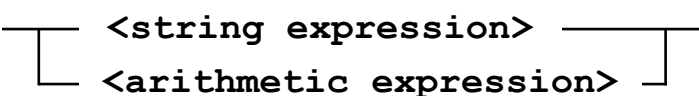
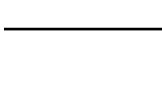
```
TT DEFINE + DISPLAY TEST_HEX:
  If Hex("00004000251") =
    ( 1 +
      5 * 16 +
      2 * 16*16 +
      4 * 16*16*16*16*16*16*16) Then
    "HEX EXPLAINED"
  Else
    "HEX IN TROUBLE"
```

See also:

[OCTAL](#)

## HEXSTRING

string function

- HEXSTRING - (  ) 

The HEXSTRING function allows conversion of a real or string expression valued into the hexadecimal string representation of that value.

For example:

```
HexString(256) will return the string "100"
HexString(99.99) will return the string "2518FF5C28F6"
HexString("A") will return the string "C1"
```

SUPERVISOR example

```
TT DEFINE + DISPLAY TEST_HEXSTRING:
  If HexString(99.99) = "2518FF5C28F6" Then
    "HexString EXPLAINED"
  Else
    "HexString IN TROUBLE"
```

## .INCLUDES

boolean method

[Definition](#)

# INDEXOF

## arithmetic function

— INDEXOF — ( — <boolean expression> — ) —————|

The INDEXOF function allows the OPAL programmer the capability to determine which entity of a set of tests evaluated to TRUE.

While it is intended to handle a complex <boolean expression> such as:

```
$User Incl { "META", "OPS", "LIVE", "PRIV", ... }  
$S Incl { "INVALID INDEX", "DIVIDE BY ZERO", "SEG ARRAY" ..... }
```

It may also be used to evaluate simple, multiple expressions such as

```
$S = "A" OR $S = "B" OR $S = "C" OR ...
```

Semantically, the <boolean expression> shown in the examples above can be regarded as a simple set membership expression or a list of multiple-OR expressions. Where no <boolean expression> returns TRUE, INDEXOF will return -1 (minus one).

Where there is only one possible <boolean expression> – for example TRUE, A=B, X NEQ Y, and so on – then if the <boolean expression> returns TRUE, INDEXOF will return a value of 0 (zero).

Where there are "multiple" elements within <boolean expression>, INDEXOF returns the index of the **first** element to evaluate to TRUE.

See the following example:

```
IndexOf( Text Incl { "ADAM", "BOB", "COLIN", "DAVE", "EDDIE" } )
```

If TEXT contains the string "ADAM" the above statement will return 0 (zero). If TEXT contains the string "EDDIE" then it will return 4. If TEXT = "ZEBEDEE" then INDEXOF will return -1 (minus one).

Similarly, the statement:

```
IndexOf( Text = "MONDAY" OR $Day = "FRIDAY" )
```

will return 0 if TEXT holds the string "MONDAY", 1 if the variable "DAY" holds the string "FRIDAY" and -1 otherwise.

Note that if Both Text="MONDAY" and \$Day="FRIDAY" then 0 is returned (the first true condition found)

In the following example, INDEXOF combined with CASE can be very effective. The value returned by INDEXOF is used directly in the <arithmetic expression> required by CASE.

## SUPERVISOR example

```
TT DEFINE + ODTSEQUENCE INDEXOF_MSG: (MESSAGE)
  Case IndexOf(Text HdIs {"REASON ",           % 0
                        "REPORTING ",           % 1
                        "CALL ID",             % 2
                        "NO PARAMETER",         % 3
                        "ADDITIONAL PARAMETERS", % 4
                        "SEVERITY",             % 5
                        "EVENT ",               % 6
                        "COMPONENT", }) Of      % 7

  Begin
    0: Show("PRIORITY 1: ",Text);
    1: Show("LOGGING   : ",Text);
    2: Show("CALL INFO : ",Text);
    4: Show("PARAMS    : ",Text);
    3: 5: 6: 7:
      Show("PRIORITY 2: ",Text);
  Else:
    Show("INFO        : ",Text);      % SHOULDN'T GET HERE
  End;
```

## .INSERT

### empty string method

#### Definition

## INPUT

### string function

### Supervisor/Trim only

— INPUT —————|

The **INPUT** function allows external text to be passed to a waiting ODTSequence from a Supervisor COMS window. The function returns a string consisting of the next user input from that window.

Example:

```
DEFINE + ODTSEQUENCE INPUT_EX:
  $MyText:=Input;
  Show("User response was ", $MyText);
TT DO INPUT_EX
```

The above DO will cause the ODTS to wait for input. The response to an EV ? command reports '(WAIT Input)' for any ODTS which is waiting for screen input.

Response:

```
-- SUPERVISOR WHEN STATUS (Limit = 40, Active = 25, Queued = 0) --  
W 251 23896                DO MYODTS                (WAIT Input)
```

Any subsequent input from this Supervisor window (#251) will be directed to the waiting ODTSequence directly into the variable \$MYTEXT.

This new function is intended for use in interactive ODTSequences only; it is not permitted by Flex, DISPlays or SITUations.

If an ODTSequence executes an **INPUT** function when it is not running from a Supervisor window, the OPAL program will be aborted. Entering ?Close on a supervisor window will not cause an ODTs, waiting on Input via the INPUT function, to fault. The ODTs will terminate normally.

When a Supervisor window is closed using the WINDOW - command a simple "Window closed" message is shown

A new Supervisor examples file, called OPALS/MENU, illustrates the use of this function (this file is unwrapped during the INSTALL process) to control various operations themes. Within this file, please refer to the ODTSequence MENU\_CFGHLP for additional documentation.

Preceding input with a ? character will cause any text not recognised by COMS as a command to be passed to Supervisor as normal.

Preceding input with ?? will cause the text to be sent directly to Supervisor, with the exception of ??END, ??WRU and ??CLOSE.

??END and ??CLOSE will kill the waiting ODTs and free the screen for normal use.

## INTEGER

### arithmetic function

— **INTEGER** — ( — **<arithmetic expression>** — ) —————|

OPAL, ALGOL, and WFL all share the same definition for the **INTEGER** function. The **INTEGER** function returns a result equal to the **<arithmetic expression>** but without any fractional part.

SUPERVISOR Example

```
DEFINE + DISPLAY DISK_AVAIL:  
    "DISK HAS ", Integer(DU("DISK")/TotalSectors("DISK")*100) ,  
    " AVAILABLE SPACE"
```

# JULIAN

## arithmetic function

— JULIAN — ( — <string expression> — ) —————|

JULIAN takes a string parameter and converts to an integer Julian date, if possible.

For example, if the SUPERVISOR/FLEX option USDATES is FALSE:

```
Julian("12/7/94")          will return      1994193
```

and if USDATES is TRUE then the same expression will return

```
1994341
```

If the string passed to Julian is NOT a valid military date then a negative value is passed back to the caller , identifying the error as follows :

```
-1    = non-numeric format
-2    = bad day and/or month combination
-3    = short date format
-4    = slash expected
-5    = bad year
```

SUPERVISOR Example

```
DEFINE + DISPLAY JULIAN(MESSAGE):
    #Jul:=Julian(Text) , ,
    DatetoText(#Jul, DDMMYYYY)
```

**See also:**

[Date Formats](#)

[Date Arithmetic](#)

# KEYIN

## string function

— KEYIN — ( — <string expression> — ) —————|

The KEYIN function, available only from SUPERVISOR, allows the capture of ODT command responses from an OPAL program. In particular, this means that system information that SUPERVISOR may be unable to extract (e.g. BNA network information) can be easily obtained. KEYIN requires a single string parameter and can be used in any OPAL program type, regardless of its context, and includes SITUations and DISPLAYs.

For example, to SHOW the output for any ODT command:

```
TT DEFINE + ODT$ KEYIN(MSG) :  
  If $Response:=KeyIn(Text) NEQ Empty Then  
    Show( $Response )  
  Else  
    Show("No response");
```

Running this:

```
TT DO KEYIN PER PK  
----- PK STATUS -----  
44*B\      [380049] #1 PACK (36)  
45*B\      [380380] #1 DISK (131)  
46*B\      [381046] #1 WORK (10)  
47*C       [380047:380048:48] #2 DBSPACK (31)  
48*B\      [380048] #1 DBSPACK (44)
```

When KEYIN is executed, the command given by the <string expression> is passed to SUPERVISOR's GRINDER process and handled by the DCALGOL DCKEYIN function. During this process, the calling OPAL program is suspended until the system has returned a response.

Note that SUPERVISOR itself is not halted during this time and will continue to process events and commands normally. KEYIN returns the response of the command as a string result.

Any OPAL programs waiting on a KEYIN response can be seen in the output from a TT WHEN ? command where the program status is marked as "(WAIT KEYIN)". This is usually a transitory state since the OPAL program is immediately reactivated as soon as a response is received.

If KEYIN is used within an OPAL program linked to an event-driven WHEN, such as the MX, LOG or PER contexts, events received by Supervisor whilst the WHEN is awaiting a KEYIN response will be queued and then handled normally as soon as the WHEN is reactivated. This behaviour is unlike the WAIT statement that suspends event detection whilst the ODTSequence is suspended.

Normally, the DCALGOL DCKEYIN function returns a multi-line response with each line split by a null (48"00") character and the message terminated by a full stop and ETX (48"03") character. However, before the response is returned to the calling OPAL, the GRINDER process replaces all null characters with carriage returns (48"0D") plus the trailing "." and ETX characters are removed. This allows KEYIN responses to be easily displayed on the calling station.

The maximum response returned by KEYIN is controlled by the MCP limit of 255 lines or the OPAL maximum string limit of 1,999,999 characters. Any graphic highlighting characters, such as bright, reverse or underline, are individually replaced by space characters before being returned to the caller.

## SUPERVISOR Example

If it is required to extract a specific line from a KEYIN response, it is necessary to parse the string using carriage return as a target.

The following OPAL code is an example of how to achieve this:

```
TT DEFINE + ODS NW_KEYIN:
    $NW:= KeyIn("NW_HOSTS");
    While $Line:=$NW.SPLIT(/) Neq Empty DO
        Show("LINE: ", $Line);
```

## LENGTH

arithmetic function

— **LENGTH** — ( — **<string expression>** — ) —————|

OPAL, ALGOL, and WFL all share the same definition for the LENGTH function. The LENGTH function returns the number of characters contained in the value of the string expression.

For example:

```
Length("SUPERVISOR")           returns 10
```

### SUPERVISOR Example

```
DEFINE + DISPLAY PER(PER):
    If Myself(Entries)=1 THEN
        #(" REPORT FOR ", HostName, /,
          " ===== === ", Repeat("=", Length(HostName))
    Else
        Empty,
    UnitNo 4, Label
```

### FLEX Example

```
REPORT Title 60, If Length(Title) > 60 Then "...." Else ""
```

## LOWER

string function

— **LOWER** — ( — **<string expression>** — ) —————|

The LOWER function converts all ALPHA characters in the given <string expression> parameter into lower case.



## Example

`Lower("AbCdE")` will translate to `"abcde"`

See also:

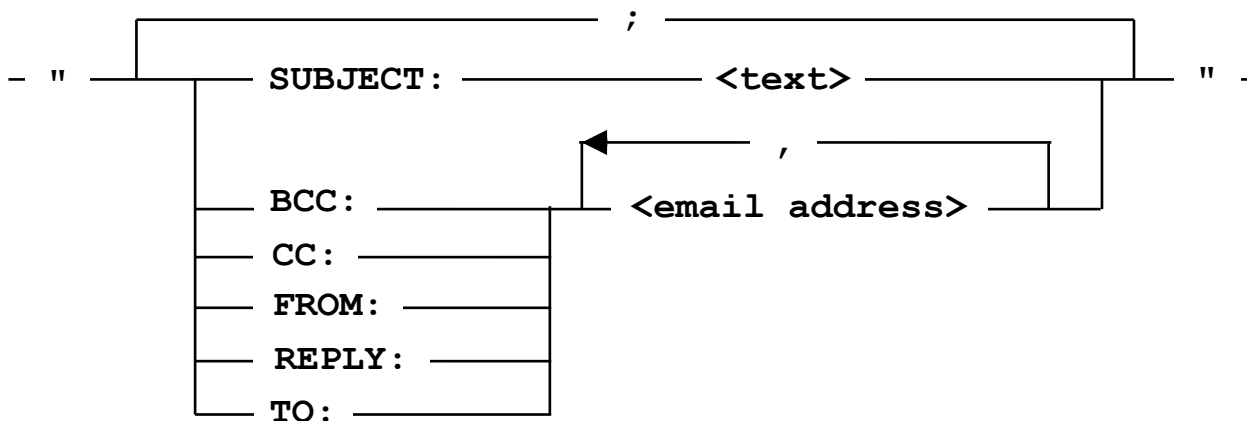
[UPPER](#)

## MAIL

arithmetic function

- **MAIL** - ( - <header string> - , - <string expression> - ) - }

<header string>



The MAIL function allows the generation and sending of email messages from Supervisor via an accessible SMTP mail server visible to the local network. The Metalogic MAIL implementation does NOT require any additional Unisys components to be present except for Unisys or third-party TCP/IP.

The implications of using MAIL from Supervisor are significant because of the abilities of external PC/Internet software to convert emails into electronic pager requests or calls to mobile phones. Many companies are providing such software and Supervisor is now capable of sending user-formatted emails, to one or more recipients, when triggered by waiting entries, abnormal job terminations, critical system messages etc.

## MAIL semantics

The MAIL function requires two string parameters:

The <Header String>expression allows the specification of multiple email headers allowing a 'Subject:' and 'To:', 'Bcc:', 'Cc:', 'From:' or 'Reply:' addresses to be specified. For 'To:', 'Bcc:' and 'Cc:', multiple email addresses can be specified using comma as a delimiter. Each individual header requires a colon ':' before each address list and each header must be separated from the next by a semi-colon.

For example:

```
"To: ian@metalog.com,bob@metalog.com; Subject: Test Message;" &
"From:alert@metalog.com;"
```

Note that Supervisor will expect email addresses to have valid format i.e. a domain must be present signified by the '@' character. A "To:" header is mandatory and **must** be present; all other headers, including "Subject:" are optional.

The second parameter is a simple OPAL string expression that formats the text or the body of the message. A '/' (new line) character may be used to denote a new line, control paragraphing, etc.

For example:

```
#("This is line 1",/,"And line 2",/,,,"... Line 4")
```

Note the use of the # (..) hash-paren construct, which makes the construction of an OPAL string expression much simpler.

MAIL will return an integer value; a positive value indicates the number of messages that were generated and successfully sent. A negative value denotes an error in the message format, no recipients etc. An OPAL attribute, MAILERROR, can return a string that provides explanatory text about the reason for the error.

Upon receipt of a MAIL command, SUPERVISOR will invoke a sub-task called MAILHANDLER, if it is not running, and handles library linkage to the Metalogic MAILLIB library. MAILLIB is responsible for handling all email generation activities. MAILLIB is highly configurable and the site can customise many aspects of an email configuration.

MAIL will permit a total message size (including text and headers) of 1,999,999 characters; the number of text 'lines' permitted in the message body cannot exceed 3000. Individual lines of text will be managed in 1000-character segments if no CR or LF characters exist in an individual line of text.

For more information about the Metalogic MAILLIB library and its configuration, please refer to the **Metalogic Mail Library reference Manual**.

An ODTSequence could be used to send an email when a critical waiting entry has been detected.

SUPERVISOR example

```
TT DEFINE + ODTSEQUENCE WAITING(MX) :
    #MRes:=Mail ("To: ian@metalog.com, bob@metalog.com;" &
        "Subject: Critical waiting entry",
        #("Waiting entry detected at ",Time(TimeOfDay),/,
        "Mix : ", MixNumber,/,
        "Task: ", Name 60,/,
        "RSVP: ", RSVP));
    IF #MRes <0 AND #MRes NEQ -999 Then
        Display("MAIL send error = ",MailError(#MRES));
```

# MAX

## arithmetic function

— **MAX** — ( — <arithmetic expression> — , —————> —  
—————> <arithmetic expression> — ) —————|

MAX returns the value of the larger parameter. It differs from its ALGOL namesake in allowing only two parameters.

SUPERVISOR Example

```
DEFINE + ODTSEQUENCE WAITABIT(MSG) :  
    #Time:= Max(60, Decimal(Text)); % at least 60 secs  
    Wait(#Time);  
TT DO WAITABIT 30          % would still wait 60 secs
```

# MEMBER

## arithmetic function

— **MEMBER** — ( — <member target> —————> —  
—————> , — <member source> ————— ) —  
—————> , — ELSE ————— ) —

<member target>

—————> <arithmetic expression> —————|  
| <string expression> —————|  
| <arithmetic primary> —————|

If <member target> is an <arithmetic expression>

<member source>

—————> <arithmetic expression> —————|  
| <arithmetic expression> —————|  
| { <arithmetic expression> ————— } —————|

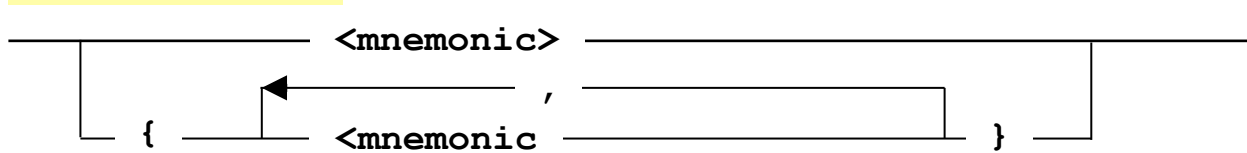
If <member target> is an <string expression>

<member source>

—————> <string expression> —————|  
| <string expression> —————|  
| { <string expression> ————— } —————|

If <member target> is an <attribute primary>

&lt;member source&gt;



MEMBER provides a method to distinguish between alternate possibilities and to return an index indicating which, if any, of the possibilities are true. It is useful in turning a complex range of possibilities into the <arithmetic expression> required for CASE expressions. MEMBER is evaluated as follows:

The <member target> is compared with each item in the <member source> in turn until the <member source> is equal to the <member target>, or the <member source> list is exhausted.

If a match is found, an index corresponding to the particular expression is returned. The component expressions of the <member source> list are numbered sequentially from 0 to N-1 where N is the number of expressions in the list.

If the list is exhausted, the value returned by MEMBER will be N if an ELSE is present, otherwise a (−1).

## SUPERVISOR Examples

### DEFINE + DISPLAY DATE:

DayInMonth,

**Case Member (DayInMonth MOD 10, 1, 2, 3, Else) Of**

```
("st", "nd", "rd", "th"),
```

" of ", MonthName(Month)

**DEFINE + DISPLAY MONTHNAMETONUM (MSG) :**

```
If #I:=member(Lower(trim(text)),"january","february","march",
```

```
"april", "may" , "june", "july", "august",
```

"september", "october", "november",

"december") +1 =0 Then

```
"Bad monthbname: " &Text
```

Else

```
Text&" is month "&String(#I,*)
```

In Flex `MEMBER` could be used to classify all values of the `FILEKIND` attribute into 6 classes, returning an integer number between 0 and 5.

## FLEX Example

```
Member(PDFFileKind("TESTFILE ON TESTPACK"),
{LIBRARYCODE,JOVIALCODE,JOBCODE,DMALGOLCODE,SANSCODE,
 38,49,51,52,55,56,57,58,59,60,61},
  % CASE 0: FILEKINDS RESERVED FOR UNISYS PLANTS
{NDLIICODE,NDLCODE,DCPCODE},
  % CASE 1: DATACOM ONLY FILEKINDS
{MCPCODEFILE,FUNCTIONFILE,CODEFILE},
  % CASE 2: SOFTWARE MAINTENANCE STUFF ONLY
COMPILERCODEFILE,
  % CASE 3: COMPILERS
{NEWPSYMBOL,NEWPCODE,ESPOLSYMBOL,ESPOLCODE},
  % CASE 4: NEWP AND ESPOL CODE,SYMBOLICS
ELSE) % CASE 5: ALL OTHERS
```

## MIN

## arithmetic function

— MIN — ( — <arithmetic expression> — , —————▶  
▶————— <arithmetic expression> — ) —————|

MIN returns the value of the smaller parameter. It differs from its ALGOL namesake in allowing only two parameters.

## SUPERVISOR Example

```
DEFINE + ODTSEQUENCE WAITABIT(MSG):
    #Time:=Min(60, Max(20, DECIMAL(TEXT)));
    Wait(#Time); % will be min 20 secs, max 60
TT DO WAITABIT 2
```

will force a wait of at least 20 seconds

TT DO WAITABIT 120

will force a wait of at most 60 seconds

## MONTHNAME

## string function

— MONTHNAME — ( — <arithmetic expression> — ) —

MONTHNAME returns as a string the name of the month indicated by the <arithmetic expression> parameter. The result of <arithmetic expression> will be adjusted to fall in the range 1 to 12, by means of the formula

```
(M-1) MOD 12 + 1
```

where M is the value of the <arithmetic expression>. If a negative or zero parameter is passed to MONTHNAME, the following error message is given:

```
BAD ARGUMENT PASSED TO DAYNAME/MONTHNAME IN <OPAL name>
```

SUPERVISOR Example

```
DEFINE + ODTSEQUENCE MONTH_CHK:
  If MonthName(Month) NEQ "DECEMBER" Then
    Odt("STARTJOB JOB/NORMAL")
  Else
    Odt("STARTJOB JOB/ENDOFYEAR")
```

**See also:**

[DAYNAME](#)

[Date Formats](#)

## NABS

arithmetic function

— NABS — ( — <arithmetic expression> — ) —

As in ALGOL, NABS returns the value of <arithmetic expression> as a negative number.

SUPERVISOR Example

```
DEFINE + DISPLAY CALC_DATE(MSG) :
  "SUBTRACTED DATE: ",
  #D:= NewDate(Today,Nabs(Decimal(Text))),
  " : ",
  DateToText(#D,YMMMDD) ;
```

Always use NEWDATE if you are adding or subtracting days from a Julian date.

**See also:**

[ABS](#)

# NEWDATE

## arithmetic function

— NEWDATE — ( — <arithmetic expression> — , —————> —————<arithmetic expression> — ) —————|

NEWDATE returns a new Julian date a specified number of days before or after a given Julian date with due regard to leap years. The first parameter must be a Julian date, the second an integer whose sign determines the 'direction' of computation (negative computes a new date before the given date, positive a date after the given date).

Regarding Year 2000, if a four-digit year is passed as a parameter then a four digit year will always be returned, otherwise a 3 or 2 digit year will be returned.

See [Date Formats](#) for more details.

### Examples

```
NewDate(99365,1)           returns 100001
NewDate(1999365,1)        returns 2000001
NewDate(100001,-1)        returns 99365
```

### SUPERVISOR Example

```
DEFINE + DISPLAY NEW_DATE:
  "Tommorow is ",
  If Holiday(NewDate(TodaysJulianDate,1)) Then ""
  Else "NOT",
  " a holiday  (" ,
  DateToText(NewDate(TodaysJulianDate,1),MMDDYY),") "
```

A typical response from this DISPlay (assuming USDATES reset) might be:

```
Tomorrow is NOT a holiday (02/04/94)
```

### FLEX Example

```
SELECT NewDate(Today, -40) > CreateDay
```

If the Julian date provided in the first parameter is a 7-digit Julian date then a 7-digit Julian date will be returned. If a 5-digit Julian date is given, a 5-digit date will be returned.

Generally, OPAL will accept 5 or 7 digit Julian dates where appropriate. The 7-digit date handling was introduced to support year 2000 changes.

### See also:

[Date Formats](#)

[Date Arithmetic](#)

# NUMBER

string function

— NUMBER — ( — <arithmetic expression> — ) —

NUMBER returns the English text of the value of the integer value of the above (<arithmetic expression>).

For example,

Number (52)	returns	"FIFTY TWO"
-------------	---------	-------------

SUPERVISOR Example

```
DEFINE + DISPLAY NUMERIX(MSG) :  
    Number(Decimal(Text))  
TT DISP NUMERIX 54678  
FIFTYFOUR THOUSAND SIX HUNDRED AND SEVENTY EIGHT
```

FLEX Example

```
REPORT Title,, "has been executed ",  
    Number(PDExecutions(Title))," times"
```

# OBJECTS

string function

Supervisor/Trim only

— OBJECTS — ( —

<MX context>
<PER context>
PRINTS
<TAPEDB context>
VDBS
VL
WHEN

: —>

>— <boolean expression> — , <limit> — ) —

The OBJECTS function is available in the SUPERVISOR and the TRIM modules only. Similar to the COUNT function, it provides a way to calculate the number of occurrences of a SITUation. OPAL evaluates the <boolean expression> as a SITUation for each mix number, unit number, print request or tape serial number, as described by the <context> type, and maintains a string-valued list of all the entities that returned TRUE. Each entity in the list is delimited by a comma.

For example,

Objects (PER:AvailableToGroup)	may give	"1,2,3,49,50,112,113"
--------------------------------	----------	-----------------------



As with SITUations of context PER, MX and TAPEDB, the **OBJECTS** function may use sub-contexts to restrict the search to specific areas within that context.

Note that a TAPEDB COUNT, without a sub-context to select an individual DMSII set, will always scan the entire METATAPELIB4 database.

Some sub-context examples:

```
Objects (MX=DBS,ACTIVE:"META" ISIN NAME)
Objects (MX=GOING,WAITING:True)
Objects (PRINTS:LinesToPrint > 5000)
Objects (VL:KIND=Pack And VolumeName HdIs "T")
```

## OBJECTS and contexts

As with SITUations of context PER,MX and TAPEDB, the **OBJECTS** function may use sub-contexts to restrict the search to specific areas of that context.

Note that a TAPEDB OBJECTS call, without a sub-context, will scan the entire METATAPELIB database, so using TAPEDB sub-contexts should always be preferred.

Some sub-context examples:

```
Objects (MX=LIBS,WAITING:True)
Objects (PER=MT-:AvailableToGroup)
Objects (TAPEDB=ByName:SerialNo HdIs "Z")
```

OBJECTS using the PRINTS context causes SUPERVISOR to scan the current PRINTS queue (as seen by a PS SHOW ALL). Using the VL context causes the system Volume Library to be scanned, examining all volumed pack and tape entries (only applicable on cataloging systems). The VDBS context searches for active DMSII databases and can be used to examine database attributes normally only available via SM STATUS.

For a list of valid sub-contexts, the Supervisor command 'HELP CONTEXT' from any Supervisor window will show this information about each context. It should be noted that evaluation of a OBJECTS function is relatively expensive, TAPEDB in particular, and especially if an unqualified context is used.

Optionally, for contexts such as TAPEDB in particular, the <limit> modifier, allows an "early" termination of an OBJECTS call without having to scan the entire database, returning only the number of entries desired.

For example

```
Objects (TAPEDB=SCRATCH:Density=1250,3)    might return

"B00011,B00555,C01234"
```

An OBJECTS call within the body of another OBJECTS is not allowed. If this were allowed, a multiplicative overhead would occur. Such recursion can always be avoided.

## OBJECTS in WAIT statements

If a WAIT statement occurs within an ODTSequence of context PER or MX, and the WAIT contains an OBJECTS call, SUPERVISOR will give a syntax error unless the body of WAIT is a Boolean expression of the form:

```
Objects(.....) <relational operator> <string expression>  
Wait(Objects(PER=MT:Scratch) NEQ Empty)
```

This restriction is imposed to simplify the analysis of just what the ODTSequence must be woken up to evaluate. The constructs allowing a program to wait for a program to enter from a given mix state (see WAIT statement) usually provide clearer and more efficient ways of avoiding OBJECTS in WAIT statements. Where they do not, a good technique is to generate an EVALUATE command or a ONCE command within an ODTSequence.

## OBJECTS and PER context

The OBJECTS(PER=<unit type>.... syntax allows a minus sign after the PER subcontext such that all units on the system, belonging to that device type, are returned regardless of status. For example:

```
Objects(PER=PK:True)
```

The above will look at all on-line, visible packs.

```
Objects(PER=PK-:True)
```

The above will examine all packs on the system even if not ready, FREED, etc.

Please note that if multiple sub-contexts are used (for example, PER=PK,LP-) then a minus sign following any of the device types in the list will cause the OBJECTS function to act as if a minus had followed all of the device types.

The special case OBJECTS(PER-: has not been considered here since a normal OBJECTS(PER: already looks at all units, regardless of status.

Example

```
DEFINE + ODTS MY_OBJECTS:  
    $WList:=Objects(MX=WAITING:"TEST WAIT" IsIn RSVP);  
    Odt($WList,"DS0");
```

The above two-line ODTSequence is very powerful; taking advantage of the MCP feature to apply certain ODT commands against a list of mix numbers, the **OBJECTS** call will return a list of mix numbers that are in the waiting entries with a RSVP whose text includes "TEST WAIT". The resultant string, held in the variable \$WLST, is then used to build a single ODT command DS them all. With the MX context in particular, care should be taken if using OBJECTS to construct an ODT mix number list command.

**See also:**

[COUNT function](#)

[VIA function](#)

## OCTAL

string function

— **OCTAL** — ( — **<string expression>** — ) —————|

OCTAL converts an octal string to an arithmetic value.

For example:

<b>Hex ("10")</b>	<b>returns 8 DECIMAL</b>
<b>Hex ("77")</b>	<b>returns 63 DECIMAL</b>

## PAD

string function

— **PAD** — ( — **<string expression>** — , —————▶  
 ▶————— **<arithmetic expression>** — ) —————|

PAD takes two parameters: a string and an integer. The function will return the contents of a string right-justified with leading spaces up to specified integer.

Example:

<b>"String=", PAD("ABC",12), "."</b>	<b>returns</b>
<b>"String=       ABC."</b>	

If the size of the string is bigger than the integer then the resultant string is simply truncated at the right.

## Example:

**Pad("ABCDEFGH",4) gives "ABCD"**

## SUPERVISOR example

```
TT DEFINE + DISPLAY C_MX(MX):
  MixNumber 4,, Name 38,,
  Pad(Time(AccumProcTime),9),,
  Pad(Time(ElapsedTime),9), " "
```

PAD is useful for producing tabular reports where the contents displayed in a column should be right justified.

## FLEX Example

**REPORT Title 60,, Pad( #(SEGMENTS),12),,  
CreationDate**

## See also:

## TRIM function

# PING

integer function

```
- PING - ( <host/ip address> _____ ) -  
                |_____, <count>_____|
```

The PING function has been implemented to allow a NW TCPIP PING command to be tracked by an OPAL script. PING accepts two parameters: the first is a hostname or IP address string while the second is an optional integer indicating the number of ping messages to be sent to that host. If the second parameter is absent, one message only will be sent.

If a hostname is provided, PING will use the SYSTEM/RESOLVERSUPPORT library to resolve the name to an IP address. An error will be returned if the host lookup fails. If a valid IP address has been determined, SUPERVISOR issues a NW TCPIP PING command to that address and the script will wait for the ping to complete.

Since TCPIP PING operations are processed asynchronously, beware that it is possible for another PING to the same IP address to be detected prematurely. Unfortunately, Unisys NW TCPIP PING encoded command and report entries do not

have any markers to identify an individual request though SUPERVISOR uses a specific SIZE specification in the PING command to help identify that the command has been processed.

A maximum of 20 messages can be sent by any PING command; note that if the IP address is not contactable, each ping will time-out after 60 seconds. This means that a 20-message ping to an unavailable IP address will take up to 20 minutes for PING to complete.

Note that if PING is used inside an event-based WHEN, then events normally destined for that WHEN will not be received whilst the slot is waiting for the PING to complete. An EV? interrogation of the waiting slot will show "(WAIT PING)" for the slot status and a SLOT command will show the IP addressed being PING-ed.

Once SUPERVISOR has detected that the PING has completed, an integer result value is returned to the caller and the script can determine information about the ping using a number of new attributes. The mnemonic values returned can be viewed with the HELP ATTR PINGRESULT command. The attribute PINGTEXT returns a string representing the life of the last PING function executed in this slot.

For example:

```
DEFINE + ODTS PING(MSG) :
    #P:= PING(TRIM(TEXT),3) ;
    SHOW(PINGTEXT) ;

TT DO PING DELL8500
PING to DELL8500
Hostname resolved to 10.0.0.35
PING result was PINGSUCCESS
PING has sent 3 messages, received 3 messages, 0 % loss
PING request duration was 0.024 seconds
Average PING response time was 0.008 seconds

TT DO PING 10.0.0.71
PING to 10.0.0.71
ERROR: NO RESPONSE TO PING
```

Note that PING looks up the hostname using RESOLVERSUPPORT so any mapped TCPIP hostnames that do not have a DNS entry will return a DNSUNKNOWNHOST error.

## POST

string function

— POST — ( — <slotno> — , — ————>

▶ ————— <text> ————— ) —————|

POST may be used to post a message to a CUSTOM Context Slot.

# REPEAT

## string function

— REPEAT — ( — <string expression> — , —  
▶ — <arithmetic expression> — ) —

REPEAT works, as in ALGOL, by returning a string that concatenates <arithmetic expression> copies of <string expression> together.

Normal restrictions on string length apply. Any attempt to make a string more than 1,999,999 characters will cause the program to be terminated. A negative count in the REPEAT function will return a null string.

For example:

```
Repeat("##",2) gives "####"
```

Assuming Attribute CU = 40000

```
Repeat("<", (CU-10000)/10000) gives "<<<"
```

SUPERVISOR Example

```
DEFINE + DISPLAY HEADING:  
  $Hdr:= "This is an underlined heading", /,  
  Repeat("=", Length($Hdr))
```

FLEX Example

```
REP HEAD $Hdr:="another underlined heading", /,  
  Repeat("=", Length($Hdr))
```

# RESPOND

## string function

Only valid in Supervisor HTTP context

RESPOND - ( — <status code> — <mime type> — <text> — ) —  
          — <entity> —  
          — <file key> - <mime type> - <text> - <URL> —  
          — WEBSOCKET - <mime type> - <text> - <URL> —

The HTTP context is still under active development and requires a component which is not currently released. At present it is included in the manual for the convenience of Metalogic staff.

The RESPOND function is used to send a response to an HTTP request. The function returns either an empty string if there is no error, or else it returns an error message.

<status code> ::= <integer>

<entity> ::= STATUS | HEADER | BODY | CHUNK | ERROR | FILE | DIRECTORY

<file key> ::= FILE | DIRECTORY | EVENTSTREAM | SMTP

<mime type> ::= <string expression>

<text> ::= <string expression>

A Respond using <status code> as the first parameter is referred to as a simple respond.

The simple Respond builds a complete HTTP response which is returned to the client. The Respond function cannot be used again within this invocation of the HTTP ODS.

Several of the common <mime types> are provided by the MIMETYPE attribute.

See Help Att MIMETYPE:

```
HELP ATT MIMETYPE
---- HELP ATTRIBUTES ----
MIMETYPE      (SYSTEM) Returns STRING
Parameters : 1. mnemonic values : PLAIN(1) HTML(2) XHTML(3) XML(4) XUL(5)
              XSL(6) CSS(7) JAVASCRIPT(8) JPEG(9) GIF(10) PNG(11)
              XPI(12) DOWNLOAD(13) SVG(14) XLINK(15) XFORMS(16) PDF(17)
              EVENTSTREAM(19).
Semantics : Returns the specified MIMETYPE definition
```

The file Respond ,identified by <file key> as the first parameter and two additional parameters, is used to send a file or directory listing to an HTTP client. The Respond function cannot be used again within this invocation of the HTTP ODS.

The <text> is either an MCP file title or directory specification.

The entity Respond ,identified by an <entity> as the first parameter and only one other parameter, is used to send the parts of an HTTP response to a client.

The FILE and DIRECTORY entities are provided so that HEADERS other than those provided by the file respond can be provided.

The DIRECTORY respond will accept an optional <URL> parameter.

An HTTP connection may be upgraded to a WEBSOCKET using the syntax, RESPOND(WEBSOCKET,<mimetype>,<text>,<url>).

## **.REVERSE method**

### **string method**

#### Definition





## SUPERVISOR Example

```

DEFINE + ODTSEQUENCE UNIT_STATS (PER) :
$Var:=String(Unitno,*);
#$Var:= RFERRORS;
IF #$Var > 50 Then
    Display("UNIT", UNITNO, " HAS ",
            #$Var, " ERRORS!");

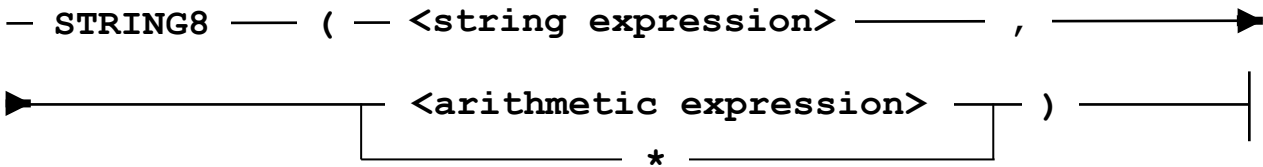
```

## FLEX Example

```
REPORT If Segments > 0 Then
    (Title & " SIZE = ", String(Segments, *))
Else
    (Title & " EMPTY FILE")
```

# STRING8

## string function



The `STRING8` function differs from the `STRING` function in that it does not integerise and it suppresses leading zeroes.

The decimal field width option of an OPAL string is often more convenient.

## SUPERVISOR Example

```

DEFINE + DISPLAY STRINGEIGHT:
  "INTEGER = ", 412.5633/48, /,
  "REAL     = ", String8(412.5633/48,*), /,
  "Formatted= ", 412.5633/48 2.3

```

## When Run

```

TT / StringEight

INTEGER = 9
REAL     = 8.59506875
Formatted= 8.595

```

## .SUM method

## arithmetic method

### Definition

# TAG

## string function

- TAG - ( — <string expression> — , —————>  
————— <string expression> — ) ———|

The TAG string function can be used to enclose a string expression in XML compliant tags. The function takes two string expressions as parameters. The first parameter describes the tag, the second is the string to be tagged. All text up to the first space in the 'tag' parameter is treated as the Tag, the rest of the text being attributes of the tag.

Example

```
$E:=Tag("td","1234");  
$Line:=Tag('table width="694" border="0"',  
Tag("tr",$E));
```

Would Leave

```
$Line = <table width="694" border="0"><tr><td>1234</td></tr></table>
```

# TAIL

## string function

- TAIL - ( - <string expression> ——— , —————>  
▶————— <string expression> ——— ) ———|  
          └ Not ─┘           └ ALPHA —————┘

TAIL is identical to the same function in WFL except that no run-time errors occur if the <string expression> has zero length .

SUPERVISOR Example

```
DEFINE + DISPLAY MCPCOMPILE:  
"MCP: ", MCP, " ",  
MCPMark 1, MCPLevel 1, ".", MCPCycle 3, /,  
"Compiled: ", Drop(Tail(MCPTimeStamp, Not ","), 1)
```

FLEX Example

```
SELECT Tail(Title, Not "/" ) HdIs "/FIRSTLEVEL"
```

The above SELECT has the same effect as

```
SELECT FileID(Title,2) HdIs "FIRSTLEVEL"
```

## string function

The TAKE function works as in WFL and ALGOL except no run time errors occur if the value of <arithmetic expression> is less than zero or greater than the length of the string.

```
MAX(0,MIN(LENGTH(<string expression>
          ,ABS(<arithmetic expression>)))
```

Take ("ABCDEFGHI", -3)	returns	"GHI"
------------------------	---------	-------

```
Drop ("ABCDEFGH", Length ("ABCDEFGHI") - 3)
```

```

DEFINE + SITUATION META_TAPE (TAPEDB) :
  If (Take (SerialNo,2) = "ME" And
      Length (Tail (Drop (SerialNo,2) , "0123456789"))=0)
      Or Take (SerialNo,4) = "META"
  Then "  METALOGIC  "
  Else "  COMPANY NAME  "

```

```
SELECT Take(Title,5) = "*META"
```

## string function

The TAKEFILEIDS function is similar to the TAKE function but treats the string expression as a file title and returns the requested number of levels from the start of the title.

## Opal Functions

If the number of levels requested is zero then the usercode and ON part are returned.

If a negative number of levels is requested the levels are taken from the end of the fileid.

If the file title is usercoded or prefixed with \* then the usercode or \* will be returned with the requested levels.

If the file title has an ON part then it will be returned with the requested levels.

Examples

```
$S:="A/B/C"
TakeFileIds($s,2) returns A/B
TakeFileIds ($s,-2) returns B/C
TakeFileIds($S,10) returns the empty string
$X:=" (BOB)A/B/C ON DEV"
TakeFileIds($X,1) returns (BOB)A ON DEV
TakeFileIds($X,-1) returns (BOB)C ON DEV
TakeFileIds($X,0) returns (BOB) ON DEV
$Bad:="a/b/c"
TakeFileIds($bad,1) returns the empty string
$Good:="'a"/B/C'
TakeFileIds($Good,2) returns "a"/B
```

## TDADDRESS

string function

— TDADDRESS ———— <arithmetic expression> — , —————>  
————— <arithmetic expression> — ) —————|

TDADDRESS generates a string of control characters to move the cursor of a Unisys ET1100 or equivalent terminal to the row and column specified. The first expression is the row and the second the column. The string will be four EBCDIC characters, beginning with an escape (ESC) followed by a quote: #[ESC]". The remaining two characters are interpreted in the terminal as the address requested.

If specified at compile-time, the first expression (row) must satisfy the following relation:

```
1 <= row <= 24
```

and the second (column) must satisfy the relation:

```
1 <= column <= 80
```

If either row or column is not constant, its value will be adjusted at run time according to the formulae

```
1+((row-1) MOD 24) and 1+((column-1) MOD 80)
```

### SUPERVISOR Example

```
TT DEFINE + DISPLAY HL_DISPLAY
  TDAddress(1,1), "#[ESC]J",
  TDAddress(1,10), "#[SUB]HALT-LOAD"
```

### FLEX Example

```
REP HEAD TDAddress(1,1), "#[ESC]J",
  TDAddress(1,10), "#[SUB]FILE ",
  "STATISTICS ", ,
  TDAddress(2,11), " ==== ====="
```

## TDPAGE

### string function

— **TDPAGE** — ( — <arithmetic expression> — ) —

The TDPAGE function generates a string of control characters to move the cursor of a Unisys ET1100 or equivalent terminal to the start of the page specified by the <arithmetic expression>. The string will be three EBCDIC characters, beginning with an escape (ESC) and followed by a dollar #[ESC]\$. The remaining character is the page requested.

The <arithmetic expression> must satisfy the relation

```
1 <= <arithmetic expression> <= 95
```

If it is not constant, the value will be adjusted at run time according to the formula

```
1+((<arithmetic expression>-1) MOD 96)
```

### SUPERVISOR Example

```
TT DEFINE + DISPLAY PAGE_SWITCH:
  TDPAGE(1), "Writing top page one", /,
  TDPAGE(2), "Now to page two"
```

### FLEX Example

```
REP HEAD TDPAGE(1), "HEADING ON THIS PAGE", #PG.STORE(2),
  TDPAGE(#PG), "... AND ON THIS PAGE TOO"
```

## TIME

### string function

— **TIME** — ( — <arithmetic expression> — ) —

TIME generates a string in hours, minutes, and seconds separated by colons. <arithmetic expression> is assumed to be a value in seconds.

For example:

```
Time(7266)
returns
"02:01:06"
```

SUPERVISOR Example

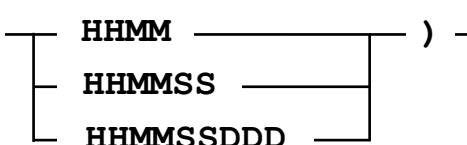
```
TT DEF + DISP MCPTIME:
    "Time of day HH:MM:SS = ", Time(TimeOfDay), /,
    "Time since last H/L = ", Time(MCPTime)
```

FLEX Example

```
REPORT HEAD "File Report prepared at ", Time(TimeOfDay),
            " on ", DateToText(Today,DDMMYYYY)
```

## TIMETOTEXT

string function

– **TIMETOTEXT** – ( – <arithmetic exp>  ) |

TIMETOTEXT generates a string with hours, minutes and seconds and fractions of a second, separated by colons, according to the mnemonic specified as the second parameter. <arithmetic exp> is assumed to be a real value in seconds.

Example:

```
TIMETOTEXT(7266.5,HHMMSSDDD) returns 02:01:06.500
TIMETOTEXT(7266.5,HHMMSS)   returns 02:01:07
TIMETOTEXT(7266.5,HHMM)     returns 02:01
```

## TRANSLATE string function

— **TRANSLATE** — ( — <string expression> — , — <how> — ) —

The Translate function returns the text in the <string expression> ,translated according to the mnemonic in <how>. The valid mnemonics and their translate actions are described in the following table.

<how>	Action
ASCIIToEBCDIC	Translates the <string expression> using the MCP translate table ASCIITOEBCDIC.
Base64Decode	Translates the source string containing a Base64 encoding to its ASCII value and then translates it to EBCDIC for use within OPAL.
Base64Encode	Assumes the source string consists of graphic characters destined for a remote ASCII system. Translates the source string to ASCII and then encodes it as Base64
BinaryToBase64	Translates the source string value to Base64.
BlankNonGraphics	Replaces any non graphic character with a space.
DOUBLEQUOTE	Using the first quote in a string as the model, it doubles all occurrences of that quote in the remainder of the string.  For Example: Translate('*PDUMP/"TASK 1"',DOUBLEQUOTE) would result in the string *PDUMP/""TASK 1"".
EBCDICToASCII	Translates the <string expression> using the MCP translate table EBCDICTOASCII.
EBCDICToHex	Translates <string expression> using the MCP translate table EBCDICTOHEX. For example, the EBCDIC string "F0F1" would translate to "01".
EntityDecode	Decodes the standard xml entities back into their. character values.

<how>	Action
EntityEncode	Encodes the standard xml entities & to &amp; < to &lt; > to &gt; ' to &apos; " to &quot;
FirstLetter	Will change the first letter of the source to upper case and the rest to lower case.
FormUrlDecode	Like URLToEBCDIC with the addition of translating the plus sign (+) to a space.
FormURLEncode	URL encodes the <string expression> and in addition translates spaces to a + sign.
JavaScriptEscape	Translates ' " and \ using the JavaScript escape syntax \' \' " and \\.
LowerToUpper	Same as Upper function.
RemoveNonGraphics	Translates <string expression> by removing any non graphic characters.
Squashed	Removes leading, trailing and multiple embedded spaces from <string expression>.
ToHexString	Same as HexString function.
ToHexIfNotDisplay	If the string expression contains any non graphic EBCDIC characters, excluding HT,FF,CR and LF, then the string expression is translated to its hex representation.
UNQUOTED	Removes matching single or double quotes from beginning and end of string. If the first and last characters are not quotes or do not match, no change is made.
UpperToLower	Same as Lower function.
URLEncode	URL encodes the <string expression>. See RFC 1738.
URLToASCII	Translates URL encoded characters in <string expression> to their ASCII character value



<how>	Action
URLToEBCDIC	Translates URL encoded characters in <string expression> to their ASCII character value and then translates the ASCII characters to their EBCDIC equivalents.
URLToFileIDs	Translates a URL to file title string which can be used with the FILEID function. It removes any leading or trailing slashes. Any level which has non identifier characters is quoted. A quoted level which only contains valid identifier characters has the quotes removed.

## Examples

```

TT / (Translate(' "Test" ',EntityEncode)
&quot;Test&quot;;

TT / (Translate("  now      is      the      time",Squashed)
now is the time

```

# TRANSMIT

## string function

— **TRANSMIT** — ( — **<FileHandle>** — , — **<TEXT>** — ) —

The Transmit function is used to transmit text to a Remote File Program, run via the SUPervisorWFL function, which is waiting for Input.

The FileHandle parameter is a real value used to identify the remote file and can be obtained from the FileHandle attribute in the Job context. See the Supervisor ON JOBMESSAGE statement for more details.

<text> is a string expression. If it is successfully transmitted to the program, the function returns EMPTY, otherwise it returns the the DCWrite Error Message.

If the <text> is ?END, an EOF indication is sent to the Remote File.

# TRIM

## string function

— **Trim** ( — **<String Expression>** — , — **Left** —  
— **Right** —  
— **Both** —  
— **All** — ) —

TRIM will remove from its string parameter any leading or trailing spaces or both(default). The Special case 'All' is like 'Both' but also replaces multiple embedded spaces with a single space. The exception to this behaviour is that quoted text sections inside the target are left untouched.

For example, the following OPAL expressions:

```
"Start=",Trim("      ABCDEF   ABC      "),"."  
would give the result  
"Start=ABCDEF   ABC."  
  
"Start=",Trim("      ABCDEF   ABC      ",Left),"."  
would give the result  
"Start="ABCDEF   ABC      ."  
  
"Start=",Trim("      ABCDEF   ABC      ",Right),"."  
would give the result  
"Start=      ABCDEF   ABC."  
  
"Start=",Trim("      ABCDEF   ABC      ",All),"."  
would give the result  
"Start=ABCDEF ABC."  
      TRIM(' This   is a   test with embedded " Quoted   string"',ALL))  
would give the result  
"This is a test with embedded " Quoted   string"
```

There is no effect if the string is null or has no leading or trailing spaces.

### SUPERVISOR Example

```
DEFINE + SITUATION TRIM_IT(MSG) :  
  "THE MCP IS      " HdIs TEXT And  
  Trim(Decat(Text, "THE MCP IS",1)) NEQ ""
```

DECAT drops the leading string "THE MCP IS" while TRIM removes leading and trailing blanks.

## TT

string function  
Supervisor Only

— **TT** — ( — <string expression> — ) —

This string function, similar to KEYIN, permits the user to process any TT command (without the prefix) and cause the calling OPAL to wait until the command has been completed. The difference from KEYIN is that special slot commands such as WHEN, ONCE, EVAL and DO will also cause the caller to wait, instead of processed independently. Like KEYIN, TT returns the response of the SUPERVISOR command as the function string result.

CONTROLLER allows multiple commands, separated by CR to be entered. This is disallowed for the TT function, so that a parameter for an ODTs that contains a CR is not truncated. The TT function is meant to be used for a single activity, with a result being returned in the function value.

An important aspect of any WHEN, EVAL or DO invoked by the TT function, is that the variable heap is automatically shared with the environment associated with the new DO or EVAL. When the DO or EVAL completes, the updated variable heap is then automatically assigned to the caller which then continues processing. Whilst the caller is waiting for a TT call to complete, the slot will show with a status of 'WAIT TT #2' where #2 is the slot number of the WHEN,DO or EVAL.

Previously, sharing variable information between OPALs in this way was only possible using GLOBAL variables and then continuously checking for any called OPAL programs to complete. The TT mechanism gives much greater control to the caller.

All SUPERVISOR commands passed with the TT function will always return a response. If the command is an in-line OPAL or DEFINE that fails syntax, the error list will be returned. THE TT function can *\*only\** be used in ODTSEQUENCES.

The following example would display the response from a TT FP command:

```
DEFINE + ODTS FP:
    $R:= TT("FP");
    Show($R);
```

The EVAL command passed in the TT function below would cause EVALTEST to wait for the EVAL to complete. Any variables updated inside the LOG ODTSEQUENCE will be immediately available for use by EVALTEST after TT completes.

Example

```
DEFINE + ODTS EVALTEST(MSG):
    $R:= TT ("EVAL (LOG:TRUE) DO ($Pg:=&LogText) [50]");
```

If EVALTEST was triggered by a WHEN, then the WHEN will be suspended until the EVAL completes; however, no message events will be missed during this time.

A WHEN ? might show:

```
W 218 65145 WHEN MSG DO EVALTEST (WAIT TT #5) 04
                                     TIMES:CPU=00:00:11,IO=00:00:00,ET=00:0
      16 evals, 16 ODTS entries
W 218 65210 EVAL INLINE_LOG_5 DO INLINE_LOG_5 RUNNING 05
                                     TIMES:CPU=00:00:00,IO=00:00:00,ET=00:0
      44 evals Limit 10000 @ 17:08:01 18/01/07 (0% of #000296), 44 ODTS
```

The MSG WHEN has a status of 'WAIT TT #5' which indicates that the WHEN is suspended until the inherited EVAL in slot #5 completes. In the above case, the number of queued notices can be seen in the response to a 'SLOT 4' command.

NOTE: The WHEN command may only be used inside a TT function if there are limits imposed on the number of user-specified evaluations and/or ODTS entries. If neither of these limits are provided, the WHEN will be rejected.

For example, the following WHEN will be permitted:

```
$Z:= TT ("WHEN MSG [50] DO MSG");  
$Z:= TT ("WHEN MSG DO MSG [100]");
```

Even with limits imposed, use of WHEN statements inside a TT function called from another event-driven WHEN should be used with caution. Any attempt to use WHEN without limits will generate a run-time error.

SUPERVISOR will only allow a nesting of one additional TT call.

Any attempts to use more than 1 nested TT call will cause the calling slot to fail with the error "Only one nested TT call allowed". Also, any DO commands executed from a nested TT function call will not be logged in the SUPERVISOR log or the SUMLOG, regardless of the setting of LOGMINIMAL and MONITORING, to aid performance.

## UNZIP

string function

– UNZIP – ( – <string expression> — ) —————|

UNZIP can only be used to decompress a single data stream which has been compressed using MZIP, GZIP or ZLIB. MZIP is Metalogic's own compression mechanism and is optimised for MCP systems.

An UNZIP function example:

```
$ZIPSTRING:=$Zipped.Perm  
If Not $Z:=UNZIP($ZIPSTRING) Hdis "Err" Then  
    SHOW("Z")  
Else  
    SHOW("Zip Error-> ", $Z);
```

All UNZIP errors are returned as the function result and are prefixed by "Err". In SUPERVISOR environments, the attribute MZIPLASTERROR returns a non-zero value if the operations fails.

## .UNZIPFILE method

string method

Definition

# UPPER

## string function

— UPPER — ( — <string expression> — ) —————|

The UPPER function converts all lower case ALPHA characters in the given <string expression> parameter into upper case. Characters that are already upper case are unaffected.

Example

```
Upper ("AbCdE") = "ABCDE"
```

See also:

[LOWER](#)

# USERFN

## string function

— USERFN — ( — <arithmetic expression> — , —————▶  
————— <string expression> ————— ) —————|

In compiled languages on Unisys A Series systems, there are functions supplied by the system software, but it is also possible to call user-written library procedures. USERFN allows the OPAL programmer to call a user-written procedure. When USERFN is called from OPAL, a procedure called USERFN is called in a library called OPALUSERLIB.

The procedure has the following ALGOL declaration:

```
STRING PROCEDURE USERFN (CASENO, STRNG) ;  
VALUE CASENO ;  
INTEGER CASENO ;  
STRING STRNG ;  
LIBRARY OPALUSERLIB ;
```

An example Library showing this capability is supplied for both SUPERVISOR and FLEX products. This may be modified at will to allow a site to specify its own OPAL functions. It is available in the Metalogic release container or CD.

Named:

(METASOFT) OPAL/USERFNLIB

Note that any error in the user programming of **USERFN** could lead to an abnormal termination of a FLEX utility run; SUPERVISOR calls of **USERFN** are now protected.

# VALID

boolean function

— VALID — ( — <attribute primary> — ) —

The VALID function allows the caller to verify whether a particular attribute is valid. An attribute's validity can depend on various factors; for example:

VALID will return FALSE if the value of an attribute is undefined. See the section on "Attribute Types" in Chapter 6 for values returned by undefined attributes.

The SECURITYGUARD attribute is not valid if the SECURITYTYPE of the file is PUBLIC.

FLEX Example

```
SELECT Valid(SecurityGuard) And
      "*MYGUARD/FILE" = $GFILE:=SECURITYGUARD
```

VALID (...) check is much faster than checking two strings.

SUPERVISOR Example

```
DEFINE + SITUATION IS_COMPILER(MX=ACTIVE)
      Valid(COMPILENAME) And
      "DCALGOL" IsIn CompilerName
```

# VIA

function primary

Supervisor/Tape Library only

— VIA — ( — <reference attribute primary> — ) —

<reference attribute primary>

<reference attribute>	_____
PrintsStr	( — <string expression> — ) —
TapeDB	
TaskStr	
UnitStr	
VDBSStr	
PDKEY	
Prints	( — <arithmetic expression> — ) —
Task	
Unit	
VDBS	

The VIA function is a very powerful mechanism for retrieving attribute information from a completely different OPAL environment.

This feature allows task, database, unit, Print System or Tape Library information about a single object to be dynamically retrieved from any OPAL context or program. Previously, the COUNT or OBJECTS functions were commonly used to extract information using OPAL techniques and could be expensive if data for only one specific object was needed.

The various reference attributes, TASK, UNIT, PRINTS, and VDBS require <arithmetic expression> as a parameter whereas TASKSTR, UNITSTR, PRINTSSTR, TAPEDB and VDBSSTR each expect a <string expression>. This parameter should evaluate to a valid system object e.g. a mix number for TASK, a unit number for UNIT etc.

The following table describes each reference attribute and its semantics.

Reference attribute	Expression Context	Parameter requirement	Parameter Examples
PRINTS	PRINTS	Valid PrintS request number in arithmetic form	12345, #REQ
PRINTSSTR	PRINTS	Valid PrintS request number in string form	"23456", \$MYREQ
TAPEDB	TAPEDB	Valid tape serial number in TRIM database in string form	"B10023", \$SN
TASK	MX	Valid active mix number in arithmetic form	1234, #MIXNO
TASKSTR	MX	Valid active mix number in string form	"45678", \$TSK
UNIT	PER	Valid peripheral unit number in arithmetic form	99, #MYUNIT
UNITSTR	PER	Valid peripheral unit number in string form	"3001", \$UNIT
VDBS	VDBS	Valid active database mix number in arithmetic form	1234, #DB
VDBSSTR	VDBS	Valid active database mix number in string form	"5678", \$DBS
PDKEY	PD	Valid file title	"MYFILE ON DEV"

When a VIA function is encountered at run-time, Supervisor will temporarily "switch" to the specified context and extract the attribute information specified by <expression>. It depends on how and where the VIA is used as to the value returned by the function; the OPAL compiler allows considerable flexibility in its handling.

Consider some examples:



```

Via (UNIT (50) :Label)
$UINF:=Via (UnitStr ("50") :Label)
$T:=Via (Task (#MIX) :# (Name 40,,Time (AccumProcTime)))
#ACC:=Via (TaskStr ($MixNo) :AccumProcTime)
Via (TAPEDB ("B00012") :# (SerialNo,,Title,,CreationDate))
Via (VDBS (1234) :# (AllowedCore,,OverlayRate 3.4))
$FILE:= "*SYSTEM/FILEDATA ON DISK";
  SHOW ("INFO = ",VIA (PDKEY ($FILE) :# (RELEASEID,,CREATIONDATE)))

```

The above OPAL expressions may be used in any environment or OPAL program type as long as the parameter equates to a valid mix or unit number etc. The # ( ) ("hash-paren") function is useful for building complex strings inside the <expression> component, if needed.

The following powerful but simple OPAL MESSAGE program allows a file name to be passed as a TEXT parameter to the PDUSERMIXLIST attribute. If the parameter is a valid file and is currently in-use by one or more programs, a list of mix numbers will be stored in the string variable \$MIXL. The list is then parsed using SPLIT and each entity in the list is passed to a TASKSTR function to extract the usercode and task name.

Example:

```

TT DEFINE + ODTs LIST_USERS (MSG) :
  $MIXL:= PDUserMixList (Text) ;
  While $MX:=$MIXL.Split NEQ Empty Do
    Show (Via (TaskStr ($MX) :# (MixNumber,,User 17,,Name 40))) ;

```

To find those tasks are currently using the METALOGIC/SUPERVISOR codefile, just pass the file name as a parameter with the DO statement.

A very typical response is shown below:

```

TT DO LIST_USERS *METALOGIC/SUPERVISOR

7356 SUPERVISOR *METALOGIC/SUPERVISOR
7357 TAPELIB      (SUPERVISOR)METALOGIC/SUPERVISOR/TAPELIB
7362              (SUPERVISOR) SUPERVISOR/GRINDER
THE DO WILL BE DONE

```

Using the VIA function and SPLIT method, as shown in the above examples, is an excellent method of extracting information for individual entries in a list of Mix, Prints, Tape or Per entities. The OBJECTS function, coupled with OPAL attributes like PDUSERMIXLIST and LIBUSERLIST, is also perfectly suited for this type of information handling.

Example using .MX method:

```
TT DEFINE + ODTs LIST_USERS(MSG) :  
    $MIXL:= PDUserMixList(Text) ;  
    While $Mix:=$MIXL.Split NEQ Empty Do  
        Show($Mix.Mx(#(MixNumber,,User 17,,Name 40)) ;
```

## OPAL reference attributes

In SUPERVISOR and TRIM modules, a number of OPAL Attributes are documented as type MX REFERENCE or PER REFERENCE. They have two uses — the first behaves as a normal attribute, returning a mix number or unit number in normal usage within their program contexts. However, these attributes can also be used to access context information about a program or physical unit, referenced by the value of the attribute, from OPAL programs other than MX or PER.

The most common reference attributes are MIXNUMBER and UNITNO. These attributes are available in a number of different SUPERVISOR program contexts.

For example, the reference attribute MIXNUMBER in a PER context can be used, using the VIA function, to return information about that task directly, within the PER program.

As shown below:

```
TT DEFINE + DISPLAY PER_EX(PER) :  
    If InUse Then  
        #("Unit ",UnitNo," is in use by ",Via(MixNumber:Name))  
    Else  
        "Not in use"
```

Here, the reference attribute MIXNUMBER is common to both the PER and MX context and allows the retrieval of the name of the task, currently accessing the specified unit, by way of the VIA function.

Note that the expression used inside a VIA function can return any type (Boolean, arithmetic, or string), depending on the attribute or attributes used.

The use of this all kinds of VIA adds a degree of overhead. To see why, consider how an operator would evaluate the question answered by TAPEOWNER below. He would enter something like OL MT 23, determine the mix number of the program using the tape, and then enter a Y system command for that mix number. SUPERVISOR must also perform these two operations.

As of MCP release level 53.1, the current reference attributes are:

Name	Context	Name	Context
BACKUPEU	SYSTEM	MIX	PER
FATHER	MX	MIXNO	MSG

Name	Context	Name	Context
HLEU	SYSTEM	OFFSPRING	MX
HLUNIT	SYSTEM	ORGUNIT	MX
HLUNITNO	SYSTEM	SIBLING	MX
JOBNUMBER	MX		

For their semantics, see HELP ATTRIBUTES.

## SUPERVISOR Examples

```
DEFINE + SITUATION SV_MSGS (MESSAGE) :
    "PC-MAINFRAME TRANSFERS ACTIVATED" IsIn Text And
    Via (MixNumber:Name) = "*METALOGIC/SUPERVISOR"
```

The above OPAL will check the name of the originating task that issued the display message.

The following program uses the TASKSTR function:

```
TT DEFINE + ODTSEQUENCE FIND_MX (MSG) :
    $Patt:= Trim(TEXT) ;
    $List:= Objects (MX=A,LIBS,S,W:Name EQW $Patt) ;
    $Disp:= #(" ----- TASK PATTERN: ",Text," -----",/)) ;
    While $MX:=$List.Split NEQ Empty Do
        $Disp:= &Via (TaskStr ($MX) :
            # (MixNo,,User 12,,Name 40,,Time (AccumProcTime),/))) ;
    SHOW ($Disp) ;
```

The FIND\_MX program will perform a similar task to the AA NAME <wildcard> ODT command that was introduced in MCP 4.1, except that this uses OPAL to find all programs that match the given pattern.

Note the use of the EQW operator to handle the pattern matching.

Example:

```
TT DO FIND_MX =SUPER=
```

would find all mix entries with the literal string "SUPER" in the task name.

Example:

```
DEF + SITU JOBUSERCHANGE (MX) :
    User NEQ Via (JobNumber:User)
```

would identify mix entries where the task usercode did not match the usercode of the parent job.

Example:

```
DEF + DISP TAPEOWNER (PER) :  
    "TAPE ",@," IN USE BY ",  
    Via (MIX:User)
```

would show the usercode if the task using a tape drive.

Example:

```
DEF + ODTs NODISP (MSG) :  
    ODT ("SS ",Via (MixNo:SourceStation) ,  
        "NO DISPLAYS!!") ;
```

would send a message to the source station of the task.

**See also:**

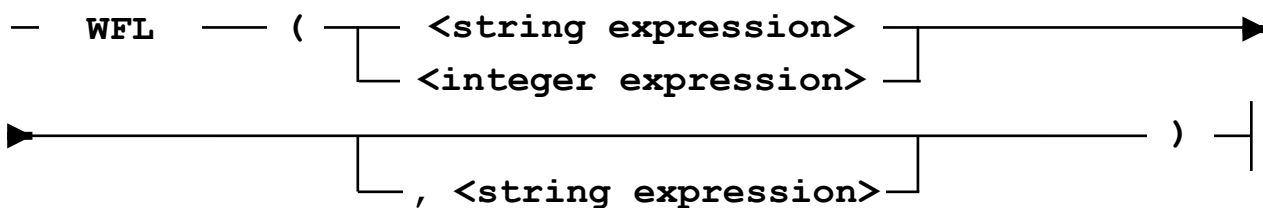
**Functions:** [COUNT](#); [OBJECTS](#)

**Methods:** [SPLIT](#) , [MX](#), [PER](#), [PRINTS](#), [TapeDB](#), [VDBS](#), [WHEN](#)

## WFL

string function

Supervisor Only



The WFL function allows WFL-style commands such as PROCESS, RUN and START to be processed from a SUPERVISOR ODTSequence. Although this activity can be simply achieved using the ODT statement, the main difference is that all job and task messages emitted by the WFL can be captured within the same OPAL program using a new statement, [ON JOBMESSAGE](#).

If a string passed to the ODT statement or WFL function is intended to be treated as a WFL command it is always safest if it starts with "BEGIN JOB;" This tells the ODTs statement that this is a WFL Job. Without BEGIN JOB Supervisor has to guess if the command is for CONTROLLER or for a JOB.

If sent to controller the usercode is not passed. Controller recognises some commands as being WFL commands and adds a "BEGIN JOB;" to the front and passes the message to controlcard. The reserved words recognised by controller as of MCP 54.1 are:

RESTORE,ADD,PASSWORD,SECURITY,STARTJOB,EXECUTE,COMPILE,PROCESS,DISPLAY,  
ARCHIVE,CATALOG,REMOVE,CHANGE,UNWRAP,ACCESS,VOLUME,MODIFY,BEGIN,RERUN,  
ABORT,PRINT,ALTER,MKDIR,CLASS,QUEUE,BIND,COPY,USER,END,PTD

Supervisor uses this same list to determine WFL commands.

The WFL function allows SUPERVISOR users to track single or multiple WFL activities from a single OPAL program. All job messages, BOJ and EOJ information are returned and made available via a special OPAL context called JOB. It is not possible to DEFINE an ODTSequence with a context of this type.

The WFL function is only available on MCP 48.1 or later; it may only be used from an ODTSequence.

The WFL function requires a string parameter, passing the name of the job to be invoked or other Unisys WFL commands. Optionally, it will accept a second string parameter that can be used to identify the job - this can be used to assist the tracking and reporting of multiple jobs from the same ODTSequence. This parameter cannot exceed 17 characters.

The value held in the first <string expression> can represent a job name, which will be START-ed by SUPERVISOR, or commands such as PROCESS, RUN or BEGIN JOB. If successful, the WFL functions returns the Job Number associated with the command. If the function fails a negative value is returned, reflecting the failure reason.

The WFLERRORTEXT attribute may be used to interpret any errors.

```
HELP ATT WFLERRORTEXT
---- HELP ATTRIBUTES ----
WFLERRORTEXT (SYSTEM) Returns STRING
Parameters : 1. INTEGER
Semantics : The parameter is the value returned by the WFL
            function. Returns the error text for that value. Returns
            the empty string if there is no error (Value Geq 0).
```

For example:

```
#JOBNO:= WFL(" (TEST) BATCH ON DEV")
IF #Jno:=WFL("RUN *OBJECT/TESTPROGRAM ON HLPACK") > 0 THEN
    #T:= WFL('BEGIN JOB J;DISPLAY "TEST!";END JOB')
Else
    SHOW("WFL EROR ",WFLERRORTEXT(#Jno));
WFL(" (LIVE) BATCH/JOB ON PROD", "BATCHID");
```

SUPERVISOR will apply any usercode attributes assigned from the WHEN slot but this can be overridden by using FOR syntax prefixing the START command. The FOR modifier syntax is identical to that used by the AFTER command.

For example:

```
WFL("FOR META/PW ACCESS T/PW1 CHARGE C123 START JOBNAME")
```

The above job will run under the usercode 'META', the accesscode 'T' and a chargecode of 'C123'. Within SUPERVISOR, there are ways to execute these jobs, without using passwords, by removing the **NODEFAULTUSE** Userdatafile attribute from the SUPERVISOR or by running the DO under the ODTSECURITY usercode.

Note that a valid WFL command *\*must\** be provided after the FOR modifier.

If the WFL function fails for any reason, one of the following negative values may be returned:

-1	BADWFL	: Various; particularly START of illegal file
-2	NOFILE	: START of non-resident JOBSYMBOL file
-3	NOTIMP	: Not available on pre-48.1 MCP
-4	NOENTP	: Missing MAGUS support
-5	PRGFLT	: Internal Metalogic fault
-6	BADFOR	: Bad FOR modifier (usercode/accesscode/charge)
-7	EMPTYID	: User supplied an empty id string parameter
-8	DUPID	: Id parameter already being used
-9	LONGID	: Id parameter exceeds 17 characters

As described above the WFLERRORTEXT attribute may be used to interpret these errors

Once one or more WFL jobs have been invoked, the MCP automatically passes all job-related information to the requesting WHEN slot. These messages remain queued in the slot until an ON JOBMESSAGE block is entered. (See [ON JOBMESSAGE](#) for details)

Examples:

```
WFL("START (META) JOB ON DEV")
#A:= WFL(' (META) J("TEST",2003011) ')
WFL("RUN *OBJECT/APP ON PACK")
WFL("FOR META/PW START *BATCH/JOB ON DISK", "MYBATCHID")
```

The WFL function should be used with care; it is not possible to use ON JOBMESSAGE blocks inside an ODTSEQUENCE linked to a WHEN because of event-handling issues.

If the first parameter is an integer then it should represent a JobNumber. This mechanism is used to control jobs restarted after a Halt/Load.

A WFL Job identified by its <jobnumber> can only be restarted if it is waiting in a System Queue. The WFL function associates the current slot with the <jobnumber>, so that messages are returned to the ON JOBMESSAGE block, and it forces the WFL Job out of its Queue.

Ex.

```
#JobNo:=WFL(<jobnumber>,"<id">) restarts the WFL Job specified  
by the <jobnumber>, after a  
Halt/Load.
```

## Remote File IO

An ODTS may participate in Remote File I/O by adding modifiers to the Job identity.

Example:

```
#JobNo:=WFL('BEGIN JOB;STATIONNAME=A;RUN OBJECT/REM',  
'MYID(STATIONNAME=A,MYUSE=OUT)');
```

If the <id> part of the WFL function is followed by an open parenthesis, then a comma separated list of attributes is expected, followed by a closing parenthesis. These attributes define how the ODTS will participate in Remote File I/O.

The attributes are,

<b>STATIONNAME=&lt;filename&gt;</b>	<b>(Required)</b>
<b>PAGE=&lt;integer&gt;</b>	<b>(Optional,Default 24)</b>
<b>WIDTH=&lt;integer&gt;</b>	<b>(Optional,Default 80)</b>
<b>MYUSE=IN OUT IO</b>	<b>(Optional,Default OUT)</b>

The STATIONNAME specified in the attribute list must match the STATIONNAME attribute in the WFL Job ,for participation in Remote File I/O.

If MYUSE is OUT, then a Pseudo Station with the specified STATIONNAME is allocated.

The ON JOBMESAGE context receives a WFLOPEN message when the Remote File is opened, a WFLOUT message when the programs writes to the Remote File, and a WFLCLOSE message when the Remote File is closed.

The JOBTXT attribute contains information about the Remote File, or the data written, depending on the JOBMSGTYPE.

If MYUSE is IN or IO, then a Schedule Station is allocated, and the STATIONNAME attribute in the Job is changed to be a STATION attribute with the LSN of the Schedule Station.

When the Program Reads from the Remote File a WFLIN message is received by the ODTS and the Program waits for Input.

Input can be sent to the Remote File using the OPAL [Transmit](#) Function. This function requires an FRSN, which is available from the FRSN attribute of the JOB Context Object.

The LSN and FRSN attributes have been added to the JOB Context, and are valid for WFLOPEN, WFLOUT, WFLIN and WFLCLOSE messages.

Please refer to [ON JOBMESAGE](#) in [Opal Statements](#) for more information on this feature.

The WFL function may also be used in conjunction with the VTCONTEXT to provide Remote File Scripting.

When a program writes to a Remote File, the message is interpreted by the TD830 Terminal Emulator and presented on the Screen. When Scripting, the Terminal Emulator function is provided by the VTContext.

A message sent to a Terminal through a Remote File is received within the SUPERVISOR JOB Context as a WFLOUT message. This message can be sent to the VTContext to be interpreted.

There are 2 Interfaces to the VTContext.

DO | EVAL <odts>[EDIT IN SL VTCONTEXT] '<td830 message>'

The EDIT interface interprets a message destined for a TD830 Terminal Emulator. The text and control codes are interpreted to create a Virtual Screen.

The <td830 message> must be enclosed in single quotes (') or double quotes ("), and may be preceded by a <string code>, either 7 for ASCII or 8 for EBCDIC.

When the VTCONTEXT has loaded the <td830 message> it sends Message Objects to the ODTs. The type of Message Object is given by the MSGTYPE attribute. %

These MSGTYPES may be received.

"LOADED" - The <td830 message> was loaded and the Virtual Screen and its attributes are available.

"STARTFORM" - Only sent if the <td830 message> contains a valid formed screen and the ODTs was started by an EVAL.

"FIELD" - If a STARTFORM was received, it is followed by a FIELD message for each field in the form. The index of the first field is 0.

"ENDFORM" - Indicates the end of FIELD messages.

"TRANSMIT" - This message indicates that an ESC ( (Transmit Page) command was embedded in the <td830 message>. The MSGTEXT contains the text that would have been transmitted at that time.

"READSCRATCHPAD" - This message indicates that an ESC RTaaaann (Read ScratchPad Memory) command was embedded in the <td830 message>. The MSGTEXT contains the scratchpad memory read from the configuration.

"WRITESTATUSLINE" - This message indicates that the <td830 message>



wrote to the Status Line, the text of which is in MSGTEXT. The STATUSLINE attribute can be read at any time to retrieve the current Statue Line text.

"VERSION" - This message indicates that an ESC sp V (Transmit Version) command was found in the <td830 message>. The MSGTEXT contains the version information, so that it may be transmitted using the OPAL Transmit Function.

"ASTERISKS" - This message indicates that an ESC RC (Transmit Asterisks) command was found. The MSGTEXT contains the uniquely crafted asterisks.

This is an example of a VTContext ODTs using the MSGTYPE.

```
Define + ODTSequence VTF_HELP(CUSTOM):  
    % Called for each Field in the Instructions Form  
    Case MsgType Of  
    Begin  
    "STARTFORM":  
        $S:=FillIn("0","FILES");  
        Display(SaveAs("FLEXFILESHELP","20"));  
    "FIELD":  
        Display("[",FieldIndex,"] ",LocationOf(FieldStart),"..",  
            LocationOf(FieldEnd)," (" ,FieldWidth,") ",  
            FieldType,," ,Text=",FieldText);  
    "ENDFORM":  
        $FormData:=FormData;  
    Else;;  
    End;  
    \
```

These attributes are defined for the EDIT Interface of the VTContext.

#### ATTLIST

Returns a list of the Attributes in the EDIT Interface.

#### COLUMNOF

`CUMNOF(Index)` returns the `CUMN (1..CUMNS)` of the character at `Index`.

#### `CUMNS`

The number of Columns `1..CUMNS` in the Virtual Screen.

#### `COPY`

The `COPY(START,END)` copies the data beginning at the index given by `START` up to and including the index specified by `END`.

#### `CURSOR`

The index `0..PAGESIZE-1` of the Cursor.

#### `FIELDEND`

The index `0..PAGESIZE-1` of the last data character in the current or specified Field.

#### `FIELDINDEX`

The Index of the Field in the Form `-1..FIELDS-1`.

#### `FIELDS`

The number of Fields `0..FIELDS`.

#### `FIELDSTART`

The index `0..PAGESIZE-1` of the last data character in the current or specified Field.

#### `FIELDTEXT`

The text in the current or specified Field.

#### `FIELDTYPE`

The type of the current or specified Field.

`LEFT` - Left Justified Unprotected,

`RIGHT` - Right Justified Unprotected,

`PROTECTED` - Transmittable Protected

#### `FIELDWIDTH`

The width in characters of the current or specified Field.

#### `FILLIN`

`FILLIN(FIELDINDEX,TEXT)` fills in the specified field with the `TEXT` supplied.

## FIND

The FIND(TARGET,START) Method searches the Virtual Screen, starting at the index given by START for the specified TARGET. It returns the Index 0..PAGESIZE-1 of the TARGET or -1 if Not Found. It searches for an exact match.

## FORM

Returns TRUE if the Virtual Screen is in Forms Mode, otherwise EMPTY.

## FORMDATA

Returns the Form data that would be sent to the Host were the Transmit Key pressed with the Cursor at the Home Position.

## HELP

Returns Attribute Information for the VTContext.

## LOCATIONOF

LOCATION(Index) returns the location of the character at Index as a Co-ordinate (ROW,COLUMN).

## MSGTEXT

The Object Message types TRANSMIT,READSCRATCHPAD,WRITESTATUSLINE, VERSION and ASTERISKS have associated text, which is returned as EBCDIC.

## MSGTYPE

The Type of Object Message is one of LOADED,STARTFORM,FIELD,ENDFORM,TRANSMIT,READSCRATCHPAD, WRITESTATUSLINE,VERSION,ASTERISKS.

## PAGES

The number of Pages configured.

## PAGESIZE

The page size (ROWS x COLUMNS) of the Virtual Screen.

## ROWOF

ROWOF(Index) returns the ROW (1..ROWS) of the

character at Index.

#### ROWS

The number of rows 1..ROWS in the Virtual Screen.

#### SAVEAS

SAVEAS(TITLE). The current page is formatted as HTML and saved in a file with the specified TITLE. If any error occurs, the function returns ERROR:<reason> otherwise EMPTY.

#### SPCFY

Location. SPCFY(INDEX) moves the Cursor to Index, returns the escape sequence for that location. The escape sequence is returned in EBCDIC.

#### STATUSLINE

STATUSLINE returns the text of the Status Line.

The 2nd VTContext Interface is BUILD.

DO <odts>[BUILD IN SL VTCONTEXT]

The BUILD interface may be used to create a Virtual Screen, using Text and TD830 Control Codes.

When the VTCONTEXT has loaded the <td830 message> it sends a LOADED object (MSGTYPE="LOADED").

These attributes are defined,

#### ATTLIST

Returns a list of the Attributes in the BUILD Interface.

#### BLINK

#### BRIGHT

#### CLEAREOL

#### CLEAREOP

Return Control Codes.

#### COLUMNS

The number of Columns 1..COLUMNS in the Virtual Screen.

#### CR

#### DELETE

#### DELETELINE

DOWN

Return Control Codes.

EDIT

Take the TD830 formatted message in P1, translate it to ASCII and send it to the EDIT Interface. Return the Virtual Screen created in EBCDIC.

ENDFIELD or EFIELD

ENDHIGHLIGHT

ENTERFORM

ESC

EXITFORM

FF

Return Control Codes.

GOTO

GOTO(ROW,COLUMN) returns the Control Codes for moving the Cursor to the specified location.

HELP

Returns Attribute Information for the VTContext.

HOME

HOMECLEAR

INSERT

INSERTLINE

LEFT

LF

LEFTFIELD or LFIELD

MOVELINEDOWN

MOVELINEUP

Return Control Codes.

MSGTYPE

The Type of Object Message is LOADED.

PROTECTEDFIELD or PFIELD

REVERSE

RIGHTFIELD or RFIELD

RIGHT

ROLLDOWN

ROLLUP

SECURE

UNDERLINE

UP

Return Control Codes.

WRITE

Returns the text provided in P1.

There is an example of using the EDIT Interface to script \*OBJECT/FLEX in the file \*SUPERVISOR/VTCONTEXT/EXAMPLE, which is available from Metalogic on request.

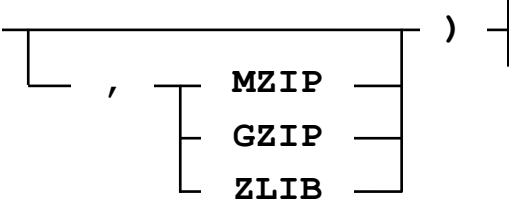
## .WRITE method

string method

[Definition](#)

## ZIP

string function

- ZIP - ( - <string expression>  )

Both ZIP can only be used to compress and decompress a single data stream. For ZIP, either MZIP, GZIP or ZLIB should be specified though MZIP is the default. MZIP is Metalogic's own compression mechanism and is optimised for MCP systems. Files that have been compressed by MZIP cannot currently be unpacked on non-MCP systems.

A ZIP function example:

```
$ZIPSTRING:=Repeat("ABCDEF",20000);      %12,000 chars
If Not $Z:=ZIP($ZIPSTRING,GZIP) Hdis "Err" Then
    SHOW("Compressed string-> ",Length($ZIPSTRING))
Else
    SHOW("Zip Error-> ",$Z);
```

All ZIP and UNZIP errors are returned as the function result and are prefixed by "Err". In SUPERVISOR environments, the attribute MZIPLASTERROR returns a non-zero value if the operations fails.

## **.ZIPFILE method**

string method

[Definition](#)

## OPAL Attributes

The raw material of an OPAL program is the information it has about the machine. To allow easy access to this information, each item is uniquely identified by one or more key words, known as Attributes. Almost every piece of information in the MCP is easily accessible from some kind of OPAL program by using an Attribute. They can be Strings, Reals, Booleans, or Mnemonics, just like WFL File or Task Attributes.

The basic syntax of an OPAL <attribute primary> is shown in the following railroad diagram:



OPAL Attributes are analogous to File and Task Attributes in WFL and will be a familiar concept to anyone with knowledge of that language (the OPAL Attribute for the FILEKIND of a file is FILEKIND), but in OPAL there are Attributes for items of operational data. For example, the Attribute that gives the current amount of save core on the machine is called SAVECORE, or the reason for the last halt-load can be interrogated through HLREASON.

Just as File Attributes describe some characteristic of a file, OPAL Attributes describe some characteristic of a file, task, unit, completed entry, tape label, or the current state of the system. In many cases, the name of an OPAL Attribute is the same as the name of a File or Task Attribute. Task Attributes are a way of manipulating task related values; where an OPAL Attribute has the same name, it returns the same value.

The same rule applies where an OPAL Attribute has the same name as a File Attribute. Other OPAL Attributes have names based on system commands used by operators to set or interrogate system parameters.

All OPAL Attributes are divided into one of a set of 'contexts'. Currently, in SUPERVISOR (Version 530.22), there are the following Attribute contexts:

Context	Description
AFTER	Supervisor schedule information
COMPLETED	Completed tasks or jobs
CUSTOM	Allows access to user defined contexts
DATABASE	LOG database open and close records
DMS	LOG database FREEZE and RESUME
DBACCESS	DMSII access log entries
DEFINE	Defined Opal programs
EI	Establish identity events
FILECLOSE	LOG file close records
FILEOPEN	LOG file open records



FILESTATUS	LOG file status records
HTTP	Requests from web browsers.
JOB	Limited job-tracking using ON JOBMESSAGE
JOBQUEUE (SQ)	Capture of queued WFL job information
JOBREJECT	LOG tracking of rejected jobs
LOG	Generic SUMLOG records
LOGBOJ	LOG beginning-of-job records
LOGEOJ	LOG end-of-job records
LOGHTTP	LOG HTTP request
LOGOFF	LOG session log-on records
LOGON	LOG session log-off records
LOGPS	LOG PrintS event records
MAIL	Metalogic Mail program
MCSSECURITY	LOG MCS security records
METALOG	Metalogic log files
MSG	System & program display messages
MX	Task information
NAPLOG	NAPLOG events from the NAP system
OPERATOR	System commands from the ODT
PD	File information
PER	Peripheral information
PRINTS	Print System request information
SECURITY	System security violations
SESSIONS	COMS SESSIONS
SHOWOPEN	Open Files
SL	SLed Libraries
SMTP	Input from mail programs
SQ	System queue access
STATIONS	Datacom stations
SYSTEM	Global system information
TAPEDB	Tapes logged in the Metalogic TRIM system
TAPELABEL	Tape creation
USER	Usercode attributes from USERDATAFILE
VDBS	Database statistics and audit info
VL	Volume Library directory analyser
VOLUME	Volume status changes of tapes, packs, CDs
WHEN	Supervisor 'when' slots
WINDOWS	COMS Windows open and close actions

In FLEX, all the Attributes are of context SYSTEM or DIRECTORY.

## Accessing OPAL Attribute information

Both SUPERVISOR and FLEX have built-in HELP commands that allow retrieval of information about a specific attribute or the entire dictionary. The dictionary printout, in particular, gives alphabetical lists of Attributes by context. Each Attribute entry includes the value returned, mnemonics if applicable, parameter details and an explanation of the meaning.

In SUPERVISOR, this is done with the "PRINT ATT" command, and in FLEX INQUIRY by using "PRINT HELP ATT". The printouts give alphabetical lists of Attributes by context. Each Attribute entry includes the value returned, mnemonics if applicable, parameter details, and an explanation of the meaning.

PRINT ATTRIBUTES indents each entry by 15 characters if the attribute name length is less than 16 characters. For longer name lengths the entry is indented by one character more than the attribute length.

Both SUPERVISOR and FLEX also support interactive Attribute information through a "HELP ATT" command. Like many SUPERVISOR commands, HELP ATT has full wildcard support for restricting the search on an Attribute name or names.

For example:

```
HELP ATT LOG=
---- HELP ATTRIBUTES ----
LOGBLOCKS          LOG          LOGRECLEN          LOG
LOGDATE            LOG          LOGRESULT          LOG
LOGDAY             LOG          LOGROWSZ           SYSTEM
LOGDUMP            LOG          LOGSELECT          USER
LOGENDDAY          LOG          LOGSEQNO           LOG
LOGENDTIME         LOG          LOGSTARTDAY        LOG
LOGENDTS           LOG          LOGSTARTTIME       LOG
LOGFIELD           LOG          LOGSTARTTS         LOG
LOGFILENO          LOG          LOGSYSNO           LOG
LOGGEDCPUTIME      SYSTEM SS  LOGTEXT            LOG
LOGGEDIOTIME       SYSTEM SS  LOGTEXT            NAPLOG
LOGICALDBNO        DATABASE  LOGTIME            LOG
LOGICALVOLID       PER        LOGTIMESTAMP       LOG
LOGINSTALL         USER        LOGTITLE           LOG
LOGJOBNO           LOG          LOGTS              LOG
LOGMAJOR           LOG          LOGTSDAY           LOG
LOGMINOR           LOG          LOGTYPE            LOG
LOGMIXNO           LOG          LOGVALIDWORDS      LOG
LOGOTHERS          USER        LOGVISIBILITY       LOG
LOGPRINT           LOG          LOGWORD            LOG
LOGRAW             LOG

Window S/1 at POWEREDGE
```

Using a wild-card search, as above, will only return a list of attributes (and their contexts) that match the pattern.

The Help Attributes command when passed a pattern, lists the attributes in two columns. Depending on the length of the longest attribute name to be displayed and the width of the screen, some data may need to be truncated to fit. If either an entry in the first column or the second column would be truncated, only the entry in the first column is shown. The entry which would have been in the second column is moved to the next line.

FLEX Inquiry has similar interactive help for its own attribute subset.

The search can be restricted to a context:

```
HELP ATT LOG=:log
---- HELP ATTRIBUTES ----
LOGBLOCKS      LOG      LOGSYSNO      LOG
LOGDATE        LOG      LOGTEXT       LOG
LOGDAY         LOG      LOGTIME      LOG
LOGDUMP        LOG      LOGTIMESTAMP LOG
LOGENDDAY      LOG      LOGTITLE     LOG
LOGENDTIME     LOG      LOGTS        LOG
LOGENDTS       LOG      LOGTSDAY     LOG
LOGFIELD       LOG      LOGTYPE      LOG
LOGFILENO      LOG      LOGVALIDWORDS LOG
LOGJOBNO       LOG      LOGVISIBILITY LOG
LOGMAJOR       LOG      LOGWORD      LOG
LOGMINOR       LOG
LOGMIXNO       LOG
LOGPRINT       LOG
LOGRAW         LOG
LOGRECLEN      LOG
LOGRESULT      LOG
LOGSEQNO       LOG
LOGSTARTDAY    LOG
LOGSTARTTIME   LOG
LOGSTARTTS     LOG
Window S/1 at POWEREDGE
```

For help on an individual attribute:

```
HELP ATT LOGTEXT
---- HELP ATTRIBUTES ----
LOGTEXT (LOG) Returns STRING
Semantics : Returns the text of the LOGANALYZER output for this
            record with multiple spaces removed.
LOGTEXT (NAPLOG) Returns STRING
Semantics : The analyzed text of the NAP log entry as decoded by the
            NAPLOGANALYZER utility.
```

When an attribute name appears in more than one context, entries for all contexts will be displayed. To list all attributes belonging to a specific context, the context name can be passed as a modifier.

For example, the following command returns all attributes (signified by "=") that belong to the LOG context.

```
HELP ATT =:log
---- HELP ATTRIBUTES ----
JOBNO          LOG      LOGSTARTTIME  LOG
JOBNUMBER      LOG      LOGSTARTTS   LOG
LOGBLOCKS      LOG      LOGSYSNO     LOG
LOGDATE        LOG      LOGTEXT      LOG
LOGDAY         LOG      LOGTIME      LOG
LOGDUMP        LOG      LOGTIMESTAMP LOG
LOGENDDAY      LOG      LOGTITLE     LOG
LOGENDTIME     LOG      LOGTS        LOG
LOGENDTS       LOG      LOGTSDAY     LOG
LOGFIELD       LOG      LOGTYPE      LOG
LOGFILENO      LOG      LOGVALIDWORDS LOG
LOGJOBNO       LOG      LOGVISIBILITY LOG
LOGMAJOR       LOG      LOGWORD      LOG
LOGMINOR       LOG      MIXNO         LOG
LOGMIXNO       LOG      MIXNUMBER     LOG
LOGPRINT       LOG
LOGRAW         LOG
LOGRECLEN      LOG
LOGRESULT      LOG
LOGSEQNO       LOG
LOGSTARTDAY    LOG
Window S/1 at POWEREDGE
```

Where more than one page of attributes is available, the user can scroll forward through the list using the SUPERVISOR **TT NS** command.

# Attribute Types

Attributes may return the following value types:

- INTEGER
- REAL
- MX REFERENCE
- PER REFERENCE
- STRING OF HEXADECIMAL CHARACTERS
- STRING
- BOOLEAN

Attributes that return the BOOLEAN values TRUE or FALSE are known as <boolean attributes>.

Attributes that return numeric values are known as <arithmetic attributes> (or, more specifically, <integer attributes> or <real attributes>, depending on whether their value is always an integer number).

Attributes that return STRING or HEXADECIMAL STRING values are known as <string attributes>. The value of a <string attribute> is always a string of EBCDIC characters. In the case of Attributes that return a HEXADECIMAL STRING, the individual string characters are digits 0 through 9 and letters A through F.

Some Attributes return a mix number or a unit number; these are known as <MX ref attributes> and <PER ref attributes> respectively. Their value is an integer number, and they may be used both as a simple <integer attribute> and as a <reference attribute> as used by the [Via function](#) and [Object Variable Methods](#).

In some cases, the values of an <integer attribute> are restricted to a small set of integers. Often these values are given names or mnemonic values. Where an Attribute has mnemonic values, the mnemonics are shown in the documentation produced by "HELP ATT" together with their associated integer values. The OPAL compiler will allow attributes that normally return mnemonic values to be optionally compared with integer values.

If an Attribute has no defined value, OPAL constructs a default value depending on the type of <attribute>.

These defaults are summarised in the following table:

OPAL type	Default value
BOOLEAN	FALSE
STRING	"" ( EMPTY string)
HEXADECIMAL STRING	"" (EMPTY string)
REAL, INTEGER	-0 (minus zero)

## Attribute Parameters

In some cases, the name of an Attribute is not sufficient on its own to retrieve its value. Such Attributes take one or more parameters. For example, if one needs to know the amount of available space on a disk pack, the necessary Attribute is DU (of context SYSTEM in SUPERVISOR or DIRECTORY in FLEX). However, the name of the family is required.

For DU, this is a string parameter and is of the form

```
DU (<string expression>)
```

For example

```
DU ("PACK")
```

The DU system command allows the specific family index to be specified. The DU Attribute has an optional second parameter that specifies the family index.

For example:

```
DU ON PACK (2)
```

would be written

```
DU ("PACK" , 2)
```

Where an Attribute has parameters, they are described in the documentation.

## Attribute Context

Attributes are said to belong to a Context. SUPERVISOR has many contexts including MX, MSG, COMPLETED, SYSTEM, TAPEDB, TAPELABEL, PER, LOG and the many variants of LOG. The TRIM Tape Library system only has access to the last four contexts – SYSTEM, TAPEDB, TAPELABEL, and PER. FLEX Attributes have two contexts – SYSTEM and DIRECTORY. The report produced by "PRINT ATT" or "PRINT HELP ATT" sorts the Attributes alphabetically within context.

One way of thinking about context is illustrated by the following example. It is meaningful to want to know the serial number of a tape volume. It is not very meaningful to ask the serial number of a program, but the priority of a program is a useful thing to know. SERIAL is a PER Attribute. PRIORITY is a MX Attribute. In this sense, an OPAL context is very similar to a SIM entity class, and an Attribute in OPAL and SIM are also very similar.

As a general rule, Attributes in SUPERVISOR that do not clearly specify some value for one program or one peripheral are in the SYSTEM context. SYSTEM Attributes are the most numerous kind, and include several sub-contexts.

MX Attributes are derived from the MCP by the DCALGOL function Getstatus, using the MIXREQUEST variant.

PER Attributes are derived from the Getstatus UNITREQUEST variant.

COMPLETED Attributes are derived from the MCP's Completed list.

TAPELABEL and TAPEDB Attributes are derived from the MCP TAPELABEL EVENT Entry.

MSG Attributes are derived from the system messages which are returned by the MSG ODT input.

LOG context Attributes, such as LOGMAJOR, LOGMINOR map directly to control information held in individual SUMLOG records. The related contexts, such as LOGEOJ, FILEOPEN, FILECLOSE and SECURITY for example, each have their own subset of attributes that also map directly to various fields in the log record. A list of individual attributes that belong to a context is reported by using the Supervisor HELP command with a context modifier.

Example:

```
HELP ATT =:FILEOPEN" or "HELP ATT =:PER".
```

A list of all current Supervisor contexts can be seen by using the command:

```
HELP CONTEXTS
```

```

TT +
----- CONTEXTS -----
AFTER = SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
        COLDSTART, TODAY
C(COMPLETED)
CUSTOM
DATABASE = OPEN, CLOSE
DBACCESS
DEFINE = SITU, ODTs, DISP, MEMO, COMMAND
DMS = START, STOP
EI
FILECLOSE = CLOSE, PLI
FILEOPEN
FILESTATUS = CREATIONCOPY, CREATION, REMOVAL, TITLE(TITLECHANGE), SECURITY,
            COPY, DESTROY, RELEASE
HTTP = <INTEGER>
JOB
JOBREJECT
SQ(JOBQUEUE) = <INTEGER>
LOG = <INTEGER>, <INTEGER LIST>
LOGBOJ = BOJ, BOT
LOGEOJ = EOJ, EOT
LOGHTTP
LOGOFF
LOGON
LOGPS = CREATED, COMPLETION, STARTED, PRINTED
MAIL
MCSSECURITY
METALOG = SUPERVISOR(SUP), TRIM, MAIL(MAILLIB), SITE, FLEX
MSG(MESSAGE)
MX(MIX) = A(ACTIVE), LIBS(LIBRARIES), S(SCHEDULED), W(WAITING), GOING, PR,
        (PRIORITY), BOT(BOJ), DBS
NAPLOG = <INTEGER>
OPERATOR = SETSTATUS, CONTROLLER, PS, PRIMITIVE
PD
PER = MT, LP(TP), DK, SC(ODT), CD(CDROM), TSP, HY, CR, CP, DC, PK, UD, HC, CDR,
        SPC, ARP, IP, NP, VSID, LBP, VC
PRINTS
SECURITY
SESSIONS
SHOWOPEN = DK, PK, CD
SL
SMTP
STATIONS = NIF, PSEUDO
SYSTEM
TAPEDB = PENDING(BYPENDING), BYNAME(NAME), BYFAMILY(FAMILY), BYLOCATION,
HELP CONTEXTS
----- CONTEXTS -----
        (LOCATION), SCRATCH(BYSCRATCH), BYTS(TS), BYPOOL(POOL), BYVOL
TAPELABEL
U(USER)
VDBS
VL = TAPE, DISK, PACK, CD, CDIMAGE(CDR), FARM(REMOTE)
VOLUME = ONLINE, OFFLINE, TAPEPG, TAPEEXPIRY, TAPENew, TAPEUSE, TAPEHOLD
WHEN
WINDOWS = REMOTE, MCS, DIRECT

```

The Unisys **System Log Programming Reference** manual is also a useful source for providing additional attribute information.

In FLEX, DIRECTORY Attributes come from the Getstatus DIRECTORY call. Some FLEX DIRECTORY Attributes are available in SUPERVISOR as SYSTEM Attributes.

Other SYSTEM Attributes are taken from many sources including other kinds of Getstatus call, Systemstatus, and the METATAPELIB database.

# OPAL Reporting

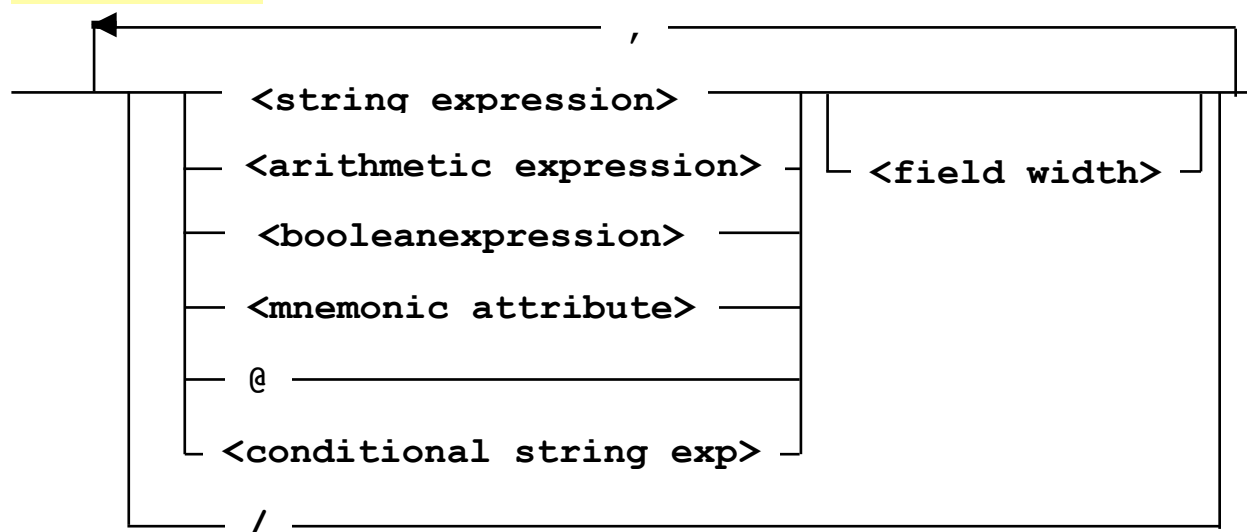
This chapter discusses the various options available in generating reports, to printer or terminal screen, using OPAL functionality. The two main topics covered here are OPAL strings and lookup functions.

## OPAL Strings

The OPAL string is a convenient way to build strings in OPAL for displays and commands. It allows items of many different types to be mixed together with type conversions performed implicitly.

OPAL strings can be used in both Flex and SUPERVISOR. In the Flex environment, simple REPORT statements consist of OPAL strings whereas, in the SUPERVISOR world, they appear in DISPLAY programs or in ODTSequences that use statements such SHOW, ODT, RECORD or DISPLAY.

<OPAL string>



OPAL will automatically 'coerce' non-string expressions into string values during the construction of the OPAL string. Each element in an OPAL string is separated by a single comma `,` or a double-comma `, ,`. A double-comma is permitted – this causes a blank character to appear between the elements.

Example:

`123 , , 456` is equivalent to `123 , " " , 456`

<string expression>

This is the most common component of any OPAL string, allowing string constants, string primaries or variables, string attributes and functions.



Usage in SUPERVISOR:

```
DEFINE + DISPLAY MCP_AND_TIME:
    "MCP is ",/,MCP/,/,"Time: ", Time(TimeOfDay)

ODT("REMOVE ",$Directory," FROM ",DLBACKUP)
```

where DLBACKUP is a SYSTEM attribute representing the MCP setting from the ODT command DL BACKUP.

<arithmetic expression>

OPAL will automatically convert an <arithmetic expression> into a string; when the resulting string is displayed, OPAL always integerises the value.

Example:

```
TT / ("R= ",,1.234*10.4555,, "S= ",#A+.123)
```

gives

```
R=13 S=6
```

To display a real with decimal places, a <field width> modifier must be used with a decimal place indicator. See [<field width>](#) for more information.

<mnemonic attribute>

<mnemonic attributes> translate a mnemonic value, predominantly for an attribute, as a string value.

Example:

```
DEFINE + ODTS REPORT_TAPE(PER):
    If Density NEQ BPI11000 Then
        Display ("TAPE :",Title,, "DENSITY: ",Density)
```

For a tape whose density is not BPI11000

Example:

```
SUPERVISOR: TAPE:METATAPELIBDUMP DENSITY:FMTDLT20
```

The DENSITY attribute actually has an integer value but it is translated into a mnemonic string.

Similarly, consider the following DO file in FLEX:

```
SELECT SecurityType NEQ Controlled
REPORT Title 50,, SecurityType
FILES *HIGHSECURITY/=
```

This might generate a report such as:

<b>*HIGHSECURITY/UNGUARDED/FILEONE</b>	<b>PRIVATE</b>
<b>*HIGHSECURITY/PUBLIC/FILE</b>	<b>PUBLIC</b>

<boolean expression>

SUPERVISOR handles <boolean expression> in various ways. For example, <boolean attributes> such as ConfigSwitched will return the values ON or OFF whereas boolean literals such as TRUE or FALSE return 1 and 0 respectively.

Example:

```
TT / ("Config state: ", ConfigSwitched)
```

returns

```
Config state:  OFF
```

Whereas the FLEX script:

```
REPORT Title 50,,CodeFile  
FILES *SYSTEM
```

might generate a snapshot:

<b>*SYSTEM/DUMPALL</b>	<b>ON</b>
<b>*SYSTEM/TRAINABLES</b>	<b>OFF</b>

@ character

An '@' character may be used in OPAL programs with parameters in SUPERVISOR or TRIM Tape Library system. In ODTSequences and DISPlays of context MX or COMPLETED, it is identical to the attribute MIXNUMBER, and for PER to the attribute UNITNO.

/ character

An OPAL strings is limited to a maximum 1,999,999 characters, regardless of the number of '/' elements. Each time a '/' is encountered, the current line is terminated, and a new line is started. Consecutive '/'s do not need to have spaces between them.

<field width>

— <integer> — . <single digit integer> —

If a <field width> is specified as a single integer value, OPAL will automatically adjust the length of the specified string to that number of characters. If the expression

being operated on is arithmetic, the string will be right-justified with trailing spaces; if the expression is a string, the resultant string is left-justified with leading spaces.

This feature is particularly useful for generating formatted tables with fixed column widths.

For a MX-based OPAL string:

```
DEF ODT5 TEST_PROC (MX) :  
If ExtraEntry Then  
  Show("MIX" 5,, "NAME" 40,, "CPU TIME");  
  Show(MixNo 5,, Name 40,, ProcTime);
```

Result:

TT EV (MX:"SUPERVISOR" IsIn Nam DO TEST_PROC		
MIX	NAME	CPU TIME
61843	*METALOGIC/SUPERVISOR/TTINTERFACE	0
60781	*METALOGIC/SUPERVISOR	350
60790	*METALOGIC/SUPERVISOR/RECORDER	15
57008	*METALOGIC/SUPERVISOR/WAITWATCHER	486
60784	(SUPERVISOR) SUPERVISOR/TAPELIBUPDATER	5
60785	(SUPERVISOR) SUPERVISOR/SILOHANDLER	0
60791	(SUPERVISOR) SUPERVISOR/GRINDER	0
60792	(SUPERVISOR) SUPERVISOR/WINDOWS	0
61025	(SUPERVISOR) SUPERVISOR/MAILHANDLER	2

```
help att proctime:mix
```

```
---- HELP ATTRIBUTES ----
```

```
PROCTIME (MIX) Returns REAL in seconds
```

```
ACCUPTIME is a synonym for this attribute
```

```
ACUMPTIME is a synonym for this attribute
```

```
Semantics : PROCTIME returns the accumulated processor time of the  
task in seconds.
```

```
help att extraentry
```

```
---- HELP ATTRIBUTES ----
```

```
EXTRAENTRY (SYSTEM) Returns BOOLEAN
```

```
LASTEVAL is a synonym for this attribute
```

```
Semantics : This attribute can be used to cause and identify an  
extra entry into and ODT5 or DISPLAY which is linked to a  
SITUATION. an extra entry is made after the last valid entry  
with EXTRAENTRY set to TRUE.
```

```
This attribute will be most useful when linked to a time based  
WHEN or an EVAL. In these cases it can be used to invoke  
totalling code or to determine if any entries were found.
```

```
Ex.
```

```
DEFINE ODT5 PRINTIT(TAPELABEL):
```

```
IF NOT EXTRAENTRY THEN
```

```
PRINT(SERIALNO,,TITLE)
```

```
ELSE
```

```
IF MYSELF(ENTRIES)=0 THEN
```

```
PRINT('NO TAPES FOUND')
```

```
Note that when EXTRAENTRY is TRUE all other attributes other  
than SYSTEM attributes are invalid.
```

PROCTIME returns the number of seconds of CPU used by the program.

If <field width> is represented as a decimal value, e.g. 4.2, then OPAL will depict an <arithmetic expression> to the specified number of decimal places, as specified by the single digit number following '.' i.e. 2.

The above report can be modified to show PROCTIME values to 2 decimal places.

Example:

```
DEF ODTS TEST_PROC(MX): If ExtraEntry Then
    Show("MIX  ", "NAME" 40, "CPU TIME")
Else
    Show(MixNo 5, Name 40, ProcTime 4.2);
```

This would generate a similar report fragment:

TT EV	(MX:"SUPERVISOR" IsIn Nam DO TEST_PROC	
MIX	NAME	CPU TIME
61843	*METALOGIC/SUPERVISOR/TTINTERFACE	0.02
60781	*METALOGIC/SUPERVISOR	350.67
60790	*METALOGIC/SUPERVISOR/RECORDER	15.05
57008	*METALOGIC/SUPERVISOR/WAITWATCHER	486.33
60784	(SUPERVISOR) SUPERVISOR/TAPELIBUPDATER	4.70
60785	(SUPERVISOR) SUPERVISOR/SILOHANDLER	0.00
60791	(SUPERVISOR) SUPERVISOR/GRINDER	0.30
60792	(SUPERVISOR) SUPERVISOR/WINDOWS	0.27
61025	(SUPERVISOR) SUPERVISOR/MAILHANDLER	2.49

Specifying an integer greater than 255 will result in a syntax error; similarly, OPAL will only display arithmetic expressions to a precision of 9 decimal places.

Example:

```
tt / (MCP 256)
* (LINE:1)
Display length should be an integer constant < 256 - 256 (LINE:1)
```

If no <field width> is specified, the <arithmetic expression> is coerced into a string. using:

```
STRING(<arithmetic expression>, * )
```

While evaluation is in process, any limits on line length are ignored. The setting of the terminal attributes (TERM system command) may cause lines to be truncated in the final display.

FLEX Example

```
REP Title 40,,FileKind 12,,Segments 10
```

SUPERVISOR Examples

```
DEFINE ODTS TEST:
    Record [6] ("A TEST OF RECORD ON MACHINE:", //, HostName)
```

To put some summary information on an ET1100 status line:

```
DEFINE + DISPLAY TIME:
```

```
"#[ESC]RS37",  
String(HourOfDay,2) , ":",  
String(MinuteOfDay,2) ,  
" MX=", MX 2, ",JOBS=", JOBS 2,  
",WAIT=", MXWaiting 2, ",SCHED=", MXScheduled 2,  
", DBS=", MXDBS 2, ",ML=", ML 2," "
```

```
tt / time
```

```
13:30 MX=19,JOBS= 0,WAIT= 0,SCHED= 0, DBS= 1,ML= 0
```

## Lookup functions

```
_____ #[ _____ ] _____  
          |  
          | <character lookup> |  
          |  
          | MX = <program name> |  
          | PK = <pack name>   |  
          |_____|
```

A lookup function defines a target for possible text replacement in an OPAL string. When an OPAL display string is sent for output or to be passed to an ODTs statement, the string content is scanned for lookup patterns. If any valid lookups are detected, the string is translated into a mix number list, unit number list or graphic character, depending on the lookup content.

The <character lookup> is available in all forms of OPAL and allows special characters that may not be available graphically on the terminal being used to create the OPAL programs.

In the #[MX variant, the target is the full task name of one or more mix entries and, should any matching tasks be found, the lookup text is replaced by a comma-delimited list of mix numbers. The <program name> may include an 'ON <familyname>'. If this is present, the lookup will find only those programs with a matching 'ON' part. If no 'ON <familyname>' appears, the 'ON' part of the <program name> is ignored.

For the #[PK variant, the target is the name of an on-line disk pack family and the string will be replaced by the corresponding unit numbers. If multiple unit numbers are returned, each unit is delimited by a comma.

It is possible, using the hash-paren function, to retrieve MX or PK information directly into a string variable. For example:

```
$A:= "# [MX=MARC] ")
```

will result in \$A holding the string "# [MX=MARC] "

but using the 'hash-paren' function forces the lookup to be analyzed immediately:

```
$A:=# ("# [MX=MARC] ") )
```

will result in \$A holding, for example, the string "1234,1236" where these are the mix numbers for all MARC tasks.

The #[MX and #[PK variants are only available in licensed SUPERVISOR and TRIM Tape Library systems.

OPAL will replace the <character lookup> by the appropriate character in the EBCDIC character set. There are two categories of characters which may be looked up. A <lower case mnemonic> is used to allow lower case characters from terminals without a lower case keyboard. They have the form 'LOW' + the upper case letter, e.g. #[LOWA] is "a".

The <special character mnemonic>s are derived from the TD830 documents and are the mnemonics for the special ASCII control codes.

The mnemonics are:

```
NUL (or NULL), SOH, STX, ETX, HT, DEL, VT,  
FF, CR, SO, SI, DLE, DC1, DC2, DC3, NL, BS,  
CAN, EM, FS, RS, US, LF, ETB, ESC, ENQ, ACK,  
SYN, BEL (or BELL), EOT, DC4, NAK, and SUB
```

A special Mnemonic NOLOOK is available in Supervisor. This will prevent any subsequent look ups from being evaluated.

## Character Lookup function

The <character lookup function> is most often used to control special features in the ODT consoles.

Here are some common ones:

EXPRESSION	<CTRL>H synonym	Function
#[CAN]	8	Blink
#[SUB]	:	Bright
#[BEL]	'	Buzz buzzer

EXPRESSION	<CTRL>H synonym	Function
#[SO]	.	Reverse Video
#[SI]	/	Underline
#[EM]	9	Secure Video
#[ESC]	;	Escape
#[ESC] L		Insert line
#[ESC] RS		Write to status line
#[ESC] ;		Print

# OPAL Statements

OPAL statements are similar to statements used in Unisys WFL or ALGOL. However, OPAL statements are custom-designed for controlling the system or passing system data to the outside world.

Some OPAL statements are common to both Flex and Supervisor, some are exclusive to Supervisor and some are exclusive to Flex.

An OPAL statement list can appear in a Supervisor ODTSequence or Command and in a Flex report block.

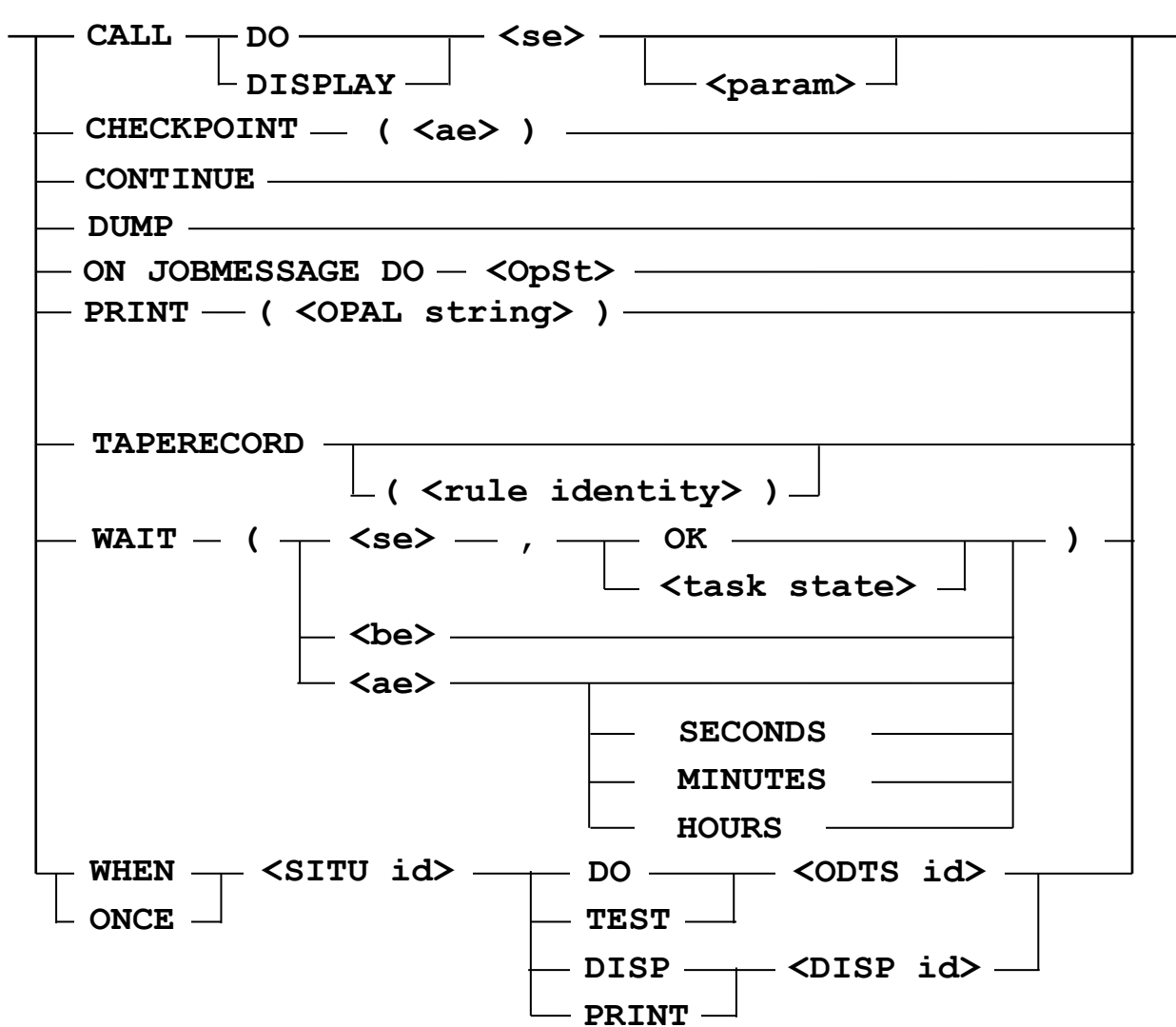
## <OPAL statement List>



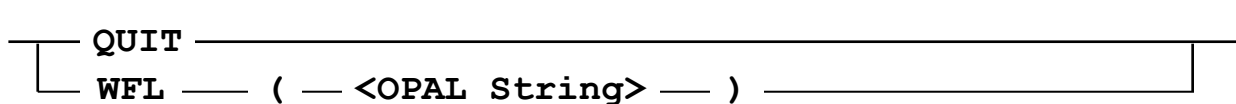
In the following diagrams <string expression> is abbreviated to <se>, <arithmetic expression> is abbreviated to <ae> , <boolean expression> is abbreviated to <be> and <OPAL statement> is abbreviated to <OpSt>



## Supervisor OPAL statements



## FLex OPAL statements



## Common OPAL statements

```
— ABORT — ( <OPAL string> ) —
— BEGIN — <OPAL statement list> — END —
— CASE — ( <ae> ) — OF — <case body> —
    [ ( <se> ) ]
— DISPLAY — ( <OPAL string> ) —
— DO — <ODTS> — UNTIL — <be> —
— EXIT —
— IF — <be> — THEN — <OpSt> —
    [ ELSE — <OpSt> ]
— ODT — ( <OPAL string> ) —
— RECORD [ <integer> ] — ( <OPAL string> ) —
    [ <mnemonic> ]
— SHOW — ( <OPAL string> ) —
— WHILE — <be> — DO — <ODTS> —
```

## Opal statement semantics

### ABORT statement

```
— ABORT — ( <OPAL string> ) —
```

ABORT allows a user to terminate an ODTsequence, whether standalone or linked to a SITUation via a WHEN or EVAL statement. If linked to a WHEN or EVAL, these programs will be deactivated immediately. ABORT expects an OPAL string as parameter that is displayed when the statement is executed.

#### Example

```
DEFINE + SITUATION TEST(MX=WAITING) :
    True
DEFINE + ODTSEQUENCE TEST(MX) :
    If "STOP MONITORING" IsIn RSVP Then
        Abort("WHEN DEACTIVATED BY JOB REQUEST")
    ELSE
        Call Do "HANDLE_WAIT";
```

If the above SITUation and ODTSequence were linked via a WHEN:

```
TT WHEN TEST DO TEST
```

then when the RSVP "STOP MONITORING" appears, the following is seen:

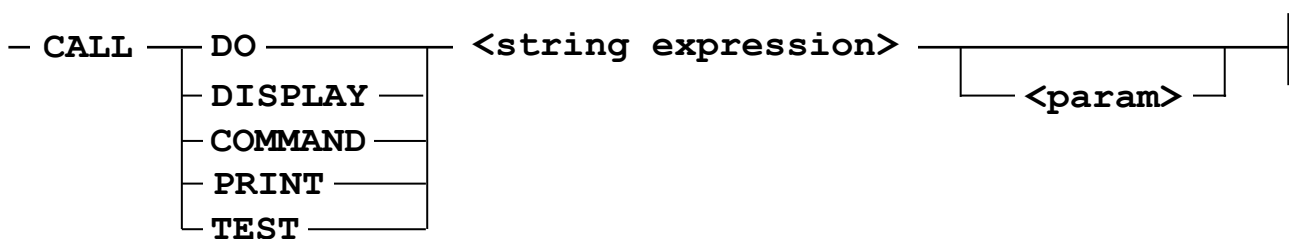
```
SUPERVISOR/TEST+TEST ABORT: WHEN DEACTIVATED BY JOB REQUEST  
SUPERVISOR/TEST+TEST[4293]: ABORT  TERMINATION
```

This mechanism is particularly useful for early termination of an EVAL where the information required has been extracted. In such circumstances, ExtraEntry will be honoured.

If the string passed to ABORT is empty e.g. ABORT(EMPTY) then SUPERVISOR will not emit the usual abort message but allows a WHEN, DO or EVAL to terminate silently and gracefully.

## CALL statement

Supervisor Only



<param> ::= <string expression>

The CALL statement suspends the execution of an ODTSequence and immediately passes control to the ODTSequence or DISPlay whose name is the value of the <string expression>. The <string expression> need not be constant and will be evaluated when the CALL statement is executed. The DO variant is used to invoke an ODTSequence while the DISPlay variant will cause an Opal Display program to be executed.

If the CALLing ODTSequence is running under a locking usercode, SUPERVISOR will look for an ODTSequence or DISPlay defined under that locking usercode. If none is found, it searches for an ODTSequence or DISPlay defined without a usercode. If no candidate is found, the CALLing ODTSequence will be terminated and an error message displayed.

If the target ODTSequence or DISPlay can be located, additional consistency checks are made before the CALLED target is invoked. These ensure that the target ODTSequence or DISPlay is of the same context as the calling ODTSequence or that the target is of the SYSTEM context. If these checks fail, or the candidate is not found, the CALLing ODTSequence will be terminated and the following message displayed:

```
NON EXECUTABLE CALL ON WHEN(ONCE) STATEMENT
```

If the CALLED ODTSequence or DISPlay has a context of MX, PER, MSG or COMPLETED then the current parameter (ie. the information currently in-use by the caller) is passed automatically to the CALLED routine.

The CALLED code runs in the same slot as the calling ODTs so any variables created in the calling ODTs are available in the called code and vice versa.

If the target of a CALL DO is an ODTSequence of type MSG the CALLing ODTSequence is allowed to pass an optional Message Text parameter to the target routine. This optional parameter replaces the Message Text seen by the CALLED ODTSequence with the String specified by the parameter. When control returns to the CALLing ODTSequence the parameter is discarded and the original Message Text is still available to that routine.

## CALL DO parameters

The following examples illustrate the use of the optional parameter:-

```
CALL DO "MYODTS" "Example Message Text"  
CALL DO "MYODTS" $MYPARAM
```

After successful termination of the CALLED ODTSequence or DISPLAY, control returns to the original ODTSequence at the statement after the CALL statement. If the CALLED ODTSequence or DISPLAY fails, then the CALLing ODTSequence will also be terminated along with any attached SITUation.

The COMMAND variant is like the DO variant but will execute a COMMAND type Opal instead of an ODTs.

The TEST variant of the CALL statement is like the SUPERVISOR TEST command whereby all ODT and RECORD statements within the CALLED ODTSequences are converted into displays.

The CALL PRINT variant is an alternative to CALL DISPLAY and will convert all OPAL display strings into print lines that are then queued for output. When the ODTSequence has terminated, SUPERVISOR will automatically print this output.

The DEFINED Attribute can be used to find out whether an OPAL program is CALLable (i.e. it is in the SCHEDULE) before actually attempting to invoke it. For example, the following code will check for the presence of the ODTSequence 'MYOPAL' before invoking the routine. Constructing the call in this way avoids the possibility of abnormally terminating the CALLer should the target not be present for any reason.

```
help att defined
```

```
---- HELP ATTRIBUTES ----
```

```
DEFINED (SYSTEM) Returns BOOLEAN
```

```
Parameters : 1. String
```

```
Semantics : is TRUE if the definition indicated in the STRING
```

```
Parameter is available. If the STRING contains just a name, all
```

```
DEFINES are scanned. If the name is preceded by a type, i.e.
```

```
SITU, DISP, ODTs, or MEMO, only that type will be scanned. E.g.
```

```
IF DEFINED('JAMCONTROL') OR DEFINED('ODTS MT') THEN
```

## Example

```
If Defined("ODTS MYOPAL") Then
    CALL DO "MYOPAL"
Else
    Display ("CAN'T CALL MYOPAL")
```

The CALL statement can be compared with an ODT statement like

```
ODT("TT DO MYODTS")
```

This method of initiating other ODTSequences has two problems. First, if there are insufficient WHEN slots allocated, the command will be rejected by SUPERVISOR. Secondly, it does not invoke the ODTsequence synchronously.

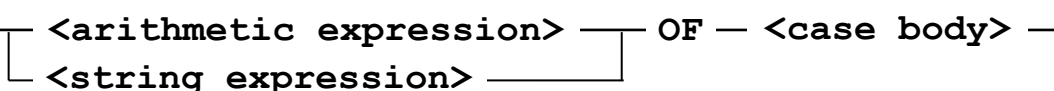
If an ODTSequence executes a CALL DISPlay and it is no longer possible to output on the terminal that originated the ODTSequence, the CALL will be considered a no-op.

## Examples

```
Call Do "EX_SECSCROAK"
Call Display "EX_BARS"
Call Do $DONAME
Call Display $NAME $TEXTPARAM
```

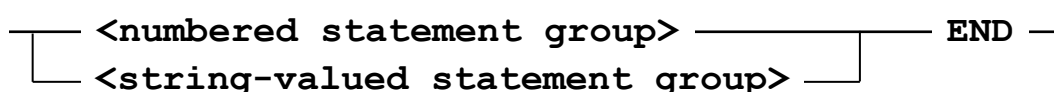
Note: There can be a limit to the calling depth of a series of Call DOs. If ODTS1 executes a Call DO of ODTS2 which in turn executes a Call DO of ODTS3 then the calling depth is 2. If the calling depth is greater than 10 and the ODTS being Called contains a statement which would cause a Wait then the Call DO will cause an abort. Various statements and functions, in addition the the Wait statement, can cause an ODTs to Wait. Ex. TT, COUNT, OBJECTS,ON JOBMESSAGE.

## CASE statement

— CASE —  —

The CASE statement implementation is very similar to that of ALGOL. The CASE <arithmetic expression> or <string expression> will be referred to as the <selection expression>. If an <arithmetic expression> is used it is evaluated to an integer value.

<case body>

— BEGIN —  —

The <case body> always consists of a BEGIN..END block composed of multiple case-action entries. Multiple selectors can 'point' to the same case statement group.

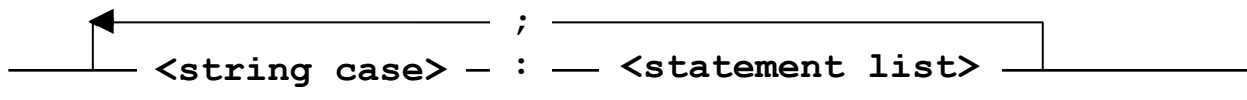
#### <numbered statement group>



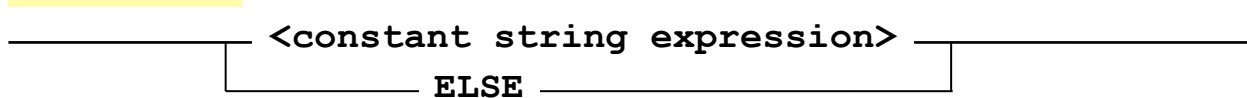
#### <case number>



#### <string valued statement group>



#### <string case>



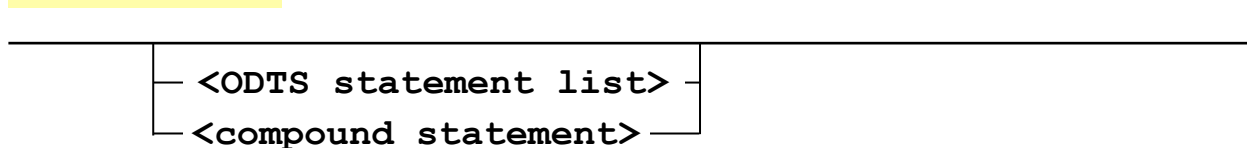
The ELSE case is optional and will be executed if the value determined within the <selection expression> is not equal to any of the selector values. If it is omitted and the value cannot be matched to a selector, a run-time fault will occur. If a <case number> greater than 65535 is used then a syntax error is given.

#### Example

##### BAD INDEX TO CASE EXPRESSION IN ODTs

This fault will always automatically cause an abnormal termination of the offending ODTsequence.

#### <statement list>



As in Algol, the <statement list> does not require a BEGIN..END around it though this may be added to improve clarity. All other OPAL rules for block construction apply.

Note that <statement list> can be empty i.e. no action.

Example of a CASE using <arithmetic expression> selector

```
Case #A OF
Begin
0: 1: Show("Case 0 / 1");
   2: Begin
       $MyStr:="Case 2 in a BEGIN..END";
       Show($MyStr);
   End;
3: #RESULT:=#IO/#TOTALIO;
Else:
   Show("Other values");
End;
```

Example of a CASE using <string expression> selector:

```
Case $A OF
Begin
"ABC": % Statements are free-format
       Show("Case ABC");
       Show("Another statement without BEGIN..END");
"XY": If #A=1 Then
       Show("Case XY with A=1")
     Else
       Show("Case XY");
Else:
   SHOW("Other values");
End;
```

## CHECKPOINT statement

### Supervisor Only

— CHECKPOINT — ( - <arithmetic expression> - ) —————|

Whenever SUPERVISOR restarts any ODTSequences which were running before a QUIT or halt-load are restarted from the first statement. To help control this behaviour the CHECKPOINT statement is used..

The CHECKPOINT statement stores the value of the <integer expression> (valid range 0–4095). This value can be seen in the WHEN ? display and can also be interrogated in OPAL using the MYSELF(CHECKPOINT) attribute. The intention is that CHECKPOINT and MYSELF(CHECKPOINT) can be used to control the behaviour of ODTSequences after restart.

## Example

```
DEFINE + ODTS STARTER:
If Myself(CheckPoint) < 1 Then
Begin
    ODT("START JOB1 ON PACK");
    CheckPoint(1);
End;
If Myself(CheckPoint) < 2 Then
Begin
    Wait("JOB1",MX);
    Wait("JOB1",C);
    CheckPoint(2);
End;
If Myself(CheckPoint) < 3 Then
Begin
    ODT("START JOB2 ON PACK");
    CheckPoint(3);
END;
```

The attribute MYSELF(RESTARTS) returns the number of times that the SITUATION/ODTSequence in this slot has been restarted. The main intended use of this is to prevent two copies of an ODTSequence that is scheduled for AFTER HL being run. This would normally happen if a second haltload occurred while the ODTSequence was running.

## Example

```
DEFINE + ODTS HL:
If Myself(Restarts) > 0 Then
    Display("TERMINATING AFTER RESTART, AF HL WILL HANDLE ")
Else
Begin
    .....
    .....
End;
```

The WHEN ? display (EV ? is a synonym) reports checkpoint and restart information. If a slot has been restarted, then the number of restarts is reported on the Evals/Entries line. If CHECKPOINT has been called for a slot, the checkpoint value is reported on the TIMES line preceded by CP:

```
ev ? bob1
----- SUPERVISOR WHEN STATUS (Limit = 40, Active = 26, Queued = 0) -----
W 177*62375      DO  BOB1      (WAIT TO 15:19.52)      23
CP:100          TIMES:CPU=00:00:00,IO=00:00:00,ET=00:00:04
Text:
    0 evals, 1 ODTS entry, 2 restarts
```



## Compound (BEGIN ... END) statement

— BEGIN ———<ODTS statement list>————— END ———|

This time-hallowed device, handed down from the misty days of ALGOL 60, allows statements to be grouped together. Vile modern neologisms such as { and } in the manner of C are not allowed as synonyms.

And quite right too.

Examples

**Begin**

    ODT("1234 OK");

**End**

**Begin**

    Display("ALL DONE");

    Record("DONE DISPLAY");

**End**

## CONTINUE statement

Supervisor Only

—— CONTINUE —————|

The CONTINUE statement may only be used inside an ON JOBMESSAGE block and when encountered, will force the OPAL machine to exit to the next executable statement outside the block.

Note that ON JOBMESSAGE is only valid following the use of a WFL function to invoke a valid Unisys WFL job.

After executing CONTINUE, any WFL job messages will continue to be queued for the WHEN slot and can be processed in any subsequent ON JOBMESSAGE block. Any unprocessed job messages will be automatically discarded by Supervisor when the calling ODTS terminates.

The use of CONTINUE is optional, if it does not appear inside an ON JOBMESSAGE block, Supervisor will only exit the block once EOJ notices have been received for all outstanding WFL jobs.

In the following example, CONTINUE will cause the block to be exited prematurely upon receipt of a job message.

## Example

```
#J:=WFL (" (META) JOB/BATCH ON DEV")
If #J > 0 Then
  On JobMessage Do
  Begin
    If JobMsgType=WFLMsg And JobText Incl "BATCH OK" Then
      Continue;
  End;
```

Please refer to [WFL](#) function and [ON JOBMESAGE](#) statement for more information.

## DBS Function

The DBS Function may be used as a statement in this case the normal result is discarded.

See [DBS Function](#)

## DISPLAY statement

— DISPLAY — ( — <OPAL string> — ) —————|

When executing a DISPLAY statement, SUPERVISOR evaluates the <OPAL string> and concatenates it to a prefix that identifies the ODTSequence. If the ODTSequence has been invoked because it is linked to a SITUation, the prefix also identifies the SITUation.

If "my text" is the value of the <OPAL string>, then the display will look like

```
SUPERVISOR/<Situation name>+<ODTSequence name>:my text
```

or

```
SUPERVISOR/<ODTSequence name>:my text
```

If the DISPLAY statement appears in an ODTSequence of context MX, the message will appear to originate from the program in question. In other cases, it will appear as a message from SUPERVISOR itself. Messages from MX ODTSequences are not displayed on the task if it is scheduled or was started from a foreign host.

## Example

```
DEFINE + ODTSEQUENCE DSNOFIL (MX) :
  Display("WAITING FOR ",Time(StateTime) ,"- DSED");
  WAIT(1);
  ODT(MixNumber, "DS0");
```

The above ODTs could be linked to a SITUation called NOFILE which sought out waiting entries.

If the following were done from a CANDE terminal:

```
WFL COPY MYPROG FROM NOTAPE TO MYPACK(PACK)
#1234 BOT (META)WFLCODE
#1235 BOT *LIBRARY/MAINTENANCE
#1235 NO FILE NOTAPE(MT)
```

The DISPLAY statement in the ODTSequence might produce a message like:

```
SUPERVISOR/NOFILE+DSNOFILE:WAITING FOR 00:02:15 - DSED
```

On the CANDE terminal, you would see:

```
1235# SUPERVISOR/NOFILE+DSNOFILE:WAITING FOR 00:02:15 - DSED
#1235 O-DS LIBRARY/MAINTENANCE
```

## DO...UNTIL statement

— DO — <ODT statement> — UNTIL — <boolean expression> —

The DO statement is identical to the Unisys WFL and ALGOL DO statement. It repeats the <ODTS statement>, which may of course be a <compound statement>, until the <boolean expression> becomes TRUE. Then it passes on to the next statement. The <ODTS statement> is always executed first before the expression is tested.

The syntax of the DO statement is the same as that for ALGOL, and unlike WFL does not allow a " ; " before the UNTIL.

Infinite loops in ODTSequences are potentially quite dangerous because of the high priority of an MCS such as SUPERVISOR.

A DO loop should normally contain a WAIT statement to reduce the processor rate when many iterations are needed.

If an ODTSequence runs for more than 30 seconds elapsed without interruption, and in that time uses more than 5 seconds of CPU, it will be terminated with a message of the form:

```
PROCESSOR LOOP WITHIN <ODTSequence id>
```

Sometimes, it is possible to get into a loop that SUPERVISOR does not abort. This is usually because of variations in machine kind – the work consumed by one second of a NX6830 processor is very much greater than one second of LX150 processor.

In such cases, SUPERVISOR may appear to stop responding. Input of "HI999" to the SUPERVISOR stack will, as a last resort, abort any ODTSequences that are currently executing.

## Example

Do

```
Wait("#[SUB] NO PACK SEGMENTS LEFT",OK)
```

```
Until DUMax("PACK") > 100;
```

## DUMP statement

Supervisor Only

— DUMP —————

DUMP allows a running OPAL program to perform a primitive 'program dump' of the slot environment.

The dump, which always goes directly to printer backup, consists of the following data about the active SUPERVISOR slot:

- Slot-related information similar to WHEN ?
- OPAL string and real variables, both local and Global
- Run-time stack information
- Wait state details

If the OPAL is waiting on an internal event e.g. time or OPAL functions such as KEYIN or TAPEDB, this information will be displayed. The SLOT command will only give a response when OPAL code is in a WAIT state or not currently executing code.

For the OPAL programmer, this is a simple way to dump OPAL variable information at specific times in the execution of the code.

Consider the following ODTSequence:

```
TT DEFINE + ODTSEQUENCE DUMP_TEST:
  $A:="THIS IS A SIMPLE STRING";
  $MCP:=MCP;
  #R:= 0.56792;
  #TimeOfDay:= TimeOfDay;
  $Time:= Time(TimeOfDay);
  Dump;
```

Executing the above ODTSequence produces the listing shown below:

Entered by : DO DUMP\_TEST

```
---- OPAL DUMP FOR SLOT #23 At 15:37:58 ---
Handled 0 Restarts
Times: CPU= 0:00:00.00, IO= 0:00:00.00, Elapsed= 0:00:00.06
Originated from WINDOW 247 Session 48923

--- OPAL STACK INFO ---
OPAL Version=5353008, MyTI=28

MySITUATIONTYPE = 024000000000    MyAWAITS          = 000000000000
MyWAITCLASS      = 000000000000
```

```
Code syllable 0201, Operator= DUMP
TOPOFSTACK = 11
STAK[11] = 0000000000000
STAK[10] = 0000000000000
STRINGTOS = 0
STR2B is STRINGS[0,*], STR is STRINGS[0,*]
```

```
--- String stack is empty ---
```

```
--- Dump of Global String variables ---
```

```
-- MENU_OPTIONS (160) --
0000000:optemail=bob.x.nicol@metalog.com;OptPermEmail=Y;OptPermW=;OptPermHot=Y;0
0000072:ptFamEmail=;OptFamW=;OptFamHot=Y;OptPermMins=10;OptFamMins=30;OptTimeSto
0000144:red=200734043433
-- MENU_FAM_DATA (189) --
0000000:Fam=CDIMAGE,ChunkSz=30000,Limit=10000,Contig=10000;Fam=DEV,ChunkSz=10000
0000072:,Limit=4000,Contig=3000;Fam=DISK,ChunkSz=20000,Limit=10000,Contig=10000;
0000144:Fam=LOGS,ChunkSz=3000,Limit=4000,Contig=2000;
```

```
--- Dump of Global Real variables ---
```

```
RECSECS 000000000003C Value = 60
REC_MAXHOST 000000000000A Value = 10
```

```
--- HeapSize 5 ---
```

```
--- Dump of local real variables ---
```

```
[002] R 26C8B19A415F Value = 0.56792
[003] TIMEOFDAY 239B7AD94D7E Value = 56278.790709
```

```
--- Maximum OPAL String Length is 2000000---
```

```
--- Dump of local string variables ---
```

```
-- [0] A (23) --
0000000:THIS IS A SIMPLE STRING
-- [1] MCP (31) --
0000000:*SYSTEM/511/DELTA/MCP/511_IC127
-- [4] TIME (8) --
0000000:15:37:59
```

```
-- WHEN STATE INFO --
```

```
WHENSTATEX : 461EE5F8DB00 WHENAWAITS : 0000000000000
WHENEVALSTART : 0000000000001 WHENEVALLIM : 007FFFFFFFFFE
WHENDOSTART : 0000000000001 WHENDOLIM : 007FFFFFFFFFE
WHENMISC : 03C0000000000 WHENMISC2 : 0000000000000
WHENMISC3 : 0000000000000 WHENWAITFLAGX : 0000000000000
WHENWAITFLAG2X: 0000000000000
```

```
-- Array A --
```

```
0000000:000000 000002 000000 020000 000000 014000 000000 001701 000000 220000 000000 000000
0000006:000000 1F0006 000000 000000 000000 000000 000000 000001 000000 000000 000000 000000
0000012:000000 000000 000000 000000 242640 000000 007FFF FFFFFE 000000 000000 000575 B2EA41
0000018:000000 00BF1B 008000 120022 220205 06E2E8 E2E3C5 D403F5 F1F105 C4C5D3 E3C103 D4C3D7
0000024:09F5F1 F16DC9 C3F1F2 F70000 000000 000000 000000 000000 254000 000000 000000 0000F7
0000030:000000 000000 000000 000000 000000 000000 E2E8E2 E3C5D4 0E0101 0AE2E4 D7C5D9 E5C9E2
0000036:09C4E4 D4D76D E3C5E2 E3E131 6D9E30 4253C2 000000 000000 000000 000017 000000 000000
0000042:000000 000000 000000 000000 000000 000000 000000 003E47 000000 000B46 000000 000000
0000048:000000 000000 000000 40004B 000000 20004F 000000 002003 FFFF00 000001 000000 000000
0000054:000000 000000 000000 001689 000000 000000 000000 000000 000000 000000 000000 000000
0000060:000000 000000 2D0304 0AE2E4 D7C5D9 E5C9E2 D6D90B D6C4E3 E2C5D8 E4C5D5 C3C508 C3D6D4
0000066:D7C9D3 C5D909 C4E4D4 D76DE3 C5E2E3 000000 0E0101 0AE2E4 D7C5D9 E5C9E2 D6D900 000000
0000072:0D0101 09C4E4 D4D76D E3C5E2 E30000 000000 180202 09D4C5 E3C1D3 D6C7C9 C30AE2 E4D7C5
0000078:D9E5C9 E2D6D9 0A0101 06F5F3 F04BF2 F30000 000000 000000 000000 000000 000000 000000
0000084:000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
```

An OPAL environment dump can be obtained on an active OPAL using the SUPERVISOR SLOT command.

**TT SLOT 6**

**TT PRINT SLOT 12**

## EXIT statement

— EXIT —

In Supervisor the execution of an ODTSequence normally finishes when the flow of control drops past the last statement. However, because OPAL has no <GO TO statement>, it can be inconvenient to structure the program when a simple test at the top of the program determines that no more need be done. The EXIT statement allows the programmer to finish execution at any statement, just as if that statement were the last one in the ODTSequence.

Example

```
DEFINE + ODTS STOPPER(MX) :  
If MCS Or ACR Then  
    Exit;  
ODT (MixNumber, "ST") ;
```

**Note:** that the EXIT only terminates the ODTS; it does not terminate a WHEN.

In Flex the EXIT statement allows the current REPORT code to be exited early, passing control to the next file.

## IF statement

— IF — <boolean Expression> — THEN — <ODTS statement> —>  
└────────── ELSE ─── <ODTS statement> ───┘

The syntax of the IF statement is the same as that for ALGOL, and unlike WFL does not allow a ";" before the THEN.

The IF statement tests the expression, and if TRUE, executes the first statement, and proceeds to the statement after. If FALSE, and no ELSE part is present, it goes straight to the next statement. If FALSE, and an ELSE part exists, the second statement will be executed.

## Examples

```
DEFINE + ODTS CHECK_MX (MX=BOT) :  
  I UserData (PU) And Name = "SYSTEM/MAKEUSER" Then  
    RECORD[1] ("SECURITY VIOLATION BY ",  
              User, "@", SourceStation) ;  
  
DEFINE + ODTS FAVOURITISM (MX) :  
  If User = "BIGBOSS" Then  
  Begin  
    ODT (MixNumber, "PR 99") ;  
    Display ("Helping hand from OPS") ;  
  END  
ELSE  
  ODT (MixNumber, "DS 0") ;
```

## INPUT Function

The Input Function may be used as a statement. In this case the normal result is discarded.

See [INPUT Function](#)

## KEYIN Function

The KEYIN Function may be used as a statement. In this case the normal result is discarded.

See [KEYIN Function](#)

## LOG statement

Supervisor only

```
— LOG — ( — TP — , <text> — ) —  
           | — MAIL —  
           | — SITE —  
           | — SUP —  
           | — , <category> —
```

The Log statement is used to write entries to the various Metalogic log files.

The MAIL, SUP and TP mnemonics allow write access to the MAILLIB, SUPERVISOR and TRIM logs respectively. <Category> is optional and is constrained to a 3 character string expression. If unprovided, "Msg" is assumed.

SITE is a special log file for customer use and is fully supported from SUPERVISOR using the new SITE LOG command.

The Opal LOG statement no longer includes originating SLOT(n) information in the text of the log entry. To retain the original behaviour, ADD #("Slot(",Myself(slot),":") to the beginning of the text to be logged.

## MAIL Function

The MAIL Function may be used as a statement in this case the normal result is discarded.

See [MAIL Function](#)

## ODT statement

Supervisor only

———— ODT — ( — <OPAL string> — ) —————|

The **ODT** statement allows ODT system commands and WFL statements to be executed. The <OPAL string> is evaluated, and the resulting string is sent to CONTROLLER.

If a string passed to the ODT statement or WFL function is intended to be treated as a WFL command it is always safest if it starts with "BEGIN JOB;" This tells the ODTS statement that this is a WFL Job. Without BEGIN JOB Supervisor has to guess if the command is for CONTROLLER or for a JOB.

If sent to controller the usercode is not passed. Controller recognises some commands as being WFL commands and adds a "BEGIN JOB;" to the front and passes the message to controlcard. The reserved words recognised by controller as of MCP 54.1 are:

RESTORE,ADD,PASSWORD,SECURITY,STARTJOB,EXECUTE,COMPILE,PROCESS,DISPLAY,  
ARCHIVE,CATALOG,REMOVE,CHANGE,UNWRAP,ACCESS,VOLUME,MODIFY,BEGIN,RERUN,  
ABORT,PRINT,ALTER,MKDIR,CLASS,QUEUE,BIND,COPY,USER,END,PTD

Supervisor uses this same list to determine WFL commands.

One special case is recognised and simulated as a primitive run. If the value of the <OPAL string> begins with "??RUN" and is followed by a valid code file title, SUPERVISOR will start the nominated program as if a primitive run had been executed.

Evaluation may involve obtaining values of attributes. When all components of an <OPAL string> have been evaluated, the resulting string is examined for the appearance of <lookup functions>, as described earlier.



If the ODTSequence is executed by a TEST, for example:

```
TT TEST MYODTS
```

or

```
WHEN MYSITU TEST MYODTS
```

the <ODT statement> is converted to a DISPLAY message with a different header attached to the text.

SUPERVISOR commands input with an ODT statement should always be prefixed with a TT, even if the command does not need the TT when used from a COMS window station.

Note that the <ODT Statement> is not the same as the <ODTS Statement>, which is the name for all statements that can appear in an ODTSequence.

### ODT statement command logging

SUPERVISOR in no way verifies that the string obtained as a result of evaluating the associated <OPAL string> is a valid ODT input. The string sent, and CONTROLLER's reply, can be seen by HARDCOPY, use of the MONITORING option, or by logging the session number of the ODTS session.

As <lookup function>s are evaluated only after all component <string expressions> have been evaluated, it is possible to build the text which defines a <lookup function> by concatenation.

A common error centres on the syntax of <conditional string expression>, which coupled with <OPAL strings> may cause unintended results.

For example, the expression

```
If CU > 5000 Then "BAD" Else "GOOD", " CU"
```

will generate either

```
BAD CU
```

or

```
GOOD CU
```

not 'BAD' or 'GOOD CU'.

The example is exactly equivalent to the following expression

```
(If CU > 5000 Then "BAD" Else "GOOD") & "CU"
```

Examples:

```
ODT("MQ 1 ML 10")
ODT("MQ 1 ML ",Min(0,QF(10,ML)-1))
ODT(MixNumber,"DS")
ODT(MixNumber,If Reply(OF) Then "OF" Else "DS:0")
ODT(#[MX=SYSTEM/CANDE],"SM TO ",USER,"NOFILE WAIT TOO LONG")
ODT("#[MX= ",NAME,"] SM SS ALL SHUTTING DOWN NOW")
ODT('AT ELSEWHERE DISPLAY "HELP ... SYSTEM TROUBLE"')
ODT("BEGIN JOB PROD; COPY X TO Y;RUN Z;END JOB")
```

## ON JOBMESAGE DO statement

Supervisor Only

—— ON JOBMESAGE DO ——<ODTS statement>—————|

The ON JOBMESAGE feature allows SUPERVISOR users to track single or multiple WFL activities from a single OPAL program. All job messages, BOJ and EOJ information are returned and made available via a special OPAL context called JOB.

It is not possible to DEFINE an ODTSequence with a context of JOB.

## WFL function

The [WFL](#) function, handles WFL-style commands such as PROCESS, RUN and START to be processed from a SUPERVISOR ODTSequence. Although this activity can be simply achieved using the ODT statement, the main difference is that all job and task messages emitted by the WFL can be captured within the same OPAL program using ON JOBMESAGE.

Both the WFL and ON JOBMESAGE facilities are only available on MCP 48.1 or later.

If one or more WFL jobs have been invoked, the MCP automatically passes all job-related information to the requesting WHEN slot. These notices and messages remain queued in the slot until an ON JOBMESAGE block is entered or the WHEN/DO slot has been deactivated.

It is NOT currently possible to use ON JOBMESAGE inside an ODTSEQUENCE that is linked to an event-driven WHEN. This is because of potential event queuing in the WHEN slot because of a long wait for a job to complete inside an ON JOBMESAGE block.

## JOB context

Once ON JOBMESAGE has been executed, the context of the ODTS will switch to a type of JOB. SUPERVISOR will now mark the DO, as seen in a TT EV ? display,

with a status of '(WAIT WFL)'. When the MCP passes a job message to the WHEN slot, code associated with the ON JOBMESSAGE block will be executed.

A small number of OPAL attributes that belong to the JOB context, are now available for use.

**These attributes include:**

**JOBNUMBER**

**MIXNUMBER**

**JOBMSGTYPE**

**JOBTEXT**

For example:

```
TT DEFINE + ODTSEQUENCE JOB_EX:
  #Job:= WFL("START MYJOB ON DEV");
  Show("Job started ",#Job);
  On JobMessage Do
  Begin
    Show(JobNumber, , JobMsgType, ">", JobText);
  End;
```

The JobMsgType attribute has various integer mnemonic values which reflect the state of the job and are generally self-explanatory: queued (WFLQD), BOJ (WFLBOJ, BOT (WFLBOT), EOJ (WFLEOJ), EOT (WFLEOT), Message (WFLMSG), Going (WFLGOING), Scheduled (WFLSCHEDULED). There is, however, no indicator that a job is currently waiting but these can be detected by examining messages of type WflMsg. The JobText attribute returns the actual text of each message as would be seen normally.

So, output from the above OPAL might be:

```
TT DO JOB_EX
27796 WFLEOJ>P-DS JOB TESTJOB
27796 WFLMSG>DISPLAY:TEST MESSAGE.
27796 WFLBOJ>BOJ JOB TESTJOB
27796 WFLQD>JOB IN QUEUE 0
Job started 27796
```

SHOW will scroll so the above messages appear in reverse order

If an ON JOBMESSAGE block is entered, the block code is executed until all of the WFL activities have terminated i.e. all EOJs for the jobs invoked by this slot ,and any Jobs started by those Jobs, have been received.

The invocation of other jobs using the WFL function inside an ON JOBMESSAGE block is permitted and any job event and messages generated from these jobs will be processed in the same block.

At least one WFL function must have been used prior to any ON JOBMESSAGE block or the block will be exited immediately.

## Using the TEST command

Using the TEST command instead of DO will cause any WFL functions inside the ODTSequence to be executed but for SYNTAX only. To assist testing, SUPERVISOR will generate a dummy 'EOJ' notice so that the script can be tested.

For example:

```
TT TEST JOB_EX

1000000 WFLEOJ
>EOJ START MYJOB ON DEV
Job started 1000000
```

SUPERVISOR supplies an internal job number (starting from 1000000) and increments it for each subsequent WFL job. The JOBTXT attribute holds the original command text passed to that WFL function call e.g. 'START MYJOB ON DEV'.

Whilst processing a WFL function or waiting for job messages to arrive for ON JOBMESSAGE processing, the DO will appear as '(WAIT WFL)' in a TT EV ? response. A TT DO- command will terminate the ODTSequence as normal and discard any pending job messages.

## WFL job identity

Further, a new attribute subset within the **SYSTEM** context supports access to WFL job information for the calling ODTSequence. These attributes all require a previously assigned job identity and allow retrieval of various job info.

The attributes include :

```
WFLELAPSED
WFLJOBNO
WFLTIMESTAMP
WFLEOJREASON
```

For example:

#### DEFINE + ODTSEQUENCE STARTER:

```
#JOBA:= WFL(" (META)J ON DEV", "JOB_A");
#JOB:= WFL(" (META)J1 ON DEV", "JOB_B");
ON JOBMESAGE DO
BEGIN
    ....
    ....
END;
SHOW("JOB_A:#",WFLJOBNO("JOB_A"),
      " ELAPSED=", ,WFLELAPSED("JOB_A"));
```

#### Attribute Descriptions

Attribute	Result
WFLJOBNO("JOB_A")	The real job number of '(META)J ON DEV'
WFLELAPSED("JOB_A")	The elapsed time in seconds of the same job;
WFLTIMESTAMP("JOB_A")	The time that the WFL was executed as a string e.g. "09:12:03,02/08/03".
WFLEOJREASON	a string representing the termination status of the job e.g. GOODEOJ, BADEOJ, SYNTAX, COMPILED, P-DS, F-DS, O-DS etc.

These attributes remain active until the calling ODTSequence has terminated.

## PING Function

The PING Function may be used as a statement in this case the normal result is discarded.

See [PING Function](#)

## PRINT Statement

Supervisor Only

—— PRINT — ( —<OPAL string>— ) —————|

When executing a PRINT statement, SUPERVISOR evaluates the <OPAL string> and queues it internally, in preparation for output to a line printer backup file.

When the ODTSequence has terminated, either because the DO has finished or the associated EVAL or WHEN has also been terminated, Supervisor will process a subtask to handle the print file generation and to create a print request in the PrintS system. This print file will receive a DESTINATION assignment using Supervisor's TT USE DESTINATION setting, if applicable.

If the <OPAL string> evaluates to the string "SPLIT", any previous queued output for this OPAL program will be sent immediately to the printer. Without this "SPLIT", output will only be sent to the printer if the OPAL terminates, as stated earlier, or as a side-effect when the SUPERVISOR Transfer Log (TL) command is entered.

Example

```
PRINT ("[" , SERIALNO , "]" , TITLE 40 , , Dates (EXPIRY , "DDMMYY" ) ) ;  
IF MYSELF (ENTRIES) = 500 THEN  
    PRINT ("SPLIT" ) ;                %RELEASE THE PRINT FILE
```

The example statements might appear in an ODTSequence linked to a TAPELABEL SITUation.

When the WHEN or DO is terminated (or a SPLIT is processed), SUPERVISOR will then generate a print process to direct output to printer backup. A separate process is invoked to handle this and will appear in the completed entries

e.g. the following EOT is the print process for an in-line DO running in WHEN slot 6:

```
---Job--Task--Time--Hist----- 44 COMPLETED ENTRIES -----  
*21321\21322 07:32 EOT (SUPERVISOR) (SUPERVISOR)DO/INLINE_6
```

## PRINT and Virtual Mail

It is now possible to pass a user-supplied PRINTDEFAULTS to a WHEN, DO or EVAL that processes output using the PRINT statement. This is accomplished by passing a string into PRINT using the prefix 'PRDEF=' and can appear at any time in the print file generation.

For example, to force conventional print to Virtual Mail:

```
PRINT ( 'PRDEF=NOTE="To: support; Subject: Test;" , ' ,  
        'DESTINATION="MAIL"' ) ;
```

When executed within an ODTSequence, SUPERVISOR will apply the above print defaults to the LINE file generated by the printing task. Any current PRINTDEFAULTS assigned to the SUPERVISOR station, if applicable, will be overridden.

If TT USE DESTINATION setting is set to 'MAIL' (or starts with 'MAIL'), SUPERVISOR will now check the EMAIL attribute assigned to the log-on usercode if

no PRINTDEFAULTS exist. If a WHEN or DO is executed with a FOR modifier then SUPERVISOR will look for PRINTDEFAULTS or EMAIL settings on the FOR usercode before checking the usercode associated with the logged-on user (if one exists).

This mechanism can be used to direct printing to any desired print device not just Virtual Mail.

## QUIT statement

### Flex Only

— QUIT —————|

The QUIT statement terminates the current scripted function, such as a FILES command, but later activities in the script will be processed normally.

## RECORD statement

— RECORD [ [ <integer> ] - ( <OPAL string> ) ————|  
          [ <mnemonic> ]

The RECORD statement allows textual information, alerts,, etc. to be passed by Supervisor or Flex where it can be written to disk or, to the outside world, via a port file.

The OPAL verb RECORD passes a string from an ODTSequence to a program called RECORDER, generally for the purpose of logging some condition to a disk file or to pass information to a HOTLINE or alert program.

Example

```
RECORD[6] ("This is a test message")
```

By default, RECORDER uses predefined file type mappings for each RECORD index:

```
0 map to header files (HDR)
0,1,2,3,4,5,11 map to disk files (DISK)
6,7,8,9,10,11 map to port files (PORT)
50,51,52,53map to variable length files (VAR)
```

However, it is also possible to configure or override the above settings with user-specified mappings. This is controlled by a Magus configuration variable called REC\_FILEMAP that is maintained by the Supervisor REC NAMES command (please refer to the **Metalogic Supervisor Manual - Control Commands** for more detailed information).



For example, defining the mnemonic TESTMNEM:

```
RECORD[TESTMNEM] ("This is a test message")
```

If a RECORD statement references a Mnemonic which is not defined on that system, a compile time warning Warning is given. At Run Time a reference to an undefined Mnemonic results in termination of the ODTs with the message BAD RECORD MNEMONIC.

The following might be a typical responses from a **TT REC NAMES** command:

```
TT REC NAMES
```

```
--- 0 of 18 Disk Files currently open ---  
RECORD [01] mapped to CRITCL          DISK,PORT  
RECORD [03] mapped to EOJ             DISK,PORT  
RECORD [04] mapped to RSVP            PORT  
RECORD [05] mapped to EOT             PORT  
RECORD [06] mapped to CONFIG          DISK,PORT  
RECORD [13] mapped to HEROIX          PORT=10.0.0.26
```

```
Default DISK files NOT mapped 0,2,11  
Default PORT files NOT mapped 9,10,11,47  
Default VAR  files NOT mapped 50,51,52,53
```

In the response shown earlier, a RECORD[4] statement could now be coded in OPAL as RECORD[RSVP]; its output will be directed through a PORT file instead of a DISK file (which is the normal default). Similarly, RECORD[6] can be written as RECORD[CONFIG] and its output will go to DISK as well as the default PORT file.

Any changes to the Name configuration, using the REC NAMES command, are applied immediately; RECORDER does not require a restart.

## Recording to PORT files

RECORD sub files 6 thru 11, by default, are used to send messages to external TCPIP or BNA clients via the network. The Metalogic HOTLINE program is a customisable program that can receive RECORDER generated messages from multiple Clearpath systems using either protocol. The HOTLINE program is discussed in more detail later in this chapter.

RECORDER now supports a specific TCPIP port file to a nominated IP address. The REC NAME syntax has been extended to allow the specification of an IP address.

Example:

```
TT REC NAME + HEROIX PORT=10.0.0.26
```

A TT REC QUIT should be done after setting or changing any PORT mappings.



Record messages passed to this port have no additional data added to (i.e. Not timestamp or Hostname). The port is closed after each write.

This change was made to enable Supervisor to create HEROIX ROBOEDA events. The layout of the events is controlled by Opal routines which are freely available to anyone interested in passing events to the ROBOEDA application.

The TT REC TCPIP command is used to specify the port number to be used (2919 is the default value for ROBOEDA).

## Recording to disk files

In the default environment, RECORDER creates up to six disk files, 0 through 5, at one time. To RECORD to files 1 through 5, enter the file number in brackets and the text to be logged in parentheses.

For example:

```
Record [3] (TimeDate(YYYYMMDDHHMMSS) , , "User" , , User , ,  
           "program" , , Name , , "security violation (COMPSEC)");
```

RECORDER does not automatically log either the time and date, or the identification of the originating OPAL. If these are required they should be included in the text (as shown in the example) so that you can identify the messages. This is especially important if several of your OPALs perform RECORDs to the same file number.

Another example where record index mapping has been established and RECORD[18] is mapped to DISK with an <id> of ALERT.

Might be:

```
Record [ALERT] ("Alert message text")
```

The file naming convention used when the disk files are created is detailed later in this section. See [RECORD file names](#).

## Recording to disk file 0

Recording to disk file 0 can be accomplished in the way as recording to disk file 1 to 5 by substituting a file index of 0. RECORDER treats file [0] in a special way and adds an additional line, showing the time of day and identifying the originating OPAL, to the output by RECORDER.

The BLOCKSIZE of the file is 450 words and the file is CLOSED after each record entry is written. Furthermore, if the log string begins with the text "QUIT", RECORDER will go to EOT after logging the entry.

## Recording to disk file 11

Recording to file index number eleven (11) is a special case where RECORDER will both send the message text to HOTLINE, using a PORT file, and at the same time log it to a disk file.

The disk file name does not follow the usual convention described later in this section but instead has the general form:

```
*SUPERVISOR/RECORD/<dayname>/HOTLINE
```

This allows a site to keep a record of messages sent to HOTLINE.

### **Recording to disk files 50,51,52,53,54,55 (type VAR)**

RECORDER will now create variable length record disk files when a RECORD with a file index in the range 50 to 55 is performed. Each disk file record will have length equal to the length of the string recorded. These files are intended for transfer to PCs for later processing. The only way to view them on the A-series system is by using the SYSTEM/DUMPALL utility.

Messages written to files of type VAR have a maximum length of 9133 characters; messages longer than this will be truncated.

### **Causing RECORDER to quit**

As well as using the TT REC QUIT command, issuing a RECORD[0] with a message starting with the string "QUIT" will cause RECORDER to terminate. SUPERVISOR will automatically re-initiate RECORDER the next time a RECORD statement is executed. In normal operation, the only reason to do this is to force RECORDER to create new disk files immediately after midnight.

For example, the following ODTSequence and scheduled entry could be used:

```
TT DEF + ODTS QUIT_RECORDER:  
    Record [0] ("QUIT");  
  
TT AF 0000 DAILY: DO QUIT_RECORDER
```

### **RECORD file names for DISK and VAR**

By default, RECORDER files of type DISK are created with FILEKIND of JOBSYMBOL with MAXRECSIZE=15 words. RECORDER files of type VAR are variable length record files with BLOCKSTRUCTURE=VARIABLE and MINRECSIZE=1. VAR files were originally intended for creating files that were to be transferred to PC platforms but have been superseded by the BYTE type.

Both these files are typically called:

```
*SUPERVISOR/RECORD/<dayname>/<filenumber>
```

where <dayname> is the day of the week that the file was created e.g. MONDAY, TUESDAY etc., and <filenumber> is the RECORD file index or message class.

Two examples are shown below:

```
*SUPERVISOR/RECORD/MONDAY/3
*SUPERVISOR/RECORD/SUNDAY/53
```

If a mapped mnemonic configuration item is used, this title format is changed. If, referring to the first example above, message class 3 had been mapped to DISK and an identity of 'SYSMON' then the last two file levels are interchanged.

Example:

```
*SUPERVISOR/RECORD/SYSMON/MONDAY
```

The files are automatically placed on the disk family assigned to DL BACKUP.

New files are *not* automatically created at midnight. Although open, these RECORDER files can be manually copied if desired. If new files are required for each day – which is often a sensible idea – then scheduling a TT REC QUIT command shortly after midnight will cause the files to be CLOSED as RECORDER terminates.

SUPERVISOR will automatically re-initiate RECORDER, thereby creating new files for each day.

All RECORDER disk files have the attributes PROTECTION=SAVE and SYNCHRONIZE=TRUE which means that the files are entered into the disk directory as soon as the file is first opened; also, any logical write to the file is immediately forced as a physical write. This mechanism ensures that the files are very secure, even through halt-loads, and can be copied or accessed, without loss of data, whilst the files are still open.

## RECORD file names for BYTE

RECORD files of type BYTE are Unisys stream files with attributes of FILESTRUCTURE=STREAM, EXTMODE=ASCII and MAXRECSIZE=1. Since stream files on Unisys mainframes running NX/Services are readily accessible from mapped MCP shares, stream files created by RECORDER have a '.TXT' extension.

Example:

```
*SUPERVISOR/RECORD/CRITCL/"MONDAY.TXT"
```

By default, stream files are unavailable in an unmapped REC NAMES configuration so BYTE files can only be created with a mnemonic name and not a number.

It is not possible to view these files, using PC applications like NotePad or Microsoft Word, whilst the files are open on the mainframe.

The file must first be closed using the REC CLOSE command:

```
TT REC CLOSE CRITCL
```

## Examples

```
RECORD[RSVP] (TIME(TIMEOFDAY), MIXNO,/,RSVP);  
RECORD[#A] ("THIS IS A TEST STRING")  
RECORD[12] (TEXT)
```

## RESPOND Function

The RESPOND Function may be used as a statement in this case the normal result is discarded.

See [RESPOND Function](#)

## SHOW statement

—— SHOW — ( —<OPAL string>— ) —————|

The SHOW statement is very similar in concept to a CALL DISPLAY statement in that they both use the <OPAL string> to format output to the originating station. However, Supervisor will only display such queued output to the station when the ODTSequence code has been exited or a WAIT statement has been invoked. In contrast, the SHOW statement will output the data from the provided <OPAL string> immediately onto the screen and is much easier to maintain since a secondary OPAL program is not required.

If an output line is greater than 80 characters then, if TERM TRUNCATE is TRUE, the display line will be truncated. If TERM TRUNCATE is FALSE then the display will be split into multiple lines, as designated by TERM WIDTH.

SHOW is especially useful for debugging purposes allowing variables to have their contents easily displayed without having to use CALL DISPLAYs or, even worse, DISPLAY statements.

For example, an ODTSequence containing the following two lines

```
TT DEFINE + ODTSEQUENCE TEST_SHOW:  
    Show(TIME(TIMEOFDAY),,"This is a test message");  
    Show("MCP is called ",MCP);  
    Show("Really the last line, but appears at the top!");
```

would display the following scrolling effect

```
TT DO TEST_SHOW  
Really the last line, but appears at the top!  
MCP is called *SYSTEM/481/DELTA/MCP  
10:12:04 This is a test message
```

Alternatively, a multi-show SHOW statement such as

```
TT DEFINE + ODTSEQUENCE TEST_SHOW2:  
    SHOW("We shall clear the page here",/,  
        "and follow this with the second line",  
        /,,,% Two CR/LF  
        "and on the fourth output line");
```

would display the following:

```
TT DO TEST_SHOW2  
  
We shall clear the page here  
and follow this with the second line....  
  
and on the fourth output line
```

Prior to output being displayed on a screen by SHOW, Supervisor will search for any lookup functions present in the text. If any valid entries are found, then #{MX= or #[PER= text strings will be replaced by unit or mix number lists.

**See also:**

[CALL](#) and [DISPLAY](#) statements

## **TAPERECORD statement**

**Supervisor Only**

```
— TAPERECORD —————  
    ( <rule identity> )
```

The TAPERECORD statement sends a fixed format message containing all that is known about a specified tape volume to the TAPELIBUPDATER program. This data is then transferred into TRIM's METATAPELIB database. The statement is only valid in TAPELABEL or PER=MT contexts.

By default, if the SUPERVISOR option, TAPELIB, is set, then all tape creation (including tape purges) will be logged to the METATAPELIB database. If it is required that some tapes should NOT be logged on the database, then a TAPELABEL SITUation can be used to filter those unwanted tapes but there must be an associated TAPELABEL ODTSequence which has a TAPERECORD statement to capture all other tape information.

By default, TAPERECORD causes the METATAPELIB4 rules database to be searched to match all retention rules against the tape details. The entry recorded in the TRIM database will use the name of the calling SITUation (if present) and ODTSequence for task and job names respectively.

For example:

```

TT DEFINE + SITUATION TAPERECORD (TAPELABEL) :
    SerialNo HDIS "0" % ONLY CAPTURE NUMERIC SERIALS

TT DEFINE + ODTSEQUENCE TAPERECORD (TAPELABEL) :
    TapeRecord; % CAPTURE EVERYTHING ELSE

```

The TAPERECORD statement will accept an optional string parameter <rule identity> that allows the user to override TRIM rule matching during tape creation.

Typically, TAPERECORD is used in a TAPELABEL WHEN to allow sites to filter tape creation notices from being applied to the database. If the site wishes to change the rule id for a newly created tape so that specific rule processing can be applied, then a TAPELABEL WHEN should be used.

For example:

```

TT DEFINE + ODTSEQUENCE TL_FILTER (TAPELABEL) :
    If Not Scratch And DayInWeek = 4 And
        Title HdIs "(META)DAILYDUMP" Then
    Begin
        $MyRule:="(META)TESTRULE" ;
        TapeRecord ($MyRule) ;
    End;

```

In the above example, any tapes whose title (a composite attribute that uses OWNER, VOLUMEID and FILEID) starts with "(META)DAILYDUMP" and today is a Wednesday, the rule assignment usually determined by TRIM will be overridden by the rule (META)TESTRULE

The rule must be valid; all expiry and generation criteria from the new rule will be applied as if it had just been created in the replaced family. Entries will be written in the TAPELOG to indicate if the rule identity was changed or not.

This feature should be used with some degree of caution. The changing of rule identity should be tested carefully before use in production.

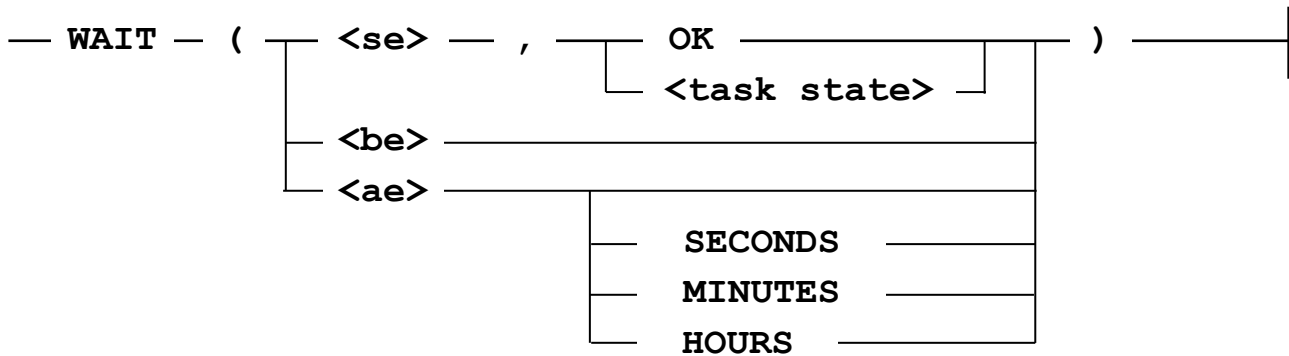
## TT Function

The TT Function may be used as a statement in this case the normal result is discarded.

See [TT Function](#)

## WAIT statement

### Supervisor Only



**<se>::=<string expression>**

**<be>::=<boolean expression>**

**<ae>::=<arithmetic expression>**

**<task state> ::=**

**MX / ACTIVE / SCHEDULED / WAITING / DBS / COMPLETED**

**LIBS / LIBRARIES / NOT ACTIVE / NOT SCHEDULED / NOT WAITING**

**NOT DBS / NOT LIBS / NOT LIBRARIES**

The WAIT statement allows an ODTSequence to suspend execution until specified conditions are met. The conditions may be as follows

### WAIT(<string expression>,OK)

The ODTSequence is suspended until an OK command is entered by the operator. SUPERVISOR constructs a prompt message for the operator. This message is displayed every 60 seconds until the operator responds. It consists of a SUPERVISOR generated prefix, followed by the results of evaluating the **<string expression>**. If 'text' is the string obtained from evaluating the **<string expression>**.

The display looks like

```
SUPERVISOR:ENTER 'TT OK <ODTSequence id>' TO text
```

If an empty string is passed to the Opal WAIT(<text>, OK) statement, then SUPERVISOR will not issue any of the usual "ENTER 'TT OK ....' TO <text>" messages nor will a RSVP task appear if SO WAITOKTASK is set. This special behaviour is intended for internal OPAL scripting purposes only.

### WAIT(<boolean expression>)

The **<boolean expression>** is evaluated. If it is true, execution continues with the next statement; if false, execution is suspended until the SITUation described by the **<boolean expression>** is true. If the context of the ODTSequence performing the WAIT is event-driven (e.g. MX=WAITING, EOJ) then the WAIT will be triggered each time the relevant MCP event is received by SUPERVISOR who will then re-evaluate the **<Boolean expression>**. If the context is not event-driven, then



SUPERVISOR will automatically evaluate the <boolean expression> every 60 seconds.

### **WAIT(<time delay>)**

The ODTSequence is suspended for the specified time period. If no units follow the <real expression>, its value is taken to be the number of seconds to suspend execution.

### **WAIT(<string expression>, <task state>)**

The MX, ACTIVE, SCHEDULED, DBS and LIBRARIES (LIBS) variants cause SUPERVISOR to search all or part of the mix for entries whose name is the value of <string expression>. <string expression> should be a string expression evaluating to a valid value of the task attribute NAME. If found in the appropriate subset of the mix, the wait terminates, otherwise execution of the ODTSequence is suspended until either a new entry enters the mix (MX variant) or an existing entry changes state into the selected subset of the mix. The COMPLETED variant causes SUPERVISOR to search the whole mix for an entry whose name is the value of <string expression>. If not found the WAIT terminates, otherwise execution of the ODTSequence is suspended until the mx entry terminates.

The NOT ACTIVE, NOT SCHEDULED, NOT DBS and NOT LIBS variants cause SUPERVISOR to search all or part of the mix for entries whose name is the value of <string expression>. If not found in the appropriate subset of the mix, the wait terminates otherwise execution of the ODTSequence is suspended until an existing entry changes state out of the selected subset of the mix.

If the <string expression> can be evaluated at compile time, SUPERVISOR will verify that it is a valid program name. No check is possible if it is not a constant string expression, and badly formed strings can lead to WAITs of indefinite duration, in which case the ODTSequence must be terminated manually.

When searching through the mix, SUPERVISOR applies the following algorithm to each NAME (<se> is <string expression> here).

Example:

```
If Decat(<se>," ON ",4) = <se> Then
    Decat(Name," ON ",4)
ELSE
    Name
```

ODTSequences with WAIT statements are permitted to be linked to a SITUation with the WHEN command. It should be noted that when an ODTSequence is in a WAIT state, it will ignore any events that the linked SITUation would normally react to.



## Examples

```
Wait("TERMINATE DATACOM?",OK) ;  
Wait(10) ;  
Wait(2 Hours) ;  
Wait(Count(MX=A:Take(Drop(Name,1),3) = "DCP") > 0 )  
Wait("DCP/0",MX)  
Wait("*SYSTEM/KEYEDIO",Libraries)
```

## WAIT statement

### Flex Only

— WAIT — ( — <arithmetic expression> — ) ————|

The Flex WAIT statement allows users to control PROCESSOR LOOP detection problems inside a WHILE or DO loop for CPU-intensive scripts. The WAIT statement is a simple variant of its SUPERVISOR cousin, only allowing an arithmetic expression parameter to represent the delay time in seconds.

Unlike SUPERVISOR, a WAIT in FLEX causes the script to wait the specified time and reset the CPU loop counters. Note that multiple WAITs in a script will accumulate - therefore extending script execution time - and that queued screen output (e.g. SHOW outputs) is not displayed to the user at that time.

## WFL Function

### Supervisor Only

The WFL Function may be used as a statement in this case the normal result is discarded.

See [WFL Function](#)

## WFL statement

### Flex only

— WFL — ( - <Opal string> — ) ————|

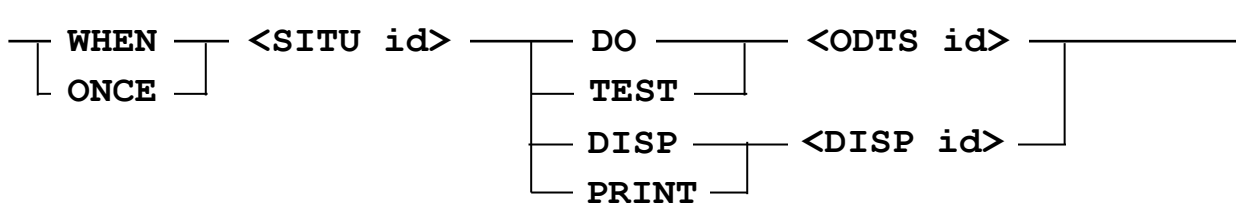
The WFL statement allows WFL-only commands to be locally processed (i.e using the current session attributes such as USERCODE and FAMILY) and, when associated with a REPORT, will act on each file returned by a SELECT. Because of this, the WFL statement should be used with care.

Example:

```
SELECT Days(Today,CreateDay) > 7
REPORT Begin
    WFL("REMOVE "&FileName); % acts on current file
End;
```

## WHEN/ONCE statement

### Supervisor Only



ODTSequences are executed in a slot within SUPERVISOR's WHEN environment (see **SUPERVISOR Reference Manual Activating SITUations in WHEN slots**). If an ODTSequence executes a WHEN or ONCE statement, the effect is to terminate that ODTSequence and execute the statement as if it were the corresponding WHEN or ONCE SUPERVISOR command.

However, unlike the command, the specified SITUation begins monitoring in the slot vacated by the terminating ODTSequence, which ensures that the incoming SITUation cannot be queued waiting for a slot. Unlike the CALL statement, execution does not automatically return to the caller after the new WHEN finishes.

There is no syntax to specify a Supervisor DELAY (as with the Supervisor WHEN command), and so, if the ODTSequence executes from a time-based WHEN, the DELAY for the new slot is taken from that set for the current slot. If there is no current DELAY and the context of the OPAL programs is time-based, the default delay of 60 seconds is supplied.

It is also not possible to start a SITUation under another usercode using the WHEN statement because a FOR clause cannot be supplied.

Note also that if the parent WHEN is invoked with the TEST modifier, the new WHEN will inherit that TEST status regardless of its own TEST or DO qualifications.

The WHEN statement offers the following advantages:

- it is allowed to switch contexts
- it preserves environment variables – any assigned variables can be used in the new WHEN.
- it is cheaper to do than setting up a new WHEN slot.

In cases where the WHEN statement is restrictive, an alternative method is to use the ODT statement to input a WHEN command.

## Example

```
DEFINE + ODTSEQUENCE FA_PER(PER) :  
    When FA_MIX Do FA_MIX
```

## WHILE...DO statement

— WHILE — <boolean expression> — DO — <OPAL statement> —

The WHILE statement is similar to the Unisys ALGOL WHILE statement. It tests the expression, and if TRUE, executes the statement, then repeats the test again. It only passes on to the next statement when the expression becomes FALSE.

Infinite loops in ODTSequences are potentially quite dangerous because of the high priority that an MCS, such as SUPERVISOR, runs at. A WHILE loop should normally contain a WAIT statement to reduce the processor rate when much iteration is needed. If an ODTSequence runs for more than 30 seconds elapsed without interruption, and in that time uses more than 5 seconds of CPU, it will be terminated.

With a message of the form:

```
PROCESSOR LOOP WITHIN <ODTSequence id>
```

## Example

```
While Running("OLAYSCOUT") Do  
Begin  
    Record [1] ("OLAYSCOUT RUNNING");  
    Wait("#[SUB] NO OVERLAY SECTORS",OK);  
End;
```

**See also:**

[DO...UNTIL](#) statement

# Obsolete OPAL variable handling

This section documents the previous OPAL variable handling for both Supervisor and Flex prior to the introduction of direct assignment.

The old OPAL variable name identifiers have similar characteristics to program identifiers. A variable identifier can be constructed from a simple string constant to form a "normal" variable or by the run-time evaluation of a non-constant, string expression. The latter are referred to as "complex" or "dynamic" variables.

For both normal and complex variables, the maximum length of a variable name is 17-characters. For a normal variable identifier, the OPAL compiler will give an error if the string literal exceeds this limit.

If a complex string expression is involved, then the OPAL machine will, if necessary, automatically truncate the variable name to 17 characters at the time the expression is evaluated.

## ACCUM

### arithmetic function

— **ACCUM** — ( — **<string expression>** — , ——————▶  
▶————— **<arithmetic expression>** — ) ——————|

The ACCUM function adds a new arithmetic value from the <arithmetic expression> parameter to the variable named in the <string expression> parameter. It returns the <arithmetic expression> as its result.

A similar function is SUM, which is identical to ACCUM, except that it returns the current total rather than its second parameter.

### SUPERVISOR Example

```
DEFINE + ODTs ACCUM:
    STORE ("TOTL", 17) ;
    STORE ("ADDED", 12) ;
    SHOW ("ADDED = ", ACCUM ("TOTL", GET ("DIFF")) ) ;
    SHOW ("TOTAL = ", GET ("TOTL")) ;

TT DO ACCUM
ADDED = 12
TOTAL = 29
```

### See also:

[SUM](#), [DELTA](#)

# DELTA

## arithmetic function

— DELTA — ( — <string expression> — , —————>  
————— <arithmetic expression> — ) —————|

DELTA is one of the functions used by OPAL to implement arithmetic variables. It stores a new arithmetic value from the <arithmetic expression> parameter into the variable named in the <string expression> parameter. It returns the difference between the new value and the prior value. This is useful when dealing with many of the Attributes in SUPERVISOR and TRIM Tape Library system which are totals since the last haltload. For example, COOLAYS or IOBYTES.

as in:

```
DELTA("WRITES",WRITES) - DELTA("CO",COOLAYS)
```

If no prior value exists, it is assumed to be zero, i.e. the result is equal to the value of the <arithmetic expression>.

A common problem occurs when an Attribute appears in the <arithmetic expression> which is a total since the last Halt Load. The first value will then be enormous. If the value is used in, say, a REPEAT function, this can lead to problems. It is usually safer to initialise the variables with a PUT the first time they are used.

### SUPERVISOR Example

```
DEFINE + SITUATION PROCTIME_CHK(TIME, MX=ACTIVE) :  
  "CPUHOG" IsIn Name And  
  (If Get(#(MixNumber)) = 0 Then  
    Store(#(MixNumber), AccumProcTime) NEQ ""  
  ELSE  
    Delta(#(MixNumber), AccumProcTime) > 10)
```

The previous SITUation checks for all task with "CPUHOG" in the name, looking for any that have incurred more than ten seconds of CPU time since the last time of checking. The ten seconds is an arbitrary limit and depends on the delay parameter assigned to the WHEN used to activate the SITUation.

Note the use of #(MIXNUMBER) which sets up a complex variable name from the task's mix number. The check to see if GET(#(MIXNUMBER)) is zero lets us know if this is the first time we have seen this mix number.

## arithmetic function

GET returns the current real value of the variable named in the <string expression> parameter. If the variable named does not exist, it defaults to -0 (minus zero). The variable name may be a maximum of 17 characters in length.

The PERMANENT modifier allows the retrieval of real-valued data stored **permanently** by the equivalent STORE or PUT functions. This feature enables "global" data to be accessed from any OPAL program at any time, in both the Supervisor and Flex environments. Permanent real variables are preserved across halt-loads and Supervisor or Flex restarts.

GET is now the less-preferred method of retrieving values from OPAL real variables. #A is preferred to GET("A").

Examples:

```
GET ("VAR")
GET ("LONGVARNAME")
GET ("GLOBAL VAR", PERM)
```

## SUPERVISOR Example

```

DEFINE ODTs ACTION(PER):
    If Get("STATE") = 2 Then      % RY unit
        ODT("RYMT", UnitNo)
    Else
        ODT("CLMT", UnitNo)      % CL unit

```

### FLEX Example:

```
SELECT Store("DAYS", Days(Today, AccessdayDay)) > 30
REPORT Title 50, " not accessed for ", Get("DAYS"), " days"
```

The FLEX example checks for all files satisfying the SELECT criterion of not being accessed for more than 30 days.

**See also:**

## GETSTR, PUT, PUTSTR, STORE

# GETSTR

## string function

`-GETSTR - ( - <string expression> , [ PERManent | GLOBAL | CONFIG | FILE ] )`

GETSTR returns the string value held in the current string variable named by the <string expression> parameter. If the variable name specified does not exist, the returned value defaults to a null string (EMPTY).

Note that this behaviour does not apply where the FILE modifier is used.

An OPAL string variable may have a name up to 17 characters in length and may hold a maximum string length of 1,999,999 characters.

The PERMANENT modifier allows the retrieval of string data stored **permanently** by the equivalent STORE or PUTSTR functions. This feature enables the global data to be accessed from any OPAL program at any time, in both the Supervisor and Flex environments. Permanent variables are preserved across halt-loads and Supervisor or Flex restarts.

The CONFIG modifier allows retrieval of string-valued METALOGIC installation and environment information, usually controlled using the INSTALL utility or the Supervisor USE command. See the **Metalogic Install Reference Manual** for more details of configuration variables.

The FILE modifier permits GETSTR to return the contents of any byte stream file as a string, the title of the file is specified in <string expression>. Byte stream files use a carriage return followed by a line feed character between each record. However, the line feed characters are stripped automatically out of the returned string.

If the file is not available or is NOT a byte stream file, the returned string will start with "Error:", followed by a description of the problem. If the file contains more than 1,999,999 characters then only the first 1,999,999 will be returned.

GETSTR is now the less-preferred method of retrieving values from OPAL real variables; using the form \$A instead of GETSTR("A").

Examples:

```
GetStr("A")
GetStr("GLOBAL_VAR", PERM)
GetStr("SUP_USERCODE", CONFIG)
PutStr("FYLE", GetStr("*SUPERVISOR/RECORD/TEST", File))
```

Since the variable name may be an OPAL expression, it is possible to construct a name using attributes or even other OPAL variables:

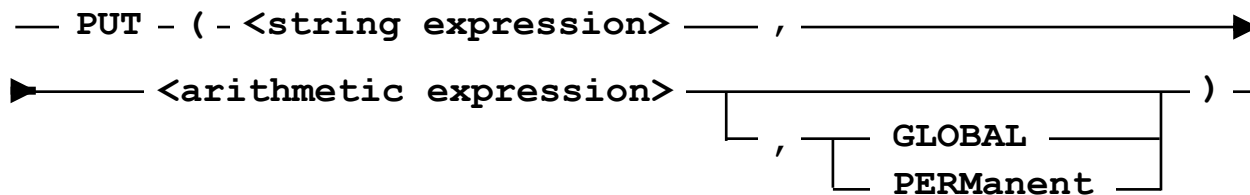
```
GetStr(String(MIXNUMBER,*))
GetStr(#(MIXNUMBER))
GetStr(GetStr("VARNAME"))
GetStr(#(Get("MIXNO")))
```

See also:

[GET](#), [PUT](#), [PUTSTR](#), [STORE](#)

## PUT

arithmetic function



PUT stores the value from the <arithmetic expression> into the variable named in the <string expression>.

It also returns the value of the <arithmetic expression> as the function result.

```
Put("VAR", 456)
Put("VAR", GET("A")*10.45/3.23)
```

The PERMANENT modifier permits the assignment of a decimal value to a variable to be stored permanently. If used, the values will be stored into the MAGUS maintained configuration file:

**METALOGIC/MAGUS/CONFIGURATIONDATA**

Using PERMANENT allows Opal programs to store information into variables which may be used by any other program environment, including both Supervisor and Flex. Permanent variables are retained over Supervisor restarts and halt-loads.

Some examples:

```
Put("MYPERM", 1.34434595, PERM)
Put("YYYDD", Today, PERM)
```

STORE may also be used as an alternative to the PUT function but always returns a null string as the function result.

SUPERVISOR Examples

```
DEFINE + DISP EX_PUT:
    "This PUTs and displays ", PUT("A", 400), /,
    "while GET returns back ", GET("A")
```



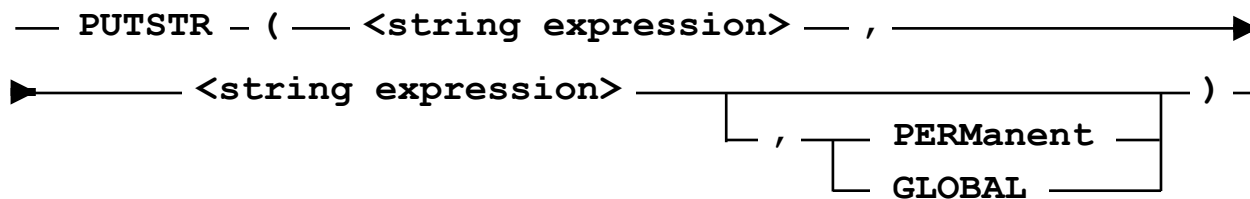
```
SELECT PUT("SEGS", SEGMENTS) GTR 10000
REPORT TITLE 60, GET("SEGS")
```

See also:

[GET](#), [GETSTR](#), [PUTSTR](#), [STORE](#)

## PUTSTR

string function



PUTSTR stores the string value given by the second parameter in the string variable named by the first parameter. The second parameter value is returned as its result.

STORE has a similar effect, but this function always returns a null string

If the second expression, is preceded by the "&" (ampersand) character then the string is appended to the string value held in the variable identified by the first parameter, instead of replacing it.

Consider the following list of OPAL expressions:

PutStr("VAR", "ABC")	returns	"ABC"	VAR="ABC"
PutStr("VAR", &"DEF")	returns	"DEF"	VAR="ABCDEF"
Store ("VAR", &"GHI")	returns	null string	VAR="ABCDEFGHI"

The **PERMANENT** modifier permits the assignment of a string value to a variable to be stored permanently. If used, the values will be stored in the MAGUS maintained configuration file:

**METALOGIC/MAGUS/CONFIGURATIONDATA**

Using the PERMANENT modifier allows Opal programs to store information into variables that may be used by any other program environment, including both Supervisor and Flex. Permanent variables are retained over Supervisor restarts and Halt-loads.

Example

```
PutStr("TEMP", "Temporary string")
PutStr("MYPERM", "This is a perrmanent variable", PERM)
```

To physically delete a permanent string variable, a PUTSTR call storing a null string or EMPTY is required.

## Example

```
PutStr("GLOBAL",EMPTY",PERM)
```

## SUPERVISOR Example

```
DEFINE + DISP EX_PUTSTR:  
    "This PUTSTRs and displays ", PutStr("A", "My string"), /,  
    "while GETSTR returns back ", GetStr("A")
```

## FLEX Example

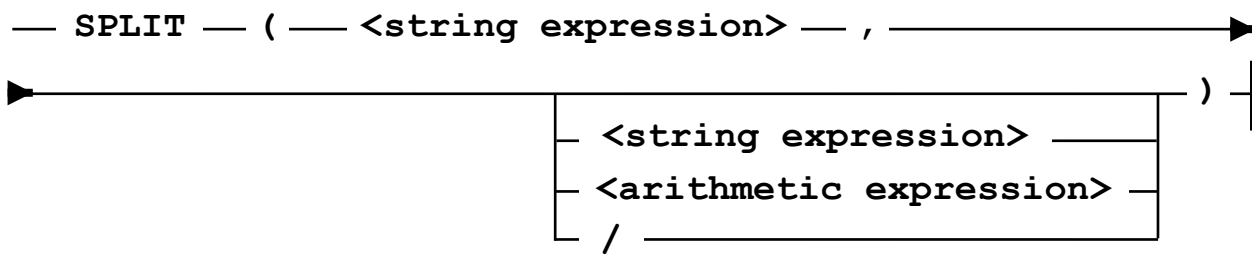
```
SELECT PutStr("TITLE", Title) HdIs "*DOWNLOAD"  
REPORT GetStr("TITLE") 60, Segments
```

## See also:

[GET](#), [GETSTR](#), [PUT](#), [STORE](#)

# SPLIT

## string function



**SPLIT** is a string function that may be used to assist with the simple parsing of "lists" held in OPAL string variables. Such a list might be a series of mix or unit numbers such as

```
"1234,1236,8999,9000"
```

The first **<string expression>** parameter represents a string variable name as used by the **PUTSTR** or **GETSTR** functions. The optional second **<string expression>** parameter is a delimiter or target string, similar to that used by **DECAT**. If the second **<string expression>** is NOT present, then a target of **'** (comma) is assumed.

Semantically, **SPLIT** returns the string of all characters held in the variable represented by the first parameter, up to but not including the first occurrence of the target. The string stored in the variable name is then updated to include all characters following the first occurrence of the target but NOT including the delimiter. Multiple and leading delimiters are skipped over.

Alternatively, if the optional **<arithmetic expression>** parameter is present, **SPLIT** will return the appropriate number of characters from the head of the string held in the variable specified by the first parameter. At the same time, these characters are removed from the string and the resultant string is stored back into the variable.

SPLIT allows the replacement of multiple STORE and DECAT/TAKE/DROP/HEAD/TAKE expressions into one simpler representation.

For example:

```
Store("FIRST", Split("AB"))
```

is equivalent to the following two actions:

```
Store("FIRST", Decat(GetStr("AB"), ",", 4))  
Store("AB", Decat(GetStr("AB"), ",", 1))
```

In the following examples, assume that the OPAL string variable VAR has been set up.

As follows:

```
Store("VAR", "\",1234,,5678,HIT,4567,9123")
```

Now SPLIT the string held in variable "VAR", using comma as the default delimiter.

Example:

```
Split("VAR")                      will return "1234"
```

and the variable "VAR" now holds the truncated string:

```
"5678,HIT,4567,9123"
```

Similarly, using SPLIT again with a target of "HIT".

Gives :

```
Split("VAR","HIT")              will return      "5678,"
```

and the variable "VAR" will now hold

```
",4567,9123"
```

Using an <arithmetic expression> as second parameter, consider the following case.

Example:

```
Store("TEST", "abcdefghijk1") ;
```

Store the first 5 characters of the string in the "TEST" variable into the variable "TOP" and also store the truncated string back into "TEST".

Example:

```
PutStr("FIRST", Split("TEST",5))              will return      "abcde"
```

One of the various equivalents might be

```
PutStr("FIRST", Take(GetStr("TEST"),5))  
Store("TEST", Drop(GetStr("TEST"),5))
```

Also, SPLIT allows the special character, /, as the second parameter. Using a slash is equivalent to using #(/) or #[CR].

Used within a WHILE..DO block, the SPLIT function is ideal for simple manipulation of mix, unit or serial number lists as returned by the OBJECTS function.

For example, the ODTSequence below allows one OBJECTS call to return all waiting entries and SHOW a single line of information for each.

Example:

```
TT DEFINE + ODTs SPLIT_TEST:
  Store("MIX", Objects(MX=WAITING: TRUE));
  While PutStr("ITEM", Split("MIX")) NEQ Empty Do
    Show(Via(GetStr("MIX"):#(MixNumber 4,, Name 40,,
                                TIME(StateTime))));

TT DO SPLIT_TEST

1299 (PROD)OBJECT/WAITER                05:12:00
1305 *SYSTEM/DUMPALL                    01:03:11
1408 *SYSTEM/MYMCS                      04:12:02
```

The following code would process each line of a file:

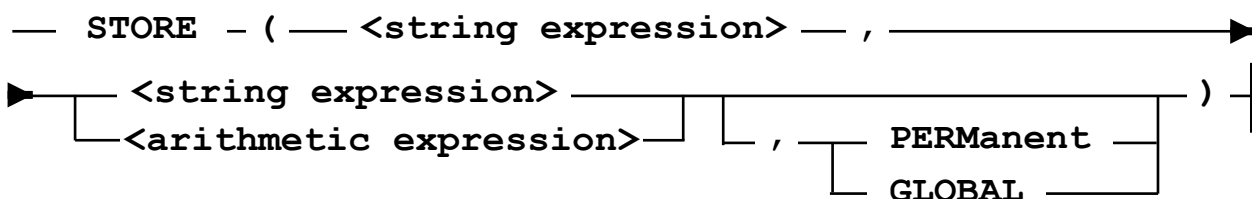
```
Store("FILE",
      GetStr('*SUPERVISOR/RECORD/TEST/"BOB.TXT" On DEV'
            ,File);
While GetStr("FILE") Neq Empty Do
  Show(Split("FILE",/))
```

See also:

[DECAT](#), [HEAD](#), [TAIL](#), [TAKE](#)

## STORE

string function



STORE assigns the type and value given by the second parameter into the OPAL variable named by the first parameter. The second parameter can be either an arithmetic or string value. STORE always returns a null string and can also be used as a statement in an ODTSequence since an ODTSequence will process the STORE as expected but ignore the resulting null string, unlike PUT or PUTSTR where the value of the second parameter is returned.

Note that OPAL keeps string and arithmetic variables separately.

For example, after

```
Store("X",2)
Get("X") gives 2
GetStr("X") gives ""
```

If a subsequent **STORE** is done:

```
Store("X","Y")
Get("X") gives 2
GetStr("X") gives "Y"
```

If the second parameter, is a string expression is preceded by the "&" (ampersand) character then the string is appended to the string value held in variable identified by the first parameter, instead of replacing it.

Consider the following list of OPAL expressions:

<b>PutStr("VAR","ABC")</b>	<b>returns</b>	<b>"ABC"</b>	<b>VAR="ABC"</b>
<b>PutStr("VAR",&amp;"DEF")</b>	<b>returns</b>	<b>"DEF"</b>	<b>VAR="ABCDEF"</b>
<b>Store ("VAR",&amp;"GHI")</b>	<b>returns</b>	<b>null string</b>	<b>VAR="ABCDEFGHI"</b>

The **PERMANENT** modifier permits the assignment of a string value to a variable to be stored permanently. If used, the values will be stored using the **MAGUS** maintained configuration file:

**METALOGIC/MAGUS/CONFIGURATIONDATA**

Using **PERMANENT** allows Opal programs to store information into variables that may be used by any other program environment, including both Supervisor and Flex. Permanent variables are retained over Supervisor restarts and Halt-loads.

To physically delete a permanent string variable, a **STORE** call using a null string or **EMPTY** is required:

```
Store("GLOBAL",Empty,Perm)
```

The following **SUPERVISOR** example illustrates how the variable 'FLAG' can be set to one of two values dependent upon a condition being satisfied. At the same time the **STORE** result is compared with "" (Null string) and will cause the **SITUATION CHECK** to return **TRUE**.

Example

```
DEFINE + SITUATION CHECK(MX=WAITING)
  If "SECTORS REQUIRED" IsIn RSVP Then
    (Store("FLAG", 1) = "")
  Else
    If "ACCEPT:" IsIn RSVP Then
      (STORE("FLAG", 2) EQL "")
    Else
      False
```

The following example assists with the parsing of the TEXT attribute for a system or display message. The two DECAT operations are using " " (space) as a target into the text string and return everything before the space in the case of "X" and only the string following the first space for variable "Y".

Example:

```
DEFINE + ODTs PARSE(MSG) :  
    Store("X", Decat(Text, " ", 4)) ;  
    Store("Y", Decat(Text, " ", 1)) ;  
    Show("X = ", GetStr("X"), " ", "  
        "Y = ", GetStr("Y")) ;  
  
TT DO PARSE TEST MESSAGE FOR STORE FUNCTION  
X = TEST, Y = MESSAGE FOR STORE FUNCTION
```

FLEX Example

```
SELECT Segments > 10000 And Store("TITLE", Title) = ""
```

OPAL's conditional AND means "TITLE" is only set up if SEGMENTS > 10000.

**See also:**

[GET](#), [GETSTR](#), [PUT](#), [PUTSTR](#)

## SUM

arithmetic function

— SUM — ( — <string expression> — , —————>  
▶————— <arithmetic expression> — ) —————|

SUM adds a new arithmetic value from the <arithmetic expression> parameter to the variable named in the <string expression> parameter. It returns the current sum after the <arithmetic expression> has been added.

A similar function is ACCUM, which is identical to SUM, except that it returns its second parameter rather than the current total.

SUPERVISOR Example

```
DEFINE + ODTSEQUENCE DOIT:  
    While Sum("LOOP", 1) NEQ 10 Do  
        ODT('STARTJOB JOB/BATCH(" ', GET("LOOP"), ' ")');
```

FLEX Example

```
SELECT FileKind=DCALGOLCODE AND Sum("SEGS", Segments) GTR -1  
REP FOOT "TOTAL SEGMENTS FOR ALL FILES SCANNED=", Get("SEGS")
```

**See also:**

[ACCUM](#), [GET](#), [GETSTR](#), [PUT](#), [PUTSTR](#)



# Optimisation in the OPAL Compiler

The OPAL Compiler performs three types of optimisation of which users should be aware and which are described here. All expressions involving literals are evaluated at compile time. All Boolean operations are converted to their conditional equivalents. A <boolean expression> is therefore evaluated only partially wherever possible. The operators converted are:

- AND becomes CAND
- OR becomes COR
- IMP becomes CIMP

The GETSTATUS function has an extremely flexible calling sequence and it is possible to gather much related information in a single call. As the major overhead of evaluating SITUations is the number of calls on GETSTATUS, OPAL expends considerable effort on minimising the number of distinct GETSTATUS calls. For this purpose two algorithms are used:

If two adjacent operands are ATTRIBUTES and the two distinct calls on GETSTATUS needed to obtain the values may be combined, they will be combined. For example, consider an expression using the ATTRIBUTES USERCODE, NAME and OP(AUTODC). If the order of the expression is:

```
USERCODE <operator> NAME <operator> OP (AUTODC)
```

then two GETSTATUS calls will be made. The first can get **USERCODE** and **NAME** in the same call, and the second call gets **OP (AUTODC)** . However, if the order of the expression had been:

```
USERCODE <operator> OP (AUTODC) <operator> NAME
```

the three individual calls would be made. It can be seen that it is possible to reduce evaluation overhead simply by changing the order of operands in <boolean expression>s thus optimizing the number of GETSTATUS calls needed.

For some MX ATTRIBUTES there are two ways to get the value from GETSTATUS. If one of the calling sequences would permit minimising the calls on GETSTATUS as above, that alternative is used.



# Date Formats

Many OPAL functions (and attributes) deal with formatting of dates. The two commonly used formats are conventionally referred to as Julian and Gregorian (or sometimes military). It should be noted that these terms are not strictly accurate but they are used in the widely understood Unisys sense.

A Gregorian or military date is (in this loose context) a string or unsigned integer composed of the year of century, the month of year, and the day of month. The order is dependent on national conventions. For example, in Britain the accepted order is DDMMYY while in America it is MMDDYY. In many OPAL functions, the order can be specified by parameter.

A Julian date is a string or an unsigned integer composed of the year of century and day of year. The sequence is from high order to low order (left to right): year of century, day of year as in YYDDD.

For example:

**July 1, 2002** is expressed as **02182**.

These two formats have been extended in OPAL to cater for the millennium change. In the case of Gregorian dates, the two-digit year can be extended to four digits as DDMMYYYY. Julian dates can be similarly extended to seven digits as YYYYDDD. These extended formats allow for the explicit definition of years in either century as, for example, 1995 and 2002.

Metalogic software date routines now handle dates beyond the year 2000 by checking the year in any date parameter. If the 2-digit year is less than 70 then the full date will be considered to be beyond the year 2000; if greater than or equal to 70 then the year will be considered as this century, i.e. 1970–1999.

For example, the Julian dates

99365	will be interpreted as	31st December 1999
01365	will mean	31st December 2001

OPAL date functions that take a Gregorian date parameter and return a Julian date will now handle 4-digit years and return an appropriate result.

For example:

<b>Julian("31/12/1999")</b>	will return	<b>1999365</b>
<b>Julian("31/12/99")</b>	will return	<b>99365</b>

The OPAL TODAY attribute has been extended to allow matching with file dates in the years 2000 to 2035 on MCP releases 4.5 and later. The Julian date returned by the MCP for January 1 2000 will be 100001; this convention will work until 2036. TODAY will now return 100001 for Jan 1 2000 in order to ease transition to four digit years. For example, CREATIONDAY=TODAY is now valid for dates up to 2035365.

For older attributes which return Julian dates with a three digit year there will often be a newer equivalent which will return a four digits year. Typically the new attributes will have the literal TS within the name of attribute.

Example:

```
----- HELP ATTRIBUTES -----
TODAY (SYSTEM) Returns INTEGER as Julian Day -1900 YYYYDDD
Semantics : Today's date as YYYYDDD. Use TSTODAY for YYYYDDD.

HELP ATT =TSDAY
----- HELP ATTRIBUTES -----
ACCESSTSDAY          PD
ALERTSDAY            PD
BACKUPCREATETSDAY    PD
CATALOGTSDAY         PD
CMMCPTSDAY           SYSTEM
CREATETSDAY          PD
CREATETSDAY          VL
CREATIONTSDAY        PER
CREATIONTSDAY        PD
DUPTSDAY             FILESTATUS
ENTERTSDAY           JOBQUEUE
FAMILYTSDAY          PER
LOGTSDAY             LOG
NEWTSDAY             FILESTATUS
NEXTWORKTSDAY        SYSTEM
OLDTSDAY             FILESTATUS
PDACCESSTSDAY        SYSTEM PD
PDALERTSDAY          SYSTEM PD
PDARCTSDAY           SYSTEM PD
PDCREATETSDAY        SYSTEM PD
PDCREATIONTSDAY      SYSTEM PD
PDLASTACCESSTSDAY    SYSTEM PD
PDTSDAY              SYSTEM PD
PDUSETSDAY           SYSTEM PD
STARTTSDAY           JOBQUEUE
TSDAY                PD
TSDAY                EI
USETSDAY             PD
VLTSDAY             SYSTEM VL
VSCREATIONTSDAY      SYSTEM VL
VSTSDAY              SYSTEM VL
```

HELP ATT CMMCPDAY

----- HELP ATTRIBUTES -----

CMMCPDAY (SYSTEM) Returns INTEGER as Julian Day -1900 YYDDDD

Parameters : 1. STRING

Semantics : CMMCPDAY returns the creation date of the MCP codefile  
on the nominated Family. Use CMMCPTSDAY for YYYYYDDD format.

HELP ATT CMMCPTSDAY

----- HELP ATTRIBUTES -----

CMMCPTSDAY (SYSTEM) Returns INTEGER as Julian Day YYYYYDDD

Parameters : 1. STRING

Semantics : CMMCPTSDAY returns the creation date of the MCP  
codefile on the nominated Family.

# Date Arithmetic

Date arithmetic can be fraught with danger. Dates are treated as integer values and so any arithmetic done on them takes no account that they are dates.

It would seem that to get yesterday's date the expression `TSTODAY-1` would work.

This would be the case except on the first day of the year.

```
If TSTODAY=2016001
TSTODAY-1=2016000 an invalid date
```

The `NewDate` function should be used to avoid this problem.

```
NewDate (TSTODAY, -1)=2015365
```

Finding future dates could also cause problems.

```
If TSTODAY=2015365
TSTODAY+7=2015372 an invalid date
NewDate (TSTODAY, 7)=2016007
```

It might seem that subtracting one date from another would give the number of days but this is only true in limited cases.

```
To find the number of days since the MCP was created.
If TSTODAY= 2016001 and CMMCPTSDAY("DISK")=2015272
TSTODAY-CMMCPTSDAY("DISK")=733 clearly the wrong answer
DAYS (CMMCPTSDAY("DISK"), TSTODAY)=98 the correct answer.
```

**It is very strongly recommended that the `Days` and `NewDate` functions are always used for date arithmetic.**

See also:

[Date Formats](#), [NewDate](#), [Days](#)

# Background

Artificial intelligence (AI) is the ability of an artificial mechanism to exhibit intelligent behaviour. The term artificial intelligence was coined in 1956, when a group of interested scientists met for a summer workshop in the United States. Those attending included Allen Newell, Herbert Simon, Marvin Minsky, Oliver Selfridge, and John McCarthy. By the early 1960s, scientists Newell, Simon, and J.C. Shaw were offering their "logical theorist" computer program, and the concept of symbolic processing. Instead of building systems based on numbers, they attempted to build systems that manipulated symbols – a powerful approach that is fundamental to most work in AI where knowledge is expressed as rules; for example, "If x is a dog, then x can bark". If such an AI system determines or is told that a spaniel is a dog, then it can infer that the spaniel can bark.

AI has shown greatest promise in the area of expert systems, or knowledge-based expert programs, which can be extremely powerful when answering questions within a specific domain. Examples of artificially intelligent systems include computer programs that perform medical diagnoses, speech understanding, natural language processing, problem solving, and learning.

An AI system receives input from its environment, determines an action or response, and delivers an output to its environment. A mechanism for interpreting the input is needed so that it is represented in a form that can be manipulated by the machine. To do this, knowledge representation techniques are invoked. The interpretation, together with knowledge obtained previously, is internally manipulated by a mechanism or algorithm to arrive at an internal representation of the appropriate response or action. This requires techniques of expert reasoning, common sense reasoning, problem solving, planning, signal interpretation, and learning.

Finally, the system must construct a response that will be effective in its environment. This requires techniques of natural-language generation.

One of the most useful ideas that emerged from AI research is that facts and rules (declarative knowledge) can be represented separately from decision-making algorithms (procedural knowledge). By adopting a particular procedural element, called an inference engine, development of an AI system is reduced to obtaining and codifying sufficient rules and facts from the problem domain. This codification process is called knowledge engineering.

Following on from the idea of representing knowledge declaratively, logic programming was born, most notably with the computer language PROLOG. PROLOG is actually an inference engine that searches declared facts and rules to confirm or deny a hypothesis. A drawback of PROLOG is that the programmer cannot alter it.

One obvious area where AI could be applied was in the operation of computers themselves. Routine and repetitive tasks could (theoretically) be "programmed" into the machine, reducing the need for human intervention. However, as the size and complexity of mainframe systems grew almost exponentially, their control systems

became equally complex. One manufacturer, Burroughs, provided better controls than most but even so, running an efficient operations room was something most sites could only dream about.

In the late 1970s, METALOGIC brought out the first generation of its own application of artificial intelligence: SUPERVISOR. This powerful knowledge based system was designed to work in the domain of Burroughs Large Systems – now Unisys A Series and Clearpath machines.

Since that first release, the software has been continuously developed and enhanced by METALOGIC's team of experts whose detailed knowledge of A Series architecture has been built into the system. Other intelligent products were subsequently developed by METALOGIC such as the TRIM Tape Library system, FLEX, and JAMPAK. All needed some common method of rule and response definition. However, METALOGIC was careful not to fall into the same trap as the developers of systems such as PROLOG and from the outset provided a powerful, flexible language for programming a site's own individual knowledge base: OPAL.

The best language for solving a given problem is one that is designed specifically with that problem in mind. Such a language is called a Problem Oriented Language (as distinct from a General Purpose Language such as COBOL or FORTRAN).

# Copyright

Copyright © 1979-2020 Metalogic S.à R.L., Luxembourg.  
All rights reserved.

Last revision: June 2020

METALOGIC believes that the information described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material.

The information contained herein is subject to change. Revisions may be used to advise of such changes and/or additions.

## ACKNOWLEDGEMENTS

Many people, over a long period of time, have contributed to this software. To all of our contributors, many thanks.

METALOGIC S.à R.L.  
PO Box 11  
L-7508 Lorentzweiler  
Luxembourg

<http://www.metalogic.eu.com>