



**Language Reference Manual for
ASNA Visual RPG 8.0 for Microsoft Visual Studio .NET 2005**

Contents

Language Reference	1
What's New in the Visual RPG Language?	1
Biggest Changes from AVR Classic 4.0	1
What's Changed in the Visual RPG Language?	2
Collapsing and Hiding Code	2
Built in Functions.....	4
%ABS Function - (Absolute Value)	8
%BOOL Function - (Get Boolean Value)	8
%CHAR function - (Convert to Character Data).....	8
%CHECK Function - (Check Characters)	9
%CHECKR Function - (Check Characters Reverse)	9
%DATE Function - (Convert to Date).....	10
%DAYS Function - (Number of Days).....	11
%DEC Function - (Convert to Packed Decimal Format)	11
%DECH Function - (Convert to Packed Decimal Format with Half-Adjust)	12
%DEC POS Function - (Get Number of Decimal Positions)	13
%DIFF Function (Difference Between Dates, Times, or Timestamps).....	13
%DIGITS Function.....	14
%DIV Function - (Return Integer Portion of Quotient).....	14
%EDITC Function - (Edit Value Using an Edit Code)	15
%EDITFLT Function - (Convert to Float External Representation)	16
%EDITW Function - (Edit Value Using an Edit Word).....	17
%ELEM - (Get Number of Elements).....	17
%EOF Function - (Return End or Beginning of File Condition).....	17
%EQUAL Function - (Return Exact Match Condition).....	18
%ERROR Function - (Return Error Condition).....	19
%FLOAT Function - (Convert to Floating Format)	19
%FOUND - (Return Found Condition)	20
%HOURS Function - (Number of Hours)	21
%INT Function - (Convert to Integer Format).....	21
%INTH Function - (Convert to Integer Format with Half Adjust)	22
%KDS Function (Search Arguments in Data Structure).....	23
%LEN Function - Get or Set Length	23
%LOOKUP Function (Look Up an Array Element).....	24
%MINUTES Function - (Number of Minutes)	25
%MONTHS Function - (Number of Months)	26
%MSECONDS Function - (Number of Microseconds).....	26
%NULLIND Function- (Query or Set Null Indicator).....	27
%OCCUR Function (Set/Get Occurrence of Data Structure).....	27
%OPEN Function- (Return File Open Condition).....	28
%REM Function - (Return Integer Remainder).....	28
%REPLACE Function - (Replace Character String)	29
%SCAN Function - (Scan for Characters).....	30
%SECONDS Function - (Number of Seconds).....	31
%SQRT Function (Square Root of Expression).....	31
%SUBARR Function (Get/Set Portion of an Array).....	31
%SUBDT Function (Extract Portion of Date, Time, or Timestamp).....	32
%SUBST Function - (Get Substring)	33
%TIME Function (Convert to Time)	34
%TIMESTAMP Function (Convert to Timestamp)	34
%TLOOKUP Function (Look Up a Table Element).....	35
%TRIM Function - (Trim Blanks at Edges).....	35
%TRIML Function - (Trim Leading Blanks).....	36
%TRIMR Function - (Trim Trailing Blanks)	36
%XFOOT Function - (Sum Array Expression Elements)	37

%XLATE Function (Translate)	38
%YEARS Function - (Number of Months)	38
Class Library	39
ASNA.VisualRPG.Runtime	40
Database Members	41
Database Methods	42
CreateName Method	42
DeleteName Method	42
GetNames Method	43
UpdateName Method	43
Database Properties	44
CurrentUserLibl Property	44
DBName Property	45
InitialException Property	45
InitialLibl Property	45
IsOpen Property	46
Label Property	46
Password Property	46
PlatformAttribute Property	47
PoolingTimeOut Property	47
Port Property	48
Server Property	48
Text Property	48
User Property	49
DataGateException Class	49
DataGateException Members	50
DataGateException Methods	50
ToString Method	50
DBFile Class	50
DBFile Members	51
DBFile Properties	52
FormatName Property	52
RecCount Property	52
RecNum Property	52
DBFile Fields	53
BlockingFactor Field	53
FilePath Field	54
IsFormatID Field	54
MemberName Field	54
QryFileName Field	55
QryKeyFlds Field	55
QrySelect Field	56
ShareType Field	57
WaitRec Field	58
MemoryFile Class	58
MemoryFile Members	59
MemoryFile Properties	59
DataSet Property	59
SflRRN Property	59
MemoryFile Fields	60
sflRRN Field	60

MessageFileReader Class	60
MessageFileReader Members.....	62
MessageFileReader Methods.....	63
GetMessageText Method.....	63
GetSecondMessageText Method	64
PrintFile Class	65
PrintFile Members.....	66
PrintFile Properties	68
Collate Property	68
Copies Property	68
CursorY Property	68
Duplex Property	69
FooterSize Property.....	69
FormName Property	70
InOverflow Property	70
IsOpen Property.....	70
LastException Property.....	71
ManuscriptPath Property	71
Orientation Property.....	71
PaperLength Property.....	72
PaperSize Property.....	72
PaperSource Property	73
PaperWidth Property	73
Preview Property	74
PrintableLength Property	74
PrintableWidth Property	75
Printer Property.....	75
PrintSetup Property	75
ProcessMacro Property	76
Quality Property	76
ReportName Property.....	77
Scale Property	77
TTOption Property	78
UseColor Property	78
YResolution Property.....	78
PrintFile Fields	79
DclPrintFileName Field	79
FilePath Field.....	79
IsFormatID Field	80
IsOpen Field.....	80
LastException Field.....	81
ASNA.VisualRPG.Windows.Controls Namespace	81
GridArgs Class.....	82
GridArgs Members	83
GridArgs.GridArgs Constructors	83
GridArgs.GridArgs Constructor (integer, integer).....	83
GridArgs Fields	84
GridArgs.CurrRow Field.....	84
GridArgs.PriorRow Field	84
RecordDeleteArgs Class	84
RecordDeleteArgs Members.....	85

RecordDeleteArgs.RecordDeleteArgs Constructors	85
RecordDeleteArgs.RecordDeleteArgs Constructor (integer, boolean)	85
RecordDeleteArgs Fields	86
RecordDeleteArgs.Cancel Field	86
RecordDeleteArgs.Row Field	86
RecordNewArgs Class	86
RecordNewArgs Members	87
RecordNewArgs.RecordNewArgs Constructors	87
RecordNewArgs.RecordNewArgs Constructor (integer)	87
RecordNewArgs Fields	88
RecordNewArgs.Row Field	88
SubfileControl Class	88
SubfileControl Members	89
SubfileControl Events	93
Bottom Event	93
DoubleClick Event	93
KeyDown Event	94
KeyPress Event	95
KeyUp Event	95
RecordDelete Event	96
RecordNew Event	96
RecordSwitch Event	97
SelectionChanged Event	97
Top Event	98
SubfileControl Methods	98
Clear Method	98
Select Method	98
SubfileControl Properties	99
AllowSort Property	99
Browse Property	99
CellBackColor Property	100
CellBrowse Property	100
CellForeColor Property	101
CellValue Property	101
Changed Property	102
Column Property	102
ColumnCount Property	103
FieldBackColor Property	103
FieldBrowse Property	104
FieldColumnHeading Property	104
FieldColumnWidth Property	105
FieldForeColor Property	105
FieldHidden Property	106
Fields Property	106
FirstRow Property	107
FormatName Property	108
GridLineColor Property	108
KeyboardDeleteInsert Property	108
RecordBackColor Property	109
RecordBrowse Property	109
RecordChanged Property	110
RecordForeColor Property	110
Row Property	111
RowCount Property	111
RowSelected Property	112
RRNName Property	112

SingleSelect Property	113
UpperCaseOnly Property	113
WholeRows Property	113
Constants.....	114
Hexadecimal Numeric Constant	114
Data Types	115
*Binary Data Type	115
*Boolean Data Type	115
*Byte Data Type	116
*Char Data Type	116
*Date Data Type	116
*Decimal Data Type	117
*Float Data Type	117
*Float4 Data Type	118
*Float8 Data Type	118
*Form Data Type	118
*Ind Data Type	119
*Integer Data Type	119
*Integer2 Data Type	119
*Integer4 Data Type	120
*Integer8 Data Type	120
*IntPtr Data Type	121
*OneChar Data Type	121
*Packed Data Type	121
*String Data Type	121
*Time Data Type	122
*Timestamp Data Type	122
*Zoned Data Type	123
Date and Time.....	124
Adding or Subtracting Dates	124
Calculating Durations between Dates	124
*Date Data Type	125
Date Formats	125
Date Operations	126
Default Values for all Time Formats	127
Default Values for all Date Formats	127
Duration Codes	128
Moving Date-Time Data	128
*Time Data Type	129
Time Formats	130
*Timestamp Data Type	131
Figurative Constants	131
*ALL'x.'	131
*BLANK/*BLANKS	132
*BASE	132
*ByRef Operator	132
*Date	133
*Day	133
*FALSE	133
*HIVAL	134
*LOVAL	134
*Month	135
*NOTHING	135
*ON	135
*This	135
*Year	136
*TRUE	136
*ZERO/*ZEROS	136

*OFF	137
Language Elements	137
Remarks	137
Continuation Lines	138
Control Language	138
Declarations	138
General Subroutines	139
Multiple Occurrence Data Structures	139
Names	139
Printing	140
Reserved Names	141
Simple Types	141
Subroutine Parameters	142
Type Conversions	142
Literals	142
Boolean Literal	142
Character Literal	142
Date Literal	143
Hexadecimal Literal	143
Numeric Literal	144
OneChar Literal	144
Time Literal	144
Timestamp Literal	145
Unicode Literal	145
Methods	146
CreateName Method	146
DeleteName Method	146
GetNames Method	147
UpdateName Method	147
Operation Codes	148
ADD	148
ADD DUR - ADD a Duration	149
ADDHANDLER	151
BEEP	152
BEGCLASS	153
BEGCONSTRUCTOR	155
BEGENUM	156
BEGFUNC	158
BEGGE	160
BEGININTERFACE	161
BEGPROP	163
BEGSE	166
BEGSR	167
BITOFF	169
BITON	170
CAB	171
CALL	171
CASE	173
CASEDFT	174
CATCH	174
CHAIN	175
CHECK	177
CHECKR	179
CLEAR	181
CLOSE	182
COMMIT	183
COMPARE	184
CONCAT	185

CONNECT	186
DCLALIAS	187
DCLALIASGROUP	189
DCLARRAY	190
DCLCONST	194
DCLDB	195
DCLDELEGATE	197
DCLDISKFILE	198
DCLDS	206
DCLDSFLD	210
DCLENUMFLD	212
DCLEVENT	212
DCLFILECONT	214
DCLFLD	220
DCLKFLD	224
DLKLIST	225
DCLMEMORYFILE	225
DCLNAMESPACE	229
DCOVERLAYGROUP	229
DCLPARAM	230
DCLPLIST	233
DCLPRINTFILE	233
DCLSRPARAM	237
DCLSUBFILE	239
DCLSUBFILEFLD	241
DCLWORKSTNFILE	243
DELETE	245
DELETERANGE	246
DISCONNECT	248
DIV	249
DO	250
DOEVENTS	252
DOUNTIL	252
DOWHILE	253
DSDUMP	254
DSLOAD	255
ELSE	256
ELSEIF	256
END	257
ENDCLASS	257
ENDCONSTRUCTOR	257
ENDCS	258
ENDDO	258
ENDENUM	259
ENDFOR	259
ENDFUNC	260
ENDGET	260
ENDIF	261
ENDINTERFACE	261
ENDPROP	262
ENDSET	262
ENDSL	263
ENDSR	263
ENDTPM	264
ENDTRY	264
ENDWITH	265
ENTERLOCK	265
EVAL	266
EXFMT	267
EXITAPP	268
EXITLOCK	268
EXSR	268

EXTRACT	269
FEOD	270
FINALLY	271
FOR	272
FOREACH	273
GOTO	274
HIDE	275
IF	275
ITERATE	276
LEAVE	276
LEAVESR	276
LOADPICTURE	277
LOOKUP	278
MOVE	279
MOVEARR	285
MOVEL	287
MOVEREM	291
MSGBOX	292
MULT	294
OCCUR	295
OPEN	297
OSEXEC	299
OTHER	302
READ	303
READC	305
READE	306
READP	308
READPE	309
READRAND	311
READRANGE	313
REMOVEHANDLER	315
RNMFLDS	315
ROLLBACK	316
SCAN	317
SELECT	319
SET	319
SETASSEMBLYATTRIBUTE	320
SETGT	320
SETLL	323
SETMOUSEPTR	325
SETOFF	326
SETON	327
SETRANGE	328
SHOW	332
SLEEP	333
SORTARR	333
SQRT	334
STARTTPM	335
SUB	336
SUBDUR	337
SUBSTR	340
SUMARR	342
TAG	343
TESTBITS	344
TESTNUM	345
TESTTIME	346
THROW	347
TIME	347
TRY	349
UPDATE	350
UDAY	350
UMONTH	350

UNLOAD	351
UNLOCK	351
UPDATE	352
USING	352
UYEAR	353
WHEN	353
WITH	354
WRITE	355
XFOOT	356
XLATE	357
ZADD	358
ZSUB	360
Operators	361
+ Operator	361
* Operator	361
/ Operator	361
** Operator	361
>> Operator	362
*As Operator	363
*ASFLD Operator	363
*BitAnd Operator	364
*BitNot Operator	365
*BitOr Operator	366
*BitXOr Operator	367
*ByRef Operator	368
*Is Operator	369
*ISFLD Operator	369
*New Operator	370
*NOT	370
*TypeOf Operator	370
Properties	371
Database Properties	371
DBFile Properties	371
MemoryFile Properties	372
PrintFile Properties	372
Special Values	374
*END	374
*PropVal	374
*START	374
Syntactical Elements	375
Commands	375
Expressions	376
Assignments	376
Expression Operators and their Precedence	376
Operands	377
Expression Overview with QrySelect	377
Simple Types	379
Type Conversions	379
Variables	380
Array Element	380
Character Expression	380
Constant Expression	380
Data Structure Element	381
Field	381
Global Variables	381
Indicator	382
Indicator Variable	382

Integer Literal.....	382
Labels	383
Language Elements	383
Lexical Elements.....	383
Local Variables	383
Numeric Expression.....	384
Numeric Variable	384
Record Format.....	384
Rules for User Date	384
Subroutine	385
User Date Special Words.....	385
Variable Operand.....	385

Language Reference

What's New in the Visual RPG Language?

Because the Visual RPG language is integrated with the Visual Studio interface and .NET framework, it has acquired by default, many new and improved language features — such as inheritance, interfaces, and overloading — that make it a powerful object-oriented programming language. As a Visual RPG developer, you can now create multithreaded, scalable applications using explicit multithreading.

What's New in Visual RPG for .NET 8.0?

Partial Classes

Partial Classes are now supported, making it possible to split the definition of a class over two or more source files. Each source file contains a section of the class definition, and all Parts are combined when the application is compiled.

Using partial classes allows you to indicate that other Parts of the class be defined within the namespace. All the Parts must use the proper keyword. All of the Parts must be available at compile time to form the final type. All the Parts must have the same accessibility, such as public, private, and so on.

All of the Parts that specify a base class must agree, but Parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all of the interfaces listed by all of the partial declarations. Any class or interface members declared in a partial definition are available to all of the other Parts. The final type is the combination of all the Parts at compile time.

Partial classes are defined by setting the BEGCLASS Partial parameter to *YES. See BEGCLASS for more information.

Biggest Changes from AVR Classic 4.0

The following is a listing of the biggest changes from AVR Classic.

- Everything is an object.
- There are no VRF, VRP, or VRC members.
- All source members are VR's.
- No call operation for forms or programs.
- No Main C-Specs. In Visual RPG.Net, there are no procedural programs or forms, since there are only "classes".
- Each source member is a class.
- No More STARTUP Programs – Allows Static or Shared files in a class. Any class can have global variables. Startup serves as entry point and global variables. Define which class is the entry point. Shared = class members only.
- Forms must be explicitly instantiated to use them.
- .NET exception error handling.
- .NET system namespaces provide lots of power.
- LR means nothing, and conditional indicators no longer exist.
- Visual RPG converts existing conditional indicators to an "IF" statement.

- Rules of scope change dramatically. Scope(*Public|*Private) is replaced by: Access (*Public | *Private | *Protected | *Internal)
- All arrays (and data structures) are zero-based - even RPG's arrays.
- IDE is all new - new editor and debugger. For example, can now collapse and hide code.
- Browser-based development is much, much, easier.

What's Changed in the Visual RPG Language?

The following is a listing of the changes that were made in the Visual RPG language for the current releases.

Operation Codes

Added:

- BEEP
- SET

Changed:

- DCLDISKFILE - New ServerCursor parameter.
- BEGCLASS - New Partial class parameter.

Operators

- Added:
- *AsFld
- *IsFld

Collapsing and Hiding Code

The Visual RPG code editor has the ability to collapse and hide code regions. These regions follow the general trend of making it easier to read your code, thus serving the function of reducing clutter. This will allow you to see, or to hide the sections of code that take place "behind the scenes". For Example, the code to create and draw forms is tucked away in collapsed regions.

Developers who want or need more complete access to the underlying code have it; all others can simply let VS.NET and Visual RPG draw the windows for them.

The /Region directive enables you to collapse and hide sections of code in Visual RPG for .NET files. /Region lets you specify a block of code that you can expand or collapse when using the /Region and /EndRegion commands. The ability to hide code selectively makes your files more manageable and easier to read.

/Region directives are not supported within functions.

Syntax

The syntax for /Region is as follows:

```
/Region "identifier string" //  
  code region  
/EndRegion
```

The identifier string requires the quotes and acts as the title of the region when it is collapsed. The region is collapsed by default.

To collapse and hide a section of code

Place the section of code between the `/Region` and `/EndRegion` statements. The following Example shows a section of system-generated code that the editor has added the collapse/hide ability for you. `/region` Windows Form Designer generated code

// Required method for Designer support - do not modify // the contents of this method with the code editor.

```
BegSr InitializeComponent
//
// Form1
// *this.AutoScaleBaseSize = *new System.Drawing.Size(5, 13)
*this.ClientSize = *new System.Drawing.Size(292, 266)
*this.Name = "Form1"
*this.Text = "Form1"
EndSr
/endregion
```

Multiple /Region Blocks

The `/Region` and `/EndRegion` block can be used multiple times in a code file; thus, users can define their own blocks of procedures and classes that can, in turn, be collapsed.

`/Region` blocks also can be nested within other `/Region` blocks.

Built in Functions

Visual RPG built-in functions are similar to RPG operation codes in that they provide an action that returns a result. However, built-in functions can be used in expressions). (eg, `x = %XFOOT( MyArray )`). Built-in functions always start with a percent sign (%).

Usage

```
function-name( (argument{:argument...}) )
-or -
function-name( (argument{,argument...}) )
```

As defined by IBM, built-in functions use a colon (:) to delimit arguments being passed to a function. For Example, the formal IBM syntax for using the %EDITC function is:

```
x = %EDITC( NumVal : '3' )
```

where the colon delimits the functions arguments.

Visual RPG offers an optional, more intuitive syntax, that uses the comma (,) to separate function arguments. For Example,

```
x = %EDITC( NumVal, '3' )
```

is also valid with Visual RPG code.

You can use whichever syntax you prefer.

Arguments for the function may be variables, constants, expressions, a prototyped procedure, or other built-in functions. An expression argument can include a built-in function.

See the individual built-in function descriptions for details on what arguments are allowed.

Built-In Function	Argument(s)	Value Returned
%ABS	numeric expression	Absolute value of the numeric expression.
%BOOL	boolean expression	True or False value.
%CHAR	numeric, date, time or timestamp expression	Value in character format.
%CHECK	comparator, base {,start}	The first position of <i>base</i> (<i>start</i> , then left to right) that contains a character that does NOT appear in <i>comparator</i> ; otherwise 0. <i>start</i> default is 1.
%CHECKR	comparator, base {,start}	The last position of <i>base</i> (<i>start</i> , then right to left) that contains a character that does NOT appear in <i>comparator</i> ; otherwise 0. <i>start</i> default is the length of the string.
%DATE	(character, numeric, or timestamp expression {,*ISO *JUL *USA *EUR *JIS *MDY *DMY *YMD})	Value of expression converted to type date or the current system date in *ISO date format.

%DAYS	number	The result is a date or timestamp value with the appropriate <i>number</i> of days added or subtracted.
%DEC	numeric expression { ,digits ,decpos }	Value in packed numeric format.
%DECH	numeric expression,digits, decpos	Half-adjusted value in packed numeric format.
%DECPOS	numeric expression	Number of decimal digits.
%DIFF	op1, op2, { *MSECONDS *SECONDS *MINUTES *HOURS *DAYS *MONTHS *YEARS }	The difference (duration) between two date, time, or timestamp values.
%DIGITS	Numeric,start{,length}	Value in integer format.
%DIV	dividend , divisor	The quotient from the division of two arguments; packed numeric if either operand is a packed, zoned, or binary numeric value, or integer if either operand is an integer numeric value.
%EDITC	non-float numeric expression , edit code {, *ASTFill *CURSYM currency-symbol})	Value in string format representing the edited value.
%EDITFLT	numeric expression	Character external display representation of float.
%EDITW	non-float numeric expression , edit word	Value in string format representing the edited value.
%ELEM	array, table, or multiple occurrence data structure name	Number of elements or occurrences.
%EOF	{file name}	'1' if the most recent read operation or write to a subfile ended in an end of file or beginning of file condition; otherwise, it returns '0'.
%EQUAL	{file name}	'1' if the most recent relevant operation found an exact match; otherwise, it returns '0'.
%ERROR		'1' if the most recent operation with extender 'E' specified resulted in an error condition. '0' and remains unchanged following the operation if no error occurs.
%FLOAT	numeric expression	Value in float format.
%FOUND	{file name}	'1' if the most recent relevant file operation found a record (CHAIN, DELETE, SETGT, SETLL), an element (LOOKUP), or a match (CHECK, CHECKR, SCAN). Otherwise, this function returns '0'.

%HOURS	number	The result is a time or timestamp value with the appropriate <i>number</i> of hours added or subtracted.
%INT	numeric expression	Value in integer format.
%INTH	numeric expression	Half-adjusted value in integer format.
%KDS	dsn name {,number-keys}	The data structure reference as indicated by setting the <i>Key</i> parameter in the CHAIN, DELETE, READE, READPE, SETGT, or SETLL using this function.
%LEN	expression	The length of a program defined variable or database variable defined as variable length, or also the current length of such fields.
%LOOKUP	argument, array {,start-index {,num-elements}}	Returns the array index of the item in <i>array</i> that matches the <i>argument</i> ; otherwise, zero if no match is found.
%MINUTES	number	The result is a time or timestamp value with the appropriate <i>number</i> of minutes added or subtracted.
%MONTHS	number	The result is a date or timestamp value with the appropriate <i>number</i> of months added or subtracted.
%MSECONDS	number	The result is a time or timestamp value with the appropriate <i>number</i> of microseconds added or subtracted.
%NULLIND	fieldname	The name of the field to query or set the null indicator for null-capable fields.
%OCCUR	dsn name	Returns an unsigned numeric value of the current occurrence number of the dsn name (data structure). Also can be used to set the occurrence number.
%OPEN	file name	'1' if the specified file is open. '0' if the specified file is closed.
%REM	, dividend, divisor	The remainder from the division of the two arguments.
%REPLACE	Replacement string , source string {,start position{,source length to Replace}}	String produced by inserting Replacement string into source string, starting at start position and replacing the specified number of characters.
%SCAN	Search argument, string to be searched {,start position]	First position of search argument in string, or zero if not found.
%SECONDS	number	The result is a time or timestamp value with the appropriate <i>number</i> of seconds added or subtracted.
%SQRT	numeric expression	The square root of the numeric expression.

%SUBARR	array ,start-index {,num-elements}	Returns a section of the specified <i>Array</i> starting at Start-index . The number of elements returned is specified by <i>num-elements</i> , or the remainder of the array if <i>num-elements</i> is not specified.
%SUBDT	date, time, or timestamp value, {*MSECONDS *SECONDS *MINUTES *HOURS *DAYS *MONTHS *YEARS}	An unsigned numeric value with the extracted portion of a date, time, or timestamp value.
%SUBST	String,start{,length}	Substring.
%TIME	(character, numeric, or timestamp expression {,*ISO *HMS *USA *EUR *JIS})	Value of expression converted to type time or the current system time in *ISO format.
%TIMESTAMP	(character, numeric, or timestamp expression {,*ISO *ISO0})	Value of expression converted to type timestamp or the current system timestamp in *ISO format.
%TLOOKUP	Argument, search-table {, alternate_table}	Sets the current table element to the element that satisfies the condition and returns the value *ON, otherwise, *OFF if no match is found.
%TRIM	string	String with left and right blanks trimmed.
%TRIML	string	String with left blanks trimmed.
%TRIMR	string	String with right blanks trimmed.
%XFOOT	array expression	Sum of the elements.
%XLATE	from, to, string {,start position}	Value translated from <i>String</i> , with any <i>From</i> character values replaced with the corresponding <i>To</i> character values.
%YEARS	number	The result is a date or timestamp value with the appropriate <i>number</i> of years added or subtracted.

%ABS Function - (Absolute Value)

%ABS built-in function returns the absolute value of the numeric expression specified as the parameter.

```
%ABS (Numeric Expression)
```

Parts

Numeric Expression

Required. Numeric expression can be any valid numeric expression. If the value of the numeric expression is non-negative, the value is returned unchanged. If the value is negative, the value returned is the value of the expression but with the negative sign removed.

Remarks

For date, time, or timestamp data, the returned value includes any separator characters. The format and separators of the result are the same as that of the parameter.

See Also

[Built-in Function Overview](#)

%BOOL Function - (Get Boolean Value)

%BOOL built-in function returns the boolean value of True or False.

```
%BOOL (value)
```

Remarks

Returns boolean value as follows.

- A negative or positive number is true and any value that contains zero is false.
- A string is false if it is zero-length or filled with zeroes only, and true if its length is greater than zero and non-zero characters are present.
- An object of a type other than the four basic types is converted to a boolean in a way that is dependent on that type.

See Also

[Built-in Function Overview](#)

%CHAR function - (Convert to Character Data)

%CHAR built-in function converts the value of the expression from numeric, date, time or timestamp data to type character.

```
%CHAR (Expression)
```

Parts

Expression

Required. Can be any valid numeric expression. If the parameter is a constant, the conversion will be done at compile time.

Remarks

For date, time, or timestamp data, the returned value includes any separator characters. The format and separators of the result are the same as that of the parameter.

The **%CHAR** command cannot accept *BOOLEAN and *IND data types.

See Also

Built-in Function Overview

%CHECK Function - (Check Characters)

%CHECK built-in function returns the first position of the string *Base* that contains a character that does NOT appear in string *Comparator*. If all of the characters in *Base* are also in *Comparator*, this function returns 0.

```
%CHECK (Comparator, Base, {Start})
```

Parts

Comparator

Required. Can be a field name, array element, named constant, data structure name, data structure subfield name, literal, or table name; to compare against

Base.

Required. Can be any valid numeric or character expression to compare with Comparator, and must be of the same type as Comparator.

Start

Optional. Contains the position, relative to the leftmost position in Base, in which to start. If specified, Start must be a non-float numeric with zero decimal positions. If not specified, the starting position defaults to 1.

Remarks

Checking begins at the Start or leftmost position of Base and continues character by character left to right. Each character of Base is compared with the Comparator characters and if a match exists, the next Base character is checked. If no match is found, the function returns the position within Base of the non-matched character. If all of the characters in Base are also in Comparator, this function returns 0.

The **%CHECK** command cannot accept *BOOLEAN and *IND data types.

See Also

Built-in Function Overview | CHECK | CHECKR | %CHECKR

%CHECKR Function - (Check Characters Reverse)

%CHECKR built-in function returns the last position of the string *Base* that contains a character that does NOT appear in string *Comparator*. If all of the characters in *Base* are also in *Comparator*, this function returns 0.

```
%CHECKR (Comparator, Base, {Start})
```

Parts

Comparator

Required. Can be a field name, array element, named constant, data structure name, data structure subfield name, literal, or table name; to compare against *Base*.

Base

Required. Can be any valid numeric or character expression to compare with *Comparator*, and must be of the same type as *Comparator*.

Start

Optional. Contains the position, relative to the leftmost position in *Base*, in which to start. If specified, *Start* must be a non-float numeric with zero decimal positions. If not specified, the starting position defaults to the length of the string.

Remarks

Checking begins at the *Start* or rightmost position of *Base* and continues character by character right to left. Each character of *Base* is compared with the *Comparator* characters and if a match exists, the next *Base* character is checked. If no match is found, the function returns the position within *Base* of the non-matched character. If all of the characters in *Base* are also in *Comparator*, this function returns 0. Note the return value is relative to the left although checking is done from the right.

The **%CHECKR** command cannot accept *BOOLEAN and *IND data types.

See Also

[Built-in Function Overview](#) | [CHECK](#) | [CHECKR](#) | [%CHECK](#)

%DATE Function - (Convert to Date)

%DATE built-in function converts the value of the expression from character, numeric, or timestamp data to type date. If no expression is specified, the current system date is returned.

```
%DATE ((Expression {, *ISO | *JUL | *USA | *EUR | *JIS | *MDY | *DMY  
| *YMD}))
```

Parts

Expression

Optional. Expression can be character, numeric, or timestamp data to be converted to type date.

The second parameter is the date format of character and numeric input. If *Expression* is a timestamp, *DATE, or UDATE, do not specify this parameter as the system knows the format of the input. Regardless of the input date format, the returned date is *ISO format.

Remarks

If no parameters are present, this function returns the current system date in *ISO format.

See Also

[Built-in Function Overview](#) | [Date Formats](#) | [Default Values Date_Formats](#) | [*DATE](#) | [UDATE](#) | [%TIME](#) | [%TIMESTAMP](#)

%DAYS Function - (Number of Days)

%DAYS built-in function converts a number into a duration that can be added to a date or timestamp value.

```
%DAYS (Number)
```

Parts

Number

Required. The number to be converted to duration and added to or subtracted from a date or timestamp value.

Remarks

%DAYS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a date or timestamp value with the appropriate number of days added or subtracted. For a date, the resulting value is *ISO format.

See Also

Built-in Function Overview | Adding or Subtracting Dates | %DIFF | %HOURS | %MINUTES | %MONTHS | %MSECONDS | %SECONDS | %YEARS

%DEC Function - (Convert to Packed Decimal Format)

%DEC built-in function converts the value of the numeric expression to packed decimal format with precision digits and decimal places.

```
%DEC (Numeric Expression , {Precision, DecimalPlaces})
```

Parts

Numeric Expression

Required. Contains the numeric expression in which to convert to packed decimal format.

Precision, DecimalPlaces

Optional. Must be numeric literals, named constants that represent numeric literals, or built-in functions with a numeric value known at compile-time. They may be omitted if the type of numeric expression is not float. If these parameters are omitted, the precision and decimal places are taken from the attributes of the numeric expression.

Remarks

%DEC is the same as %DECH, except that with **%DECH**, if the numeric expression is a decimal or float value, half-adjust is applied to the value of numeric expression when converting to the desired precision.

Example

```
BEGSR Dec_Ex
  DclFld Name( A )      Type( *ZONED ) Len( 9, 5 )
  DclFld Name( B )      Type( *ZONED ) Len( 9, 5 )
  DclFld Name( C )      Type( *ZONED ) Len( 9, 6 )
  DclFld Name( Result1 ) Type( *ZONED ) Len( 9, 5 )
  DclFld Name( Result2 ) Type( *ZONED ) Len( 7, 5 )
  DclFld Name( Result3 ) Type( *ZONED ) Len( 8, 5 )

  A = 1234.567
  B = 73.73442
  C = 123.456789
```

```

    Result1 = %DEC(A) + 0.011
    MsgBox Result1
    Result2 = %DEC(B, 5, 0)
    MsgBox Result2
    Result3 = %DECH(C, 5, 2)
    MsgBox Result3

    // The value of "Result1" is now 1234.57800
    // The value of "Result2" is now 73.00000
    // The value of "Result3" is now 123.46000
ENDSR

```

See Also

Built-in Function Overview | %DECH Function

%DECH Function - (Convert to Packed Decimal Format with Half-Adjust)

%DECH built-in function is the same as %DEC, except that if the numeric expression is a decimal or float value, half adjust is applied to the value of numeric expression when converting to the desired precision. No message is issued if half adjust cannot be performed.

```
%DECH (Numeric Expression, {Precision, Decimal Places})
```

Parts

Numeric Expression

Required. Contains the numeric expression in which to convert to packed decimal format.

Precision, Decimal Places

Required. Precision and decimal places must be numeric literals, named constants that represent numeric literals, or built-in functions with a numeric value known at compile-time.

Remarks

Unlike **%DEC**, all three parameters are required.

Example

```

BEGSR Dech_Ex
    DclFld  Name( A )           Type( *ZONED ) Len( 9, 5 )
    DclFld  Name( B )           Type( *ZONED ) Len( 9, 5 )
    DclFld  Name( C )           Type( *ZONED ) Len( 9, 6 )
    DclFld  Name( Result1 )     Type( *ZONED ) Len( 9, 5 )
    DclFld  Name( Result2 )     Type( *ZONED ) Len( 7, 5 )
    DclFld  Name( Result3 )     Type( *ZONED ) Len( 8, 5 )

    A = 1234.567
    B = 73.73442
    C = 123.456789

    Result1 = %DEC( A ) + 0.011
    MsgBox Result1
    Result2 = %DEC( B, 5, 0 )
    MsgBox Result2
    Result3 = %DECH( C, 5, 2 )
    MsgBox Result3

    // The value of Result1 is now 1234.57800
    // The value of Result2 is now 73.00000
    // The value of Result3 is now 123.46000
ENDSR

```

See Also

[Built-in Function Overview](#) | [%DEC Function](#)

%DEC POS Function - (Get Number of Decimal Positions)

%DECPOS built-in function returns the number of decimal positions of the numeric variable or expression. The value returned is a constant, and so may participate in constant folding.

```
%DEC POS (Numeric Expression)
```

Parts

Numeric Expression

Required. Must not be a float variable or expression.

See Also

[Built-in Function Overview](#) | [%DEC Function](#)

%DIFF Function (Difference Between Dates, Times, or Timestamps)

%DIFF built-in function produces the difference (duration) between two date or time values. The first and second parameters must be the same or compatible types.

```
%DIFF (Op1, Op2, { *MSECONDS | *SECONDS | *MINUTES | *HOURS | *DAYS | *MONTHS  
| *YEARS })
```

Parts

Op1, Op2

Required. These two parameters must be the same or compatible types. The following combination are valid:

- Date and Date
- Time and Time
- Timestamp and Timestamp
- Date and Timestamp
- Time and Timestamp

*MSECONDS, *SECONDS, *MINUTES, *HOURS, *DAYS, *MONTHS, *YEARS

Required. This specifies the duration. Valid codes depend upon the previous parameters as follows:

- For Date and Date, you can specify *DAYS, *MONTHS, or *YEARS.
- For Time and Time, you can specify *SECONDS, *MINUTES, or *HOURS.
- For Timestamp and Timestamp, you can specify *MSECONDS, *SECONDS, *MINUTES, *HOURS, *DAYS, *MONTHS, or *YEARS.
- For Date and Timestamp, you can specify *DAYS, *MONTHS, or *YEARS. **Note** Only the date portion of the Timestamp is considered.
- For Time and Timestamp, you can specify *SECONDS, *MINUTES, or *HOURS. **NOTE:** Only the time portion of the Timestamp is considered.

Remarks

The result will be rounded down with any remainder discarded. For Example, 61 minutes is equal to 1 hour, and 59 minutes is equal to 0 hours. Visual RPG places an 18-digit limit on durations. A duration with more than 15 significant digits will cause errors or truncation. The value returned by this function is compatible with both type numeric and type duration. You can add or subtract the result to a number (type numeric) or a date, time, or timestamp (type duration). See [Adding or Subtracting Dates](#) for more information.

See Also

[Built-in Function Overview](#) | [Date and Time Topics](#) | [Adding or Subtracting Dates](#) | [%DAYS](#) | [%HOURS](#) | [%MINUTES](#) | [%MONTHS](#) | [%MSECONDS](#) | [%SECONDS](#) | [%YEARS](#) | [ADDDUR](#) | [SUBDUR](#)

%DIGITS Function

%DIGITS built-in function can be used to get and set digits of a Fixed Decimal Numeric. When getting, the digits are returned as a decimal number and when used for setting should be assigned a decimal expression.

```
%DIGITS (Source, Start, Length)
```

Parts

Source

Required. Must be a Fixed Decimal Numeric (*zoned, *packed, or *binary) from which the digits are extracted.

Start

Required. Represents the starting position from where to extract the digits of the fixed decimal. The starting position must be greater than zero.

Length

Required. Represents the number of digits to be extracted from the **Source** parameter.

Remarks

%Digits can also be used to replace digits of a fixed decimal numeric when used as a result of an assignment expression. **%Digits** returns a decimal representing the number of digits, hence if the digits start with a 0, that 0 will not be a part of the number returned.

Example

```
BEGSR Digits_Ex
DclFld Name(Phone) Type (*Packed) Len( 10, 0) INZ ( 2105551212)
ioField1.Number = %Digits (Phone, 3, 7) // 55512 ( 055512 as a valid decimal
is 55512 )
ENDSR
```

See Also

[Built-in Function Overview](#) | [EVAL](#)

%DIV Function - (Return Integer Portion of Quotient)

%DIV built-in function returns the integer portion of the quotient that results from dividing operands **N** by **M**.

```
%DIV (N,M)
```

Parts

N,M

Required. Must be numeric values with zero decimal positions.

- If either operand is a packed, zoned, or binary numeric value, the result is packed numeric.
- If either operand is an integer numeric value, the result is integer. Otherwise, the result is unsigned numeric.
- Float numeric operands are not allowed.

Remarks

%DIV and %REM built-in function have the following relationship:

$$\%REM(A,B) = A - (\%DIV(A,B) * B)$$

If the operands are constants that can fit in 8-byte integer or unsigned fields, constant folding is applied to the built-in function. In this case, %REM can be coded in the definition specifications.

Example

```
BEGSR Div_Ex
DclFld Name(A) Type(*ZONED) Len(5,0)
DclFld Name(B) Type(*ZONED) Len(5,0)
DclFld Name(Result) Type(*ZONED) Len(5,0)

A = 123
B = 27

Result = %DIV (A,B)
MsgBox Result

Result = %Rem (A,B)
MsgBox Result

Result = %Rem (A,B) + %Div (A,B)
MsgBox Result

// Now, "Result" = 4, 15, and 19
ENDSR
```

See Also

[Built-in Function Overview](#) | [%REM Function](#)

%EDITC Function - (Edit Value Using an Edit Code)

%EDITC built-in function returns a character result representing the numeric value edited according to the edit code.

```
%EDITC (Numeric Expression, EditCode, (*ASTFill | *CURSYM | Currency-
symbol))
```

Parts

Numeric Expression, EditCode

Required. In general, the rules for the Numeric Expression and Edit Code parameters are identical to those for editing numeric values in output specifications.

When using the %EDITC built-in-function, you can only enter the alpha representation for the desired edit code. You cannot enter the numeric representation.

***ASTFill, *CURSYM, Currency-symbol**

Optional. The third parameter is optional, and if specified, must be one of the following:

***ASTFill** - Indicates that asterisk protection is to be used. This means that leading zeros are %EDITC'd with asterisks in the returned value. For Example,

```
%EDITC(-0012.5 , 'K' , *ASTFill) returns '***12.5'
```

CURSYM - Indicates that a floating currency symbol is to be used. The actual symbol will be the one specified on the control specification in the *CURSYM keyword, or the default, '\$'. When *CURSYM is specified, the currency symbol is placed in the result just before the first significant digit. For Example,

```
%EDITC(0012.5 , 'K' , *CURSYM) returns // $12.5 //
```

Currency-symbol - Indicates that floating currency is to be used with the provided currency symbol. It must be a 1-byte character constant (literal, named constant or expression that can be evaluated at compile time). For Example,

```
%EDITC(0012.5 , 'K' , £) returns // £12.5 '
```

See Also

[Built-in Function Overview](#) | [Numeric Expression](#) | [Edit Code Usage](#)

%EDITFLT Function - (Convert to Float External Representation)

%EDITFLT built-in function converts the value of the numeric expression to the character external display representation of float. The result is either 14 or 23 characters. If the argument is a 4-byte float field, the result is 14 characters. Otherwise, it is 23 characters.

```
%EDITFLT (Numeric Expression)
```

Parts**Numeric Expression**

Required. If specified as a parameter to a definition specification keyword, the parameter must be a numeric literal, float literal, or numeric value constant name or built-in function.

When specified in an expression, constant folding is applied if the numeric expression has a constant value.

Example

```
BEGSR
DclFld Name(A) Type(*ZONED) Len(5,0)
DclFld Name(Result) Type(*String)

A = 50000
Result = "Float value is " + %EditFlt (A) + "."
MsgBox Result

// The value of "Result" is "Float value is +5.000000e+004."
ENDSR
```

See Also

[Built-in Function Overview](#)

%EDITW Function - (Edit Value Using an Edit Word)

%EDITW built-in function returns a character result representing the numeric value edited according to the edit word.

```
%EDITW (Numeric, Editword)
```

Parts

Numeric, Editword

Required. The rules for the **Numeric** value and **Editword** are identical to those for editing **numeric** values in output specifications.

Example

If the values of **mySalary** is 2451.53, and **myEditWd** is "\$,**Dollars Cents ", then the edited version of **mySalary * 12** is:

```
(amount = 'The annual salary is // + %editw (mySalary * 12 , myeditwd)
```

See Also

Built-in Function Overview | Edit Word Usage

%ELEM - (Get Number of Elements)

%ELEM built-in function returns the number of elements in the specified array, table, or multiple-occurrence data structure.

```
%ELEM (Table_name)
%ELEM (Array_name)
%ELEM (Multiple_occurrence_data_structure_name)
```

Parts

Table_name, Array_name, Multiple_occurrence_data_structure_name

Required. The parameter must be the name of an array, table, or multiple occurrence data structure.

Remarks

The value returned is in unsigned integer format (type U). It may be specified anywhere a numeric constant is allowed in the definition specification or in an expression in the extended Factor 2 field.

The parameter must be the name of an array, table, or multiple occurrence data structure.

The array must be declared using the DCLARRAY command to be supported by **%ELEM**.

See Also

Built-in Function Overview

%EOF Function - (Return End or Beginning of File Condition)

%EOF built-in function returns **True** if the most recent read operation or write to a subfile ended in an end of file or beginning of file condition; otherwise, it returns **False**.

```
%EOF {(File_name)}
```

Parts

File_name

Optional. The name of the file. **%EOF** is only available if the file name is specified for primary and secondary files.

Remarks

The operations that set **%EOF** are:

- **READ** (Read a Record)
- **READC** (Read Next Changed Record)
- **READE** (Read Equal Key)
- **READP** (Read Prior Record)
- **READPE** (Read Prior Equal)
- **WRITE** (Create New Records - subfile only)

When a full-procedural file is specified, this function returns **True** if the previous operation in the list above, for the specified file, resulted in an end of file or beginning of file condition.

For primary and secondary files, **%EOF** is available only if the file name is specified.

This function is allowed for input, update, and record-address files; and for display files allowing **WRITE** to subfile records.

See Also

[Built-in Function Overview](#) | [READ](#) | [READC](#) | [READE](#) | [READP](#) | [READPE](#) | [WRITE](#) | [IsEof Property](#)

%EQUAL Function - (Return Exact Match Condition)

%EQUAL built-in function returns **True** if the most recent relevant operation found an exact match; otherwise, it returns **False**.

```
%EQUAL {(File_name)}
```

Parts

File_name

Optional. The name of the file.

Remarks

The operations that set **%EQUAL** are:

- **SETLL** (Set Lower Limit)
- **LOOKUP** (Look Up a Table or Array Element)

If **%EQUAL** is used without the optional **File_name** parameter, then it returns the value set for the most recent relevant operation.

For the **SETLL** operation, this function returns **True** if a record is present whose key or relative record number is equal to the search argument.

For the **LOOKUP** operation with the EQ indicator specified, this function returns True if an element is found that exactly matches the search argument.

If a file name is specified, this function applies to the most recent **SETLL** operation for the specified file. This function is allowed only for files that allow the **SETLL** operation code.

See Also

Built-in Function Overview | LOOKUP | SETLL | IsEqual Property

%ERROR Function - (Return Error Condition)

%ERROR built-in function returns **True** if the most recent operation with error operation extender specified resulted in an error condition.

%ERROR

Remarks

This is the same as the error indicator being set on for the operation. Before an operation with an error operation extender begins, **%ERROR** is set to return False and remains unchanged following the operation if no error occurs. All operations that allow an error indicator can also set or reset the **%ERROR** built-in function.

The implementation of RPGIV's **%ERROR** built-in function is that each program has its own **%ERROR**. The "program" analog in a .NET Windows application is the current thread. **%ERROR** is NOT unique per class - it can be used by shared and non-shared members. **%ERROR** is a per-thread shared flag, and is therefore thread-safe when used in multithreading. **%ERROR** is stored in thread local storage using **Thread.GetNamedDataSlot** and **Thread.SetData** (which means it cannot be quick-watched). Because all AVR .NET objects in the thread share the same **%ERROR** built-in function, use of the **Try/Catch/Throw** is advised when you need more granular error handling.

Example

```
CHAIN From(myfile) Key(myKey) Err(*EXTENDED)
  If %Error
    .
    . // error handling here
    .
  Endif
```

See Also

Built-in Function Overview | IsError Property

%FLOAT Function - (Convert to Floating Format)

%FLOAT built-in function converts the value of the numeric expression to float format.

%FLOAT (Numeric Expression)

Parts

Numeric Expression

Required. The numeric expression to convert.

Remarks

This built-in function may only be used in expressions.

Example

```
BEGSR Float_Ex
DclFld Name(A)           Type(*PACKED) Len(1,0)
DclFld Name(B)           Type(*PACKED) Len(1,0)
DclFld Name(Result)      Type(*PACKED) Len(6,5)
A = 1
B = 3

Result = %Float (A / B)
MsgBox Result

// The value of "Result" is now 0.33333
ENDSR
```

See Also

[Built-in Function Overview](#)

%FOUND - (Return Found Condition)

%FOUND built-in function returns **True** if the most recent relevant file operation found a record, a string operation found a match, or a search operation found an element. Otherwise, this function returns **False**.

```
%FOUND{(File_name)}
```

Parts

File_name

Optional. The name of the file.

If the **file_name** parameter is not specified, then it returns the value set for the most recent relevant operation.

If the **file_name** parameter is specified, then it applies to the most recent relevant operation on that file.

Remarks

For **file** operations, **%FOUND** is opposite in function to the "no record found NR" indicator, or the NotFnd parameter.

For **string** operations, **%FOUND** is the same in function as the "found FD" indicator, or the Fnd parameter.

For the **LOOKUP** operation, **%FOUND** returns True if the operation found an element satisfying the search conditions.

The operations that set **%FOUND** are:

File operations:

- CHAIN (Random Retrieval from a File)
- DELETE (Delete Record)
- SETGT (Set Greater Than)
- SETLL (Set Lower Limit)

String operations:

- CHECK (Check Characters)
- CHECKR (Check Reverse)
- SCAN (Scan String)

Built-in function %SCAN does not change the value of %FOUND.

Search operations:

- LOOKUP (Look Up a Table or Array Element)

See Also

[Built-in Function Overview](#) | [%SCAN Function](#) | [CHAIN](#) | [CHECK](#) | [CHECKR](#) | [DELETE](#) | [LOOKUP](#) | [SCAN](#) | [SETGT](#) | [SETLL](#) | [IsFound Property](#)

%HOURS Function - (Number of Hours)

%HOURS built-in function converts a number into a duration that can be added to a time or timestamp value.

```
%HOURS (Number)
```

Parts

Number

Optional. The number to be converted to duration and added to or subtracted from a time or timestamp value.

Remarks

%HOURS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a time or timestamp value with the appropriate number of hours added or subtracted. For a time, the resulting value is *ISO format.

See Also

[Built-in Function Overview](#) | [Adding or Subtracting Dates](#) | [%DAYS](#) | [%DIFF](#) | [%MINUTES](#) | [%MONTHS](#) | [%MSECONDS](#) | [%SECONDS](#) | [%YEARS](#)

%INT Function - (Convert to Integer Format)

%INT built-in function converts the value of the numeric expression to integer.

```
%INT (Numeric Expression)
```

Parts

Numeric Expression

Required. The numeric expression to convert.

Remarks

Any decimal digits are truncated. This built-in function may only be used in expressions. **%INT** can be used to truncate the decimal positions from a float or decimal value allowing it to be used as an array index.

Example

```

BEGSR Int_Ex
  DclFld Name( A )           Type( *ZONED ) Len( 7,3 )
  DclFld Name( B )           Type( *ZONED ) Len( 7,5 )
  DclFld Name( C )           Type( *ZONED ) Len( 7,3 )
  DclFld Name( Result1 )     Type( *ZONED ) Len( 9,5 )
  DclFld Name( Result2 )     Type( *ZONED ) Len( 7,5 )
  DclFld Name( Result3 )     Type( *ZONED ) Len( 8,5 )   A = 1234.567
  B = 73.73442
  C = 123.789   Result1 = %Int ( A ) + 0.011
  Result2 = %Int ( B )
  Result3 = %Inth ( C )   MsgBox Result1
  MsgBox Result2
  MsgBox Result3

  //The value of "Result1" is now 1234.01100
  //The value of "Result2" is now 73.00000
  //The value of "Result3" is now 124.00000
ENDSR

```

See Also

[Built-in Function Overview | %INTH Function](#)

%INTH Function - (Convert to Integer Format with Half Adjust)

%INTH built-in function is the same as **%INT**, except that if the numeric expression is a decimal or float value, half-adjust is applied to the value of the numeric expression when converting to integer type.

```
%INTH (Numeric Expression)
```

Parts**Numeric Expression**

Required. The numeric expression to convert.

Remarks

No message is issued if half-adjust cannot be performed.

Example

```

BEGSR Inth_Ex
  DclFld Name(A)             Type(*ZONED) Len(7,3)
  DclFld Name(B)             Type(*ZONED) Len(7,5)
  DclFld Name(C)             Type(*ZONED) Len(7,3)
  DclFld Name(Result1)       Type(*ZONED) Len(9,5)
  DclFld Name(Result2)       Type(*ZONED) Len(7,5)
  DclFld Name(Result3)       Type(*ZONED) Len(8,5)

  A = 1234.567
  B = 73.73442
  C = 123.789

  Result1 = %Int (A) + 0.011
  Result2 = %Int (B)
  Result3 = %Inth (C)

  MsgBox Result1
  MsgBox Result2
  MsgBox Result3

```

```
// The value of "Result1" is now 1234.01100
// The value of "Result2" is now 73.00000
// The value of "Result3" is now 124.00000
ENDSR
```

See Also

Built-in Function Overview | %INT

%KDS Function (Search Arguments in Data Structure)

%KDS built-in function is allowed as the search argument for any keyed input or output operation (CHAIN, DELETE, READE, READPE, SETGT, SETLL). The search argument is specified by the subfields of the data structure name coded as the first argument of the built-in function.

```
%KDS (DSN_name {, Number-keys})
```

Parts

DSN_name

Required. Contains the name of the Data Structure.

Number-keys

Optional. Contains the number of subfields of the data structure to use as the search argument.

Remarks

The individual key values in the compound key are taken from the top level subfields of the data structure.

Subfields used to form the compound key must not be arrays.

The types of all subfields (up to the number specified by **Number-keys**) must match the types of the actual keys. Where lengths and formats differ, the value is converted to the proper length and format.

If the data structure is defined as an array data structure, an index must be supplied for the data structure.

See Also

Built-in Function Overview | Multiple Occurrence Data Structures | CHAIN | DELETE | READE | READPE | SETGT | SETLL

%LEN Function - Get or Set Length

%Len built-in function can be used to get the length of a program defined variable or database variable defined as "Varying". It can also be used to set the current length of such fields.

%Len is not supported for non-varying fields or expressions.

```
%Len (Expression)
```

Parts

Expression

Required. The expression to which the **%Len** function will apply.

Remarks

When used on the left-hand side of an expression, this function sets the current length of a varying field. If the set length is greater than the current length, the characters in the field between the old length and the new length are set to blanks. A varying field length can then be read and changed at run-time using `%Len ( FieldName )`.

The following are rules to follow when using `%Len`:

The maximum length you can specify is the length that was specified for that field when declaring the field (**DCLFLD**). A run-time error will occur if the maximum length is exceeded.

By assigning a string of Length X - the length of the field is changed to X.

```
// Given an externally described varying file of length 10:
City = "Hello" msgbox %Len( City ) // This shows a msgbox with 5, the
length of the field.
```

When increasing the length of the field from X to Y, the (Y-X) characters of the field are initialized to spaces.

```
// Given an externally described varying field of length 10:
City = "Hello" // This from Example above changes the len to 5
%Len( City ) = 2 // City would look like "He" in Quickwatch
%Len( City ) = 5 // Now City Should look like "He " in Quickwatch.
```

`%Len` can only be used on the left-hand-side of an expression when the parameter is an externally-described varying field.

Using %Len at Compile-Time

`%Len` can be used to define the length of a variable based upon the defined length of another variable or data structure. The basis variable must be defined prior in the code to the use of `%Len`. The basis variable must also be within the same scope as the using variable.

```
dclds MyDS
    dclDSfld DS1 Len( 5,0 )
    dclDSfld DS2 Len( 25 )
dclfld X len( 3,2 )
dclfld Y Len( %len(X),1 ) // Len( 3,1 )
dclfld Z Len( %len(Y),3 ) // Len( 3,3 )
dclfld D Len( %Len(MyDS)) // Len( 30 )
```

Example

```
%Len ( FieldName ) = 30 // Sets the length of the FieldName field to
30
Length = %Len ( FieldName ) // Sets the length to the length of the
FieldName field
```

See Also

[Built-in Function Overview](#) | [Character data type](#) | [DCLFLD](#) | [Variable Length Overview](#)

%LOOKUP Function (Look Up an Array Element)

`%LOOKUP` built-in function returns the array index of the item in **Array** that matches the **Argument**; otherwise, zero is returned if no match is found.

```
%LOOKUP (Argument, Array{, Start-index {, Num-elements}})
%LOOKUPLT (Argument, Array{, Start-index {, Num-elements}})
%LOOKUPGE (Argument, Array{, Start-index {, Num-elements}})
%LOOKUPGT (Argument, Array{, Start-index {, Num-elements}})
%LOOKUPLE (Argument, Array{, Start-index {, Num-elements}})
```


Parts

Argument, Array

Required. Contains the search argument and the name of the array to be searched. Both must be the same type but do not need to be the same length.

Start-index, Num-elements

Optional. Non-float numerics with zero decimal positions, containing the index in which to begin the search, and the number of elements to be searched. By default, the entire **Array** is searched.

Remarks

The search starts at index **Start-index** and continues for **Num-elements**. The function returns the array index of the item in **Array** that matches **Argument** as follows:

%LOOKUP - for an exact match.

%LOOKUPLT - for the value that is closest to, but less than Argument in value.

%LOOKUPLE - for an exact match, or the value that is closest to, but is less than Argument in value.

%LOOKUPGT - for the value that is closest to, but greater than Argument in value.

%LOOKUPGE - for an exact match, or the value that is closest to, but greater than Argument in value.

If no match is found, zero is returned.

Unlike the LOOKUP op code, %LOOKUP only applies to arrays. To look up a value in a table, use the %TLOOKUP function.

Note %FOUND and %EQUAL are **NOT** set following a %LOOKUP operation.

See Also

Built-in Function Overview | DCLARRAY | DCLALIASGROUP | DCLDSFLD | LOOKUP | %EQUAL | %FOUND | %TLOOKUP

%MINUTES Function - (Number of Minutes)

%MINUTES built-in function converts a number into a duration that can be added to a time or timestamp value.

```
%MINUTES (Number)
```

Parts

Number

Required. The number to be converted to duration and added to or subtracted from a time or timestamp value.

Remarks

%MINUTES can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of minutes added or subtracted. For a time, the resulting value is *ISO format.

See Also

[Built-in Function Overview](#) | [Adding or Subtracting Dates](#) | [%DAYS](#) | [%DIFF](#) | [%HOURS](#) | [%MONTHS](#) | [%MSECONDS](#) | [%SECONDS](#) | [%YEARS](#)

%MONTHS Function - (Number of Months)

%MONTHS built-in function converts a number into a duration that can be added to a date or timestamp value.

%MONTHS (*Number*)

Parts

Number

Required. The number to be converted to duration and added to or subtracted from a date or timestamp value.

Remarks

%MONTHS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a date or timestamp value with the appropriate number of months added or subtracted. For a date, the resulting value is *ISO format.

In most cases, when adding or subtracting a duration in months, the general rule is that the month portion is increased or decreased by the number of months in the duration and the day portion is unchanged.

There are exceptions resulting when the day portion will exceed the actual number of days in the resulting month. In this case, the resulting day portion is adjusted to the actual month end date. For Example, adding one month to '97/05/31' results in '97/06/30'. The resulting month portion has been increased by 1 and the resulting day portion has been adjusted because June only has 30 days. See [Adding or Subtracting Dates](#) for more information.

See Also

[Built-in Function Overview](#) | [Adding or Subtracting Dates](#) | [%DAYS](#) | [%DIFF](#) | [%HOURS](#) | [%MINUTES](#) | [%MSECONDS](#) | [%SECONDS](#) | [%YEARS](#)

%MSECONDS Function - (Number of Microseconds)

%MSECONDS built-in function converts a number into a duration that can be added to a time or timestamp value.

%MSECONDS (*Number*)

Parts

Number

Required. The number to be converted to duration and added to or subtracted from a time or timestamp value.

Remarks

%MSECONDS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of microseconds added or subtracted. For a time, the resulting value is *ISO format.

See Also

Built-in Function Overview | %DAYS | %DIFF | %HOURS | %MINUTES | %MONTHS | %SECONDS | %YEARS

%NULLIND Function- (Query or Set Null Indicator)

%NULLIND built-in function built-in function can be used to query or set the null indicator for null-capable fields.

```
%NULLIND (Fieldname)
```

Parts

Fieldname

Required. The name of the field.

Remarks

The nullability of a field determines if the record in the file can contain a null value for that field. A null value, or NULL, is not the same as zero (0), blank, or a zero-length character string such as "". NULL means that no entry has been made. The presence of NULL usually implies that the value is either unknown or undefined. For Example, a null value in the price field of the titles file of the pubs database does not mean that the book has no price; NULL means that the price is unknown or has not been set.

The fieldname can be a null-capable array element, data structure, stand-alone field, subfield, or multiple occurrence data structure.

This built-in function can be used on the left hand side of an assignment expression to set the Null value capability of a field. When used on the left-hand side of an expression, this function can be used to set the null indicator for null-capable fields to *ON or *OFF, or *TRUE or *FALSE. The content of a null-capable field remains unchanged.

Example

```
// Test the null indicator for a null-capable field.
IF %NULLIND (myFieldName)
..
..
Endif

// Set the null indicator for a null-capable field.

%NULLIND(myFieldName) = *True    // myFieldName is null
%NULLIND(myFieldName) = *False   // myFieldName is not null
```

See Also

Built-in Function Overview

%OCCUR Function (Set/Get Occurrence of Data Structure)

%OCCUR built-in function sets or gets the current position of a **Multiple Occurrence Data Structure**.

```
%OCCUR (DSN_name)
```

Parts

DSN_name

Required. Contains the name of the Data Structure.

Remarks

When this function is evaluated for its value, it returns the current occurrence number of the specified Data Structure. This is an unsigned numeric value. When this function is specified on the left-hand side of an EVAL statement, the specified number becomes the current occurrence. This must be a non-float numeric value with zero decimal positions. See OCCUR operation code for more information about multi-occurrence data structures.

See Also

Built-in Function Overview | Multiple Occurrence Data Structures | OCCUR

%OPEN Function- (Return File Open Condition)

%OPEN built-in function returns **True** if the specified file is open.

```
%OPEN (File_name)
```

Parts

File_name

Required. The name of the file.

Remarks

A file is considered "open" if it has been opened by the RPG program during initialization or by an **OPEN** operation, and has not subsequently been closed.

If the file is conditioned by an external indicator and the external indicator was off at program initialization, the file is considered closed, and **%OPEN** returns **False**.

Example

```
// This goes in main line code
DclDiskFile Name( CUSTMAST ) Type( *Input ) Org( *Indexed ) DBDesc ( 'ASNA
LOCAL DB' )

BEGSR Open_Ex
  MsgBox %OPEN ( CUSTMAST )
  Close File    ( CUSTMAST )
  MsgBox %OPEN ( CUSTMAST )
ENDSR
```

See Also

Built-in Function Overview | IsOpen Property

%REM Function - (Return Integer Remainder)

%REM built-in function returns the remainder that results from dividing operands *n* by *m*.

```
<result> = %REM (n,m)
```

Parts

n,m

Optional. Must be numeric values with zero decimal positions.

Returns

<result> contains the remainder that results from dividing *n* by *m*. If either operand is a packed, zoned, or binary numeric value, the result is packed numeric. If either operand is an integer numeric value, the result is integer. Otherwise, the result is unsigned numeric. Float numeric operands are not allowed..

Remarks

%REM and **%DIV** have the following relationship:

$$\%REM(A,B) = A - (\%DIV(A,B) * B)$$

The result in this case has the same sign as the dividend. If the operands are constants that can fit in 8-byte integer or unsigned fields, constant folding is applied to the built-in function. In this case, the **%REM** built-in function can be coded in the definition specifications.

Example

```
BEGSR Rem_Ex
DclFld Name(A) Type(*ZONED) Len(5,0)
DclFld Name(B) Type(*ZONED) Len(5,0)
DclFld Name(Result) Type(*ZONED) Len(5,0)

A = 123
B = 27

Result = %DIV (A,B)
MsgBox Result

Result = %Rem (A,B)
MsgBox Result

Result = %Rem (A,B) + %Div (A,B)
MsgBox Result

// Now, "Result" = 4, 15, and 19
```

See Also

[Built-in Function Overview | %DIV](#)

%REPLACE Function - (Replace Character String)

%REPLACE built-in function returns the character string produced by inserting a replacement string into the source string, starting at the position specified and replacing the specified number of characters.

```
<stringVal> = %REPLACE (ReplaceString, SourceString, {StartPos,
{SourceLengthToReplace}})
```

Parts

ReplaceString, SourceString

Required. Must be of type character, graphic, or UCS-2 and can be in either fixed or variable-length format. The second parameter must be the same type as the first.

StartPos

Optional. Represents the starting position, measured in characters, to begin replacement. If it is not specified, the starting position is at the beginning of the source string. The value may range from one to the current length of the source string plus one.

SourceLengthToReplace

Optional. Represents the number of characters in the source string to be replaced. If zero is specified, then the replacement string is inserted **before** the specified starting position. If the parameter is not specified, the number of characters replaced in **SourceString** is the length of the replacement string. The value must be greater than or equal to zero, and less than or equal to the current length of **SourceString**.

The starting position and source length may be any numeric value or numeric expression with no decimal positions.

Returns

<stringVal> contains the character string produced by inserting a replacement string into the source string, starting at the position specified and replacing the specified number of characters. The returned value is varying length if the source string or replacement string are varying length, or if the start position or source length to replace are variables. Otherwise, the result is fixed length.

Example

```
BEGSR Replace_Ex
  DclFld Name(B) Type(*CHAR) Len(20)
  DclFld Result *String
  B = "Visual RPG 3.5"
  Result = %Replace ("4.0", B,12,3 )
  IOField1.Text = Result
ENDSR
```

See Also

Built-in Function Overview

%SCAN Function - (Scan for Characters)

%SCAN built-in function returns the first position of the search argument in the source string, or 0 if it was not found. If the start position is specified, the search begins at the starting position. The result is always the position in the source string even if the starting position is specified. The starting position defaults to 1.

```
<pos> = %SCAN (SearchArgument, SourceString, {Start})
```

Parts

SearchArgument

Required. Must be of type character.

SourceString

Required. Must be the same type as the first parameter.

Start

Optional. Must be numeric with zero decimal positions.

Returns

<pos> is an unsigned integer value containing the first position of the **SearchArgument** in **SourceString**. If the **SearchArgument** is not found, 0 will be returned.

Remarks

When any parameter is variable in length, the values of the other parameters are checked against the current length, not the maximum length.

This built-in function can be used anywhere that an unsigned integer expression is valid.

Unlike the SCAN operation code, **%SCAN** cannot return an array containing all occurrences of the search string and its results cannot be tested using the %FOUND built-in function.

Example

```
BEGSR Scan_Ex
  DclConst Name(Source) Value("This is a test.")
  DclFld Name(Pos) Type(*ZONED) Len(5,0)

  Pos = %Scan ("est", Source, 5)
```

```
Msgbox Pos
ENDSR
```

See Also

[Built-in Function Overview](#) | [%FOUND Function](#) | [SCAN](#)

%SECONDS Function - (Number of Seconds)

%SECONDS built-in function converts a number into a duration that can be added to a time or timestamp value.

```
%SECONDS (Number)
```

Parts

Number

Required. The number to be converted to duration and added to or subtracted from a time or timestamp value.

Remarks

%SECONDS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of seconds added or subtracted. For a time, the resulting value is *ISO format.

See Also

[Built-in Function Overview](#) | [Adding or Subtracting Dates](#) | [%DAYS](#) | [%DIFF](#) | [%HOURS](#) | [%MINUTES](#) | [%MONTHS](#) | [%MSECONDS](#) | [%YEARS](#)

%SQRT Function (Square Root of Expression)

%SQRT built-in function returns the square root of the numeric expression.

```
%SQRT (Numeric Expression)
```

Parts

Numeric Expression

Required. The valid numeric expression for which the square root will be returned.

Remarks

If the numeric expression is of type float, the result will be also; otherwise the result is packed decimal numeric.

See Also

[Built-in Function Overview](#) | [SQRT Operation](#)

%SUBARR Function (Get/Set Portion of an Array)

%SUBARR built-in function returns a section of the specified *Array* starting at *Start-index*. The number of elements returned is specified by the optional *Num-elements* parameter, or, if not specified, the number of elements returned defaults to the remainder of the array.

```
%SUBARR (Array, Start-index {, Num-elements})
```


Parts

Array

Required. This parameter must be an array. eg, a standalone field, data structure, or subfield defined as an array. This parameter must not be a table name or procedure call.

Start-index

Required. This parameter must be a numeric value with zero decimal positions. Float numeric values are not allowed. The value must be greater than or equal to 1 and less than or equal to the number of elements of the array.

Num-elements

Optional. If specified, this parameter must be a numeric value with zero decimal positions. A float numeric value is not allowed. The value must be greater than or equal to 1 and less than or equal to the number of elements remaining in the array after applying the **Start-index** value. By default, the number of elements returned is the remaining of the **Array**.

Remarks

Generally **%SUBARR** is valid in any expression where an unindexed array is allowed. However, **%SUBARR** cannot be used in the following places:

as the array argument of built-in function **%LOOKUP**.

a parameter passed by reference.

%SUBARR may be used in the following ways:

On the left-hand side of an assignment using **EVAL**. This changes the specified elements in the specified array.

Within the expression on the right-hand side of an assignment using **EVAL** where the target of the assignment is an array. This uses the values of the specified elements of the array. The array elements are used directly; a temporary copy of the sub-array is not made.

As parameter of the **SORTARR** operation.

Passed by **VALUE** or by read-only reference (**CONST** keyword) when the corresponding parameter is defined as an array.

As the parameter of the **%XFOOT** function.

Note It is valid to use **%SUBARR** to assign part of an array to another part of the same array. However, if the source part of the array overlaps the target part of the array, unpredictable results can occur.

See Also

Built-in Function Overview | **DCLARRAY** | **DCLDS** | **DCLDSFLD** | **EVAL** | **SORTARR** | **%LOOKUP**

%SUBDT Function (Extract Portion of Date, Time, or Timestamp)

%SUBDT built-in function extracts a portion of a date, time, or timestamp and returns an unsigned numeric value.

```
%SUBDT (Value, {*MSECONDS | *SECONDS | *MINUTES | *HOURS | *DAYS | *MONTHS | *YEARS})
```

Parts

Value

Required. A date, time, or timestamp type value.

***MSECONDS, *SECONDS, *MINUTES, *HOURS, *DAYS, *MONTHS, *YEARS**

Indicates the portion of *value* you want to extract. The following table indicates the valid values for the *value* type:

For Date, you can specify *DAYS, *MONTHS, or *YEARS.

For Time, you can specify *SECONDS, *MINUTES, or *HOURS.

For Timestamp, you can specify *MSECONDS, *SECONDS, *MINUTES, *HOURS, *DAYS, *MONTHS, or *YEARS.

Remarks

This function returns an unsigned numeric value and will always return a 4 digit year.

See Also

Built-in Function Overview | %DIFF

%SUBST Function - (Get Substring)

%**SUBST** built-in function returns a portion of argument string. It may also be used as the result of an assignment with the EVAL operation code.

```
%SUBST (String, Start {Length})
```

Parts**String**

Required. Must be character. Starting position and length may be any numeric value or numeric expression with zero decimal positions. When the string parameter is of varying length, the values of the other parameters are checked against the current length, not the maximum length.

Start

Required. Represents the starting position of the substring. The starting position must be greater than zero.

Length

Optional. Represents the length of the substring. If it is not specified, the length is the length of the string parameter less the start value plus one. The length may be greater than or equal to zero.

Remarks

When specified as a parameter for a definition specification keyword, the parameters must be literals or named constants representing literals. When specified on a free-form calculation specification, the parameters may be any expression.

Example

```
BEGSR Subst_Ex
DclConst Name(City) Value("San Antonio")

ioField1.Text = %Subst (City, 5, 7)
ENDSR
```

See Also

Built-in Function Overview | EVAL

%TIME Function (Convert to Time)

%TIME built-in function converts the value of an expression from character, numeric, or timestamp data to type time. If no expression is specified, the current system time is returned.

```
%TIME ((Expression {, *ISO | *HMS | *USA | *EUR | *JIS}))
```

Parts

Expression

Optional. *Expression* can be character, numeric, or timestamp data to be converted to type time.

*ISO, *HMS, *USA, *EUR, *JIS

Optional. This is the time format of character and numeric input. If *Expression* is a timestamp, do not specify this parameter as the system knows the format of the input. The default for character and numeric input is *ISO format.

Remarks

The converted value remains unchanged but is returned as a time. If no *Expression* is specified, the current system time is returned. Regardless of the input format, the resulting value is *ISO time format.

See Also

Built-in Function Overview | Time Formats | %TIMESTAMP | %DATE

%TIMESTAMP Function (Convert to Timestamp)

%TIMESTAMP built-in function converts the value of an expression from character, numeric, or date data to type timestamp. If no expression is specified, the current system timestamp is returned.

```
%TIMESTAMP ((Expression {, *ISO | *ISO0}))
```

Parts

Expression

Optional. *Expression* can be character, numeric, or date data to be converted to type timestamp.

*ISO, *ISO0

Optional. This is the timestamp format.

For character expressions, *ISO or *ISO0 is valid.

For numeric expressions, you need not specify the timestamp format as only *ISO (the default) is valid.

For date expressions, do not specify the timestamp format. The date is converted from its current format to *ISO.

Remarks

If no *Expression* is specified, the current system timestamp is returned. Regardless of the input format, the resulting value is *ISO format.

See Also

Built-in Function Overview | Timestamp Data Types | %TIME | %DATE

%TLOOKUP Function (Look Up a Table Element)

%TLOOKUP built-in function sets the current table element to the element that satisfies the condition and returns the value ***ON**, otherwise, ***OFF** is returned if no match is found.

```
%TLOOKUP (Argument, Search_Table {, Alternate_Table})
%TLOOKUPLT (Argument, Search_Table {, Alternate_Table})
%TLOOKUPGE (Argument, Search_Table {, Alternate_Table})
%TLOOKUPGT (Argument, Search_Table {, Alternate_Table})
%TLOOKUPLE (Argument, Search_Table {, Alternate_Table})
```

Parts

Argument, Search_Table

Required. Contains the search argument and the name of the table to be searched. Both must be the same type but do not need to be the same length.

Alternate_Table

Optional. Contains an alternate table name.

Remarks

This function sets the current table element to the element that satisfies the condition, sets the current table element for the **Alternate-Table** to the same element, and returns the value ***ON** as follows:

%TLOOKUP - for an exact match.

%TLOOKUPLT - for the value that is closest to, but less than Argument in value.

%TLOOKUPLE - for an exact match, or the value that is closest to, but is less than Argument in value.

%TLOOKUPGT - for the value that is closest to, but greater than Argument in value.

%TLOOKUPGE - for an exact match, or the value that is closest to, but greater than Argument in value.

If no match is found, ***OFF** is returned.

Unlike the LOOKUP op code, **%TLOOKUP** only applies to tables. To look up a value in an array, use the **%LOOKUP** function.

Note **%FOUND** and **%EQUAL** are **NOT** set following a **%TLOOKUP** operation.

See Also

Built-in Function Overview | LOOKUP | %EQUAL | %FOUND | %LOOKUP

%TRIM Function - (Trim Blanks at Edges)

%TRIM built-in function returns the given string less any leading and trailing blanks.

```
%TRIM (String)
```

Parts

String

Required. Can be character, graphic, or UCS-2 data. When specified as a parameter for a definition specification keyword, the string parameter must be a constant.

Example

```
BEGSR Trim_Ex
  DclConst Name(String) Value(" This is a test string. ")
  ioField1.Text = %Trim (String)
ENDSR
```

See Also

[Built-in Function Overview](#) | [%TRIML Function](#) | [%TRIMR Function](#)

%TRIML Function - (Trim Leading Blanks)

%TRIML built-in function returns the given string less any leading blanks.

```
%TRIML (String)
```

Parts**String**

Required. Can be character, graphic, or UCS-2 data. When specified as a parameter for a definition specification keyword, the string parameter must be a constant.

Example

```
BEGSR TrimL_Ex
  DclConst Name(String) Value(" This is a test string.")
  ioField1.Text = %TrimL (String)
ENDSR
```

See Also

[Built-in Function Overview](#) | [%TRIM Function](#) | [%TRIMR Function](#)

%TRIMR Function - (Trim Trailing Blanks)

%TRIMR built-in function returns the given string less any trailing blanks.

```
%TRIMR (String)
```

Parts**String**

Required. Can be character, graphic, or UCS-2 data. When specified as a parameter for a definition specification keyword, the string parameter must be a constant.

Example

```
BEGSR CommandButton7 Click
  DclConst Name(String) Value("This is a test string. ")
  ioField1.Text = %TrimR (String)
ENDSR
```

See Also

[Built-in Function Overview](#) | [%TRIM Function](#) | [%TRIML Function](#)

%XFOOT Function - (Sum Array Expression Elements)

%XFOOT built-in function results in the sum of all elements of the specified numeric array expression.

```
%XFOOT (array-expression)
```

Parts

array-expression

Optional. The precision of the result is the minimum that can hold the result of adding together all array elements, up to a maximum of 30 digits. The number of decimal places in the result is always the same as the decimal places of the array expression.

For Example, if ARR is an array of 500 elements of precision (17,4), the result of %XFOOT(ARR) is (20,4).

For %XFOOT(X) where X has precision (m,n), the following table shows the precision of the result based on the number of elements of X:

Elements of X Precision of %XFOOT(X)

1 (m,n)
2-10 (m+1,n)
11-100 (m+2,n)
101-1000 (m+3,n)
1001-10000 (m+4,n)
10001-32767 (m+5,n)

Remarks

Normal rules for array expressions apply. For Example, if ARR1 has 10 elements and ARR2 has 20 elements, %XFOOT(ARR1+ARR2) results in the sum of the first 10 elements of ARR1+ARR2.

This built-in function is similar to the XFOOT operation, except that float arrays are summed like all other types, beginning from index 0 on up.

Example

```
BEGSR XFoot_Ex
  DclArray Name(Arr1) DCLFLD(2,2) Type(*ZONED) Len(10, 0)
  DclFld Name($X) Type(*ZONED) Len(2,0)
  DclFld Name($Y) Type(*ZONED) Len(2,0)

  $X=0
  DoUntil ($X=3)
    $Y=0
    DoUntil ($Y=3)
      Arr1[$X,$Y] = 0
      $Y = $Y + 1
    EndDo $X = $X + 1
  EndDo
  ioField1.Text = %XFoot (Arr1)
ENDSR
```

See Also

Built-in Function Overview | XFOOT Operation

%XLATE Function (Translate)

%XLATE built-in function translates *String*, replacing the *From* character values with the corresponding *To* character values. A *Start position* within the string can also be specified.

```
%XLATE (From, To, String {,Start_position})
```

Parts

From

Required. A list of characters that should be replaced in the existing *String*.

To

Required. A list of replacement characters which will replace the corresponding positional *From* character in *String*. For Example, the first positional *From* character will be replaced by the first positional *To* character, second *From* with second *To*, etc.

String

Required. The string to be translated.

Start_position

Optional. The starting point in *String* where translation is to begin. The default is 1.

Remarks

Translation will begin at the first or *Start position* and will continue from left to right. Each character within *String* is compared to all characters within *From*. If a match is found, the corresponding *To* character is replaced in *String*. *From*, *To*, and *String* must all be the same type. The value returned will be the same type and have the same length as *String*.

See Also

Built-in Function Overview | XLATE

%YEARS Function - (Number of Months)

%YEARS built-in function converts a number into a duration that can be added to a date or timestamp value.

```
%YEARS (Number)
```

Parts

Number

Required. The number to be converted to duration and added to or subtracted from a date or timestamp value.

Remarks

%YEARS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a date or timestamp value with the appropriate number of years added or subtracted. For a date, the resulting value is *ISO format.

In most cases, when adding or subtracting a duration in years, the general rule is that the year portion is increased or decreased by the number of years in the duration and the day portion is unchanged. There are exceptions resulting when the day portion will exceed the actual number of days in the resulting month/year. In this case, the resulting day portion is adjusted to the actual month end date. See Adding or Subtracting Dates for more information.

See Also

Built-in Function Overview | Adding or Subtracting Dates | %DAYS | %DIFF | %HOURS | %MINUTES | %MONTHS | %MSECONDS | %SECONDS

Class Library

The Visual RPG class library contains a few of its own library of classes, interfaces, and value types that are included in Visual RPG for .NET. This library also provides access to system functionality and contains links to the .NET Framework classes in which applications, components, and controls are built.

Namespaces

The Visual RPG class library provides the following namespaces:

ASNA.VisualRPG.Runtime Contains classes that support compilation and code generation using the Visual RPG .NET language. **ASNA.VisualRPG.Windows.Controls** Contains classes that add additional functionality to the Control classes that display information to the user. **ASNA.DataGate.Client** Contains classes that support the ability for client-side database access. **System** Contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions. **System.Data** Consists mostly of the classes that constitute the ADO.NET architecture. The ADO.NET architecture enables you to build components that efficiently manage data from multiple data sources. In a disconnected scenario (such as the Internet), ADO.NET provides the tools to request, update, and reconcile data in multiple tier systems. The ADO.NET architecture is also implemented in client applications, such as Windows Forms, or HTML pages created by ASP.NET. **System.Drawing** Provides access to GDI+ basic graphics functionality. **System.Windows.Forms** Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system. **System.Xml** Provides standards-based support for processing XML.

ASNA.VisualRPG.Runtime Namespace

The namespace contains classes that support compilation and code generation using the Visual RPG .NET language. Most of the classes are reserved for the Visual RPG system. Some classes and their members (which are documented here) may be employed by the Visual RPG programmer.

Namespace hierarchy

Classes

Class	Description
Database	Defines a target for database operations. That is, a database source comprises a connection to a database engine. The database object models this notion of a database source. See DCLDB for additional information.
DataGateException	Provides granular reporting for exceptions thrown by the DataGate client for certain Database, DBFile and Printfile operations.
DBFile	Defines a reference to a system-level database file object and its members. It supports I/O operations to a database file. See DCLDISKFILE for additional information.
MemoryFile	Maintains an ADO.NET Dataset as though it were a collection of "flat files". See DCLMEMORYFILE

for additional information.

MessageFileReader

Provides a method for reading message text from an ASNA .amf Message file.

PrintFile

Defines a reference to an externally described, multiple-format DataGate printer file. Supports output operations to the printer file. See DCLPRINTFILE for additional information.

Database Class

The Database class contains the information required to create a database connection.

For a list of all members of this type, see Database Members.

ASNA.VisualRPG.Runtime

ASNA.VisualRPG.Runtime.**Database**

```
<Serializable>
Syntax
```

Thread Safety

This type is safe for multithreaded operations.

Remarks

To 'load' the contents of the registry for a particular database name, set the DBName property of the database object declared with DCLDB. If a database name exists with the name, values such as server, label, and user, etc., become current and available through properties of the object.

You can use the CreateName method to take all the current values of the database name, except for the name itself, and creates a new 'Database Name' in the registry with the name given in *NewName*.

You can use the UpdateName method to write the current values of the database name object as the new values of the database name in the registry.

There are two methods which are Shared. These methods don't affect the current values of the database object.

GetNames (*Boolean Public Databases) - Returns an array of string with all the names found in the registry. The Boolean parameter indicates whether to get Public Database (**True**) or those private to the current user (**False**).

DeleteName (*String Existing Name) - This method deletes from the registry an existing database Name.

Requirements

Namespace: ASNA.VisualRPG.Runtime

Assembly: ASNA Visual RPG Runtime (in ASNA.VisualRPG.Runtime.dll)

Platforms: Windows 2000, Windows XP Professional, Windows Server 2003 family.

See Also

ASNA.VisualRPG.Runtime | Database Class Members | DCLDB

Database Members

Public Constructor

 **Database** This can not be called directly in code, instantiate using DCLDB.

Public Methods

 **CreateName** Takes all the current values of the database name, except for the name itself, and creates a new 'Database Name' in the registry with the name given in *NewName*.

 **DeleteName** Deletes an existing database name from the registry.

 **GetNames** Returns an array of strings with all the names found in the registry. The boolean parameter indicates whether to get PublicDatabase (*True) or those private to the current user (*False).

 **UpdateName** The current values of the database name object are written out as the new values of the database name in the registry.

Public Properties

 **CurrentUserLibl** Sets the current library list for the current connection.

 **DBName** Gets or sets the name of a database.

 **InitialException** Gets the first exception thrown for this database. This property is of type ASNA.DataGate.Common.dgException.

 **InitialLibl** Gets or sets the current database's initial library list.

 **IsOpen** Gets a boolean value indicating if the file is currently open for I/O operations.

 **Label** Gets or sets a string containing the label of the database to connect to.

 **Password** The password to be used in conjunction with **User** for connection authentication.

 **PlatformAttribute** Returns or sets a platform-specific attribute of the database engine

 **PoolingTimeOut** Gets or sets the amount of time (in minutes) in which a connection will remain idle in the pool until it is closed and removed from the pool.

 **Port** Gets or sets the TCP port number used by the database server for TCP/IP-based transport (default 5042).

 **Server** Gets or sets the database server host machine name or address.

 **Text** Gets or sets a description of the database name.

 **User** Gets or sets the authenticated user profile that is used to authorize the current database connection.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Properties | DCLDB](#)

Database Methods

CreateName Method

CreateName takes all the current values of the database name, except for the name itself, and creates a new 'Database Name' in the registry with the name given in **NewName**.

```
database.CreateName (NewName)
```

Parameters

NewName

Required. A string containing a new database name to be created in the registry.

Remarks

CreateName creates a new 'Database Name' in the registry with the name given in **NewName**. The content of the database name registry entries are defined by the values of the Label, PlatformAttribute, PoolingTimeOut, Password, Port, Server, Text, and User properties.

You can use the UpdateName method to write the current values of the database name object as the new values of the database name in the registry.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | Database Class Methods | UpdateName Method | DCLDB](#)

DeleteName Method

DeleteName deletes an existing database name from the registry.

```
database.DeleteName (ExistingName)
```

Parameters

ExistingName

Required. A string containing the name of an existing database name to delete from the registry.

Remarks

This method is **Shared**, and does not affect the current values of the database object.

Use the GetNames method to get an array of strings containing all of the database names found in the registry.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB | GetNames Method](#)

GetNames Method

GetNames returns an array of strings with all of the names found in the registry.

```
variable[] = database.GetNames (publicDBs)
```

Parameters

publicDBs

Required. A boolean parameter indicating whether to get Public Databases (*True) or those private to the current user (*False).

Returns a string array with all of the names found in the registry.

Remarks

This method is **Shared**, and does not affect the current values of the database object.

Private Database names are stored in the registry at KEY_CURRENT_USER\Software\ASNA\Acceler8-DB\Current\Version\Databases.

Public Database names are stored in the registry at
HEY_LOCAL_MACHINE\SOFTWARE\ASNA\Acceler8-DB\Current\Version\Data Sources*PUBLIC.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB

UpdateName Method

UpdateName takes the current values of the database name object and writes them out as new values of the database name in the registry.

```
database.UpdateName ()
```

Remarks

UpdateName

Takes the current values of the database name object and writes them out as new values of the database name in the registry.

You can use the CreateName method to take all the current values of the database name, except for the name itself, and create a new 'Database Name' in the registry with the name given in **newName**.

UpdateName modifies the 'Database Name' in the registry corresponding the name given in the *Dbname* property. The updated content of the database name registry entries are defined by the values of the Label, PlatformAttribute, PoolingTimeout, Password, Port, Server, Text, and User properties.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB | CreateName Method

Database Properties

Public Properties

 CurrentUserLibl	Sets the current library list for the current connection.
 DBName	Gets or sets the name of a database.
 InitialException	Gets the InitialException source object of a database.
 InitialLibl	Gets or sets the current database's initial library list.
 IsOpen	Gets a boolean value indicating if the file is currently open for I/O operations.
 Label	Gets or sets a string containing the label of the database to connect to.
 Password	The password to be used in conjunction with User for connection authentication.
 PlatformAttribute	Gets or sets a platform-specific attribute of the database engine.
 PoolingTimeOut	Gets the amount of time (in minutes) in which a connection will remain idle in the pool until it is closed and removed from the pool.
 Port	Gets or sets the TCP port number used by the database server for TCP/IP-based transport (default 5042).
 Server	Gets or sets the database server host machine name or address.
 Text	Gets or sets a description of the database name.
 User	Gets or sets the authenticated user profile that is used to authorize the current database connection.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Methods | Database Class Members | DCLDISKFILE

CurrentUserLibl Property

Sets the current library list for the current connection.

```
database.CurrentUserLibl = string[]
```

Property Value

String Array. Sets the current library list for the current connection. This operation must be conducted upon an open database connection.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | DCLDB | InitialLibl Property

DBName Property

Gets or sets the **name** of a database.

```
database.DBName = <stringVal>
```

Property Value

<stringVal> containing the name of a database.

Remarks

When set to a valid database name, the referenced database server becomes the database object upon connection. Setting the database name is a common means of overriding the database connection. When set to a valid database name, the following properties are also set from the collateral content of the Database name's registry content: Label, PlatformAttribute, PoolingTimeOut, Port, Server, Text, User.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | DCLDB

InitialException Property

Gets the InitialException source object of a database.

```
<dgException> = database.InitialException
```

Property Value

<dgException> is of type **ASNA.DataGate.Common.dgException** and contains the description of the database InitialException.

See Also

ASNA.VisualRPG.Runtime | Database Class

InitialLibl Property

Gets or sets the current database's initial library list.

```
database.InitialLibl = <stringVal>[]
```

Property Value

<stringVal> is a string array containing the current database's initial library list. This list is associated with the Database name contained in the client registry.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | DCLDB | CurrentUserLibl Property

IsOpen Property

Gets a boolean value indicating if the file is currently open for I/O operations.

```
<booleanVal> = database.IsOpen
```

Property Value

booleanVal of type **Boolean*.

True indicates the file is currently open for I/O operations.

False indicates the file is currently **not** open for I/O operations.

Remarks

Refer to the CLOSE and OPEN commands.

IsOpen is equivalent to the %OPEN built-in function.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | DCLDISKFILE | CLOSE | OPEN | %OPEN

Label Property

Gets or sets a string containing the label of the database to connect to.

```
database.Label = <stringVal>
```

Property Value

<stringVal> of type **String* containing the label of the database to connect to.

Remarks

The **Label** property specifies a particular database instance on the server platform, for platforms that allow more than one server instance, such as DataGate for Windows Server and Microsoft SQL Server.

For single-instance platforms, such as the iSeries, the **Label** property is ignored. The **Label** property is also used in conjunction with the CreateName and UpdateName methods.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | DCLDB | CreateName Method | UpdateName Method

Password Property

Gets the password used in conjunction with User for connection authentication.

```
<stringVal> = database.Password
```

Property Value

<stringVal> is of type **String* and contains the password for the user profile indicated by the User property.

Remarks

Authentication is the process of verifying that a user with a unique name is the actual user that the name implies. This verification process is also applied to applications running in the security context of a user. Typically, authentication is implemented by giving the user a password.

The Password property is also used in conjunction with the CreateName and UpdateName methods, where the password is stored in the registry as encrypted data.

This property is only set if the user profile has an associated password, and only after setting the User property.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | DCLDB | CreateName Method | UpdateName Method | User Property

PlatformAttribute Property

Gets or sets the platform-specific attribute of the database engine.

```
database.PlatformAttribute = <stringVal>
```

Property Value

<stringVal> is of type **String* and contains the platform-specific attribute of the database engine.

Remarks

The **PlatformAttribute** property is used to distinguish between different database engines that can reside on the same machine. For instance, both DataGate for Windows and SQL Server can reside on the same machine. When accessing SQL Server databases, **PlatformAttribute** must be set to **"*SQLOLEDB"**. The setting of assumed **"*DATA LINK"** is used for DataGate for Windows and DataGate for iSeries. Any value other than **"*SQLOLEDB"** is presumed to be **"*DATA LINK"**.

The PlatformAttribute property is also used in conjunction with the CreateName and UpdateName methods.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Properties | Database Class Methods | DCLDB | CreateName Method | UpdateName Method

PoolingTimeout Property

Gets or sets the amount of time (in minutes) in which a connection will remain idle in the pool until it is closed and removed from the pool.

```
database.PoolingTimeout = <integerVal>
```

Property Value

<integerVal> is of type **Integer* and contains the amount of time (in minutes) in which a connection will remain idle in the pool until it is closed and removed from the pool.

Remarks

Connection pooling timeout applies only to DataGate connections to the iSeries/400. When a pooled DataGate job is idle for the timeout period, the job is terminated.

When database connection pooling is enabled, the communication to the server and to the server resources are kept in a special 'pool'. The next time the application requires database services, one of the available connections in the 'pool' is assigned to it. Thus, enabling Pooling Connections results in faster database connections.

If connection pooling is not enabled, when a database and all its associated files are closed, the connection and server resources are ended (i.e., in the case of DataGate, the DataGate Job ends). When the same application needs the database services again, a new connection and corresponding job has to be established. This overhead wastes resources and slows down the application

The **PoolingTimeout** property is also used in conjunction with the **CreateName** and **UpdateName** methods.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB | CreateName Method | UpdateName Method](#)

Port Property

Gets or sets the TCP port number used by the database server for TCP/IP-based transport (default 5042).

```
database.Port = <integerVal>
```

Property Value

<integerVal> is of type **Integer* and contains the TCP port number that is to be used for TCP/IP-based transport.

Remarks

The **Port** property is also used in conjunction with the **CreateName** and **UpdateName** methods.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | Database Class Methods | DCLDB | CreateName Method | UpdateName Method](#)

Server Property

Gets or sets the database server host machine name or address.

```
database.Server = <stringVal>
```

Property Value

<stringVal> is of type **String* and contains either a DNS host name or a TCP/IP address in "dotted decimal" notation.

Remarks

The **Server** property is also used in conjunction with the **CreateName** and **UpdateName** methods.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | Database Class Methods | DCLDB | CreateName Method | UpdateName Method](#)

Text Property

Gets or sets a description of the database name.

```
database.Text = <stringVal>
```

Property Value

<stringVal> is of type **String* and contains a description of the database name.

Remarks

The **Text** property is also used in conjunction with the **CreateName** and **UpdateName** methods.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | Database Class Methods | DCLDB | CreateName Method | UpdateName Method](#)

User Property

Gets or sets the authenticated user profile used to authorize the current database connection.

```
database.User = <stringVal>
```

Property Value

<stringVal> is of type **String* containing the authenticated user profile used to authorize the current database connection.

Remarks

Authentication is the process of verifying that a user with a unique name is the actual user that the name implies. This verification process is also applied to applications running in the security context of a user. Typically, authentication is implemented by giving the user a password.

The user profile described by the **User** property is the one used to authenticate the database connection on the server platform, not the client.

The **User** property is also used in conjunction with the **CreateName** and **UpdateName** methods.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | Database Class Methods | DCLDB | CreateName Method | UpdateName Method | Password Property](#)

DataGateException Class

The **DataGateException** class contains the information for exception errors occurring during program execution.

For a list of members of this class, see [DataGateException Members](#).

ASNA.VisualRPG.Runtime

```
ASNA.VisualRPG.Runtime.DataGateException <Serializable>
```

DataGateExceptions : [ASNA.VisualRPG.Runtime.AvrException](#)

Thread Safety

This type is safe for multithreaded operations.

Remarks

A **DataGateException** class provides for access to the current exception errors during program execution.

Requirements

Namespace: ASNA.VisualRPG.Runtime

Assembly: ASNA Visual RPG Runtime (in ASNA.VisualRPG.Runtime.dll)

Platforms: Windows 2000, Windows XP Professional, Windows Server 2003 family.

See Also

ASNA.VisualRPG.Runtime | DataGateException Members

DataGateException Members

Public Methods

 **ToString** Returns a string representation of the current exception.

See Also

ASNA.VisualRPG.Runtime | DataGateException Class

DataGateException Methods

ToString Method

ToString returns a string representation of the current exception.

```
ToString ()
```

Remarks

The string returned by the **ToString** method contains the declared file name (DclFileName) and the errors that caused (CausingException) an exception during application execution. CausingException is of type ASNA.DataGate.Common.dgException.

See Also

ASNA.VisualRPG.Runtime | DataGateException Class | DataGateException Members | DCLDISKFILE

DBFile Class

The **DBFile** class defines a reference to a system-level database file object and its members. It supports /O operations to a database file.

For a list of all members of this type, see DBFile Members.

ASNA.VisualRPG.Runtime

```
ASNA.VisualRPG.Runtime.DBFile <Serializable>  
Syntax
```

Thread Safety

This type is safe for multithreaded operations.

Requirements

Namespace: ASNA.VisualRPG.Runtime

Assembly: ASNA Visual RPG Runtime (in ASNA.VisualRPG.Runtime.dll)

Platforms: Windows 2000, Windows XP Professional, Windows Server 2003 family.

See Also

ASNA.VisualRPG.Runtime | DBFile Members | DCLDISKFILE

DBFile Members

Public Constructor

 **DBFile** This can not be called directly in code, instantiate using DCLDISKFILE.

Public Properties

 **FormatName** Gets the name of the record format last read or written.

 **RecCount** Gets the current number of non-deleted records in the file.

 **RecNum** Gets the current RRN for the last input operation to a file.

Public Fields

 **BlockingFactor** Contains the client/server record blocking size.

 **FilePath** Contains the path of the file.

 **IsFormatID** Contains **True** if the opened file's format ID is equal to the described file's format ID.

 **MemberName** Contains the name of the file member to be opened when the file is opened.

 **QryFileName** Contains the name of the query file created for the **QrySelect** and **QryKeyFlds** parameters.

 **QryKeyFlds** Contains the name of one or more key fields used to arrange the query records.

 **QrySelect** Contains the selection expression used to determine which records are to be made available for processing.

 **ShareType** Contains a string defining how a file will be shared.

 **WaitRec** Contains the waiting period for a record.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DCLDISKFILE

DBFile Properties

FormatName Property

Gets the name of the record format last read or written.

```
<stringVal> = dbfile.FormatName
```

Property Value

<stringVal> of type *String containing the name of the record format last read or written.

Remarks

The iSeries database engine supports a logical file type called "multiformat". This file is based upon two or more physical files. The contents of the **FormatName** property are useful when reading from a multiformat file. When reading sequentially through a multiformat file, you can determine the format last read from the content of **FormatName**.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members

RecCount Property

The RecCount property gets the current number of non-deleted records in the file.

```
<integerVal> = dbfile.RecCount
```

Property Value

<integerVal> is of type *Integer8 and contains the current number of non-deleted records in the file.

Remarks

The file must first be Opened before the record count is available.

If the file is a logical file, the record count returned is that of the base physical file.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | DCLDISKFILE | OPEN | RecNum Property

RecNum Property

Gets the current RRN for the last input operation to a file.

```
<integerVal> = dbfile.RecNum
```

Property Value

<integerVal> is of type *Integer8 and contains the current RRN for the last input operation to a file.

Remarks

The Relative Record Number (RRN) is relative to 1.

RRN is not supported by Microsoft's SQL Server database engine.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | DCLDISKFILE | OPEN | RecCount Property

DBFile Fields

BlockingFactor Field

BlockingFactor contains the client/server record blocking size.

```
dbfile.BlockingFactor Type(*IConnteger)
```

Remarks

BlockingFactor *must* be set prior to opening the file (see OPEN command).

If **BlockingFactor** is not set, the **NetblockFactor** parameter value specified in the DCLDISKFILE command is used.

BlockingFactor determines if and how many database records will be blocked across the Network for better performance when reading groups of records. Network blocking significantly reduces the network burden of transferring records between server and client machines, and this is demonstrated by the substantial increase of data throughput due to the reduced number of TCP/IP packet requests.

The database file must be declared in the **DCLDISKFILE** command as **Type(*Input)** and **AddRec(*NO)** before network blocking can be used.

The blocking factor is expressed as the maximum number of records to block. Valid values are:

-1 The blocking factor will be computed automatically (6K/record length).

0 or 1 One record at a time will be transported across the network.

nnn nnn records at a time will be blocked.

When a file is network blocked, the block is delivered when the database engine receives the first read request. This means the client will wait to receive the block before completing the processing of the request. The subsequent read requests will retrieve the records from the cached block held by the client. When the cache is empty, the next block is retrieved when the database engine receives the next read request.

It is prudent to set the blocking factor to a value less than or equal to the number of records being processed at one time. For instance, if your application is adding 50 records at a time to a DataGrid, then the blocking factor should not be greater than 50.

When you employ SETRANGE & READ/READP and READRANGE & READ/READP idioms, the database engine will place no more records in the block than needed to satisfy the range.

Hint: If you are using a **progress bar** to show the progress of reading the next *n* records, make the blocking factor a fraction of the total records to be processed. For Example, if processing 50 records at a time, set BlockingFactor to 10.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE | OPEN

FilePath Field

FilePath contains a string that specifies the path of the file.

```
dbfile.FilePath Type (*String)
```

Remarks

FilePath must be set prior to opening a file (see OPEN command).

If **FilePath** is not set, the filepath of the **File** parameter specified in the DCLDISKFILE command is used.

Setting the **FilePath** prior to opening a file is equivalent to the iSeries process called **override database file**.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE | OPEN

IsFormatID Field

IsFormatID contains **True** if the opened file's format ID is equal to the described file's format ID.

```
dbfile.IsFormatID Type (*Boolean)
```

Remarks

When the **ChkFmtID** parameter specified in the DCLDISKFILE command is set to ***Yes** or ***Same**, **IsFormatID** returns **True** if the format ID of the file specified in the DcIDiskfile's **FileDesc** parameter is the same as the file opened at run-time.

The Visual RPG for .NET compiler stores the format of each file in the assembly when a program is compiled. At run-time, AVR for .NET can then compare the stored format ID with the format ID of the currently-opened file.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE

MemberName Field

MemberName contains the name of the file member to be opened when the file is opened.

```
dbfile.MemberName Type (*String)
```

Remarks

MemberName must be set prior to opening the file (see OPEN command).

If **MemberName** is not set, the value of the **Mbr** parameter specified in the DCLDISKFILE command is used.

In addition to specifying a named file member, **MemberName** may also be set to the following special values:

*FIRST will access the member with the oldest creation date.

*FILE will access the member name with the same name as the file name.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Class Members | DCLDISKFILE | OPEN

QryFileName Field

QryFileName contains the name of the query file created for the **QrySelect** and **QryKeyFlds** parameters.

```
dbfile.QryFileName Type (*String)
```

Remarks

QryFileName must be set prior to opening the file (see OPEN command).

If **QryFileName** is not set, the value of the **QryFileName** parameter specified in the DCLDISKFILE command is used.

QryFileName specifies the name of the query file created for DCLDISKFILE's **QrySelect** and **QryKeyFlds** parameters. If *NAME is specified, which is the default, the file name given in the DCLDISKFILE command is used.

If the database access with this file is an iSeries database, then the **QryFileName** is used both on the **OVRDBF** and the **OPNQRYF** commands used when opening the file.

For a DataGate for Windows database, a temporary logical file is created in QTemp with the **QryFileName** as its name. You can use the special name ***UNIQUE** to guarantee that there will be no conflicts with other files residing in QTemp. You will receive errors if there are duplicate file names in QTemp.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE | QryKeyFlds Field | QrySelect

QryKeyFlds Field

QryKeyFlds contains the name of one or more key fields used to arrange the query records.

```
dbfile.QryKeyFlds Type (*String)
```

Remarks

QryKeyFlds must be set prior to opening the file (see OPEN command).

If **QryKeyFlds** is not set, the value of the **QryKeyFlds** parameter specified in the DCLDISKFILE command is used.

Each key field is composed of a field name followed by optional attributes that determine the sorting order (ascending descending, or absolute value for numeric fields).

*FILE - is the default, which specifies that the value of the **QryKeyFlds** parameter specified in the **DCLDISKFILE** command is used.

*NONE - specifies that the file has no keys.

A string containing key field name(s) and corresponding sort attributes that determine the sorting order (ascending, descending, or absolute value for numeric fields).

Syntax: The syntax for the QryKeyFlds string is:

```
(KeyFieldName1 Attributes KeyFieldName2 Attributes ... KeyFieldNameN
Attributes )
```

KeyFieldName must be a valid field name in the file. Attributes consist of the following: *ASCEND, *DESCEND, or *ABSVAL. You can have as many attributes as you want after a field name as long as they conform to the rules below.

Each key field name and attribute must be separated by at least one blank.

*ASCEND and *DESCEND cannot be applied to the same field.

*ABSVAL only applies to numeric fields.

If there are any errors in the key fields' expression, they will not be detected until the file is opened at run-time.

Use QrySelect in conjunction with **QryKeyFlds** to specify the selection criteria indicating which records to select from a file. Use QryFileName to specify the name of the temporary query file created. Selection expressions are evaluated during file open time. Therefore, a runtime error will be generated during the opening of the file when a selection expression error is encountered.

Example

```
myFile.QryKeyFlds = "CMActive *DESCEND CMState *ASCEND CMCity *ASCEND CMFax  
*ABSVAL *DESCEND"
```

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE | QryFileName Field | QrySelect Field

QrySelect Field

QrySelect contains the selection expression used to determine which records are to be made available for processing.

```
dbfile.QrySelect Type (*String)
```

Remarks

QrySelect must be set prior to opening the file (see OPEN command).

If **QrySelect** is not set, the value of the **QrySelect** parameter specified in the DCLDISKFILE command is used.

QryKeyFlds is used in conjunction with **QrySelect** to specify ordering of the selected records. Use QryFileName to specify the name of the temporary query file created.

When using QrySelect, please Note:

*NONE (default) - there is no selection expression available for processing.

Selection expressions are evaluated during file open time. Therefore, a run-time error will be generated during the opening of the file when a selection expression error is encountered.

The Select statement syntax can vary depending upon the syntax supported by the target database engine. Each key field is composed of a field name followed by optional attributes that determine the sorting order (ascending, descending, or absolute value for numeric fields).

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE | QryFileName Field | QryKeyFlds Field

ShareType Field

ShareType contains a string defining how a file will be shared when opened.

```
dbfile.ShareType Type (*String)
```

Field Values

Exclusive - File is opened for update, exclusive lock.

This lock can be held by exactly one database session. No other sessions are granted any lock type on the object when an **Exclusive** lock is held, and conversely, the **Exclusive** lock will not be granted if any other type of lock is held in another session. The granted session has exclusive system access to the object.

ExclusiveRead - File is opened for update, other jobs may read.

When this lock is granted, other database sessions may access the object for read-only access, but no other sessions are granted any type of exclusive access. This is appropriate to use when the object should be restricted from any access other than read access.

ShareUpdate - File is opened for update, other jobs may read or update.

When this lock is granted, the object can be accessed for updating or reading by other sessions. When held, other sessions will be granted **ShareRead** or **ShareUpdate** locks for the same object. This lock prevents **Exclusive** access to the object. This lock type is appropriate when the target object is to be modified in the current session, and read and change access in other sessions should be allowed.

ShareNoUpdate - File is opened for read, other jobs may read.

When this lock is granted, the object can be accessed only for reading by other sessions. When held, other sessions will be granted **ShareRead** or **ShareNoUpdate** locks for the same object. This lock prevents exclusive access and update access to the object. This lock type is appropriate when the target object should not be modified by any session, but all sessions should be allowed to read the object.

ShareRead - File is opened for read, other jobs may read or update.

When this lock is granted, the object can be accessed non-exclusively by other sessions. When held, other sessions will be granted **ExclusiveRead**, **ShareUpdate**, **ShareRead** or **ShareNoUpdate** locks for the same object, so long as the lock requested does not conflict with another session's held lock. Alone, this lock prevents only **Exclusive** access. It is appropriate when the target object should not be locked exclusively by any one session.

Remarks

ShareType must be set prior to opening the file (see **OPEN** command).

If **ShareType** is not set, the value of the **ShareType** parameter specified in the **DCLDISKFILE** command is used.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | DCLDISKFILE

WaitRec Field

WaitRec contains the waiting period for a record.

```
dbfile.WaitRec Type (*Integer)
```

Field Value:

Integer values containing the time to wait for a record.

Remarks

WaitRec must be set prior to opening the file (see OPEN command).

If **WaitRec** is not set, the value of the **WaitRec** parameter specified in the DCLDISKFILE command is used. The DclDiskFile parameter values are:

*FILE (default) - indicates that the properties of the file are used to determine how long it waits for the record.

*IMMEDIATE - it won't wait for a record.

*NOMAX - it will wait indefinitely.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DBFile Members | OPEN | DCLDISKFILE

MemoryFile Class

The MemoryFile class maintains an ADO.NET Dataset as though it were a collection of "flat files". See MemoryFile Members.

```
ASNA.VisualRPG.Runtime.MemoryFile
```

Thread Safety

This type is safe for multithreaded operations.

Remarks

A MemoryFile class simply defines a memory file that results in an ADO.NET Dataset object. The file can be created and accessed similar to an externally described database file even though it is cached in memory. Currently, the memory file description is externally described as an iSeries/400 or DataGate Windows database file.

Requirements

Namespace: ASNA.VisualRPG.Runtime

Assembly: ASNA Visual RPG Runtime (in ASNA.VisualRPG.Runtime.dll)

Platforms: Windows 2000, Windows XP Professional, Windows Server 2003 family.

See Also

ASNA.VisualRPG.Runtime | MemoryFile Members

MemoryFile Members

Public Properties

	DataSet	Gets an ASNA DataGate Client ADO.NET DataSet. This property is of type ASNA.DataGate.Client.AdgDataSet.
	SfIRRN	Gets an integer value containing the RRN number for the current subfile record held by the MemoryFile .

Protected Fields

	sfIRRN	Contains a read-only integer value containing the RRN number for the current subfile record held by the MemoryFile .
---	---------------	---

See Also

ASNA.VisualRPG.Runtime | MemoryFile Class | DCLMemoryFile

MemoryFile Properties

DataSet Property

Gets the dataset held by the **MemoryFile**.

```
<dataset> = memoryfile.DataSet
```

Returns

<dataset> of type ASNA.DataGate.Client.AdgDataSet that is held by the MemoryFile.

Remarks

The **DataSet** property is a **DataGate Client Dataset** derived from **System.Data.DataSet**. **DataSet** may be assigned to any data source that supports a **System.Data.DataSet** object. The **DataSet** may also be assigned to a **System.Data.DataSet** object.

See Also

ASNA.VisualRPG.Runtime | MemoryFile Class | DCLMEMORYFILE

SfIRRN Property

Gets an integer value containing the RRN number for the current subfile record held by the **MemoryFile**.

```
<integerVal> = memoryfile.SfIRRN
```

Property Value

<integerVal> is of type *Integer and contains the RRN number for the current subfile record held by the **MemoryFile**.

See Also

ASNA.VisualRPG.Runtime | MemoryFile Class | DCLMEMORYFILE

MemoryFile Fields

sf1RRN Field

Contains a read-only integer value containing the RRN number for the current subfile record held by the **MemoryFile**.

```
memoryfile.sf1RRN Type (*Integer)
```

Field Value

Integer value containing the relative record number (RRN) for the current subfile record held by the **MemoryFile**.

See Also

ASNA.VisualRPG.Runtime | MemoryFile Class | DCLMEMORYFILE

MessageFileReader Class

The **MessageFileReader** class supports a message file structure which is composed of a individual text messages identified by a unique prefix and message number (ID) combination. Typical use for message files is to easily provide multiple language support, error message text, and for externalizing the text property of controls.

ASNA.VisualRPG.Runtime

```
ASNA.VisualRPG.Runtime.MessageFileReader <Serializable>  
Syntax
```

Thread Safety

This type is safe for multithreaded operations.

Remarks

ASNA Message Files are not intended to be a replacement for Visual Studio's Dynamic properties.

An ASNA Message File is a compilation of uniquely identified messages that can be retrieved at runtime. The **MessageFileReader** is instantiated with the path of the compiled message file. The **GetMessageText** and **GetSecondMessageText** functions return first and second level text strings.

To create a message, you can use any text editor - including the IDE - to create a text file. Each message must contain the following information. The information must be specified in this order.

Message Prefix.....3 characters (case insensitive)

Message Number(ID).....From 1 - 65535

Message severity.....From 00 - 99

First level text.....Any alphanumeric data

Second level text must be entered on the line following the message details listed above. It also allows any combination of letters and numbers.

To specify substitution text in either first- or second-level message text, enter the ampersand character (&) followed immediately by a one or two digit number. If you provide more substitution variables than the number of replacement identifiers (&1...&99), the extra variables are ignored. If you provide fewer substitution variables than the number of replacement identifiers, the identifiers are removed and no text is replaced.

Example

Prefix	Msg#	Severity	Text
VRT	2034	50	Invalid array index value: &1&nExpected array index is between 1 and &2
VRT	2035	50	Runtime error occurred in Program &1 at Line &2:
// Below are for setting button text property at runtime			
FRM	10	00	E&&xit

3 Character Prefix

The 3-character prefix will help you in associating, or identifying text in a message file. This prefix can be any 3 characters, and you can have more than 1 prefix in a message file. For Example, you may want to use a prefix of ERR for all your error messages, or CTL for all of your Control messages, etc. The prefix is case-insensitive.

Message Number (ID)

The number can either be a hexadecimal number, or a decimal number.

If you enter the number in hexadecimal, then valid numbers can be anywhere between 0x0000 and 0xFFFF.

If you enter the number in decimal, then valid numbers can be anywhere between 0 and 65535.

Hint: The format used should remain consistent throughout the message file. For Example, to avoid duplication of numbers, do not use decimal numbers and hexadecimal numbers in the same message text file.

Both the prefix and number are used together to identify a particular message text, so you can have duplicate numbers assigned to different prefixes.

Any of the following entries are correct, and will generate the same number. Note that if you have duplicate numbers assigned to the same prefix, a Warning showing the prefix and message number will display when you compile the message file.

```
<0x00F0
0xF0
240
00240
```

Severity

This is a 2 digit numeric number that you can use to denote a level of severity to the message that you have defined. It is for your use only and is optionally returned to your application by the GetMessageText method.

Message Text

Within the message text, you can specify access keys, the text to be retrieved, replacement text and 2 levels of message text (Level 1 and Level 2), as explained below.

Two levels of message text

Level 1 text is usually a brief message text. It follows the severity and stops at the end of the line.

Level 2 is extended message information. It is entered on the line immediately following Level 1 message text. All Level 2 message text must be indented by at least one space or TAB. You may start as many new lines as you wish for Level-2 message text as long as each line is indented as noted.

Replacement Text

You can specify any number of replacement text tokens for each message text. Replacement text is specified by a single ampersand, followed by the replacement number; i.e., &1, &2, ... , &99.

The replacement text is replaced at run-time by the **GetMessageText** method.

Keyboard Navigation Keys

Within the message text, you can specify access keys by entering two ampersands (&&) in front of the character you wish to assign to the access key. Typically for use with menu and button text; for Example, "E xit" would be in the message text as "E&&xit".

If you actually want an ampersand to display as part of the text, merely enter "&&". If you want 2 ampersands to display, then you will need to enter "&&&&", and so on.

Display Text on Next Line:

You can specify an (&n) anywhere in the message text for a new line just prior to the text you want to display on the next line.

Compiling the message file:

When all messages have been entered, execute the CRTMSGF.EXE file passing the text file name as the first argument and the compiled message file name as the second argument. The CRTMSGF.EXE file is located in the "%Program Files\Common Files\ASNA Shared" folder.

Requirements

Namespace: ASNA.VisualRPG.Runtime

Assembly: ASNA Visual RPG Runtime (in ASNA.VisualRPG.Runtime.dll)

Platforms: Windows 2000, Windows XP Professional, Windows Server 2003 family.

See Also

ASNA.VisualRPG.Runtime | MessageFileReader Methods | GetMessageText Method | GetSecondMessageText Method

MessageFileReader Members

Public Methods

 GetMessageText	Overloaded. Returns the first-level message text from the specified prefix and id number, with additional overload options.
 GetSecondMessageText	Overloaded. Returns the second-level message text from the specified prefix and id number, with optional replacement values.

See Also

ASNA.VisualRPG.Runtime | MessageFileReader Class

MessageFileReader Methods

GetMessageText Method

Overloaded. **GetMessageText** returns message first-level data from an ASNA Message File. See **MessageFileReader Class** for a description on how to establish a Message File.

Overload List

Returns the message text from the specified prefix and number.

```
BegFunc type( *String ) Access( *Public )
    dclSRParm prefix Type( *string ) By( *Value )
    dclSRParm id      Type( *Integer2 ) By( *Value )
```

Returns the message text from the specified prefix and id number, and sets the severity byte equal to the severity level of the requested message.

```
BegFunc type( *String ) Access( *Public )
    dclSRParm prefix      Type( *string ) By( *Value )
    dclSRParm id          Type( *Integer2 ) By( *Value )
    dclSRParm severity    Type( *Byte ) By( *Reference )
```

Returns the message text from the specified prefix and id number, and assigns the replacement text for the symbolic placeholders put in the message file.

```
BegFunc type( *String ) Access( *Public )
    dclSRParm prefix      Type( *string ) By( *Value )
    dclSRParm id          Type( *Integer2 ) By( *Value )
    dclSRParm replacementValues Type( *String ) By( *Value ) Rank( 1 )
```

Returns the message text from the specified prefix and id number, sets the severity byte equal to the severity level of the requested message, and assigns the replacement text for the symbolic placeholder put in the message file.

```
BegFunc type( *String ) Access( *Public )
    dclSRParm prefix      Type( *string ) By( *Value )
    dclSRParm id          Type( *Integer2 ) By( *Value )
    dclSRParm severity    Type( *Byte ) By( *Reference )
    dclSRParm replacementValues Type( *String ) By( *Value ) Rank( 1 )
```

Parameters

Prefix

The 3 character string prefix under which the requested message is stored.

Id

The numeric value under which the requested message is stored.

Severity

The out parameter is set to the severity level of the requested message.

Replacement Values

An array of objects meant to replace the symbolic placeholders placed in the message file.

Example

Using ANSA.VisualRPG.Runtime

```

DclFld MsgReader      Type( MessageFileReader )
DclFld MsgSeverity     Type( *Byte )
DclFld MsgText1       Type( *String )    // 1st level
DclFld MsgText2       Type( *String )    // 2nd level
DclArray ReplText     Type( *Object) Dim(20)
DclFld MsgFileName    Type( *String) Inz("MessFile.msg")
ReplText{0} = "Hello"
ReplText{1} = "World"

MsgReader = *New MessageFileReader(MsgFileName)

// no replacement, no severity
MsgText1 = MsgReader.GetMessageText( "ENG", 1200 )

// no replacement, return severity by reference
MsgText1 = MsgReader.GetMessageText( "ENG, 1200, *ByRef MsgSeverity )

// no severity, replace text
MsgText1 = MsgReader.GetMessageText( "ENG, 1200, ReplText )

//return severity by reference, replace text
MsgText1 = MsgReader.GetMessageText( "ENG", 1200, *ByRef MsgSeverity,
ReplText )

```

See Also

ASNA.VisualRPG.Runtime | MessageFileReader Class | MessageFileReader Members |
GetSecondMessageText Method

GetSecondMessageText Method

Overloaded. **GetSecondMessageText** returns second-level message data from an ASNA message file. See MessageFileReader Class for a description on how to establish a Message File.

Overload List

Returns the second level message text from the specified prefix and number.

```

BegFunc type( *String ) Access( *Public )
  dclSRParm prefix Type( *string ) By( *Value )
  dclSRParm id      Type( *Integer2 ) By( *Value )

```

Returns the second level message text from the specified prefix and id number, and assigns the replacement text for the symbolic placeholders put in the message file.

```

BegFunc type( *String ) Access( *Public )
  dclSRParm prefix      Type( *string ) By( *Value )
  dclSRParm id          Type( *Integer2 ) By( *Value )
  dclSRParm replacementValues Type( *String ) By( *Value ) Rank( 1 )

```

Parameters

Prefix

The 3 character string prefix under which the requested message is stored.

id

The numeric value under which the requested message is stored.

replacementValues

An array of objects meant to replace the symbolic placeholders placed in the message file.

Example

Using ANSA.VisualRPG.Runtime

```
DclFld MsgReader      Type( MessageFileReader )
DclFld MsgSeverity    Type( *Byte )
DclFld MsgText1       Type( *String )    // 1st level
DclFld MsgText2       Type( *String )    // 2nd level
DclArray ReplText     Type( *Object) Dim(20)
DclFld MsgFileName    Type( *String) Inz("MessFile.msg")
ReplText{0} = "Hello"
ReplText{1} = "World"

MsgReader = *New MessageFileReader(MsgFileName)

// no replacement
MsgText2 = MsgReader.GetSecondMessageText( "ENG", 1200 )

// replace text
MsgText2 = MsgReader.GetSecondMessageText( "ENG", 1200, ReplText )
```

See Also

ASNA.VisualRPG.Runtime | MessageFileReader Class | MessageFileReader Members | GetMessageText Method

PrintFile Class

The PrintFile class contains all the information for printing in a program.

ASNA.VisualRPG.Runtime

```
ASNA.VisualRPG.Runtime.PrintFile <Serializable>
Syntax
```

Thread Safety

This type is safe for multithreaded operations.

Remarks

A PrintFile class simply defines a DataGate printfile for printing at run-time. At compile time, the description of the file is read in with its record format. The file name and each record format name must be unique.

Print Files are always opened for sequential output operation.

Requirements

Namespace: ASNA.VisualRPG.Runtime

Assembly: ASNA Visual RPG Runtime (in ASNA.VisualRPG.Runtime.dll)

Platforms: Windows 2000, Windows XP Professional, Windows Server 2003 family.

See Also

ASNA.VisualRPG.Runtime | PrintFile Members | DCLPRINTFILE

PrintFile Members

Public Constructor

 **PrintFile** This can not be called directly in code, instantiate using DCLPRINTFILE.

Public Properties

 Collate	Gets or sets whether collation will be used when printing multiple copies.
 Copies	Gets or sets the number of copies to print.
 CursorY	Gets the position, in Print Units, that the printer is from the top of the page.
 Duplex	Gets or sets whether the printer file is to print duplex (prints on both sides of the paper). This property is of type <code>System.Drawing.Printing.Duplex</code> .
 FooterSize	Gets the size of the page footer.
 FormName	Gets or sets the name of the customized page size to use.
 InOverflow	Gets the overflow state of the printer file. This property is of type <code>*Boolean</code> . When the value is True , the print file is within the overflow area, otherwise False .
 IsOpen	Gets the open state of the printer file. This property is of type <code>*Boolean</code> . When the value is <code>*True</code> , the file is Open. (Same as the <code>%Open</code> built-in function.)
 LastException	Gets the last exception thrown for this file. This property is of type <code>ASNA.VisualRPG.Runtime.AvrException</code> .
 ManuscriptPath	Gets or sets the file and path name for the manuscript.
 Orientation	Gets or sets the print orientation (Horizontal or Vertical). This property is of type ASNA.DataGate.Common.PaperOrientation .
 PageLength	OBSOLETE: This property is marked obsolete and is referenced only for backward compatibility. This has been replaced with <code>PrintableLength</code> property.
 PageWidth	OBSOLETE: This property is marked obsolete and is referenced only for backward compatibility. This has been replaced with <code>PrintableWidth</code> property.
 PaperLength	Gets or sets the physical length of the page. Valid only with a continuous-form printer.
 PaperSize	Gets or sets the paper size. This property is of type <code>System.Drawing.Printing.PaperKind</code> .
 PaperSource	Gets or sets the printer tray that will be used. This property is of type <code>System.Drawing.Printing.PaperSourceKind</code> .
 PaperWidth	Gets or sets the physical width of the paper. Valid only with a continuous-form printer.

 Preview	Gets or sets whether the print preview dialog is to be displayed before printing. This property is of type ASNA.DataGate.Common.AccessMode .
 PrintableLength	Gets the length of the printer's printable area.
 PrintableWidth	Gets the width of the printer's printable area.
 Printer	Gets or sets the full path name of the printer to be used.
 PrintSetup	Gets or sets whether the print setup dialog is to be displayed before printing.
 ProcessMacro	Gets or sets whether DataGate print engine is to search for page numbering "macros" and replace them with appropriate data before sending the output to the printer.
 Quality	Gets or sets the print quality. This property is of type System.Drawing.Printing.PrinterResolutionKind .
 ReportName	Gets or sets the name of the report as seen on the printer spool. If not assigned, the default name is the name given when the print file was declared.
 Scale	Gets or sets the percentage factor by which the printed document is to be scaled.
 TTOption	Gets or sets how true type fonts are to be printed. This property is of type ASNA.DataGate.Common.PrintTrueType .
 UseColor	Gets or sets whether to print in color when a color printer is specified.
 YResolution	Gets or sets the Y-axis resolution (in dot per inch) when printing graphics.

Public Fields

 DclPrintFileName	Contains the declared print file's <i>Name</i> .
 FilePath	Contains a string that specifies the path of the file.
 IsFormatID	Contains True if the opened (run-time) file's format ID is equal to the declared (compile-time) file's format ID.

Protected Fields

 isOpen	Contains the open state of the printer file. When the value is True , the file is open.
 lastException	Contains the last exception thrown for this file. This is a protected public field of type ASNA.VisualRPG.Runtime.AvrException .

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | DCLPRINTFILE

PrintFile Properties

Collate Property

Gets or sets whether collation will be used when printing multiple copies.

```
printfile.Collate = <booleanVal>
```

Property Value

booleanVal of type **Boolean*.

True indicates the printfile is to be collated.

False indicates if the printfile is **not** to be collated.

Remarks

Must be set prior to opening the printer file (see **OPEN** command).

Setting this property to **True** will allow multiple copies to be collated. However, this property is applicable only when printing multiple copies, or when the **Copies** property is >1.

If **Collate** is not set, the collate setting specified for the print file will be used. (If **DEFAULT* is specified for the print file, then the collate setting specified for the printer will be used).

The collation is handled by the printer, rather than software. If the printer itself does not support collate, then no collation will occur.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE | Copies Property | **OPEN**

Copies Property

Gets or sets the number of copies to print.

```
printfile.Copies = <integerVal>
```

Property Value

integerVal

**Integer4*. Contains the number of copies to print.

Remarks

Must be set prior to opening the printer file, or prior to **FEOD** with reset (see **OPEN** and **FEOD** command).

If **Copies** is not set, the number of copies specified for the print file will be used. If **DEFAULT* is specified for the print file, then the default number of copies specified for the printer will be used.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | **OPEN** | **FEOD** | DCLPRINTFILE

CursorY Property

Gets the position, in Print Units, that the printer is from the top of the page.

```
<integerVal> = printfile.CursorY
```

Property Value

integerVal

*Integer4. Contains the position, in Print Units, that the printer is from the top of the page.

Print Units are measured in 1/100th of a millimeter.

To get the measurement in millimeters, divide by 100.

To get the measurement in centimeters, divide by 1,000.

To get the measurement in inches, divide by 2,540

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

Duplex Property

Gets or sets how duplex (double-sided) printing will be used.

```
printfile.Duplex = System.Drawing.Printing.Duplex.<setting>
```

Property Value

setting

Is of type System.Drawing.Printing.Duplex.

Also note that not all of the possible values for this enumeration may be valid for your printer. Each printer has a limited set of acceptable values. It is up to the programmer to select the value adequate to the actual printer used to render the output of the print file.

Remarks

Must be set prior to opening the printer file (see OPEN command).

If **Duplex** is not set, the duplex type specified for the print file will be used. (If *DEFAULT is specified for the print file, then the duplex option specified for the printer will be used).

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE | System.Drawing.Printing.Duplex Enumeration

FooterSize Property

Gets the size of the page footer.

```
<integerVal> = printfile.FooterSize
```

Property Value

integerVal

*Integer4. Contains the size of the page footer.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

FormName Property

Gets or sets the name of the customized page size to use.

```
printfile.FormName = <stringVal>
```

Property Value

stringVal

*String containing the name of the customized page size to use (see PaperSize).

Remarks

Must be set prior to opening the printer file (see OPEN command).

If **FormName** is not set, the form name for the print file will be used.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | PaperSize Property | OPEN | DCLPRINTFILE

InOverflow Property

Gets a boolean value indicating whether the printer has entered the overflow area of the page. **True** if within the overflow area; otherwise **False**.

```
<booleanVal> = printfile.InOverflow
```

Property Value

booleanVal of type *Boolean.

True if the printer has entered the overflow area.

False if the printer is not within the overflow area.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE

IsOpen Property

Gets a boolean value indicating if the file is currently open for I/O operations.

```
<booleanVal> = database.IsOpen
```

Property Value

booleanVal of type *Boolean.

True indicates the file is currently open for I/O operations.

False indicates the file is currently **not** open for I/O operations.

Remarks

Refer to the CLOSE and OPEN commands.

IsOpen is equivalent to the %OPEN built-in function.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | DCLDISKFILE | CLOSE | OPEN | %OPEN

LastException Property

Gets the last exception thrown for this file.

```
<avrexception> = printfile.LastException
```

Property Value

avrexception

Is of type **ASNA.VisualRPG.Runtime.AvrException** containing the last exception thrown for the file.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

ManuscriptPath Property

Gets or sets the path and file name for the manuscript.

```
printfile.ManuscriptPath = <stringVal>
```

Property Value

stringVal

*String containing the path and file name for the manuscript.

Remarks

Must be set prior to opening the printer file.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE | Renderer Command Options

Orientation Property

Gets or sets the print orientation.

```
printfile.Orientation = ASNA.DataGate.Common.PaperOrientation.<value>
```

Property Value

value

Is of type **ASNA.DataGate.Common.PaperOrientation** and is the enumeration value indicating the print orientation.

Remarks

Must be set prior to opening the printer file (see OPEN command).

The page orientation can be set to either Portrait or Landscape. If **Orientation** is not set, the orientation specified for the print file will be used. (If *DEFAULT is specified for the print file, then the default orientation of the printer will be used.)

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE | ASNA.DataGate.Common.PaperOrientation Enumeration

PaperLength Property

Gets or sets the physical length of the page.

```
printfile.PaperLength = <integerVal>
```

Property Value

integerVal

*Integer4 and contains the physical length of the page in the printer. This is valid only with a continuous-form printer.

Remarks

Must be set prior to opening the printer file (see OPEN command).

If **PaperLength** is not set, the paper length specified for the print file will be used. (If *DEFAULT is specified for the print file, then the default paper length, based upon the orientation of the paper in the printer will be used. These values are in tenths of a millimeter.

Print Units are measured in 1/100th of a millimeter.

To get the measurement in millimeters, divide by 100.

To get the measurement in centimeters, divide by 1,000.

To get the measurement in inches, divide by 2,540

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | Orientation Property | OPEN | DCLPRINTFILE

PaperSize Property

Gets or sets the paper size to be used when printing.

```
printfile.PaperSize = System.Drawing.Printing.PaperKind.<value>
```

Property Value

value

System.Drawing.Printing.PaperKind and is the enumeration value indicating the paper size.

Please note that not all of the possible values for this enumeration may be valid for your printer. Each printer has a limited set of acceptable values. It is up to the programmer to select the value adequate to the actual printer used to render the output of the print file.

Remarks

Must be set prior to opening the printer file (see OPEN command).

If **PaperSize** is not set, the paper size specified for in the print file will be used. (If *DEFAULT is specified for the print file, then the default paper size, based on the orientation of the printer, will be used.)

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | Orientation Property | DCLPRINTFILE | System.Drawing.Printing.PaperKind Enumeration

PaperSource Property

gets or sets the printer tray with will be used when printing.

```
printfile.PaperSource = System.Drawing.Printing.PaperSourceKind.<value>
```

Property Value

value

System.Drawing.Printing.PaperSourceKind and is the enumeration value indicating the source of the paper tray.

Please note that not all of the possible values for this enumeration may be valid for your printer. Each printer has a limited set of acceptable values. It is up to the programmer to select the value adequate to the actual printer used to render the output of the print file.

Remarks

Must be set prior to opening the printer file (see OPEN command).

If **PaperSource** is not set, the paper source specified for the print file will be used. (If *DEFAULT is specified for the print file, then the default orientation of the printer will be used.)

Printer manufacturers can implement their own paper source code values which are different from the standard PaperSource values. Common printer tray options include upper tray, lower tray, middle tray, manual feed, envelopes, etc. Please note that each printer will contain the paper tray options that are available for that printer.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE | System.Drawing.Printing.PaperSourceKind Enumeration

PaperWidth Property

Gets or sets the physical width of the paper.

```
printfile.PaperWidth = <integerVal>
```

Property Value

integerVal

*Integer4 and contains the physical width of the paper. Valid only with a continuous-form printer.

Remarks

Must be set prior to opening the printer file (see OPEN command).

If **PaperWidth** is not set, the paper width Print Units specified for the print file will be used. (If *DEFAULT is specified for the print file, then the default paper width, based upon the Orientation of the paper in the printer will be used.

Print Units are measured in 1/100th of a millimeter.

To get the measurement in millimeters, divide by 100.

To get the measurement in centimeters, divide by 1,000.

To get the measurement in inches, divide by 2,540.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | Orientation Property | OPEN | DCLPRINTFILE

Preview Property

Gets or sets whether the print preview dialog is to be displayed before printing.

```
printfile.Preview = ASNA.DataGate.Common.AccessMode.<value>
```

Property Value

value

ASNA.DataGate.Common.AccessMode and is the enumeration value for the property.

Remarks

Must be set prior to opening the printer file (see OPEN command).

Preview allows for easy previewing of printer file output. Setting this property to "PrintPreview" prior to opening the printer file will result in the print driver commands being saved to a temporary file on the client machine, and when the print file is closed, or the FEOD command is issued, a preview of the printed output is displayed (nothing is yet sent to the printer). While previewing, the user has the option to then send the output to the target printer.

IF **Preview** is not set, the preview setting specified for in the print file will be used.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | FEOD | DCLPRINTFILE | ASNA.DataGate.Common.AccessMode Enumeration

PrintableLength Property

Gets the length of the printer's printable area.

```
<integerVal> = printfile.PrintableLength
```

Property Value

integerVal

*Integer4 and contains the length of the printer's printable area.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | Orientation Property | OPEN | DCLPRINTFILE

PrintableWidth Property

Gets the width of the printer's printable area.

```
<integerVal> = printfile.PrintableWidth
```

Property Value

integerVal

*Integer4 and contains the width of the printer's printable area.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | Orientation Property | OPEN | DCLPRINTFILE

Printer Property

Gets or sets the full path name of the printer to be used.

```
printfile.Printer = <stringVal>
```

Property Value

stringVal

*String and contains the full path name of the printer to be used.

Remarks

Must be set prior to opening the printer file (see OPEN command).

To specify a **Printer**, enter the complete path and name of the printer. For Example, if the Printer is on a network, you must specify the network, then the Printer name. For Example, "\\Network\HP LaserJet 5M".

If you are unsure about the correct printer name used by the operating system, simply select *Print* in Word or other Microsoft product and the printer names displayed will be in the correct format.

If **Printer** is not set, the default printer for the operating system will be used.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE

PrintSetup Property

Gets or sets whether the print setup dialog is to display before printing.

```
printfile.PrintSetup = <booleanVal>
```

Property Value

booleanVal of type *Boolean.

True will display the setup dialog.

False will not display the setup dialog.

Remarks

Must be set prior to opening the printer file (see OPEN command).

Setting **PrintSetup** property **True** prior to opening the printer file will result in the print setup dialog being displayed when the print file is opened. This allows the user to select print settings prior to printing.

If **PrintSetup** is set **False**, the print setup dialog will not be displayed. If **PrintSetup** is not set, the print setup setting specified for the file will be used.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE

ProcessMacro Property

Gets or sets a boolean value indicating if DataGate print engine should search for page numbering "macros" and replaced them with appropriate data before sending the output to the printer.

```
printfile.ProcessMacros = <booleanVal>
```

Property Value

booleanVal of type ***Boolean**.

True to search and replace page number information in the print file before printing.

False to not search and replace page number information in the print file before printing.

Remarks

Must be set prior to opening the printer file (see OPEN command).

Setting **ProcessMacro** property **True** will cause the DataGate print engine to search for page numbering "macros" and replace them with the appropriate data as indicated below.

The following macros are currently available:

&# - This is replaced with the current page number.

&\$ - This is replaced with the page count.

Macros may only be described in a Print file label control. Only one macro per Label control is allowed (e.g. "Page &# of &\$" is not valid). You may, however, describe the same macro more than once per format and/or per print file.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE

Quality Property

Gets or sets the print quality.

```
printfile.Quality = System.Drawing.Printing.PrinterResolutionKind.<value>
```

Property Value

value

System.Drawing.Printing.PrinterResolutionKind and contains the enumeration value for the print quality.

Please note that not all of the possible values for this enumeration may be valid for your printer. Each printer has a limited set of acceptable values. It is up to the programmer to select the value adequate to the actual printer used to render the output of the print file.

Remarks

Must be set prior to opening the printer file (see OPEN command).

Set the **Quality** property to specify the print quality. *FILE indicates that the print quality specified for the print file will be used. If *DEFAULT is specified for the print file, then the default print quality specified for the printer will be used.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE | System.Drawing.Printing.PrinterResolutionKind Enumeration

ReportName Property

Gets or sets the name of the report as seen on the printer spool.

```
printfile.ReportName = <stringVal>
```

Property Value

stringVal

*String and contains the name of the report.

Remarks

If not assigned, the default name is the name given when the print file was declared. This is the name of the print file seen on the printer spool.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

Scale Property

Gets or sets the percentage by which the printed output is to be scaled.

```
printfile.Scale = <integerVal>
```

Property Value

integerVal

*Integer4 and contains the percentage by which the printed output is to be scaled.

Remarks

The specified printer must support the output to be scaled or resized.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

TTOption Property

Gets or sets how Microsoft TrueType fonts are to be printed.

```
printfile. TTOption= ASNA.DataGate.Common.PrintTrueType.<value>
```

Property Value

<value> is of type ASNA.DataGate.Common.PrintTrueType.

*FILE indicates that the True Type option specified for the print file will be used. (If *DEFAULT is specified for the print file, then the True Type option specified for the printer will be used).

BITMAP indicates that True Type fonts will print as graphics.

DOWNLOAD indicates that True Type fonts will be downloaded as soft fonts.

SUBDEVFONT indicates that device fonts will be substituted for True Type fonts.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE |
ASNA.DataGate.Common.PrintTrueType Enumeration

UseColor Property

Gets or sets whether to print in color when a color printer is specified.

```
printfile.UseColor = <booleanVal>
```

Property Value

booleanVal of type *Boolean.

True will print in color when a color printer is specified.

False will print in black even if a color printer is specified.

Remarks

UseColor is only valid for color printers and has no affect otherwise.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

YResolution Property

Gets or sets the y-axis resolution in dots *per* inch, in which graphics will be printed.

```
printfile.YResolution = <integerVal>
```

Property Value

integerVal

*Integer4 and contains the value that defines the y-axis resolution in dots per inch, in which graphics will be printed.

Remarks

Must be set prior to opening the print file (see OPEN command).

Setting **YResolution** specifies the Y-axis resolution, in dots per inch. If no **YResolution** is specified, then the y-resolution specified for the printer will be used.

Printers support one or more horizontal and vertical resolutions for faster graphics output. A resolution is an integer value specified in dots-per-inch (dpi), which indicates the number of distinct dots that the print head or inkjet can apply along the x- or the y-axis.

Typically, the x-y resolutions will be the same, such as 300 x 300, or 600 x 600, etc. Since the x-y resolutions are the same, you only need to specify the y-resolution.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | OPEN | DCLPRINTFILE

PrintFile Fields

DclPrintFileName Field

Contains the print file *Name* as declared for the print file (see DCLPRINTFILE).

```
printfile.DclPrintFileName Type(*String)
```

Field Value

String

The *Name* of the print file declared by **DCLPRINTFILE**.

Remarks

DclPrintFileName can contain, as defined by **DCLPRINTFILE**, either:

The name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

A valid program id registered on your system at compile time.

A valid class id registered on your system at compile time.

A valid interface name registered on your system at compile time.

A name of the form **Library.Object**, where **Library** is the name of one of the libraries selected in the new object libraries dialog, and **Object** is valid object name within that library.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

FilePath Field

FilePath contains a string that specifies the path of the file.

```
printfile.FilePath Type(*String)
```

Field Value

String.

The path of the print file to use when opened at file at run-time.

Remarks

FilePath must be set prior to opening a file (see OPEN command).

If **FilePath** is not set, the Library and File of the *File* parameter specified in the DCLPRINTFILE command is used.

Setting the **FilePath** prior to opening a file is equivalent to the iSeries process called **override database file**.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE | OPEN

IsFormatID Field

Contains **True** if the opened (run-time) file's format is equal to the declared (compile-time) file's format.

```
printfile.IsFormatID Type(*Boolean)
```

Field Value

*Boolean

True if the format ID of the file specified in the **DclPrintFile's FileDesc** parameter is the same as the file opened at run-time.

Remarks

IsFormatID indicates if the format ID of the file at run-time compares to the format ID of the file at compile-time.

When the **ChkFmtID** parameter specified in the DCLPRINTFILE command is set to ***Yes** or ***Same**, **IsFormatID** contains **True** if the format ID of the file specified in the **DclPrintfile's FileDesc** parameter is the same as the file opened at run-time.

The Visual RPG for .NET compiler stores the format ID of each file in the assembly when a program is compiled. At run-time, AVR for .NET can then compare the stored format ID with the format ID of the currently-opened file.

At run-time, Visual RPG compares the format ID that was stored during the compilation with the current format ID. The comparing of format ID's can be especially useful when checking if new fields have been added to a file used by a program that has been compiled months ago.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

isOpen Field

Contains **True** when the file is currently open for print operations.

```
printfile.isOpen Type(*Boolean) Access(*Protected)
```

Field Value

*Boolean

Contains **True** or **False** indicating if the printfile is currently open (**True**) for print operations.

Remarks

A file is considered "open" if it has been opened by the RPG program during initialization or by an **OPEN** operation, and has not subsequently been closed.

If the file is conditioned by an external indicator and the external indicator was off at program initialization, the file is considered closed, and **isOPEN** returns **False**.

isOpen is equivalent to the %OPEN built-in function.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE | CLOSE | OPEN | %OPEN

lastException Field

Contains the last exception thrown for this file.

```
printfile.lastException Type (ASNA.VisualRPG.Runtime.AvrException)
Access (*Protected)
```

Field Value

This is a protected public field of type **ASNA.VisualRPG.Runtime.AvrException**.

Remarks

lastException contains the last exception thrown for this file.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

ASNA.VisualRPG.Windows.Controls Namespace

The **ASNA.VisualRPG.Windows.Controls** namespace contains classes that add additional functionality to the Control classes that display information to the user.

Windows Forms controls handle user input through the keyboard and pointing devices, message routing, security and defines the bounds of a control (its position and size). It also has ambient properties so child controls can appear like their surrounding environment. An ambient property is a control property that, if not set, is retrieved from the parent control. Typically, an ambient property represents a characteristic of a control, such as **BackColor**, that is communicated to a child control. For Example, a field will have the same **BackColor** as its parent by default.

ASNA.VisualRPG.Windows.Controls classes add addition functions to further control how the information is displayed to the user and also additional update capabilities through the use of new properties, events, and fields. For Example, a field BackColor can not be set with the addition of a new FieldBackColor property.

Namespace hierarchy

Classes

Class	Description
IODateTime	Represents a Windows DateTime control. It is very similar to the AVR Classic's ioField control.

*Not currently implemented as of 5-19-06. See the object browser for a list of its properties and methods.

IODecimal

*Not currently implemented as of 5-19-06. See the object browser for a list of its properties and methods.

Represents a Windows **Decimal** control. It is very similar to the **AVR Classic's ioField** control.

IOString

*Not currently implemented as of 5-19-06. See the object browser for a list of its properties and methods.

Represents a Windows **String** control. It is very similar to the **AVR Classic's ioField** control.

GridArgs (inherited from **System.EventArgs**)

GridArgs is a derived class to hold data for the SubfileControl.RecordSwitch event.

RecordDeleteArgs (inherited from **System.EventArgs**)

RecordDeleteArgs is a derived class to hold data for the SubfileControl.RecordDelete event.

RecordNewArgs (inherited from **System.EventArgs**)

RecordNewArgs is a derived class to hold data for the SubfileControl.RecordNew event.

SubfileControl (inherited from **System.Windows.Forms.Control**)

SubfileControl provides a flexible, capable tool for browsing and editing tabular data that reduces the requisite programming to a minimum.

See Also

DCLSUBFILE | DCLSUBFILEFLD

GridArgs Class

GridArgs is a derived class to hold data for the SubfileControl.RecordSwitch event.

```
ASNA.VisualRPG.Windows.Controls
ASNA.VisualRPG.Windows.Controls.GridArgs
    public class GridArgs : System.EventArgs
```

Remarks

GridArgs class is derived from System.EventArgs as a container for the CurrRow the PriorRow parameters for the **SubfileControl.RecordSwitch** event.

See Also

ASNA.VisualRPG.Windows.Controls | GridArgs Members | SubfileControl.RecordSwitch Event

GridArgs Members

Public Constructor

 **GridArgs** This method creates a new instance of a **GridArgs** object that contains the SubfileControl.RecordSwitch event data.

Public Fields

 **CurrRow** **CurrRow** is used to indicate the 0-based record number that the selection is switching to.

 **PriorRow** **PriorRow** is used to indicate the 0-based record number that the selection is switching from.

See Also

[ASNA.VisualRPG.Windows.Controls](#) | [GridArgs Class](#) | [SubfileControl.RecordSwitch Event](#)

GridArgs.GridArgs Constructors

This method constructs a new instance of a **GridArgs** Object.

```
GridArgs (integer, integer)
```

See Also

[ASNA.VisualRPG.Windows.Controls](#) | [GridArgs Class](#) | [GridArgs Members](#) | [SubfileControl.RecordSwitch Event](#) | [SubfileControl overview](#)

GridArgs.GridArgs Constructor (integer, integer)

This method creates a new instance of a **GridArgs** object that contains the SubfileControl.RecordSwitch event data.

```
BeginConstructor GridArgs
  DclSrParm Prior (*Integer4)
  DclSrParm Curr (*Integer4)
EndConstructor
```

Parameters

Prior Required. The row number, relative to zero, of the record that the selection is switching from. **Curr** Required. The row number, relative to zero, of the record that the selection is switching to.

See Also

[ASNA.VisualRPG.Windows.Controls](#) | [GridArgs Class](#) | [GridArgs Members](#) | [SubfileControl.RecordSwitch Event](#) | [SubfileControl overview](#)

GridArgs Fields

GridArgs.CurrRow Field

CurrRow is used to indicate the 0-based record number that the selection is switching to.

```
GridArgs.CurrRow = [*integer4]
```

Settings

The row number, relative to zero, of the record that the selection is switching to.

See Also

[ASNA.VisualRPG.Windows.Controls | GridArgs Class | GridArgs Members | SubfileControl.RecordSwitch Event | Subfile Control overview | Integer4 Data Type](#)

GridArgs.PriorRow Field

PriorRow is used to indicate the 0-based record number that the selection is switching from.

```
GridArgs.PriorRow = [*integer4]
```

Settings

The row number, relative to zero, of the record that the selection is switching from.

See Also

[ASNA.VisualRPG.Windows.Controls | GridArgs Class | GridArgs Members | SubfileControl.RecordSwitch Event | SubfileControl overview | Integer4 Data Type](#)

RecordDeleteArgs Class

RecordDeleteArgs is a derived class to hold data for the **SubfileControl.RecordDelete** event.

```
ASNA.VisualRPG.Windows.Controls
ASNA.VisualRPG.Windows.Controls.RecordDeleteArgs
    public class RecordDeleteArgs : System.EventArgs
```

Remarks

RecordDeleteArgs class is derived from **System.EventArgs** as a container for the Row the Cancel parameters for the **SubfileControl.RecordDelete** event.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordDeleteArgs Members | SubfileControl.RecordDelete Event](#)

RecordDeleteArgs Members

Public Constructor



RecordDeleteArgs

This method creates a new instance of a **RecordDeleteArgs** object that contains the SubfileControl.RecordDelete event data.

Public Fields



Cancel

Cancel is used to indicate if the record delete event is to be canceled. The default is **False**.



Row

Row is used to indicate the record (row) to be deleted.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordDeleteArgs Class](#) | [SubfileControl.RecordDelete Event](#)

RecordDeleteArgs.RecordDeleteArgs Constructors

This method constructs a new instance of a **RecordDeleteArgs** Object.

```
RecordDeleteArgs (integer, boolean)
```

See Also

[ASNA.VisualRPG.Windows.Controls | RecordDeleteArgs Class](#) | [RecordDeleteArgs Members](#) | [SubfileControl.RecordDelete Event](#) | [Subfile Control overview](#)

RecordDeleteArgs.RecordDeleteArgs Constructor (integer, boolean)

This method creates a new instance of a **RecordDeleteArgs** object that contains the SubfileControl.RecordDelete event data.

```
BeginConstructor RecordDeleteArgs
  DclSrParm rowPm (*Integer2)
  DclSrParm cancelPm (*Boolean) By(*Reference)
EndConstructor
```

Parameters

rowPm

Required. Specify the row (record) number to be deleted. This is a 0-based value.

cancelPm

Required. Specify ***True** to indicate the action to delete a record is to be canceled, otherwise ***False**.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordDeleteArgs Class](#) | [RecordDeleteArgs Members](#) | [SubfileControl.RecordDelete Event](#) | [Subfile Control overview](#)

RecordDeleteArgs Fields

RecordDeleteArgs.Cancel Field

Cancel is used to indicate if the record delete event is to be canceled. The default is **False**.

```
RecordDeleteArgs.Cancel = [boolean]
```

Settings

True indicates that the action to delete a record is to be canceled, otherwise **False**.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordDeleteArgs Class | RecordDeleteArgs Members | SubfileControl.RecordDelete Event | Subfile Control overview](#)

RecordDeleteArgs.Row Field

Row is used to indicate the record (row) to be deleted.

```
RecordDeleteArgs.Row = [*integer4]
```

Settings

The row number, relative to zero, of the record to be deleted.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordDeleteArgs Class | RecordDeleteArgs Members | SubfileControl.RecordDelete Event | Subfile Control overview | Integer4 Data Type](#)

RecordNewArgs Class

RecordNewArgs is a derived class to hold data for the **SubfileControl.RecordNew** event.

```
ASNA.VisualRPG.Windows.Controls
ASNA.VisualRPG.Windows.Controls.RecordNewArgs
public class RecordNewArgs : System.EventArgs
```

Remarks

RecordNewArgs class is derived from **System.EventArgs** as a container for the **Row** parameter for the **SubfileControl.RecordNew** event.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordNewArgs Members | SubfileControl.RecordNew Event](#)

RecordNewArgs Members

Public Constructor



RecordNewArgs

This method creates a new instance of a **RecordNewArgs** object that contains the SubfileControl.RecordNew event data.

Public Fields



Row

Row is used to indicate the record (row) to be added.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordNewArgs Class | SubfileControl.RecordNew Event](#)

RecordNewArgs.RecordNewArgs Constructors

This method constructs a new instance of a **RecordNewArgs** Object.

```
RecordNewArgs (integer)
```

See Also

[ASNA.VisualRPG.Windows.Controls | RecordNewArgs Class | RecordNewArgs Members | SubfileControl.RecordNew Event | Subfile Control overview](#)

RecordNewArgs.RecordNewArgs Constructor (integer)

This method creates a new instance of a **RecordNewArgs** object that contains the SubfileControl.RecordNew event data.

```
BegConstructor RecordNewArgs  
    DelSrParm rowPm (*Integer2)  
EndConstructor
```

Parameters

rowPm

Required. Specify the row (record) number to be deleted. This is a 0-based value.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordNewArgs Class | RecordNewArgs Members | SubfileControl.RecordNew Event | Subfile Control overview](#)

RecordNewArgs Fields

RecordNewArgs.Row Field

Row is used to indicate the record (row) to be added.

```
RecordNewArgs.Row = [*integer4]
```

Settings

The row number, relative to zero, of the record to be added.

See Also

[ASNA.VisualRPG.Windows.Controls | RecordNewArgs Class | RecordNewArgs Members | SubfileControl.RecordNew Event | SubfileControl overview | Integer4 Data Type](#)

SubfileControl Class

The SubfileControl class provides a flexible, capable tool for browsing and editing tabular data that reduces the requisite programming to a minimum.

Remarks

SubfileControl uses tables like those found in databases. A table can contain multiple fields of information for each of many records.

The definition of a subfile begins with its declaration (DCLSUBFILE) and the list of fields (DCLSUBFILEFLD) it contains. For each field, the subfile employs a separate control to display and edit the field's data.

When writing records to a subfile where the *char data contains hex zeros (X'00'), these values will be displayed as a period. Hex 00 is a terminator character for a windows string. The WRITE command when run against a **SubfileControl**, allows you to write to any RRN you want up to the max number of records allowed in a subfile (32,761). If the RRN is less than the total number of records, then blank records are written to fill in the difference.

Certain properties retain their last set values regardless of the number of active rows in the subfile. These include Row and Column. Note also, the content of the RRNName property remains unchanged when the Subfile is cleared.

See Also

[ASNA.VisualRPG.Windows.Controls | SubFileControl Members | SubfileField Collection Editor](#)

SubfileControl Members

Public Properties

 AccessibleDescription (inherited from Control)	Gets or sets the description of the control used by accessibility client applications.
 AccessibleName (inherited from Control)	Gets or sets the name of the control used by accessibility client applications.
 AccessibleRole (inherited from Control)	Gets or sets the accessible role of the control.
 AllowDrop (inherited from Control)	Gets or sets a value indicating whether the control can accept data that the user drags onto it.
 AllowSort	Design-time only. Sets a boolean value indicating if sorting on columns is allowed.
 Anchor (inherited from Control)	Gets or sets which edges of the control are anchored to the edges of its container.
 BackColor (inherited from Control)	Gets or sets the background color for the control.
Supported by the .NET Compact Framework	
 BackgroundImage (inherited from Control)	Gets or sets the background image displayed in the control.
 Browse	Design-time only. Sets a indicator used to indicate if data in a subfile can be edited, or if it will be displayed without editing (read-only) in any of the Subfile cells, fields, or records.
 CausesValidation (inherited from Control)	Gets or sets a value indicating whether the control causes validation to be performed on any controls that require validation when it receives focus.
 CellBackColor	Run-time only. Gets or sets the text background color for a cell in a Subfile.
 CellBrowse	Run-time only. Gets or sets the browse state for data in an individual cell in a Subfile.
 CellData	Not implemented.
 CellForeColor	Run-time only. Gets or sets the text foreground color for an individual cell in a Subfile.
 CellValue	Run time only. Gets the unformatted value of a Subfile cell.
 Changed	Run-time only. Gets or sets a value indicating that user changes

have been made to the contents of the subfile.

 Column	Run-time only. Gets or sets the column of the active cell in the Subfile.
 ColumnCount	Run-time only. Gets the number of columns in the Subfile.
 ContextMenu (inherited from Control)	Gets or sets the shortcut menu associated with the control.
Supported by the .NET Compact Framework	
 Cursor (inherited from Control)	Gets or sets the cursor that is displayed when the mouse pointer is over the control.
 Dock (inherited from Control)	Gets or sets which edge of the parent container a control is docked to.
 Enabled (inherited from Control)	Gets or sets a value indicating whether the control can respond to user interaction.
Supported by the .NET Compact Framework	
 FieldBackColor	Run-time only. Gets or sets the text background color for a field (column) in a Subfile.
 FieldBrowse	Run-time only. Gets or sets the a boolean value indicating the browse state for data in a field (column) in a Subfile.
 FieldColumnHeading	Run-time only. Gets or sets the text to be displayed at the top of each column.
 FieldColumnWidth	Run time only. Gets or sets the width of the field column in the Subfile.
 FieldForeColor	Run time only. Gets or sets the text foreground color for a field (column) in a Subfile.
 FieldHidden	Run-time only. Gets or sets a boolean value used to turn on and off the visibility of a field to a user.
 Fields	Design-time only. The Fields property produces the SubfileField Collection Editor dialog box to specify the fields, and set their properties, contained in the Subfile.
 FirstRow	Run-time only. Gets or sets the specified row displayed at the top of the Subfile.
 Font (inherited from Control)	Gets or sets the font of the text displayed by the control.
Supported by the .NET Compact Framework	

 ForeColor (inherited from Control)	Gets or sets the foreground color of the control.
Supported by the .NET Compact Framework	
 FormatName	Design-time only. Sets the name of the record format.
 GridLineColor	Design-time and run-time. Gets or sets the color of the grid lines between cells of a Subfile.
 ImeMode (inherited from Control)	Gets or sets the Input Method Editor (IME) mode of the control.
 KeyboardDeleteInsert	Design-time only. Sets a boolean value that is used to determine if the user can use the Delete and Insert keys at run-time to delete and insert records.
 Location (inherited from Control)	Gets or sets the coordinates of the upper-left corner of the control relative to the upper-left corner of its container.
Supported by the .NET Compact Framework	
 Name (inherited from Control)	Gets or sets the name of the control.
 RecordBackColor	Run-time only. Gets or sets the text background color for a record (row) in a Subfile.
 RecordBrowse	Run-time only. Gets or sets the browse state for data in a record (row) in a Subfile.
 RecordChanged	Run time only. Gets or sets a boolean value indicating when the contents of a record have changed.
 RecordForeColor	Run-time only. Gets or sets the text foreground color for a record (row) in a Subfile.
 RightToLeft (inherited from Control)	Gets or sets a value indicating whether control's elements are aligned to support locales using right-to-left fonts.
 Row	Run-time only. Gets or sets the Row of the active cell in the Subfile.
 RowCount	Run-time only. Gets the current number of rows in the Subfile.
 RowSelected	Run-time only. Gets or sets a boolean value indicating which row(s) the user has selected.
 RRNName	Design-time only. Gets or sets the name of a field used as the relative record number.

 SingleSelect	Design-time only. Sets a boolean value indicating whether the user can only select an individual cell, or both an individual cell and a range of cells (this is the default).
 Size (inherited from Control)	Gets or sets the height and width of the control.
Supported by the .NET Compact Framework	
 TabIndex (inherited from Control)	Gets or sets the tab order of the control within its container.
 TabStop (inherited from Control)	Gets or sets a value indicating whether the user can give the focus to this control using the TAB key.
 Tag (inherited from Control)	Gets or sets the object that contains data about the control.
 UpperCaseOnly	Design-time only. Sets a boolean indicator whether to force the entry of UPPERCASE characters on a field.
 Visible (inherited from Control)	Gets or sets a value indicating whether the control is displayed.
Supported by the .NET Compact Framework	
 WholeRows	Design-time only. Sets a boolean value that indicates whether selection to an entire row occurs when an individual cell is selected.

Events

 Bottom	The Bottom event occurs when the user attempts to scroll off the bottom of the Subfile.
 DoubleClick	The DoubleClick event occurs when the user double-clicks a control with the left or right mouse button.
 KeyDown	The KeyDown event occurs when the user presses a key while the control has focus.
 KeyPress	The KeyPress event occurs when a key is pressed while the control has focus.
 KeyUp	The KeyUp event occurs when a key is released while the control has focus.
 RecordDelete	The RecordDelete event occurs when a record is about to be deleted by the user.
 RecordNew	The RecordNew event indicates a new record was added by the user.
 RecordSwitch	The RecordSwitch events occurs when the selection has been moved to a different record.

 SelectionChanged	The SelectionChanged event occurs when the current selection changes.
 Top	The Top event occurs when the user attempts to scroll off the top of the Subfile.

Methods

 Clear	The Clear method clears the contents of a Subfile.
 Select	The Select method selects an individual cell, row, column or a range of cells in the Subfile.

See Also

ASNA.VisualRPG.Windows.Controls | DCLSUBFILE | DCLSUBFILEFLD | SubfileField Collection Editor | System.Windows.Forms. **Control**

SubfileControl Events

Bottom Event

The Bottom event occurs when the user attempts to scroll off the bottom of the Subfile.

```
BEGSR mycontrol_Bottom Event(*this.mycontrol.Bottom)
  DclSrParm sender Type (*Object)
  DclSrParm e      Type (System.EventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A System.EventArgs object that contains the event data.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | System.EventArgs

DoubleClick Event

The **DoubleClick** event occurs when the user double-clicks a control with the left or right mouse button.

```
BEGSR mycontrol_DoubleClick Event(*this.mycontrol.DoubleClick)
  DclSrParm sender Type (*Object)
  DclSrParm e      Type (System.EventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A `System.EventArgs` object that contains the event data.

Remarks

SubfileControl: The `Column` and `Row` properties reference the cell that was selected. To have a double-click event recognized in a Subfile, you must first click on a cell which gets focus on that cell, then double-click to register the event.

For **objects** that receive mouse events, the events occur in this order, `MouseDown`, `Click`, `MouseUp`, and **DoubleClick**.

To distinguish between the left, right and middle Mouse buttons, the `MouseDown` and `MouseUp` events.

The double-click event will only be issued for the `ComboBox` control when its `Style` property is set to 0 - `SimpleCombo`.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | DCLEVENT | System.EventArgs](#)

KeyDown Event

The **KeyDown** event occurs when the user presses a key while the control has focus.

```
BegSr mycontrol_KeyDown Event(*this.mycontrol.KeyDown)
  DclSrParm sender Type (*object)
  DclSrParm e Type (System.Windows.Forms.KeyEventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A `System.Windows.Forms.KeyEventArgs` object that contains the event data.

Remarks

Key events occur in the following order:

`KeyDown`

`KeyPress`

`KeyUp`

To handle keyboard events only at the form level and not allow other controls to receive keyboard events, set the `KeyEventArgs.Handled` property in your form's **KeyPress** event-handling method to **True**.

Certain keys, such as `TAB`, `RETURN`, `ESCAPE`, and arrow keys are handled by controls automatically. In order to have these keys raise the **KeyDown** event, you must override the `IsInputKey` method in each control of your form. The code for the override of the **IsInputKey** would need to determine if one of the special keys is pressed and return a value of **True**.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | KeyPress Event | KeyUp Event | System.Windows.Forms.KeyPressEventArgs Class | KeyEventArgs.Handled Property | IsInputKey Method](#)

KeyPress Event

The **KeyPress** event occurs when a key is pressed while the control has focus.

```
BegSr mycontrol_KeyPress Event(*this.mycontrol.KeyPress)
  DclSrParm sender Type (*object)
  DclSrParm e Type(System.Windows.Forms.KeyPressEventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A System.Windows.Forms.KeyPressEventArgs object that contains the event data.

Remarks

Key events occur in the following order:

KeyDown

KeyPress

KeyUp

The **KeyPress** event is not raised by noncharacter keys. Use **KeyDown** and **KeyUp** events to handle any keystrokes not recognized by **KeyPress**, such as function keys, editing keys, and any combinations of these with keyboard modifiers.

To handle keyboard events only at the form level and not allow other controls to receive keyboard events, set the KeyPressEventArgs.Handled property in your form's **KeyPress** event-handling method to **True**.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | KeyDown Event | KeyUp Event | System.Windows.Forms.KeyPressEventArgs | KeyPressEventArgs.Handled Property](#)

KeyUp Event

The **KeyUp** event occurs when a key is released while the control has focus.

```
BegSr mycontrol_KeyEvent Event(*this.mycontrol.KeyUp)
  DclSrParm sender Type (*object)
  DclSrParm e Type(System.Windows.Forms.KeyEventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A System.Windows.Forms.KeyEventArgs object that contains the event data.

Remarks

Key events occur in the following order:

KeyDown

KeyPress

KeyUp

To handle keyboard events only at the form level and not allow other controls to receive keyboard events, set the `KeyEventArgs.Handled` property in your form's **KeyPress** event-handling method to **True**.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | KeyDown Event | KeyPress Event | System.Windows.Forms.KeyEventArgs Class | KeyEventArgs.Handled Property](#)

RecordDelete Event

The **RecordDelete** event occurs when a record is about to be deleted by the user.

```
Begsr mycontrol_RecordDelete Event (*this.mycontrol.RecordDelete)
  DclSrParm sender Type (*object)
  DclSrParm e      Type (ASNA.VisualRPG.Windows.Controls.RecordDeleteArgs)
EndSr
```

Parameters

sender

The source of the event. **e** An **ASNA.VisualRPG.Windows.Controls.RecordDeleteArgs** object that contains the event data.

Remarks

The user can only delete records from the Subfile if the `KeyboardDeleteInsert` property is set to **True**.

This event will be received prior to the record actually being deleted from the Subfile.

Set `e.Cancel = *True` to ignore the delete command (`RecordDeleteArgs.Cancel` field).

See Also

[ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | DCLEVENT | RecordDeleteArgs Class](#)

RecordNew Event

The **RecordNew** event indicates a new record was added by the user.

```
Begsr mycontrol_RecordNew Event (*this.mycontrol.RecordNew)
  DclSrParm sender Type (*object)
  DclSrParm e      Type (ASNA.VisualRPG.Windows.Controls.RecordNewArgs)
EndSr
```

Parameters

sender

The source of the event. **e** An **ASNA.VisualRPG.Windows.Controls.RecordNewArgs** object that contains the event data.

Remarks

The **RecordNew** event occurs only when:

The KeyboardDeleteInsert property is set to True, and the user presses the Insert (Ins) key.

The AutoExtend property is set to True, and the user types a field in the last record and leaves the field.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | DCLEVENT | RecordNewArgs Class

RecordSwitch Event

The **RecordSwitch** event occurs when the selection has been moved to a different record.

```
BegSr mycontrol_RecordSwitch Event(*this.mycontrol.RecordDelete)
  DclSrParm sender Type (*object)
  DclSrParm e Type (ASNA.VisualRPG.Windows.Controls.GridArgs)
EndSr
```

Parameters

sender

The source of the event. **e** An **ASNA.VisualRPG.Windows.Controls.GridArgs** object that contains the event data.

Remarks

Do not use the Select method within this event. The use of the method within this event subroutine will generate another **RecordSwitch** event, resulting in an endless loop and ultimately a stack overflow.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | GridArgs Class

SelectionChanged Event

The **SelectionChanged** event occurs when the current selection changes.

```
BEGSR mycontrol_SelectionChanged Event(*this.mycontrol.SelectionChanged)
  DclSrParm sender Type (*Object)
  DclSrParm e Type (System.EventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A System.EventArgs object that contains the event data.

Remarks

Processing occurring within this event handler can significantly affect the overall performance of the form designer. Try to minimize processing when handling this event.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | System.EventArgs

Top Event

The **Top** event occurs when the user has **attempted** to scroll off the top of the Subfile.

```
BEGSR mycontrol_Top Event(*this.mycontrol.Top)
  DclSrParm sender Type (*Object)
  DclSrParm e      Type (System.EventArgs)
EndSr
```

Parameters

sender

The source of the event. **e** A System.EventArgs object that contains the event data.

Remarks

When loading a Subfile in "Page" mode, this event can be used to fetch previous records to load into the Subfile.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | DCLEVENT | System.EventArgs

SubfileControl Methods

Clear Method

The **Clear** method clears the contents of a Subfile.

```
control.Clear()
```

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members

Select Method

The **Select** method selects an individual cell, row, column or a range of cells in the Subfile.

```
control.Select (FromRow, FromCol, [ToRow], [ToCol])
```

Settings

FromRowRequired. Integer value indicating the row index that defines the beginning of the range of selected cells in the Subfile. **FromColRequired.** Integer value indicating the column index that defines the beginning of the range of selected cells in the Subfile. **ToRowOptional.** Integer value indicating the row index that defines the end of the range of selected cells in the Subfile. **ToColOptional.** Integer value indicating the column index that defines the end of the range of selected cells in the Subfile.

Remarks

The first two arguments are required. To specify other arguments, you must specify all arguments that appear in the syntax **before** the argument you want to specify. For Example, you cannot specify *ToRow* without specifying *FromRow* and *FromCol*. Trailing arguments that are unspecified remain unchanged.

To select a single cell, specify only the first **two** arguments. If you specify *ToRow* but not *ToCol*, *ToCol* will assume the same value as *FromCol*.

To select a range of cells, use all **four** arguments. Specify the starting row and column, and the ending row and column. All cells in between this range will be selected.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | Integer Data Types

SubfileControl Properties

AllowSort Property

The **AllowSort** property sets a boolean value indicating if sorting on columns is allowed. The default is **False**.

```
control.AllowSort = <booleanVal>
```

Settings

<booleanVal> of type ***Boolean**

True indicates sorting on columns is allowed.

False (default) indicates no column sorting is allowed.

Remarks

The **AllowSort** property can be set at design-time only.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | *Boolean Data Types

Browse Property

The **Browse** property sets a indicator used to determines if data in a Subfile can be edited, or if it will be displayed without editing (read-only) in any of the Subfile cells, fields, or records. The default is **False**.

```
control.Browse = <booleanVal>
```

Settings

<booleanVal> of type ***Boolean**

True does not allow data to be edited by the user.

False allows data in the subfile to be edited by the user.

Remarks

The **Browse** property can be set at design-time only.

You can also change the browsing state at run-time for an individual cell (CellBrowse property), field (FieldBrowse property), or record (RecordBrowse property). However, if the **Browse** property is set to **True**, setting any of these run-time properties will have no effect.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | *Boolean Data Types

CellBackColor Property

The **CellBackColor** property gets or sets the text background color for an individual cell in a Subfile.

```
control.CellBackColor( <RowIndex>,<ColIndex> ) =  
System.Drawing.Color.<color>
```

Settings

<RowIndex> and,<ColIndex> of type *Integer8

Contain the respective row and column index, relative to zero.

Color

One of the valid enumeration values (Red, Green, Blue, etc.). See [Setting Colors](#) for more information on setting colors.

Remarks

CellBackColor is available at run-time only.

Setting this property will override the current **BackColor** for the cell with a new color. Only the color of the background in the Subfile cell will be changed, headings and row buttons will not be affected.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | FieldBackColor Property | RecordBackColor Property | System.Drawing.Color | Integer Data Types](#)

CellBrowse Property

The **CellBrowse** property gets or sets the browse state for data in an individual cell in a Subfile.

```
control.CellBrowse( <RowIndex>,<ColIndex> ) = <booleanVal>
```

Settings

<RowIndex> and,<ColIndex> of type *Integer8

Contain the respective row and column index, relative to zero.

<booleanVal> of type *Boolean

True indicates the contents of the cell is browse only, editing is not allowed.

False indicates cell editing is allowed. This is the default.

Remarks

CellBrowse is available at run-time only.

This property is valid only if the Subfile **Browse** property is **False**. If the Subfile **Browse** property is **True**, setting this property will have no effect.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | FieldBrowse Property | RecordBrowse Property | *Boolean Data Types | Integer Data Types](#)

CellForeColor Property

The **CellForeColor** property gets or sets the text foreground color for an individual cell in a Subfile.

```
control.CellForeColor( <RowIndex>,<ColIndex> ) =  
System.Drawing.Color.<color>
```

Settings

<RowIndex> and,<ColIndex> of type *Integer8

Contain the respective row and column index, relative to zero, for the specific cell.

System.Drawing.Color.< color>

Where < color> is one of the valid enumeration values (Red, Green, Blue, etc.). See [Setting Colors](#) for more information on setting colors.

Remarks

CellForeColor is available at run-time only.

Setting this property will override the current text ForeColor for the cell with a new color. Only the color of text in the Subfile cell will be changed, headings and row buttons will not be affected.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | FieldForeColor Property | RecordForeColor Property | Integer Data Types | System.Drawing.Color](#)

CellValue Property

The **CellValue** property gets the unformatted value of a Subfile cell (numbers as <integrals>.<decimal> regardless of locale, indicators as '0' or '1').

```
<stringVal> = control.CellValue
```

Settings

<stringVal>

Contains the unformatted value of the cell.

Remarks

The **CellValue** property is available at run-time only.

Within <stringVal>, numbers are returned as <integrals>.<decimal> regardless of locale, indicators as '0' or '1'.

The Column and Row properties are used to establish the Subfile coordinates for using the property, and also references the location where the user is active.

See Also

[ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | *String Data Types](#)

Changed Property

The **Changed** property gets or sets a value indicating that user changes have been made to the contents of the subfile.

```
control.Changed = <booleanVal>
```

Settings

<booleanVal> of type ***Boolean**

True indicates the contents have been changed. **False** indicates the contents have not changed (this is the default).

Remarks

The **Changed** property can be set at run-time only.

The **Changed** property is useful when you want to know if a change was made to a Subfile by user input.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | *Boolean Data Types

Column Property

The **Column** property gets or sets the column number of the active cell in the Subfile. Except as noted below, changing the **Row** and **Column** properties will not change the location of the active cell.

```
control.Column = <integerVal>
```

Settings

<integerVal> of type ***Integer**

Contains the column number relative to zero.

Remarks

The **Column** property is available at run-time only.

The **Column** and **Row** properties must be set by the application prior to updating the **CellData** property in order to specify which cell's data to set.

To position or set focus on a particular cell, use the **Select** method.

The **Column** property retains its last set value regardless of the number of active rows in the subfile.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | *Integer Data Type | Row Property

ColumnCount Property

The **ColumnCount** property gets the number of columns in the Subfile.

```
<integerVal> = control.ColumnCount
```

Settings

<integerVal> of type ***Integer**

contains the number of columns in the subfile. Hidden column are also included.

Remarks

The **ColumnCount** property can be used at run-time only and is read-only.

The **ColumnCount** Property Value also includes hidden columns.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | *Integer Data Type

FieldBackColor Property

The **FieldBackColor** property gets or sets the text background color for a field (column) in a Subfile.

```
control.FieldBackColor( <ColIndex> ) = System.Drawing.Color.<color>
```

Settings

<ColIndex> is of type ***Integer8**

Contains the column index, relative to zero.

System.Drawing.Color.< color>

Where **< color>** is one of the valid enumeration values (Red, Green, Blue, etc.). See Setting Colors for more information on setting colors.

Remarks

FieldBackColor is available at run-time only.

Setting this property will override the current **BackColor** color for the field with a new color. Only the color of the background in the Subfile field will be changed, headings and row buttons will not be affected.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | RecordBackColor Property | CellBackColor Property | System.Drawing.Color | Integer Data Types

FieldBrowse Property

The **FieldBrowse** property gets or sets a boolean value indicating the browse state for data in a field (column) in a Subfile.

```
control.FieldBrowse( <ColIndex> ) = <booleanVal>
```

Settings

<ColIndex> is of type ***Integer8**

Contains the column index, relative to zero.

<booleanVal> is of type ***Boolean**

True indicates the contents of the field is browse only, editing is not allowed. **False** indicates field editing is allowed (this is the default).

Remarks

FieldBrowse is available at run-time and can also be set at design-time in the Fields (**SubfileFields Collection Editor**) dialog box. See Also the *Browse* parameter when the field is defined (DCLSUBFILEFLD).

This property is valid only if the Subfile Browse property is **False**. If the Subfile **Browse** property is **True**, setting this property will have no effect.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | CellBrowse Property | RecordBrowse Property | SubfileField Collection Editor | *Boolean Data Types | Integer Data Types

FieldColumnHeading Property

The **FieldColumnHeading** property gets or sets the text to be displayed at the top of each column.

```
control.FieldColumnHeading( <ColIndex> ) = <stringVal>
```

Settings

<ColIndex> is of type ***Integer8**

Contains the column index, relative to zero.

<stringVal> is of type ***String**

Contains the text to be displayed at the top of the field column.

Remarks

FieldColumnHeading is available at run-time only, but can be set at design-time in the Fields (**SubfileFields Collection Editor**) dialog box. (See Also the *Heading* parameter when the field is defined (DCLSUBFILEFLD)).

FieldColumnHeading is an array property to set and get the column headings, if the *ColButtons* property is set to **True**. By default, the column headings will contain the field *Names*.

If the text is too long to fit in the column heading, the text will be wrapped on word boundaries. However, the programmer must ensure that the column headings are high enough to display all of the wrapped text. If it is not high enough, the text will be clipped.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | SubfileField Collection Editor | String Data Type | Integer Data Type

FieldColumnWidth Property

The **FieldColumnWidth** property allows the user to get or set the width of the field column in the Subfile.

```
control.FieldColumnWidth( <ColIndex> ) = < integerVal> )
```

Settings

<ColIndex> is of type *Integer8

Contains the column index, relative to zero.

<integerVal> is of type *Integer8

Contains the column width.

Remarks

FieldColumnWidth is available at run-time only, but can be set at design-time in the Fields (SubfileFields Collection Editor) dialog box. (See Also the *Width* parameter when the field is defined (DCLSUBFILEFLD).

*CALC is the default, which automatically calculates the column width based upon the font used and the maximum number of characters in the field definition or *Heading*, whichever is greater.

Width may also be an absolute value (expressed in pixels). The absolute values may be entered into the **Fields** dialog.

Reminder: When set to an absolute value, the corresponding Subfile column width will **not be** automatically sized at run-time as described above for *CALC.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | SubfileField Collection Editor | Integer Data Type

FieldForeColor Property

The **FieldForeColor** property gets or sets the text foreground color for a field (column) in a Subfile.

```
control.FieldForeColor( <ColIndex> ) = System.Drawing.Color.<color>
```

Settings

<ColIndex> is of type *Integer8

Contains the column index, relative to zero.

System.Drawing.Color.< color>

Where *< color>* is one of the valid enumeration values (Red, Green, Blue, etc.). See Setting Colors for more information on setting colors.

Remarks

FieldForeColor is available at run-time only.

Setting this property will override the current text ForeColor for the field with a new color. Only the color of text in the Subfile field will be changed, headings and row buttons will not be affected.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | CellForeColor Property | RecordForeColor Property | System.Drawing.Color | Integer Data Types

FieldHidden Property

The **FieldHidden** property gets or sets a boolean value used to turn on and off the visibility of a field to a user.

```
control.FieldHidden(<ColIndex>) = <booleanVal>
```

Settings

<ColIndex> is of type ***Integer8**

Contains the column index, relative to zero.

<booleanVal> of type ***Boolean**

True if the contents of the field are hidden.

False if the contents are not hidden. This is the default.

Remarks

The **FieldHidden** property can be set at run-time only, but can be set at design-time in the Fields (**SubfileFields Collection Editor**) dialog box. (See Also the *Hidden* parameter when the field is declared with the DCLSUBFILEFLD command).

Hiding a field (column) is useful for storing the file record key in the Subfile. Another use of the **FieldHidden** property might be to hide the description of a field (e.g., Credit Code) until the user double-clicks on the code field. On the DoubleClick event (Column property), you can reveal the column containing the descriptions. On the next **DoubleClick** in the code, you can hide the description.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | SubfileField Collection Editor | Boolean Data Types | Integer Data Types

Fields Property

At design-time, the **Fields** property produces the SubfileField Collection Editor dialog box to specify the fields, and set their properties, contained in the Subfile.

Remarks

To access the subfile fields, click on the  symbol on the far right of the "Fields" (SubfileField[] Array) line in the **SubfileControl** Properties window. The **SubfileField Collection Editor** then displays allowing you to **Add** or **Remove** fields and also to enter or change the Property Values for an existing subfile field.

The list of field properties with a brief description are listed below. These properties and values entered for the subfile field become parameter values in the subfile field declaration. See DCLSUBFILEFLD for a more complete description.

Alignment - the alignment of data within the field. Pick a selection from the drop down list.

DropDownStyle - the style for a drop down field. Pick a selection from the drop down list.

Heading - the heading for the column. Default is the field **Name**.

Hidden - indicate if the field is hidden or not. Pick a selection from the drop down list.

UpperCaseOnly - indicate if the field is forced to upper case. Pick a selection from the drop down list.

Width - the width of the column. Enter the width or select "*Calc", which is the default, from the drop down list.

Browse - indicate if the field is read-only. Pick a selection from the drop down list.

EditCode - indicate the punctuation for a numeric field according to the standard RPG edit code rules. Pick a selection from the drop down list (see Edit Code Usage).

EditWord - indicate an edit word that controls the display of blanks, signs, unwanted zeros, commas and decimal points, and named constants (see Edit Word Usage).

Decimals - the number of decimal positions in numeric fields.

Length - the length of the field.

Type - the type of field. Pick a selection from the drop down list.

Name - the name of the field.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | DCLSUBFILEFLD | SubfileField Collection Editor

FirstRow Property

The **FirstRow** property gets or sets the specified row to be displayed at the top of the Subfile.

```
control.FirstRow = <RowIndex>
```

Settings

<RowIndex> is of type *Integer8

Contains the row index, relative to zero.

Remarks

The **FirstRow** property can be set at run-time only.

Setting the **FirstRow** property causes the Subfile to scroll so that the specified row is the topmost row in the Subfile.

FirstRow can also be used to query the row currently at the top of the Subfile.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | Integer Data Types

FormatName Property

The **FormatName** property sets the name of the record format.

```
control.FormatName= <stringVal>
```

Settings

<stringVal> is of type ***String**

Contains the name of the record format.

Remarks

The **FormatName** property can be set at design-time only.

See Also the *FormatName* parameter in the DCLSUBFILE command.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | DCLSUBFILE | String Data Type

GridLineColor Property

The **GridLineColor** property determines the color of the grid lines between cells of a Subfile.

```
control.GridLineColor = System.Drawing.Color.<color>
```

Settings

System.Drawing.Color.< color>

Where **< color>** is one of the valid enumeration values (Red, Green, Blue, etc.). Only solid colors can be set for this property. If you select a dithered color, the closest solid color will be displayed. See Setting Colors for more information on setting colors.

Remarks

This property can be set at design-time or run-time.

The default settings for grid lines is black.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | System.Drawing.Color.<color> | Setting Colors

KeyboardDeleteInsert Property

The **KeyboardDeleteInsert** property sets a boolean value that is used to determine if the user can use the Delete and Insert keys at run-time to delete and insert records. The default is **False**.

```
control.KeyboardDeleteInsert = <booleanVal>
```

Settings

<booleanVal> is of type ***Boolean**.

True enables the use of the Delete and Insert keys at run-time. **False** disables the use of the Delete and Insert keys at run-time (this is the default).

Remarks

The **KeyboardDeleteInsert** property can be set at design-time only.

If the user deletes a record, the **RecordDelete** event will be received prior to the record actually being deleted from the Subfile. If the user inserts a new record, the **RecordNew** event is received by the program.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | *Boolean Data Types

RecordBackColor Property

Gets or sets the text background color for a record (row) in a Subfile.

```
control.RecordBackColor ( <RowIndex> ) = System.Drawing.Color.<color>
```

Settings

<RowIndex> is of type *Integer8

Contains the respective row index, relative to zero, for the specific record.

System.Drawing.Color. <color>

Where **<color>** is one of the valid enumeration values (Red, Green, Blue, etc.). See **Setting Colors** for more information on setting colors.

Remarks

RecordBackColor is available at run-time only.

Setting this property will override the current **BackColor** color in the record with a new color. Only the color of the background in the Subfile record will be changed, headings and row buttons will not be affected.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | FieldBackColor Property | CellBackColor Property | Integer Data Types | System.Drawing.Color

RecordBrowse Property

The **RecordBrowse** property gets or sets the browse state for a record (row) in a Subfile.

```
control.RecordBrowse ( <RowIndex> ) = <booleanVal>
```

Settings

<RowIndex> is of type *Integer8

Contains the respective row index, relative to zero, for the specific record.

<booleanVal> of type *Boolean.

True indicates the record is browse only, editing is not allowed. **False** indicates record editing is allowed (this is the default).

Remarks

RecordBrowse is available at run-time only.

This property is valid only if the Subfile Browse property is **False**. If the Subfile **Browse** property is **True**, setting this property will have no effect.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | CellBrowse Property | FieldBrowse Property | Integer Data Types | *Boolean Data Types

RecordChanged Property

The **RecordChanged** property gets or sets a boolean value indicating when the contents of a record have changed.

```
control.RecordChanged ( <RowIndex> ) = <booleanVal>
```

Settings

<RowIndex> is of type *Integer8

Contains the respective row index, relative to zero, for the specific record. Using this property with an index GREATER than the number of rows in the subfile will generate a runtime error. Use RowCount property for the current number of rows in the subfile.

<booleanVal> of type *Boolean.

True indicates the contents of the record changed. **False** indicates the contents of the record have not changed (this is the default).

Remarks

The **RecordChanged** property can be set at run-time only.

The **RecordChanged** property dictates which records are retrieved with the READC op code. Its function is similar to the AS/400's use of SFLNXTCHG. With the **RecordChanged** property, you can cause Subfile records to be considered changed even though the work station user did not change them.

Note The **RecordChanged** property does not change if the change to the Subfile was made programmatically.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | RowCount Property | Integer Data Types | *Boolean Data Types | READC

RecordForeColor Property

The **RecordForeColor** property gets or sets the text foreground color for a record (row) in a Subfile.

```
control.RecordForeColor ( <RowIndex> ) = System.Drawing.Color.<color>
```

Settings

<RowIndex> is of type *Integer8

Contain the respective row index, relative to zero, for the specific record.

System.Drawing.Color. <color> where <color>

One of the valid enumeration values (Red, Green, Blue, etc.). See Setting Colors for more information on setting colors.

Remarks

RecordForeColor is available at run-time only.

Setting this property will override the current text **ForeColor** for the record with a new color. Only the color of text in the Subfile record will be changed, headings and row buttons will not be affected.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | CellForeColor Property | FieldForeColor Property | Integer Data Types | System.Drawing.Color

Row Property

The **Row** property gets or sets the row of the active cell in the subfile. This property, in conjunction with the **Column** property, defines the coordinates of the active cell in the Subfile. Except as noted below, changing the **Row** and **Column** properties will not change the location of the active cell.

```
control.Row = <integerVal>
```

Settings

<integerVal> is of type *Integer8

Contain the row index, relative to zero, for the specific record.

Remarks

The **Row** property is available at run-time only.

The **Row** and **Column** properties must be set by the application prior to updating the **CellData** property in order to specify which cell's data to set.

To position or set focus on a particular cell, use one of the **Select** methods.

The **Row** property retains its last set value regardless of the number of active rows in the subfile.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | Column Property | CellData | Select Method | Integer Data Types

RowCount Property

The **RowCount** property gets the current number of rows in the Subfile.

```
<integerVal> = control.RowCount
```

Settings

<integerVal> of type *Integer8

Containing the number of rows currently in the subfile.

Remarks

The **RowCount** property can be used at run-time only and is read-only.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | Integer Data Types

RowSelected Property

The **RowSelected** property is an array property to get or set a boolean value indicating which row(s) the user has selected.

```
control.RowSelected( <RowIndex> ) = <booleanVal>
```

Settings

<RowIndex> is of type ***Integer8**

Contains the respective row index, relative to zero, for the specific record. Using this property with an index **Greater** than the number of rows in the Subfile will generate a runtime error. Use the RowCount property or the current number of rows in the subfile.

<booleanVal> of type ***Boolean**.

True indicates the record is selected. **False** indicates the record is not selected (this the default).

Remarks

The **RowSelected** property is available at run-time only.

Example

```
// Display each row that was selected in the Subfile
DclFld I   Type(*packed) Len(2,0)
DclFld I$ Type(*char) Len(2)
Do 0 ToVal(Subfile1.RowCount -1)I
  If Subfile1.RowSelected(I) = *ON
    I& = I
    MsgBox Msg("The selected row is +I$")
  Endif
EndDo
```

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | RowCount Property | Integer Data Types | *Boolean Data Types

RRNName Property

The **RRNName** property contains the name of a field used as the relative record number for UPDATE and WRITE operations. The RRN is also returned into this variable for Subfile CHAIN and READC operations.

```
control.RRNName = <stringVal>
```

Settings

<stringVal> is of type ***String**

Contains the name of the field containing the relative record number.

Remarks

The **RRNName** property can be set at design-time only. Note, the content of this property remains unchanged when the Subfile is cleared.

See Also the *RRNName* parameter in the command.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | SubfileControl Members | String Data Types | DCLSUBFILE | UPDATE | WRITE | CHAIN | READC

SingleSelect Property

The **SingleSelect** property sets a boolean value indicating whether the user can only select an individual cell, or both an individual cell and a range of cells (this is the default).

```
control.SingleSelect = <booleanVal>
```

Settings

<booleanVal/> of type ***Boolean**.

True allows the user to only select an individual cell. **False** allows the user to select an individual cell or a range of cells (this is the default).

Remarks

The **SingleSelect** property can be set at design-time only.

See Also

ASNA.VisualRPG.Windows.Controls | SubfileControl | Subfile Control Members | *Boolean Data Types

UpperCaseOnly Property

The **UpperCaseOnly** property sets a boolean indicator whether to force the entry of UPPERCASE characters on a field.

```
control.UpperCaseOnly = <booleanVal>
```

Settings

<booleanVal/> of type ***Boolean**.

True forces the entry of uppercase characters. **False** does not force the entry of uppercase characters. This is the default.

Remarks

The **UpperCaseOnly** property can be set at design-time only in the Fields (**SubfileFields Collection Editor**) dialog box. See Also the *UpperCaseOnly* parameter when the field is defined with DCLSUBFILEFLD.

Only lowercase letters are converted to uppercase; all uppercase letters and non-alphabetic characters remain unchanged.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | SubfileField Collection Editor | DCLSUBFILEFLD | *Boolean Data Types

WholeRows Property

The **WholeRows** property sets a boolean value that indicates whether selection to an entire row occurs when an individual cell is selected.

```
control.WholeRows = <booleanVal>
```

Settings

<booleanVal/> of type ***Boolean**.

True enables the user to set a selection to an entire row when an individual cell is selected.

False disables selecting an entire row when an individual cell is selected. This is the default.

Remarks

The **WholeRows** property can be set at design-time only.

When this property is **True**, the Subfile functions in the same manner as when the Browse property is set to **True**, in that user update of the Subfile is prohibited.

See Also

ASNA.VisualRPG.Windows.Controls | Subfile Control | SubfileControl Members | Browse Property | *Boolean Data Types

Constants

Hexadecimal Numeric Constant

Hexadecimal numeric constants take the form: **H<hex digits>** where <hex digits> is a string of hexadecimal characters (0 to 9 and A to F) enclosed in single quotes (//), with a length from 1 to 16 characters.

Remarks

Hex constants are another way to declare integral constants. The size of the integral depends on the length of the hex string (and not on the value of the hex number):

If the length is from 1 to 2 digits, then the type is ***Byte**.

if the length is from 3 to 4 digits, the type is ***Integer2**.

if the length is from 5 to 8 digits, the type is ***Integer4**.

if the length is from 9 to 16 digits, the type is ***Integer8**.

So H'00000001' will be a number 1 with type ***Integer 4**, while H'1' will be a number 1 with type ***Byte**.

Example

```
H'34B'  
H'800040ef'
```

See Also

Hexadecimal Literal | *Byte Data Type | *Integer2 Data Type | *Integer4 Data Type | *Integer8 Data Type

Data Types

*Binary Data Type

The ***Binary data type** means that the sign (positive or negative) is in the leftmost bit of the field and the numeric value is in the remaining bits of the field.

Positive numbers have a zero in the sign bit; negative numbers have a one in the sign bit and are in twos complement form. A binary field can be from one to nine digits in length and can be defined with decimal positions.

If the length of the field is from five to nine digits, the compiler assumes a binary field length of 4 bytes.

The default initialization value for numeric fields is zero.

See Also

Data Types

*Boolean Data Type

The ***Boolean** data type is a special case of an integer data type.

***Boolean** data types are used with ***True** or ***False** and can be used wherever indicators and character values '1' and '0' are used, such as with DCLFLD and DCLSRPDM.

Character fields (*Char, *String).

If the contents are all '0' (character 0) then the character string converts to ***False** when assigned to a ***Boolean**. If one of the characters is not '0' then the character string maps to ***True**. Exception to this rule (or special case) is the empty ***String**, which maps to ***False**.

Numbers (*Packed, *Zoned, *Binary, *DECIMAL | *FLOAT, *Integer).

If all digits in the number equal 0 (i.e. if the number is 0) then it maps to ***False**, otherwise it maps to ***True**.

Indicators

Indicators are characters, so they follow the character fields rule.

The equivalent .NET data type is System.Boolean.

Example

```
Dclfld myBool type(*Boolean)
myBool = *True
myBool = *False
```

See Also

Data Types | ***True** | ***False** | DCLFLD | Integer Data Type | DCLSRPDM | System.Boolean

*Byte Data Type

***Byte** variables are stored as unsigned 8-bit (1-byte) numbers ranging in value from 0 through 255.

The ***Byte** data type is used for containing binary data.

The equivalent .NET data type is System.Byte.

See Also

Data Types | Integer Data Type | System.Byte

*Char Data Type

The ***Char** data type is a finite-length string of Unicode text characters that compose an RPG field. Internally, this field is characterized as a string, but, as a field, the length of the string is always the exact length of the ***Char** field definition.

For Example:

```
dclFld CustName type( *char )      Len( 35 )
dclFld myStr      type( *string )
myStr = CustName.Trim()           // trim white space
```

See Also

Data Types | System.String | Unicode

*Date Data Type

***Date** variables are stored as IEEE 64-bit (8-byte) integers that represent dates ranging from January 1 of the year 1 through December 31 of the year 9999, and times from 0:00:00 (midnight) through 11:59:59 PM.

***Date** constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Also, dates used for I/O operations such as input fields, output fields or key fields are also converted (if required) to the necessary format for the operation.

***Date** fields have a predetermined size and format. The default internal format for date variables is *ISO.

There are two kinds of date data formats: 2-digit and 4-digit year formats. For 2-digit year formats, years in the range 1940 to 2039 can be represented. This leads to the possibility of a date overflow condition occurring when converting from a 4-digit year format to a 2-digit year format.

See the following topics for the date data type formats and ***LOVAL**, ***HIVAL** and default values.

Date Formats

Default Values for all Date Formats

The equivalent .NET data type is System.DateTime.

Example

For Examples on how to code date fields, see the Examples in the following topics:

Date Operations

ADDDUR

MOVE

Moving Date and Time

EXTRACT
SUBDUR
TESTTIME

See Also

Data Types | System.DateTime | Date and Time

*Decimal Data Type

The ***Decimal** data type variables are stored as signed 128-bit (16-byte) integers scaled by a variable power of 10.

The scaling factor specifies the number of digits to the right of the decimal point; it ranges from 0 through 28. With a scale of 0 (no decimal places), the largest possible value is +/- 79,228,162,514,264,337,593,543,950,335. With 28 decimal places, the largest value is +/- 7.9228162514264337593543950335, and the smallest nonzero value is +/- 0.00000000000000000000000001 (+/-1E-28).

The equivalent .NET data type is **System.Decimal**.

See Also

Data Types

*Float Data Type

The ***Float** data type defines a floating-point field. The decimal positions must be left blank. However, floating-point fields are considered to have decimal positions. As a result, float variables may not be used in any place where a numeric value without decimal places is required, such as an array index, do loop index, etc.

The length of a floating-point field is defined in terms of the number of bytes. It must be specified as either 4 or 8 bytes. Use ***Float4** to set to 4 bytes and ***Float8** to set to 8 bytes.

Negative values have the same range, but with a negative sign.

***Float** variables conform to the IEEE standard as supported by the AS/400. Since float variables are intended to represent 'scientific' values, a numeric value stored in a float variable may not represent the exact same value as it would in a packed variable. Float should not be used when you need to represent numbers exactly to a specific number of decimal places, such as monetary amounts.

The default initialization value for numeric fields is zero.

See Also

Data Types | *Float4 | *Float8

*Float4 Data Type

The ***Float4** data type defines a floating-point field of 4 bytes. The decimal positions must be left blank. However, floating-point fields are considered to have decimal positions. As a result, float variables may not be used in any place where a numeric value without decimal places is required, such as an array index, do loop index, etc.

Use ***Float8** to specify a floating-point field of 8 bytes.

Negative values have the same range, but with a negative sign.

***Float4** variables conform to the IEEE standard as supported by the AS/400. Since float variables are intended to represent 'scientific' values, a numeric value stored in a float variable may not represent the exact same value as it would in a packed variable. Float should not be used when you need to represent numbers exactly to a specific number of decimal places, such as monetary amounts.

The default initialization value for numeric fields is zero.

The equivalent .NET data type is **System.Single**.

See Also

Data Types | *Float8

*Float8 Data Type

The ***Float8** data type defines a floating-point field to 8 bytes. The decimal positions must be left blank. However, floating-point fields are considered to have decimal positions. As a result, float variables may not be used in any place where a numeric value without decimal places is required, such as an array index, do loop index, etc.

Use ***Float4** to specify a floating-point field to 4 bytes.

Negative values have the same range, but with a negative sign.

***Float8** variables conform to the IEEE standard as supported by the AS/400. Since float variables are intended to represent 'scientific' values, a numeric value stored in a float variable may not represent the exact same value as it would in a packed variable. Float should not be used when you need to represent numbers exactly to a specific number of decimal places, such as monetary amounts.

The default initialization value for numeric fields is zero.

The equivalent .NET data type is **System.Double**.

See Also

Data Types | *Float4

*Form Data Type

The ***Form** data type specifies that the variable contains a windows handle of a generic form. You then have access to all forms and all controls in a project.

In referencing a form, you can use either a form name or ***FORM**.

A form name allows you to reference or have access to a particular form and its controls.

***Form** allows you to reference or have access to any form in a project.

The equivalent .NET data type is **System.Form**.

See Also

Data Types

*Ind Data Type

The ***Ind** data type specifies that the variable is an indicator.

When assigning resulting indicators, remember the following:

Resulting indicators cannot be used when the result field refers to an entire array.

If the same indicator is used to test the result of more than one operation, the last operation processed determines the setting of the indicator.

The same indicator can be used to test for more than one condition depending upon the operation specified.

See Also

Data Types

*Integer Data Type

The ***Integer** data type is similar to ***Binary**, with these exceptions:

The integer format allows the full range of binary values.

The number of decimal positions for an integer field is always zero.

Its length is defined in bytes, rather than digits.

The length of an integer field is defined in terms of number of bytes; it can be either 2 or 4 bytes long.

The range of values allowed for an integer field depends on its length.

Field Length Range of Allowed Values

2-byte integer

-32768 to 32767

4-byte integer

2147483648 to 2147483647

The equivalent .NET data type is **System.Int32**.

You can also use the following data types to automatically set the length of the data type.

***Integer2** - sets the length to a 2-byte integer.

***Integer4** - sets the length to a 4-byte integer.

***Integer8** - sets the length to an 8-byte integer.

See Also

Data Types | ***Integer2** | ***Integer4** | ***Integer8**

*Integer2 Data Type

The ***Integer2** data type is similar to ***Binary**, with these exceptions:

The integer format allows the full range of binary values.

The number of decimal positions for an integer field is always zero.

Its length is defined in bytes, rather than digits.

***Integer2** automatically specifies the length as a 2-byte integer with the following field length range

-32768 to 32767

You can also use the other data types to automatically set the length to a 4 or 8-byte integer.

***Integer4** - sets the length to a 4-byte integer.

***Integer8** - sets the length to an 8-byte integer.

The equivalent .NET data type is **System.Int16**.

See Also

Data Types | [*Integer](#) | [*Integer4](#) | [*Integer8](#)

*Integer4 Data Type

The ***Integer4** data type is similar to ***Binary**, with these exceptions:

The integer format allows the full range of binary values.

The number of decimal positions for an integer field is always zero.

Its length is defined in bytes, rather than digits.

***Integer4** automatically specifies the length as a 4-byte integer with the following field length range

-2147483648 to 2147483647

You can also use the other data types to automatically set the length to a 2 or 8-byte integer.

***Integer2** - sets the length to a 2-byte integer.

***Integer8** - sets the length to an 8-byte integer.

The equivalent .NET data type is **System.Int32**

See Also

Data Types | [*Integer](#) | [*Integer2](#) | [*Integer8](#)

*Integer8 Data Type

The ***Integer8** data type is similar to ***Binary**, with these exceptions:

The integer format allows the full range of binary values.

The number of decimal positions for an integer field is always zero.

Its length is defined in bytes, rather than digits.

***Integer8** automatically specifies the length as an 8-byte integer.

You can also use the other Integer data types to automatically set the length to a 2 or 4-byte integer.

***Integer2** - sets the length to a 2-byte integer.

***Integer4** - sets the length to an 4-byte integer.

The equivalent .NET data type is **System.Int64**.

See Also

Data Types | [*Integer](#) | [*Integer2](#) | [*Integer4](#)

*IntPtr Data Type

The ***IntPtr** data type is a platform-specific type that is used to represent a pointer or a handle.

The ***IntPtr** type is designed to be an integer whose size is platform-specific. That is, an instance of this type is expected to be 32-bits on 32-bit hardware and operating systems, and 64-bits on 64-bit hardware and operating systems.

The ***IntPtr** type can be used by languages that support pointers, and as a common means of referring to data between languages that do and do not support pointers.

***IntPtr** objects can also be used to hold handles.

The equivalent .NET data type is **System.IntPtr**.

See Also

Data Types

*OneChar Data Type

The ***OneChar** value type represents a Unicode character, also called a Unicode code point, and is implemented as a 16-bit number ranging in value from hexadecimal 0x0000 to 0xFFFF.

The default initialization value for a ***OneChar** field is 0x0000. See Also **system.char** structure.

See Also

Data Types | [*Char Data Type](#) | [Unicode](#) | [OneChar Literals](#)

*Packed Data Type

The ***Packed** data type means that each byte of storage (except for the low order byte) can contain two decimal numbers. The low-order byte contains one digit in the left-most portion and the sign (positive or negative) in the right-most portion. The standard signs are used: hexadecimal **F** for positive numbers and hexadecimal **D** for negative numbers.

The sign portion of the low-order byte indicates whether the numeric value represented in the digit portions is positive or negative.

In AVR for .NET, the maximum length for a packed field is 28,0.

See Also

Data Types

*String Data Type

The ***String** data type (unicode) allows you to pass a parameter to a procedure even though the data type is not precisely known to the called procedure, (for Example, different types of strings). In these instances you can use operational descriptors to provide descriptive information to the called procedure regarding the form of the parameter. The additional information allows the procedure to properly

interpret the string. You should only use operational descriptors when they are expected by the called procedure.

See Also

Data Types | *Char Data Type | Unicode

*Time Data Type

Time constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Also, times used for I/O operations such as input fields, output fields or key fields are also converted (if required) to the necessary format for the operation.

Time fields have a predetermined size and format.

The default format for time variables is *ISO. Leading and trailing zeros are required for all time data.

See the following topics for the ***Time** data type formats and *LOVAL, *HIVAL and default values.

Time Formats

Default Values for all Date Formats

Examples

For Examples on how to code time fields, see the Examples in the following topics:

Date Operations

Moving Date and Time

ADDUR

EXTRACT

MOVE

SUBDUR

TESTTIME

See Also

Date and Time | Data Types

*Timestamp Data Type

Timestamp fields have a predetermined size and format. They can be defined on the definition specification and must be in the following format:

```
yyyy-mm-dd-hh.mm.ss.mmmmmm (length 26)    // This is *ISO
```

Microseconds (.mmmmmm) are optional for **Timestamp literals** and if not provided will be padded on the right with zeros. Leading zeros are required for all timestamp data.

The following are the default, ***LOVAL** and ***HIVAL** values for ***Timestamp** data type.

The default initialization value for a timestamp is midnight of January 1, 0001 (0001-01-01-00.00.00.000000).

The ***LOVAL** value for timestamp is 0001-01-01-00.00.00.000000.

The ***HIVAL** value for a timestamp is 9999-12-31-24.00.00.000000.

***ISO** is the only valid type for a ***Timestamp** data type.

The equivalent .NET data type is System.DateTime.

Remarks

Design-time controls (DTC's) are no longer supported.

See Also

Date and Time | Data Types | ***LOVAL** | ***HIVAL** | System.DateTime

*Zoned Data Type

The ***Zoned** data type means that each byte of storage can contain one digit or one character. In the zoned-decimal format, each byte of storage is divided into two portions: a 4-bit zone portion and a 4-bit digit portion. The zoned-decimal format looks like this:

The **zone** portion of the low-order byte indicates the sign (positive or negative) of the decimal number. The standard signs are used: hexadecimal **F** for positive numbers and hexadecimal **D** for negative numbers. In zoned-decimal format, each digit in a decimal number includes a zone portion; however, only the low-order zone portion serves as the sign.

You can specify an alternative sign format for zoned-decimal format. In the alternative sign format, the numeric field is immediately preceded or followed by a + or - sign. A plus sign is a hexadecimal 4E, and a minus sign is a hexadecimal 60.

When an alternative sign format is specified, the field length (specified on the input specification) must include an additional position for the sign. For Example, if a field is 5 digits long and the alternative sign format is specified, a field length of 6 positions must be specified.

See Also

Data Types

Date and Time

Adding or Subtracting Dates

When adding (or subtracting) a duration in months to (or from) a date, the general rule is that the month portion is increased (or decreased) by the number of months in the duration, and the day portion is unchanged. The exception to this is when the resulting day portion would exceed the actual number of days in the resulting month. In this case, the resulting day portion is adjusted to the actual month end date.

For Example:

Adding one month to '97/08/30' (*YMD format) results in '97/09/30', as expected. The resulting month portion has been increased by 1; the day portion is unchanged. On the other hand, adding one month to '97/08/31' results in '97/09/30'. The resulting month portion has been increased by 1 and the resulting day portion has been adjusted because September only has 30 days.

Subtracting one month from '95/03/30' yields '95/02/28'. In this case, the resulting month portion is decreased by 1 and the resulting day portion adjusted because February has only 28 days (in non-leap years).

Adding one year to '92/02/29' results in '93/02/28', an adjusted value since the resulting year is not a leap year.

See Also

[Date and Time](#) | [ADDDUR](#) | [Calculating Durations between Dates](#) | [SUBDUR](#)

Calculating Durations between Dates

Use SUBDUR to calculate a duration by subtracting two dates, times, or timestamps. The result of the calculation is in whole units, and any rounding is done downwards. The calculation of durations includes microseconds.

For Example:

A duration of 59 minutes requested in hours will result in 0 hours.

A duration is 374 days requested in years yields 1 complete year because there are 1.02 years in 374 days.

If adjustment takes place on a date-time addition or subtraction, then a subsequent calculation will most likely result in a different duration than the one original one added or subtracted. This is because the subsequent duration will no longer contain a complete unit, and rounding down will yield one unit less than expected.

See Also

[Date and Time](#) | [ADDDUR](#) | [Adding or Subtracting Dates](#) | [SUBDUR](#)

*Date Data Type

***Date** variables are stored as IEEE 64-bit (8-byte) integers that represent dates ranging from January 1 of the year 1 through December 31 of the year 9999, and times from 0:00:00 (midnight) through 11:59:59 PM.

***Date** constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Also, dates used for I/O operations such as input fields, output fields or key fields are also converted (if required) to the necessary format for the operation.

***Date** fields have a predetermined size and format. The default internal format for date variables is ***ISO**.

There are two kinds of date data formats: 2-digit and 4-digit year formats. For 2-digit year formats, years in the range 1940 to 2039 can be represented. This leads to the possibility of a date overflow condition occurring when converting from a 4-digit year format to a 2-digit year format.

See the following topics for the date data type formats and ***LOVAL**, ***HIVAL** and default values.

Date Formats

Default Values for all Date Formats

The equivalent .NET data type is System.DateTime.

Example

For Examples on how to code date fields, see the Examples in the following topics:

Date Operations

ADDDUR

MOVE

Moving Date and Time

EXTRACT

SUBDUR

TESTTIME

See Also

Data Types | System.DateTime | Date and Time

Date Formats

The following is a list of the date formats and separators for ***Date** data type.

RPG Name	Description	Format (Default Separator)	Valid Separators	Length	Example
*ISO	International Standards Organization	yyyy-mm-dd	-	10	1996-01-15
*JUL	Julian	yy/ddd	/	6	96/015
*USA	IBM USA Standard	mm/dd/yyyy	/	10	01/15/1996

*EUR	IBM European Standard	dd.mm.yyyy	.	10	15.01.1996
*JIS	Japanese Industrial Standard Christian Era	yyyy-mm-dd	-	10	1996-01-15
*MDY	Month/Day/Year	mm/dd/yy	/	8	01/15/96
*DMY	Day/Month/Year	dd/mm/yy	/	8	15/01/96
*YMD	Year/Month/Day	yy/mm/dd	/	8	96/01/15

The date types *YMD, *DMY, *MDY and *JUL are only valid for the years ranging from 1940 to 2039. *ISO, *USA, *EUR and *JIS date types support years ranging from 0001 to 9999.

See Also

Date and Time | *HIVAL | *LOVAL | Time Formats | Default Values for all Date Formats

Date Operations

Date operations allow you to extract portions of a date, time or timestamp field, perform date and time arithmetic; or test for valid date and time fields.

The date operation codes are used for date, time, timestamp fields, and character and numeric fields representing dates, times and timestamps.

The date operation codes are:

ADDDUR

Add a duration to a date or time.

EXTRACT

Extract part of a date, time or timestamp.

SUBDUR

Subtract a duration from a date or time, or calculate the duration between 2 dates, times or timestamps.

TESTTIME

Test for a valid date, time, or timestamp field.

See Also

Date and Time | Duration Codes

Default Values for all Time Formats

Format	Description	*LOVAL	*HIVAL	Default
name				Value
*HMS	Hours:Minutes:Seconds	00:00:00	24:00:00	00:00:00
*ISO	International Standards Organization	00.00.00	24.00.00	00.00.00
*USA	IBM USA Standard. AM and PM can be any mix of upper and lower case.	00:00 AM	12:00 PM	00:00 AM
*EUR	IBM European Standard	00.00.00	24.00.00	00.00.00
*JIS	Japanese Industrial Standard Christian Era	00:00:00	24.00.00	00.00.00

See Also

[Date and Time](#) | [*HIVAL](#) | [*LOVAL](#) | [Date Formats](#) | [Default Values for all Time Formats](#)

Default Values for all Date Formats

Format	Description	*LOVAL	*HIVAL	Default
Name				Value
*MDY	Month/Day/Year	01/01/40	12/31/39	01/01/40
*DMY	Day/Month/Year	01/01/40	31/12/39	01/01/40
*YMD	Year/Month/Day	40/01/01	39/12/31	40/01/01
*JUL	Julian	40/001	39/365	40/001
*ISO	International Standards Organization	0001-01-01	9999-12-31	0001-01-01
*USA	IBM USA Standard	01/01/0001	12/31/9999	01/01/0001
*EUR	IBM European Standard	01.01.0001	31.12.9999	01.01.0001
*JIS	Japanese Industrial Standard Christian Era	0001-01-01	9999-12-31	0001-01-01

See Also

[Date and Time](#) | [*HIVAL](#) | [*LOVAL](#) | [Date Formats](#) | [Default Values for all Time Formats](#)

Duration Codes

The valid duration codes (and there short forms) are:

- *DAYS for the day (*D)
- *MONTHS for the month (*M)
- *YEARS for the year (*Y)
- *MSECONDS for the microseconds (*MS)
- *SECONDS for the seconds (*S)
- *MINUTES for the minutes (*MN)
- *HOURS for the hours (*H)

Remarks

Use *Years (*Y), *Months (*M) and *Days (*D) if value is a **Date** or **Timestamp**.

Use *Hours (*H), *Minutes (*MN), *Seconds (*S) and *MSeconds (*MS) when value is a **Time** or **Timestamp**.

See Also

Date and Time

Moving Date-Time Data

The following considerations are provided for the **MOVE** and **MOVEL** operation codes when moving Date, Time, and Timestamp data type fields.

The combinations below are allowed for the **MOVE** and **MOVEL** operation codes:

- Date to Date
- Time to Time
- Timestamp to Timestamp
- Date to Timestamp
- Time to Timestamp (sets micro-seconds to 000000)
- Timestamp to Date
- Timestamp to Time
- Date to Character or Numeric
- Time to Character or Numeric
- Timestamp to Character or Numeric
- Character or Numeric to Date
- Character or Numeric to Time
- Character or Numeric to Timestamp

Remarks

TimFmt must be blank if:

The source or target of the move is a timestamp field.

Both the source and the target of the move are date or time fields. Otherwise, TimFmt contains the date or time format compatible with the character or numeric field that is the source or target of the operation.

Any valid format can be specified.

Keep in mind the following when specifying TimFmt:

Time format *USA is not allowed for movement between Time and numeric classes.

A two-digit year format (*MDY, *DMY and *JUL) can only represent dates in the range 1940 through 2039. An error occurs if converting to a two- or three-digit year format for dates outside this range.

When moving a value to a date field, keep in mind that **TimFmt** determines the number of digits moved.

Source is required and may contain a character or numeric, Date, Time, or Timestamp field; array; array element; a literal; or a named constant, to be converted.

The following rules apply to Source:

In general, if the field specified in **Source** is character, it must have separator characters, for Example 09/06/91.

Separator characters must be valid for the specified format.

If **Source** is not a valid representation of a date or time or its format does not match the format specified in TimFmt, an error occurs.

If Source contains *DATE, the **TimFmt** entry must be a date format with a 4-digit year.

The Target must be a Date, Time, or Timestamp field or a numeric or character class field, array or array element. The date or time is placed in **Target** according to its defined format or the format code specified in **TimFmt**. If **Target** is numeric, separator characters are removed prior to the operation. The length used is the length after removing the separator characters.

When moving from a **Date** to a **Timestamp** field, the time and microsecond portion of the timestamp are unaffected. However, an error occurs if the entire timestamp is not valid.

When moving from a **Time** to a **Timestamp** field, the microsecond's part of the timestamp is set to 000000. The date portion remains unaffected, but an error occurs if the entire timestamp is not valid.

See Also

Date and Time | Date Data Type | MOVE | MOVEL | Time Data Type | Timestamp Data Type

*Time Data Type

Time constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Also, times used for I/O operations such as input fields, output fields or key fields are also converted (if required) to the necessary format for the operation.

Time fields have a predetermined size and format.

The default format for time variables is *ISO. Leading and trailing zeros are required for all time data.

See the following topics for the *Time data type formats and *LOVAL, *HIVAL and default values.

Time Formats

Default Values for all Date Formats

Examples

For Examples on how to code time fields, see the Examples in the following topics:

Date Operations

Moving Date and Time

ADDDUR

EXTRACT

MOVE

SUBDUR

TESTTIME

See Also

Date and Time | Data Types

Time Formats

The following is a list of the Time formats and separators for *Time data type.

RPG name	Description	Format (Default Separator)	Valid Sepa- rators	Length	Example
*ISO	International Standards Organization	hh.mm.ss	.	8	13.00.00
*HMS	Hours:Minutes:Seconds	hh:mm:ss	:	8	13:00:00
*USA	IBM USA Standard. AM and PM can be any mix of upper and lower case.	hh:mm AM or hh:mm PM	:	8	01:00 PM
*EUR	IBM European Standard	hh.mm.ss	.	8	13.00.00
*JIS	Japanese Industrial Standard Christian Era	hh:mm:ss	:	8	13:00:00

See Also

Date and Time | *HIVAL | *LOVAL | Date Formats | Default Values for all Time Formats

*Timestamp Data Type

Timestamp fields have a predetermined size and format. They can be defined on the definition specification and must be in the following format:

```
yyyy-mm-dd-hh.mm.ss.mmmmmm (length 26)    // This is *ISO
```

Microseconds (.mmmmmm) are optional for **Timestamp literals** and if not provided will be padded on the right with zeros. Leading zeros are required for all timestamp data.

The following are the default, ***LOVAL** and ***HIVAL** values for ***Timestamp** data type.

The default initialization value for a timestamp is midnight of January 1, 0001 (0001-01-01-00.00.00.000000).

The ***LOVAL** value for timestamp is 0001-01-01-00.00.00.000000.

The ***HIVAL** value for a timestamp is 9999-12-31-24.00.00.000000.

***ISO** is the only valid type for a ***Timestamp** data type.

The equivalent .NET data type is System.DateTime.

Remarks

Design-time controls (DTC's) are no longer supported.

See Also

Date and Time | Data Types | ***LOVAL** | ***HIVAL** | System.DateTime

Figurative Constants

*ALL'x..'

***ALL'x...'** will cyclically repeat 'x..' to a length equal to the associated field. If the field is a numeric field, all characters within the string must be numeric (0 through 9). No sign or decimal point can be specified when ***ALL'x..'** is used as a numeric constant. You cannot use ***ALL'x..'** with numeric fields of float format.

Remarks

***ALL** is valid only for character or numeric fields. Numeric fields cannot be float format.

For numeric integer or unsigned fields, the value is never greater than the maximum value allowed for the corresponding field. For Example, ***ALL'95'** represents the value 09595 if the corresponding field is a 5-digit integer field, since 95959 is greater than the maximum value allowed for a 5-digit signed integer.

See Also

Figurative Constants | Figurative Constants Rules | Data Types

***BLANK/*BLANKS**

***BLANK/*BLANKS** will move blanks into the designated field.

Remarks

***BLANK/*BLANKS** is valid only for character, graphic, or UCS-2 fields.

The value for character is // ' (blank) or X'40'.

The value for graphic is X'4040'.

The value for UCS-2 is X'0020'.

***BLANK/*BLANKS** storage equivalent to the implied length of **Blanks** is used.

The performance of ***BLANK/*BLANKS** is equal to using spaces when the implied length of **Blanks** is less than or equal to 140.

See Also

[Figurative Constants](#) | [Figurative Constants Rules](#)

***BASE**

***BASE** refers to the base class in the .NET framework; **System.Object**.

Remarks

The **System.Object** class provides a common base class for all objects in the .NET framework, ensuring interoperability of objects developed using Visual RPG.

New classes implicitly inherit the **System.Object** class.

See Also

[Figurative Constants](#) | [Figurative Constants Rules](#)

***ByRef Operator**

The ***ByRef** operator is used to declare that the associated reference parameter is being passed by reference.

Example

```
myObject.Mymethod( Parm1, *ByRef Parm2, Parm3 )    // Parm2 is passed by  
reference
```

See Also

[Figurative Constants](#) | [Operators](#)

*Date

***Date** is a special word that allows the programmer to supply the date at run-time. These special words give the date and time when the project was started. Use the TIME operation code to obtain the current time and date while the program is running.

Remarks

The user dates can be written at output time, and are written in the format specified by the control specification.

These special words can be used in three different date formats:

Month/day/year

Year/month/day

Day/month/year

***Date** gives an 8-character (4-digit year) numeric date field.

***Date** can be edited by using the Y edit code.

User date fields are not ***Date data type** fields, but are numeric fields.

Example

```
IOFIELD1 = *DATE
```

See Also

Figurative Constants | *Day | *Month | TIME | *Year | Figurative Constants Rules

*Day

***Day** is a special word that gives the day the project was started.

Remarks

***Day** and ***Month**, will print a 2-position numeric date field when specified.

See Also

Figurative Constants | *Date | *Month | Figurative Constants Rules

*FALSE

***FALSE** and ***TRUE** special values are used in conjunction with ***Boolean** data type to indicate a true or false condition.

***FALSE** can be used wherever indicators and character values '1' and '0' are used.

Example

```
Dclfld myBool Type (*Boolean)
myBool = *True
myBool = *False
```

See Also

Figurative Constants | *Boolean | *True | Figurative Constants Rules | Figurative Constants Rules

***HIVAL**

***HIVAL** moves the highest collating value for the field type.

Field Types	Descriptions
For Character Fields:	*HIVAL moves the highest collating character for the system (hexadecimal FFs).).
For Graphic Fields:	*HIVAL moves the highest collating character for the system (hexadecimal FF's).
For Numeric Fields:	*HIVAL moves the maximum value allowed for the corresponding field (with a positive sign if applicable).
	*HIVAL for 4-byte float = 3.402 823 5E38 (x'FF7FFFFF')
For Float Fields:	*HIVAL for 8-byte float = 1.797 693 134 862 315 E308 (x'FFEEEEEEEEEEEE')
For Date Fields:	See Default Values for all Date Formats for *HIVAL values.
For Time Fields:	See Default Values for all Time Formats for *HIVAL values.
For TimeStamp Fields:	*HIVAL moves 9999-12-31-24.00.00.000000.

See Also

Figurative Constants | Default Values for all Date Formats | Default Values for all Time Formats |
***LOVAL** | Figurative Constants Rules

***LOVAL**

***LOVAL** moves the lowest collating value for the field type.

Field Types	Descriptions
For Character Fields:	*LOVAL moves the lowest collating character for the system (hexadecimal zeros).
For Graphic Fields:	*LOVAL moves the lowest collating character for the system (hexadecimal zeros).
For Numeric Fields:	*LOVAL moves all 9's with a negative sign.
	*LOVAL for 4-byte float = -3.402 823 5E38 (x'7F7FFFFF')
For Float Fields:	*LOVAL for 8-byte float = -1.797 693 134 862 315 E308 (x'7FEFFFFFFF')
For Date Fields:	See Default Values for all Date Formats for *LOVAL values.
For Time Fields:	See Default Values for all Time Formats for *LOVAL values.
For TimeStamp Fields:	*LOVAL moves 0001-01-01-00.00.00.000000.

See Also

Figurative Constants | Default Values for all Date Formats | Default Values for all Time Formats | *HIVAL
| Figurative Constants Rules

*Month

*Month is a special word that gives the month the project was started.

Remarks

*Month (and *Day and *Year) will print a 2-position numeric date field when specified in the output specifications.

See Also

Figurative Constants | *Day | *Date | *Year | Figurative Constants Rules

*NOTHING

Setting a reference type variable - that is, an object variable to *Nothing means that the variable is not associated with any object. For Example:

```
Dclfld MyObject Type( *Object ) New() // instances MyObject
MyObject = *Nothing                // No object currently referred to.
```

When you assign *Nothing to an object variable, it no longer refers to any object instance. If the variable had previously referred to an instance, setting it to *Nothing does not terminate the instance itself. The instance is terminated and the memory and system resources associated with it are released, only after the garbage collector detects there are no active references remaining.

See Also

Figurative Constants | Figurative Constants Rules

*ON

*ON moves all ones ('1' or X'F1').

Remarks

*ON is valid for boolean, character and indicator fields. Setting a boolean value to *ON sets the value to True.

See Also

Figurative Constants | *OFF | Figurative Constants Rules

*This

*This is a variable that refers to the current object, or form that your program is currently executing in.

*This allows you to access the currently executing forms properties and methods or run any command that requires a form variable (**SHOW**, **HIDE**, **UNLOAD**, etc.) without having to ensure that a form variable is declared and set to the desired form.

***This** is the equivalent of the **Me** keyword that is used in other languages such as Visual Basic and C#. Using ***This** is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure or program.

Remarks

Design-time controls (DTC's) are no longer supported.

Example

The current form is closed in the following Example.

```
*This.Close()
```

See Also

Figurative Constants | Figurative Constants Rules

*Year

***Year** is a special word that allows the programmer to supply a year that the project was started. These special words access the job date that is specified in the job description.

Remarks

***Year** will print a 4-position numeric date field.

User date fields are not ***Date data type** fields but are numeric fields.

See Also

Figurative Constants | *Day | *Date | *Month | Figurative Constants Rules

*TRUE

***True** and ***False** special values are used in conjunction with ***Boolean** data type to indicate a true or false condition.

***TRUE** can be used wherever Indicators and character values '1' and '0' are used.

Example

```
DclFld myBook Type( *Boolean )  
myBool = *True  
myBool = *False
```

See Also

Figurative Constants | *Boolean Data Type | *FALSE

*ZERO/*ZEROS

***ZERO/*ZEROS** moves either all zeros ('0' or 'X'F0') to character or numeric fields, or '0 E0' to numeric float fields.

See Also

Figurative Constants | Figurative Constants Rules

***OFF**

***OFF** moves all zeros ('0' or 'X'F').

Remarks

***OFF** is valid for boolean, character and indicator fields. Setting a boolean value to ***OFF** sets the value to **False**.

See Also

Figurative Constants | Figurative Constants | ***ON** | Figurative Constants Rules

Language Elements
Remarks

As you're writing code or reading the code Examples, you will encounter the comment symbols (`//` or start with a `/*` and ended with a `*/`). These symbols tell the Visual RPG compiler to ignore the text following it, or the *comment*. Remarks are brief explanatory notes added to code for the benefit of those reading it.

The comment to occupy an entire line starts with `//`. You do not need to add anything at the end of the comment.

The comment symbol to occupy multiple lines, or to include a comment on the same line as code starts with a `/*` and ends with a `*/`. **This type of comment can go anywhere white space (tab, new-line, and carriage return) is permitted and is treated as such.**

It is good practice to begin all subroutines and functions with a brief comment describing the functional characteristics of the procedure (what it does), for your own benefit and the benefit of anyone else who examines the code.

Remarks can follow a statement on the same line, or occupy an entire line. Both are illustrated in the following code:

Remarks that occupy an entire line (using `//`):

```
// This comment occupies an entire line.
System.Windows.Forms.Application.DoEvents()
- or -
System.Windows.Forms.Application.DoEvents() // This comment is on the same
line.
```

Remarks using `/*` to Start and `*/` to End Comment:

```
/* This comment occupies an entire line. You must end with a */.
System.Windows.Forms.Application.DoEvents()
- or -
System.Windows.Forms.Application.DoEvents() /* This comment is on the same
line. */
```

If your Remarks expands more than one line, simply use the `//` symbol at the beginning or each line, or add the `*/` symbol at the end of the last line of Remarks. `// This text is too long to fit on one line, so we'll add`

```
// another line.
- or -
/* This text is too long to fit on one line, so we'll add another line */.
```

Note the following points:

Every important variable declaration should include a **line** comment describing the use of the variable being declared.

Variables, controls, and procedures should be named clearly enough that **inline** commenting is needed only for complex implementation details.

Remarks cannot follow a **line-continuation** sequence (+) on the same line.

You can add or remove comment symbols for a block of code by selecting two or more lines of code and choosing the **Comment**  and **Uncomment**  buttons on the **Edit** toolbar.

See Also

[Continuation Lines | Language Elements](#)

Continuation Lines

Since the new line character is used in the Visual RPG Language as a command separator, a continuation character "+" is provided to span a command across line boundaries. The plus character must be the last non white-space character on the line and cannot be embedded in a string or comment. Attention should be given when continuing an expression.

Example

```
SCAN      CmpStr (Name) CmpLen (3) BaseStr (MonthTable) +
          StartPos (1) FndPos (MonthIndex) +
          Fnd (*IN01)
          Month = MonthIndex -1 / 3 +
          + 12
```

See Also

[Language Elements | Remarks](#)

Control Language

Database control operation codes are incorporated into the Visual RPG language syntax. The developer can use the strength of the Visual RPG language to control database operations anywhere in the program structure. The following are the **Control Language** operation codes and their functions.

Op code	Description
OSEXEC	Launch execution of another Windows .EXE.

See Also

[Language Elements | OSEXEC](#)

Declarations

You can place variable, constant, type and DLL subroutine **declarations** at the program level for form or code programs.

Declarations can also be local to a subroutine. These local **declarations** are not available to any other part of the application, except the subroutine in which they are declared.

See Also

[Language Elements | Event Subroutines | General Subroutines | Mainline Code](#)

General Subroutines

General subroutines are created because several different event subroutines may need the same actions performed. A good programming strategy is to put common statements in a separate subroutine (a **general subroutine**) and have event subroutines call it. This eliminates the need to duplicate code and also makes the application easier to maintain.

Once you have defined a **general subroutine** in a form program, you can call it from any of the event subroutines in that form.

See Also

[Language Elements](#) | [Declarations](#) | [Event Subroutines](#) | [Mainline Code](#) | [Figurative Constants Rules](#)

Multiple Occurrence Data Structures

Multiple Occurrence Data Structures by default behave the same as in RPG/400. The current occurrence initially is set to the first and can be changed using the OCCUR operation code or %OCCUR function. They can be declared using the DCLDS command.

Visual RPG also allows the [] characters to access occurrences. Use of the [] does not alter the current occurrence.

Example

```
MOVE      Total  MyDS[5].Total
MOVE      Prod   MyDS[5].ProdArray[3]
MOVE      Amount MyMultDCLFLDDS[5,3,2].Amount
```

See Also

[Language Elements](#) | [Data Structure Element](#) | [DCLDS](#) | [DCLDSFLD](#) | [DCOVERLAYGROUP](#) | [OCCUR](#) | [%OCCUR](#) | [Array Elements](#) | [Control Properties](#) | [Data Structure Element](#) | [Fields](#)

Names

When you are writing your Visual RPG code, you name many elements, subroutines, variables, forms, controls, labels and so on. The **names** that you declare in your code must follow the following set of rules:

Names must start with an alphabetic character (A-Z, a-z) or the character \$, @ or #.

Subsequent characters in a name can be alphabetic (A-Z, a-z), numeric (0-9) or the character \$, @, #, _ (underscore), ñ, €, £, and ¥.

There is no limit to the length of the name.

Example:

```
CUSNAM
CustomerName
Total$
#ofInvoices
```

The Visual RPG Language is case insensitive, that is, any lower case alphabetic characters are automatically turned to upper case by the compiler, unless it is seen inside a character literal.

Since Visual RPG has no restrictions on the length of a name, more descriptive, readable names can be chosen.

Example:

PRTNUM vs. PartNumber

Taking advantage of this can greatly increase the readability and thus the maintainability of your programs.

See Also

Language Elements | Reserved Names | Special Values

Printing

Local and network printing is accomplished in Visual RPG by writing to a print file contained in the Acceler8DB database. Because a user may select from a number of printer devices with different capabilities, some properties require special considerations.

Note Not all of the possible settings for the following properties may be valid as each printer device can have a limited set of capabilities. It is up to the programmer to select the setting appropriate to the actual printer device used.

See the individual properties for more information on each of these property.

Property	Description
Collate	Indicates whether collation will be used when printing multiple copies.
Duplex	Indicates whether the printer file is to print duplex (prints on both sides of the paper).
FormName	The name of the customized page size to use.
Orientation	The printer's paper orientation (Horizontal or Vertical).
PaperLength	The physical length of the page.
PaperSize	The paper (size) to be used.
PaperSource	The printer tray that will be used.
PaperWidth	The physical width of the page.
PrintableLength	The length of the printable area on the paper based on the page orientation selected.
PrintableWidth	The width of the printable area on the page based on the page orientation selected.
Quality	The print quality.
Scale	The percentage factor by which the printed document is to be scaled.
TTOption	Indicates how true type fonts are to be printed.
UseColor	Indicates whether to use color when a color printer is specified.

YResolution Indicates the Y-axis resolution (in dot per inch) when printing graphics.

See Also

Language Elements | PrintFile Class | PrintFile Properties

Reserved Names

The compiler has reserved certain names for its exclusive use. This means that variables cannot be declared or used with these names, because a syntax error will occur. These names include the following:

Command Names

Special Values

Example

```
*CHAR
*EQ
*INPUT
*OFF
```

See Also

Language Elements | Command Names | Special Values | Lexical Element Names

Simple Types

The following are the **simple types** allowed in the Visual RPG Language.

Character Types:

***CHAR** Character string. The maximum character string is 65,523.

Numeric Types:

***BINARY** Decimal binary number (2's complement). The maximum size for a 2-byte binary is four digits, and the maximum size for a 4-digit binary is 9 digits.

***PACKED** Decimal packed number (X'7' as the rightmost nibble is negative, X'3' as the rightmost nibble is positive). The maximum number of digits can be 31 and the maximum number of decimal positions is 9.

***ZONED** Decimal zoned number (X'3' as the rightmost zone is positive, X'7' as the rightmost zone is negative). The maximum number of digits can be 31 and the maximum number of decimal positions is 9.

Special Types:

***FORM** Form program handle.

***IND** Indicator (can only have the special values *ON and *OFF).

See Also

Language Elements | Visual RPG Language Reference | Syntactical Elements

Subroutine Parameters

Visual RPG allows parameters to be passed to subroutines (BEGSR). The EXSR op code may reference a DCLPLIST or be followed by a DCLPARM list. The **BEGSR** may be followed by a *ENTRY and **DCLPARM** list.

Subroutine parameters not only provide for powerful programming, but also facilitate the functionality of some events.

See Also

Language Elements | DCLPARM | DCLPLIST | BEGSR | EXSR

Type Conversions

Indicator Conversions:

A single character '1' moved to an indicator variable sets the indicator to the *ON value, and a single character '0' moved to an indicator variable sets the indicator to the *OFF value. Likewise, a move from an indicator variable to a character variable sets the first byte of the character variable to a '1' or '0'.

Numeric Conversions:

Numeric operands are promoted to a zoned number before being used on any operation. If all the operands involved are binary with the same specification, the promotion may be virtual.

Implicit Conversions:

Implicit Conversions are done when two operands in an expression are not of the same type. These conversions can only be done implicitly if the operands are both character or indicator, or are both numerics. Character or indicator operands mixed with numeric operands will result in a compiler error.

See Also

Language Elements | Visual RPG Language Reference | Syntactical Elements

Literals

Boolean Literal

The *Boolean Data Type contains **True** and **False** literals that map to the **True** and **False** state, respectively.

```
BooleanLiteral ::= True | False
```

See Also

Boolean Data Type | Character Literal | Literals

Character Literal

A **Character literal** is defined by a single or double quote, followed by any combination of characters, and terminated by a single/double quote.

Remarks

If a quote is required as part of a character literal it is represented by two quotes. For Example, the literal o'clock is coded as 'o'clock', and "Hello" is coded as ""Hello"".

Example

```
Character Literal <ABC>
```

See Also

Character Data Type | Character Expression | Literals

Date Literal

Date literals take the form D'YYYY-MM-DD' where:

D indicates that the literal is of type date.

D'YYYY-MM-DD' is a valid date in the *ISO date format.

YYYY implies a 4 digit year.

MM implies a 2 digit month.

DD Implies a 2 digit day.

See Date Formats for more information.

Remarks

D'YYYY-MM-DD' is enclosed in single or double quotes.

Example

```
D'1887-01-31'
```

See Also

Character Literal | Date Data Type | Date Formats | Hexadecimal Literal | Time Literal | TimeStamp Literal | Literals

Hexadecimal Literal

Hexadecimal literals may be used to define unicode string (U) and numeric (H) values. U'x1x2...xn' and H'x1x2...xn' can only contain the characters A-F, a-f, and 0-9.

Unicode literals take the form U'Xxxx...Yyyy' where 4 bytes are given to represent a single unicode character.

Example

```
Mystring = "Hello" + U'0009' + "World"    // U'0009' is TAB character
Number2 = Number1 + H'64'                 // add 100 to Number1
DclConst myTab value( U'0009')            // Tab constant
DclConst myNumber value( H'7FFF')         // 32767
```

See Also

Character Literal | Date Literal | Time Literal | TimeStamp Literal | Literals

Numeric Literal

Numeric literals are constant numeric values that take the form `[+/-]nnn[.nnn]` where the `+/-` is optional, the `nnn`'s are numerics (0-9) and the `[.nnn]` decimal portion is also optional.

Remarks

The `+` or `-` sign must be the leftmost character, if present. An unsigned literal is treated as a positive number.

Blanks cannot appear in a numeric literal.

The `.` character is used as a decimal point (e.g. `-12.53`).

Example

```
1.2, -45.6, +89.2
```

See Also

[Hexadecimal Literal](#) | [Date Literal](#) | [Time Literal](#) | [TimeStamp Literal](#) | [Literals](#)

OneChar Literal

A **OneChar literal** allows the specification to write `*OneChar` Data Type constants.

Remarks

The format is `O'<char>'` where `<char>` is any single character. The difference between `O'<char>'` and simply `'<char>'` is that the former is of type `*OneChar`, while the latter is of type `*Char` (or `*String`).

Example

```
O'w'  
O'4'
```

See Also

[*Char Data Type](#) | [*OneChar Data Type](#) | [Literals](#)

Time Literal

Time literals take the form `T'HH.MM.SS'` where:

T indicates that the literal is of type time.

'HH.MM.SS' is a valid time in the ISO format.

HH - Implies a 2 digit hour (0-23).

MM - Implies a 2 digit minute.

SS - Implies a 2 digit second (0-59).

Remarks

Single or double quotes are valid.

Example

```
T'05:10:00'
```


See Also

Numeric Literal | Hexadecimal Literal | Date Literal | Character Literal | TimeStamp Literal | Time Data Type | Literals

Timestamp Literal

Timestamp literals take the form `Z'yyyy-mm-dd-hh.mm.ss.mmmmmm'` where:

`Z` indicates that the literal is of type `Timestamp`.

`yyyy-mm-dd` is a valid date (year-month-day).

`hh.mm.ss.mmmmmm` is a valid time (hours.minutes.seconds.microseconds).

Remarks

`yyyy-mm-dd-hh.mm.ss.mmmmmm` is enclosed by apostrophes.

Microseconds are optional and if not specified will default to zeros.

Example

```
z'1997-07-22-10.15.05.000030'
```

See Also

Numeric Literal | Hexadecimal Literal | Date Literal | Time Literal | Character Literal | Timestamp Data Type | Literals

Unicode Literal

A **Unicode literal** allows strings to be specified as a chain of unicode characters written in hexadecimal notation (same as RPG ILE U literals).

Remarks

Each unicode character is represented with 4 hex digits, as in the format `U'xxxx...'`, where `xxxx` is the hex representation of a unicode character.

The type of a unicode constant is ***String**.

Example

```
U'00310032000d'  
u'30c3'
```

See Also

Hexadecimal Literal | String Data Type | Literals | Unicode Overview

Methods

CreateName Method

CreateName takes all the current values of the database name, except for the name itself, and creates a new 'Database Name' in the registry with the name given in **NewName**.

```
database.CreateName (NewName)
```

Parameters

NewName

Required. A string containing a new database name to be created in the registry.

Remarks

CreateName creates a new 'Database Name' in the registry with the name given in **NewName**. The content of the database name registry entries are defined by the values of the Label, PlatformAttribute, PoolingTimeOut, Password, Port, Server, Text, and User properties.

You can use the UpdateName method to write the current values of the database name object as the new values of the database name in the registry.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Properties | Database Class Methods | UpdateName Method | DCLDB](#)

DeleteName Method

DeleteName deletes an existing database name from the registry.

```
database.DeleteName (ExistingName)
```

Parameters

ExistingName Required. A string containing the name of an existing database name to delete from the registry.

Remarks

This method is **Shared**, and does not affect the current values of the database object.

Use the GetNames method to get an array of strings containing all of the database names found in the registry.

See Also

[ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB | GetNames Method](#)

GetNames Method

GetNames returns an array of strings with all of the names found in the registry.

```
variable[] = database.GetNames (publicDBs)
```

Parameters

publicDBs

Required. A boolean parameter indicating whether to get Public Databases (*True) or those private to the current user (*False).

Returns

A string array with all of the names found in the registry.

Remarks

This method is **Shared**, and does not affect the current values of the database object.

Private Database names are stored in the registry at
HKEY_CURRENT_USER\Software\ASNA\Acceler8-DB\CurrentVersion\Databases.

Public Database names are stored in the registry at
HKEY_LOCAL_MACHINE\SOFTWARE\ASNA\Acceler8-DB\CurrentVersion\Data Sources*PUBLIC.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB

UpdateName Method

UpdateName takes the current values of the database name object and writes them out as new values of the database name in the registry.

```
database.UpdateName ()
```

Remarks

UpdateName takes the current values of the database name object and writes them out as new values of the database name in the registry.

You can use the CreateName method to take all the current values of the database name, except for the name itself, and create a new 'Database Name' in the registry with the name given in **NewName**.

UpdateName modifies the 'Database Name' in the registry corresponding the name given in the *Dbname* property. The updated content of the database name registry entries are defined by the values of the Label, PlatformAttribute, PoolingTimeOut, Password, Port, Server, Text, and User properties.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Members | Database Class Methods | Database Class Properties | DCLDB | CreateName Method

Operation Codes

ADD

Adds **F1** to **F2** and places the sum in **Result**.

```

ADD
F1 (Numeric Expression | Character Expression | *NONE)
F2 (Numeric Expression | Character Expression)
Result (Numeric Variable | Character Variable)
Adj (*TRUNC | *RNDUP)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT | FLOAT4
      | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])

```

Parts

F1

Optional. If F1 is not specified, F2 is added to Result and the sum is placed in Result.

If F1 and F2 are character type (*CHAR), they will be concatenated, and the result placed in the Result. In this case, the adjust (ADJ) parameter will be ignored.

F1 and F2 may not be the Value property of an IOField control. Implicitly numeric control properties are allowed.

F2

Required. If F1 and F2 are character type (*CHAR), they will be concatenated, and the result placed in the Result. In this case, the adjust (ADJ) parameter will be ignored.

F1 and F2 may not be the Value property of an IOField control. Implicitly numeric control properties are allowed.

Result

Optional. Contains a numeric or character variable to contain the result of the addition.

Adj

Optional. Db specifies the Database Name to use when opening the file at run-time. The actual file/member that is accessed at run-time is determined by the Mbr and File parameters. If the DB parameter is not given, the same database that was accessed to get the file description will be accessed to open the file.

Pos

Optional. Pos will be turned on if the **Result** is a positive number.

Neg

Optional. Neg will be turned on if the **Result** is a negative number.

Zero

Optional. Zero will be turned on if the **Result** is zero.

Type

Optional. **Type** can be any of the types listed. *NONE is the default.

Len

Optional. **Len** defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Example

```
// The value 1 is added to RecordNbr.
RecordNbr = RecordNbr + 1
// The contents of HoursWrk are added to CurrHours.
CurrHours = HoursWrk + CurrHours
// The contents of OTimePay and RegPay are added together and placed in
TotalPay.
TotalPay = OTimePay + RegPay
```

See Also

+ Operator | CONCAT | Expressions

ADDUR - ADD a Duration

ADDUR adds the duration specified in **F2** to a date or time, and places the resulting date, time or timestamp in **Result**.

```
ADDUR
F1 (*NONE | Date | Time | Timestamp field | Subfield | Array | Array element
   | Literal | Constant)
F2 (Numeric field | Array element | Constant : Date | Time | Timestamp |
   Array | Array element)
F2Fmt (*DAYS, *MONTHS, *YEARS, *MSECONDS, *SECONDS, *MINUTES, *HOURS)
Result (Date | Time | Timestamp field | Subfield | Array | Array element)
Err (*EXTENDED | Indicator Variable)
```

Parts**F1**

Optional. May contain a date, time or timestamp field, subfield, array, array element, literal or constant. If **F1** is not specified, the duration is added to the field specified in the **Result**. If **F1** contains a field name, array or array element, then its data type must be the same data type as the field specified in **Result**.

F2

Required. May be a numeric field, array element or constant with zero decimal positions. If the duration is negative, then it is subtracted from the date.

F2Fmt

Required. **F2Fmt** must be a valid duration code indicating the type of duration. The duration code must be consistent with the **Result** data type. You can add a year, month or day duration to a date field, but you cannot add a minute).

Result

Required. **Result** must be a date, time or timestamp data type field, array or array element.

If **Result** is an array, the value in **F2** is added to each element of the array.

If **F1** is blank, the duration is added to the value in **Result**.

If **Result** is a time field, the result will always be a valid time. For Example, adding 10 minutes to 23:59:59 would give 24:09:59. Since this time is not valid, the run-time adjusts it to 00:09:59.

When adding a duration to a date in months, only the month portion is increased by the number of months in the duration. The day portion remains unchanged. The exception to this is when the resulting day portion would exceed the actual number of days in the resulting month. In this case, the resulting day portion is adjusted to the actual month-end date. See Adding or Subtracting Dates for more information.

Err

Optional. **Err** determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred. The **Err** indicator will be turned on when one of the following occurs:

The result of the operation is invalid.

The value of the date, time or timestamp field in **F1** is invalid.

F1 is blank and the value of the **Result** before the operation is invalid.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function. If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Visual RPG places an 18-digit limit on durations. Adding a duration with more than 15 significant digits will cause errors or truncation. These problems can be avoided by limiting the first factor in Factor 2 to 18 digits, or less.

Example

```
'97/07/21' ADDDUR2:*MONTH results in '97/09/21'  
'97/07/21' ADDUR1:*YEAR results in '98/07/21'  
'97/07/21' ADDUR1:*YEAR results in '98/07/21'  
'97/07/21' ADDDUR5:*DAY results in '97/07/26'
```

See Also

Adding or Subtracting Dates | Duration Codes | EXTRACT | SUBDUR

ADDHANDLER

Attaches an event handler to an object's event.

```
ADDHANDLER
SourceObject (Object)
SourceEvent (Name)
HandlerObject (Object)
HandlerSr (Name)
```

Parts

SourceObject

Required. The name of the object firing the event that you want to handle.

SourceEvent

Required. The name of the event you want to handle.

HandlerObject

Optional. Object containing the event handler subroutine.

HandlerSr

Required. Name of the subroutine that will be invoked to handle the event.

Remarks

ADDHANDLER allows the user control of event handlers. An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic. The object that raises (triggers) the event is called the event sender or source object. The object that captures the event and responds to it is called the event receiver or handler object.

When specifying event handlers for windows and web forms controls, it is usually more convenient to use the **Event** keyword on the **BegSr** command. **ADDHANDLER** is useful when the object firing the event is not defined as a global field of the class handling the event.

Use **REMOVEHANDLER** to remove the event handler from the **SourceEvent**.

Example

```
BegClass Form1 Extends(System.Windows.Forms.Form) Access(*Public)
DclDb Name( database ) DBName( "ASNA Local DB" )
DclMemoryFile Name( memCustomer ) DBDesc( database )      +
    FileDesc( "/ASNA_Example_Files/CMasterL1" )
    BegConstructor Access(*Public)
        AddHandler memCustomer.dataset.Tables[0] RowChanged +
            *this Row_Changing
    EndConstructor

    BegSr Row_Changing Access(*Private)
        DclSrParm sender Type(*Object)
        DclSrParm e Type(System.Data.DataRowChangeEventArgs)
        msgBox Title( "Row is Changing" )
        Msg( e.Row.ToString() )
    EndSr
EndClass
```

See Also

REMOVEHANDLER

BEEP

Sounds a tone through the computer's speaker.

```
BEEP
SoundType ( Special Value)
Err ( Indicator Variable | *Extended)
```

Parts

SoundType

Optional. Specifies the type of sound to be used by indicating a **special value** that is identified by an entry in the [SOUNDS] section of the registry.

*Std	Beep using the computer speaker.
*Asterisk	System asterisk.
*Exclamation	System exclamation.
*Hand	System hand.
*Question	System question.
*SysDft	System default.

Err

Optional. Will be set if any error condition is found with the file. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the Err keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

The ERR parameter determines what the runtime will do if an error occurs while executing this command. The following rules apply:

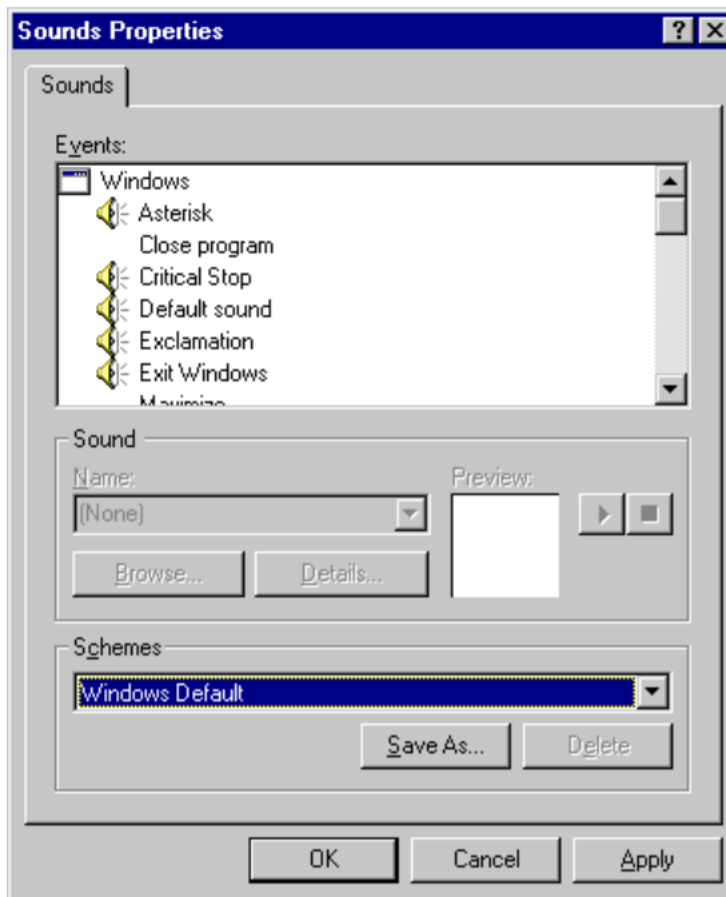
Example

The frequency and duration of the beep depend on your hardware and system software and may vary among computers.

After queuing the sound, the **BEEP** op code returns control to the calling function and plays the sound asynchronously.

If the system cannot play the specified sound, **BEEP** will attempt to play the system default sound. If the system default sound does not play, the function produces a standard beep sound.

Whether or not your system produces a sound, regardless of the type, depends on the setting in the **Settings - Control Panel - Sounds** application. If it is set to **No Sounds** scheme, then no sound will be produced, as shown below. To change, click on the arrow to the right and set to Windows Default.



See Also

Custom Attributes | ENDCCLASS | Understanding Classes

BEGCLASS

Declares the name of a class, as well as a definition of the variables, properties, events and methods that comprise the class.

```
BEGCLASS
Name (Class name)
Partial (*NO | *YES)
Access(*PUBLIC | *INTERNAL)
Modifier (*EXTENSIBLE| *NOTEXTENSIBLE | *MUSTEXTEND)
Extends (Class name)
Implements (Interface name list)
Attributes (attribute 1, attribute 2, ...)
```

Parts

Name

Required. Contains the name of the class.

Partial

Optional. Indicates if the class is partial, that is, declared in Parts, possibly in different source files. All the Parts of a partial class must use Partial(*YES). *NO is the default.

Access

Optional. Allows the contents of the class to be access either only internally by other class in the same assembly (exe or dll), or to the 'outside world'. *Public is the default.

*PUBLIC – class is accessible to the 'outside' world, that is, it can be referenced by any other assembly.

*INTERNAL - class is accessible only by other classes in the current assembly.

Modifier

Optional. Modifier controls how properties and methods are overridden in derived classes. *EXTENSIBLE is the default.

*EXTENSIBLE (default) - specifies that other classes can be defined to extend this class. The new class is called a derived class.

*MUSTEXTENDED - specifies that non-shared members of this class cannot be used on its own. The class only provides a set of fields, properties, events and methods as the basis to derive new classes. Instances of *MustExtende classes cannot be created.

*NOTEXTENSIBLE - specifies that this class cannot be extended in a new derived class. No further inheritance is allowed from this class.

Extends

Optional. Identifies the name of the base class to extend, such as System.Windows.Forms.Form. The new class inherits all the members of the base class.

Implements

Optional. Indicates that this class implements the members of an interface. An interface defines a set of properties, events and methods, but does not provide an implementation of them, just a definition. You must implement every member of every interface you specify in this keyword.

Attributes

Optional. Allows you to provide one or more attributes that apply to this class. To specify more than one attribute, simply separate them with commas.

Remarks

An ENDCLASS statement must be the last statement in the class.

For more information on classes, see Visual Studio's topics on Understanding Classes.

For more information on attributes, see Custom Attributes.

Example

The following is an Example of **BEGCLASS** showing attribute usage:

```
Using System.ComponentModel
Using System.Reflection
BEGCLASS Boxer Extends(System.Windows.Forms.TextBox) +
Attributes(DefaultProperty("myValue") +
Description("With two custom attributes"))
    DclFld AnInt Type( *Integer) Len(4) Attributes(DefaultValue(85))
    DclFld AStr Type( *String) Attributes(DefaultValue("Jupiter"))
    DclFld XColor Type(*Integer) Len(4) Access(*Public) +
Attributes (Description("Radix showing color"), +
Category("Appearance"))
ENDCLASS
```

See Also

Custom Attributes | ENDCLASS | Understanding Classes

BEGCONSTRUCTOR

Identifies, or declares a constructor.

```
BEGCONSTRUCTOR
Shared (*NO | *YES)
Access (*PRIVATE | *INTERNAL | *PROTECTED | *PUBLIC)
Base (*DFT | Expression 1, Expression 2, ...)
This (Argument1, Argument2, ...)
Attributes (Attribute 1, Attribute 2, ...)
```

Parts

Shared

Optional. Declares whether the constructor is a class constructor or an instance constructor. *NO is the default.

*NO (default) - declares an instance constructor, used to initialize an instance (that is, an object) of a class.

*YES – declares a class constructor, not associated with an object but with the class itself.

Access

Optional. Allows the constructor to be accessible to the 'outside world'. *PRIVATE is the default.

*PRIVATE (default) - constructor is declared and accessible only in the current class.

*INTERNAL – constructor is accessible by the whole assembly (all the classes in the same dll or exe).

*PUBLIC – constructor is accessible by everybody.

*PROTECTED - constructor is accessible only by the current class and classes that extend the current class.

Protected members are not as private as private members, which are accessible only to members of the class in which they are declared, but they are not as public as public members, which are accessible in any function.

Base

Optional. Only valid in instance constructors. Calling the current constructor will also invoke the constructor in the parent class (the class the current class extends) that matches the given arguments.

If you specify the Base parameter, you cannot specify the This parameter.

If you do not specify Base or This, the default (*Dft) is to call the Base constructor (the one with no arguments)

Note The **Base** constructor must exist or you will get a compiler error.

This

Optional. Allows a constructor to call another constructor in this same class. It contains the arguments to the other constructor. If you specify this parameter, you cannot specify the **Base** parameter.

If you specify the **This** parameter, you cannot specify the **Base** parameter.

If you do not specify **Base** or **This**, the default (*Dft) is to call the **Base** constructor (the one with no arguments)

Note The **Base** constructor must exist or you will get a compiler error.

Attributes

Optional. Allows you to list one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

Attributes use required, and optional arguments. The required parameters are provided in positional fashion. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For Example, the following code specifies the Developer attribute:

```
BegConstructor MyClass Attributes(Developer("John Smith", 35))
```

Note You must use positional arguments for arguments that are used with the attribute's constructor. Named arguments can only be used to set the values of fields and properties of attribute classes.

Remarks

Constructors are special methods that allow control over initialization. They are run after the program begins(class constructors) or when an instance of a type is created (instance constructors). Instance constructors specify what happens when an object is created. Unlike other members, constructors are not inherited and do not introduce a name into a type's declaration space. Constructors may only be invoked by object-creation expressions or by the .NET framework; they may never be directly invoked.

Instance constructors must always invoke (implicitly or explicitly) a constructor in the parent class. If an instance constructor declaration does not provide neither **Base** nor **This**, then the call is implicit to the parameter-less constructor in the parent class.

A **BEGCONSTRUCTOR** routine must end with the **ENDCONSTRUCTOR** command.

See Also

Custom Attributes | **ENDCONSTRUCTOR** | Using Constructors

BEGENUM

Declares, or starts an enumeration definition.

```
BEGENUM
Name (Enumeration name)
Access (*PUBLIC | *INTERNAL)
BaseType (Integral type name)
Attributes (Attribute 1, Attribute 2, ...)
```

Parts**Name**

Required. The name of the enumeration definition.

Access

Optional. Allows the enumeration definition to be accessible to the 'outside world'. *PUBLIC is the default.

*INTERNAL – enumeration definition is accessible by the whole assembly (all the classes in the same dll or exe).

*PUBLIC (default) – enumeration definition is accessible by everybody.

BaseType

Optional. If not given, it defaults to *Integer4. If given, it must be one of the integral types: *Integer2, *Byte, etc.

Attributes

Optional. Allows you to list one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

Attributes use required, and optional arguments. The required parameters are provided in positional fashion. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For Example, the following code specifies the Developer attribute:

```
BEGENUM MyClass Attributes(Developer("John Smith", 35))
```

Note You must use positional arguments for arguments that are used with the attribute's interface. Named arguments can only be used to set the values of fields and properties of attribute classes.

Remarks

Enums are used to describe enumeration constants. Their only members are DclEnumFld fields.

The **BEGENUM** routine must end with the **ENDENUM** command.

Example

```
BegEnum myEnum basetype( *byte )
  DclEnumFld q4 value(4) // <--- the default value is 0
  DclEnumFld q5          // <--- this will be 5, which is 1 plus the
previous one
EndEnum
```

See Also

ENDENUM | DCLENUMFLD | Enumeration Definition

BEGFUNC - Begin a Function

Creates or identifies a function or method for an ActiveX control or ActiveX DLL.

```

BEGFUNC
Name (Function name)
Type (*BOOLEAN | *BINARY | *BYTE | *CHAR | *DATE | *FORM | *DECIMAL | *FLOAT
      | *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 |
      *INTEGER8 | *OBJECT | *PACKED | *STRING | *TIME | *TIMESTAMP |
      *ZONED | name | String Literal | Library.Object)
Len (Length Integer, [Decimal Positions Integer])
TimFmt (*PGMDFT | *MDY | *DMY | *YMD | *JUL | *ISO | *USA | *EUR | *JIS |
      *HMS)
Rank (IntegerLiteral)
Shared (*NO | *YES)
Access (*PRIVATE | *INTERNAL | *PUBLIC | *PROTECTED)
Modifier (*NONE | *MUSTOVERRIDE | *NOTOVERRIDABLE | *OVERRIDABLE |
      *OVERRIDES)
Implements (Interface name list)
Attributes (attribute 1, attribute 2, ...)
Like (Function name)
Event (*NONE | *CHANGE | event name)

```

Parts

Name

Required. The name of the function.

Type

Required. Type can be any of the **** Types** (*CHAR, *BINARY, etc.) or it can be the name of a field already defined in the program, a string literal or a library object, as described below. See **Type Parameter** for further information.

Name is the name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

- a valid program id registered on your system at compile time.
- a valid class id registered on your system at compile time.
- a valid interface name registered on your system at compile time.

A name of the form Library.Object, where Library is the name of one of the libraries selected in the new up and coming object libraries dialog, and Object is valid object name within that library.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer, and *ZONED is assumed if there are two integers.

Len

Optional. **Len** defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *BINARY, *CHAR, *PACKED, *ZONED, *DECIMAL | *FLOAT, *INTEGER will require a **Len**. A compiler message will display if the **Len** parameter is needed.

TimFmt

Optional. **TimFmt** specifies the format for time. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Rank

Optional. Rank declares an n-dimensional array of indefinite elements. The value given for Rank defines the number of dimensions for the array.

Shared

Optional. Indicates that one or more declared programming elements are shared. **Shared** elements are not associated with a specific instance of a class or structure. You can access them by qualifying them either with the class or structure name, or with the variable name of a specific instance of the class or structure. *NO is the default, indicating that the file will not be shared.

*NO (default) - this file will not be shared.

*YES - this file will be shared.

Access

Optional. The type of access to the file. The default is *Private.

*PRIVATE (default) - access to the file is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the file is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the file is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the file is accessible from within the same project, but not from outside the project.

Modifier

Optional. Modifier controls how properties and methods are overridden in derived classes. *NONE is the default.

*NONE (default) - specifies that none of the override options are available for properties and methods.

*MUSTOVERRIDE - specifies that a property or procedure in a base class must be overridden in a derived class before it can be used.

*NOTOVERRIDABLE - specifies that a property or procedure cannot be overridden in a derived class.

*OVERRIDABLE - specifies that a property or method can be overridden in a derived class.

*OVERRIDES - specifies that a property or method overrides a member inherited from a base class.

Implements

Optional. Indicates that this class implements the members of an interface. An interface defines a set of properties, events and methods, but does not provide an implementation of them, just a definition. You must implement every member of every interface you specify in this keyword.

Attributes

Optional. Allows you to provide one or more attributes that apply to this class. To specify more than one attribute, simply separate them with commas.

Like

Optional. Like must be the name of a function already defined in the class. **Type** cannot be a named constant or a literal. In this case, the function will be defined based on the attributes of another function. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the function being defined. The function length entry is allowed only for field definitions requiring a length.

Event

Optional. The event names that are valid in **EVENT** depend upon the control being used in **Name**. In this case, *CHAR is assumed if there is one integer, and *ZONED is assumed if there are two integers.

Remarks

Use BEGFUNC when creating a method for an ActiveX control or ActiveX DLL.

A **BEGFUNC** routine must always contain a LEAVESR and end with an ENDFUNC, as shown below.

Example

```
BEGFUNC MyFunc Type(*IND) Access (*PUBLIC)
  Chain CustMaster CustNumber NotFnd(*IN60)
  LEAVESR *IN60
ENDFUNC
```

Visual RPG allows you to name a control property for the **LeaveSR** 'Value' parameter. However, you must not assume the default property for the control. Instead, you must specifically code the desired property.

See Also

BEGPROP | BEGSR | Custom Attributes | DCLSRPARM | ENDFUNC | LEAVESR

BEGGET - Begin a GET Procedure

Declares a **GET** property procedure used to assign a value to a property.

```
BEGGET
```

Remarks

BEGGET allows you to get the value of the property with the BEGPROP routine. To set the value of a property, use BEGSET. A **BEGGET** routine must always end with an ENDGET, and **BEGSET** must always end with an ENDSET.

Example

```
dclfld_Firstname    *string      INZ('')
dclfld_Lastname     *string      INZ('')

      BegProp Firstname    *string      Access( *public )
      BegSet
          _FirstName = *Propval
      EndSet
```



```

        BegGet
        LeaveSR _FirstName
    EndGet
EndProp

    BegProp Lastname      *string          Access ( *public )
    BegSet NewValname( Lname )
        _LastName = Lname
    EndSet

        BegGet
        LeaveSR _LastName
    EndGet
EndProp

// Read only property

BegProp FullName Type( *string )          Access ( *public )
BegGet
    LeaveSR Value( FirstName + " " + LastName )
EndGet
EndProp

```

See Also

BEGPROP | BEGSET | ENDGET | ENDSET

BEGININTERFACE

Declares, or starts an interface definition.

```

BEGININTERFACE
Name (Interface name)
Access (*PUBLIC | *INTERNAL)
Implements (Interface name list)
Attributes (Attribute 1, Attribute 2, ...)

```

Parts

Name

Required. The name of the interface definition.

Access

Optional. Allows the interface definition to be accessible to the 'outside world'. *PUBLIC is the default.

*INTERNAL – interface definition is accessible by the whole assembly (all the classes in the same dll or exe).

*PUBLIC (default) – interface definition is accessible by everybody.

Implements

Optional. Indicates that this class implements the members of an interface. An interface defines a set of properties, events and methods, but does not provide an implementation of them, just a definition. You must implement every member of every interface you specify in this keyword.

Attributes

Optional. Allows you to list one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

Attributes use required, and optional arguments. The required parameters are provided in positional fashion. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For Example, the following code specifies the Developer attribute:

```
BEGININTERFACE MyClass Attributes(Developer("John Smith", 35))
```

Note You must use positional arguments for arguments that are used with the attribute's interface. Named arguments can only be used to set the values of fields and properties of attribute classes.

Remarks

Interfaces can only contain methods (functions and subroutines), properties, and TYPE-declared events. Fields and PARM-declared events are not allowed. In addition to this, the interface methods (including properties' get-and set-methods) must not have a body.

Interface members must not define their Access, Modifier and Shared keywords, as they are *Public, *MustOverride, and Shared (*No) by default.

In keeping with standard practice, we recommend interface names to begin with a capital letter "I", i.e., IDisposable, IEnumerable, etc. See "Design Guidelines for Class Library Developers" in the Documentation.

The **BEGININTERFACE** routine must end with the **ENDINTERFACE** command.

Example

```
//Define an Interface
BeginInterface myInterface
  BegSr Customer
    DclSrParm Name *integer4
  EndSr

  BegFunc Function Type( *integer4 )
  EndFunc
  DclEvent EVENT type( System.EventHandler )
EndInterface

// Interface Usage
BeginClass Overloaded Access (*Public) Implements( myInterface )
  BegSr First Access (*Public) implements ( myInterface.Customer )
    DclSrParm theParm *integer4
  EndSr
```

See Also

[ENDINTERFACE | Interface Definition](#)

BEGPROP - Begin a Property

Creates a property routine.

```
BEGPROP
Name (Function name)
Type (*BOOLEAN | *BINARY | *BYTE | *CHAR | *DATE | *IND | *PACKED | *TIME |
    *TIMESTAMP | *ZONED | *FORM | *DECIMAL | *FLOAT | *FLOAT4 | *FLOAT8 |
    *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *OBJECT | *STRING | name
    | String Literal | Library.Object)
Len (Length Integer, [Decimal Positions Integer])
TimFmt (*PGMDFT | *MDY | *DMY | *YMD | *JUL | *ISO | *USA | *EUR | *JIS | *HMS)
Rank (IntegerLiteral)
Shared (*NO | *YES)
Access (*PRIVATE | *INTERNAL | *PUBLIC | *PROTECTED)
Modifier (*NONE | *MUSTOVERRIDE | *NOTOVERRIDABLE | *OVERRIDABLE |
    OVERRIDES)
Implements (Interface name list)
Attributes (Attribute 1, Attribute 2, ...)
Like (Property name)
```

Parts

Name

Required. The name of the property.

Type

Required. **Type** can be any of the '*' Types (*CHAR, *BINARY, etc.) or it can be the name of a field already defined in the program, a string literal or a library object, as described below. If **Type** is *BOOLEAN, you can use the *TRUE or *ON and *FALSE or *OFF special values to indicate **True** or **False**. See Type Parameter for further information.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

A valid program id registered on your system at compile time.

A valid class id registered on your system at compile time.

A valid interface name registered on your system at compile time.

A name of the form Library.Object, where Library is the name of one of the libraries selected in the new up and coming object libraries dialog, and Object is valid object name within that library.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer, and *ZONED is assumed if there are two integers.

Len

Optional. **Len** defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *BINARY, *CHAR, *PACKED, *ZONED, *DECIMAL | *FLOAT, *INTEGER will require a **Len**. A compiler message will display if the **Len** parameter is needed.

TimFmt

Optional. **TimFmt** specifies the format for time. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Rank

Optional. Rank declares an n-dimensional array of indefinite elements. The value given for Rank defines the number of dimensions for the array.

Shared

Optional. Indicates that one or more declared programming elements are shared. **Shared** elements are not associated with a specific instance of a class or structure. You can access them by qualifying them either with the class or structure name, or with the variable name of a specific instance of the class or structure. *NO is the default, indicating that the property will not be shared.

*NO (default) - this property will not be shared.

*YES - this property will be shared.

Access

Optional. The type of access to the property. The default is *PRIVATE, indicating that the property will not be shared.

*PRIVATE (default) - access to the property is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the property is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the property is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the property is accessible from within the same project, but not from outside the project.

Modifier

Optional. Modifier controls how properties and methods are overridden in derived classes.*NONE is the default.

*NONE (default) - specifies that none of the override options are available for properties and methods.

*MUSTOVERRIDE - specifies that a property or procedure in a base class must be overridden in a derived class before it can be used.

*NOTOVERRIDABLE - specifies that a property or procedure cannot be overridden in a derived class.

*OVERRIDABLE - specifies that a property or method can be overridden in a derived class.

*OVERRIDES - specifies that a property or method overrides a member inherited from a base class.

Implements

Optional. Indicates that this property implements the members of an interface. An interface defines a set of properties, events and methods, but does not provide an implementation of them, just a definition. You must implement every member of every interface you specify in this keyword.

Attributes

Optional. Allows you to provide one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

Attributes use required, and optional arguments. The required parameters are provided in positional fashion. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For Example, the following code calls the **Developer** constructor of an attribute with two values:

```
BegClass MyClass Attributes(Developer("Joe Cool", 17))
```

You can also use named arguments, which are optional, to directly set the value of properties and fields of the attributes. You specify a named argument by appending ":" to the name of the argument, followed by the value being specified. Unlike positional arguments, named arguments can be specified in any order. For Example, the following code sets the value ***True** for the reviewed property of the Developer Attribute, and ***False** for the Keep field:

```
BegClass MyClass Attributes(Developer("Joe Cool", 17, Reviewed:=*True))
```

Note You must use positional arguments for arguments that are used with the attribute's constructor. Named arguments can only be used to set the values of fields and properties of attribute classes.

Like

Optional. Like must be the name of a property already defined in the class. **Type** cannot be a named constant or a literal. In this case, the property will be defined based on the attributes of another property. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the property being defined. The length entry is allowed only for property definitions requiring a length.

RemarksUse **BEGPROP** when creating a property for a class. **BEGSET** and **BEGGET** procedures allow you to set and get the value of the property.

A **BEGPROP** routine *must always* end with an **ENDPROP** and the **BEGGET** routine must also contain a **LEAVESR**, as shown below.

Example

```
dclfld_Firstname    *string          INZ('')

dclfld_Lastname     *string          INZ('')

      BegProp Firstname    *string          Access( *public )
      BegSet
          _FirstName = *Propval
      EndSet

          BegGet
          LeaveSR _FirstName
      EndGet
EndProp

      BegProp Lastname     *string          Access( *public )
      BegSet NewValname( Lname )
          _LastName = Lname
      EndSet

          BegGet
          LeaveSR _LastName
      EndGet
EndProp
```

```

        // Read only property

BegProp FullName Type( *string )      Access ( *public )
BegGet
    LeaveSR Value( FirstName + " " + LastName )
EndGet
EndProp

```

See Also

BEGSR | Custom Attributes | DCLSRPARM | ENDPROP | LEAVESR

BEGSET - Begin a SET Procedure

Declares a **SET** property procedure used to assign a value to a property.

```

BEGSET
NewValName (Name)

```

Parts

NewValName

Optional. A user-given name for the Property Value being passed. If **NewValName** is not given, the user can just use *PROPVAL. For instance, if you wish to know the value that is passed to you, you can define a name for *PROPVAL, and just reference the name.

Remarks

BEGSET allows you to Set the value of the property. To **Get** the value of a property, use BEGET. A **BEGSET** routine *must always* end with an ENDSET.

Example

```

dclfld_Firstname      *string      INZ('')
dclfld_Lastname       *string      INZ('')

        BegProp Firstname      *string      Access( *public )
        BegSet
            _FirstName = *Propval
        EndSet

                BegGet
            LeaveSR _FirstName
        EndGet
    EndProp

        BegProp Lastname      *string      Access( *public )
        BegSet NewValname( Lname )
            _LastName = Lname
        EndSet

                BegGet
            LeaveSR _LastName
        EndGet
    EndProp

        // Read only property

BegProp FullName Type( *string )      Access ( *public )
BegGet
    LeaveSR Value( FirstName + " " + LastName )

```

```

        EndGet
    EndProp

```

See Also

BEGGET | BEGPROP | ENDGET | ENDSET

BEGSR - Begin a Subroutine

Identifies the beginning of a subroutine. If it is an event-handling subroutine, it also identifies the event it handles.

```

BEGSR
Name (Function name)
Event (*NONE | *CHANGE | event name)
Shared (*NO | *YES)
Access (*PRIVATE | *INTERNAL | *PUBLIC | *PROTECTED)
Modifier (*NONE | *MUSTOVERRIDE | *NOTOVERRIDABLE | *OVERRIDABLE |
          *OVERRIDES)
Implements (Interface name list)
Attributes (attribute 1, attribute 2, ...)

```

Parts

Name

Required. The name of the subroutine. This must be used on an EXSR operation to call the subroutine. Event handling subroutines are declared by specifying the name of an **EVENT**. In that case, **Name** must be the name of a form, menu option, or control declared earlier in the program. Every subroutine in a given program must contain a unique symbolic name. Only one subroutine can be defined by this name.

Event

Optional. The event names that are valid in **EVENT** depend upon the control being used in **Name**. In this case, *CHAR is assumed if there is one integer, and *ZONED is assumed if there are two integers.

Len

Optional. **Len** defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *BINARY, *CHAR, *PACKED, *ZONED, *DECIMAL | *FLOAT, *INTEGER will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Shared

Optional. Indicates if one or more declared programming elements are shared. **Shared** elements are not associated with a specific instance of a class or structure. You can access them by qualifying them either with the class or structure name, or with the variable name of a specific instance of the class or structure. *NO is the default, indicating that the property will not be shared.

*NO (default) - this property will not be shared.

*YES - this property will be shared.

Access

Optional. The type of access to the property. The default is *Private.

*PRIVATE (default) - access to the property is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the property is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the property is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the property is accessible from within the same project, but not from outside the project.

Modifier

Optional. The type of access to the property. The default is *PRIVATE.

*MUSTOVERRIDE - specifies that a property or procedure in a base class must be overridden in a derived class before it can be used.

*NOTOVERRIDABLE - specifies that a property or procedure cannot be overridden in a derived class.

*OVERRIDABLE - specifies that a property or method can be overridden in a derived class.

*OVERRIDES - specifies that a property or method overrides a member inherited from a base class.

Implements

Optional. Indicates that this class implements the members of an interface. An interface defines a set of properties, events and methods, but does not provide an implementation of them, just a definition. You must implement every member of every interface you specify in this keyword.

Attributes

Optional. Allows you to provide one or more attributes that apply to this class. To specify more than one attribute, simply separate them with commas.

Note When your project startup object is BEGSR Main and you are printing to a DataGate OLE Print File (maintained with adpfn.exe 5.41 or earlier), the following Attribute must be included: System.STAThread().

Example:

```
BegSr Main Shared( *Yes ) Access( *Public ) Attributes( System.STAThread() )
```

Remarks

Parameters may be declared for a subroutine by using DCLSRPARM commands right after the BEGSR. See DCLSRPARM for more information.

ENDSR must be the last statement in the subroutine.

See Also

Built-In Controls | BEGFUNC | Custom Attributes | DCLSRPARM | ENSDR | EXSR

BITOFF - Set Bits Off

Allows bits to be set off in the **Target** parameter based upon a mask specified in the **Mask** parameter.

```
BITOFF
Mask (Character Expression)
Target (Character Expression)
Type (*BYTE)
```

Parts

Mask

Required. **Mask** can contain:

Bit numbers 0-7. Enclose the bit numbers in apostrophes. For Example, to set bits 0, 3, 6 on, enter '036' in Factor 2.

A valid character expression.

Hexadecimal literal (x'85') for Example (Binary 10000101).

Named constant up to 8 positions long containing the bit numbers to be set OFF.

Target

Required. It can specify a one-position character field, or it can be an array element if each element in the array is a one-position character field. **In AVR.Net, this parameter of the 'BITOFF' command requires a byte type or a type that can be converted to a byte.**

Type

Optional. If specified, must be *BYTE.

Remarks

When using Bit operations to format a character, you should use both BITON and **BITOFF**. **BITON** specifies the bits to be set on (=1). **BITOFF** to specify the bits to be set off (=0). You may not get the character you want unless you explicitly set all the bits in a character On or Off.

Example

```
// Initialize HEX00 to X'00'.
BITOFF MASK('01234567') Target(HEX00)
BITON X'00' HEX00

// Initialize HEX01 to X'01'.
BITOFF '01234567' HEX01
BITON X'01' HEX01

//Initialize HEX21 to X'21'.
BITOFF '01234567' HEX21
BITON X'21' HEX21
```

See Also

BITON | TESTBITS

BITON - Set Bits On

Allows bits to be set on in the **Target** parameter based upon a mask specified in the **Mask** parameter.

```
BITON
Mask (Character Expression)
Target (Character Expression)
Type (*BYTE)
```

Parts

Mask

Required. *Mask* can contain:

Bit numbers 0-7. Enclose the bit numbers in apostrophes. For Example, to set bits 0, 3, 6 on, enter '036' in Factor 2.

A valid character expression.

Hexadecimal literal (x'85') for Example (Binary 10000101).

Named constant up to 8 positions long containing the bit numbers to be set OFF.

Target

Required. It can specify a one-position character field, or it can be an array element if each element in the array is a one-position character field. **In AVR.Net, this parameter of the 'BITON' command requires a byte type or a type that can be converted to a byte.**

Type

Optional. If specified, must be *BYTE.

Remarks

When using Bit operations to format a character, you should use both **BITON** and **BITOFF**. **BITON** specifies the bits to be set on (=1). **BITOFF** to specify the bits to be set off (=0). You may not get the character you want unless you explicitly set all the bits in a character On or Off.

Example

```
// Initialize HEX00 to X'00'.
BITOFF MASK('01234567') Target(HEX00)
BITON X'00' HEX00

// Initialize HEX01 to X'01'.
BITOFF '01234567' HEX01
BITON X'01' HEX01

//Initialize HEX21 to X'21'.
BITOFF '01234567' HEX21
BITON X'21' HEX21
```

See Also

[BITOFF | TESTBITS](#)

CAB - Compare and Branch

Computes the comparison expression, and if true, branches to the TAG operation or label associated with the LABEL parameter. Otherwise, the program continues with the next operation in the sequence.

```
CAB
Cond (Expression)
Label (Character expression)
```

Parts

Cond

Required. The expression of the condition.

Label

Required. The name of the routine to branch to.

Remarks

The **CAB** command is **not** allowed within a TRY/CATCH block.

Example

```
:
  CAB COND( DollarsEarned <> 0 ) LABEL( OK )
  MSGBOX MSG( 'Error - DollarsEarned = 0. Press OK to abort' )
  MOVE Source( *ON ) Target( *INLR )
  RETURN
  TAG Name( OK )
  DIV F1( HoursWorked ) F2( DollarsEarned ) Result( DollarsPerHour )
:
```

See Also

COMPARE | IF | TAG

CALL - Call a Program

Transfers program control to an AS/400 program.

```
CALL
Pgm (Character Expression | Form variable)
ParmList (*NONE| Parameter List Name)
Db (Database Object)
Err (Indicator Variable | *EXTENDED)
Lr (Indicator Variable)
```

Parts

Pgm

Required. **Pgm** specifies the name of the program or form that control is transferred to. This can either be an AS/400 program or a local program that is part of the same project.

ParmList

Optional. **ParmList** specifies the parameter list name. *NONE is the default and indicates that there are no parameters to pass.

Db

Required. **Db** specifies the database server location for the program being Called. Database Objects are defined with the DCLDB command.

Err

Optional. **Err** determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - the specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Lr

Optional. The **Lr** parameter will be set when the last record is read.

Remarks

Calling an AS/400 Program:

When calling an AS/400 program, PGM should include the library name or *LIBL, i.e. CALL PGM('Library/Program') or CALL PGM(*LIBL/Program'). The Call will use the library list associated with the user ID associated with the database name. The library list can be changed by calling the appropriate OS/400 command.

The maximum number of parameters that can be passed in an AS/400 call is 36. The parameters can be of any simple types (packed, char, date, zoned etc...) or data structures or multiple occurrence data structures.

The AS/400 provides for three parameter data types for Commands (e.g., QCMDEXC) and Programs (CL, RPG, etc): Character, Boolean and Numeric. Typically, numeric parameters are always Packed. You must explicitly declare numeric parameters as Type(*packed) in your Visual RPG code. It may not be presumed that the Visual RPG compiler generates packed numerics.

You can also call AS/400 API's which can require a data structure as a Parm. This is supported and all required data types are permitted. When redefining data structure subfields, only the first definition is processed and converted between ASCII and EBCDIC.

For Example:

```
dclDS Name(MYparmDS)
dclsfld FieldA *char 10 // This value is passed and converted
dclsfld FieldB *packed len(7,2) // Ignored
dclsfld FieldC *zoned len(9,2) // Ignored
```

Returning from a Form Program:

A form program returns to the caller when:

The user closes the form.

The form is unloaded with the UNLOAD op code.

The form is hidden with the HIDE op code(a called form program is automatically UNLOADED with the Hide op code).

The program returns with the *INLR indicator on.

Returning from a Program (VRP) or an AS/400 Program:

A program returns when:

The mainline code finishes.

The RETURN op code is executed.

If *INLR is off when a program returns, all the files are kept open, and all the variables remain intact. If *INLR is off when returning from a form control, the form remains intact, and control is passed to the event cycle to process user input. Subsequent calls to this program are faster.

Example

```
dclDB Name( ProdDB ) DBName ("DG Net iSeries")
DclPlist Listout
  Dclparm TimeOfDay    Type(*char)    Len(10)
  Dclparm DayOfWeek    Type(*packed)  Len(1,0)
  Dclparm MonthOfYear  Type(*packed)  Len(2,0)

// Call to a program with a PList.
Call Pgm("LIBL/MyProgram") DB(ProdDB) ParmList(Listout)

// Call to a program with parms.
Call Pgm("LIBL/MyProgram") DB(ProdDB) ParmList(Listout)
  Dclparm TimeOfDay    Type(*char)    Len(10)
  Dclparm DayOfWeek    Type(*packed)  Len(1,0)
  Dclparm MonthOfYear  Type(*packed)  Len(2,0)
```

See Also

CONNECT | DCLDB | DCLPARM | DCLPLIST | HIDE | UNLOAD

CASE - Conditional Invoke Subroutine

Allows you to conditionally select a subroutine for processing.

```
CASE
Cond (Expression)
Sr (Subroutine Name)
ParmList (*NONE | Parameter List)
```

Parts

Cond

Required. **Cond** allows you to enter the expression of the condition.

Sr

Required. **Sr** is the name of the subroutine to process when the condition is met.

ParmList

Optional. The name of the parameter list. *NONE is the default.

Remarks

The **CASE** operation allows you to use a comparison expression.

Example

```
/* The Case operation compares OnHand with UnitSold. If OnHand is greater than or equal to UnitSold,
ReturnNow is processed and the program continues with the operation after the Endcs operation. */
Case Cond(OnHand >= UnitSold) Sr(ReturnNow)

/* If OnHand is not greater than or equal to UnitSold, the program compares OnHand with OnOrder. If
OnHand is equal to OnOrder, CutOrder is processed and the program continues with the operation after
the Endcs operation. */
Case Cond(OnHand = OnOrder) Sr(CutOrder)
```

/* If OnHand is not equal to OnOrder, the Case operation causes ReviewOrder to be processed before the program continues with the operation after the Endcs operation. The Case statement is used to provide a subroutine if none of the previous Case operations have been met. */

```
CaseDft Sr(ReviewOrder)
```

// The Endcs operation denotes the end of the Case group.

```
Endcs
```

See Also

CASEDFT | ENDCS | EXSR

CASEDFT - Conditional Invoke Subroutine

Specifies the default subroutine to be run in a CASE group.

```
CASEDFT
Sr (Subroutine Name)
ParmList (*NONE | Subroutine Name)
```

Parts

Sr

Required. Contains the subroutine name to be called. The name must be a valid subroutine name declared on a BEGSR command.

ParmList

Optional. The name of the parameter list. *NONE is the default.

Remarks

The **CASEDFT** operation is functionally identical to an EXSR operation. Any **CASE** operations that follow an unconditional **CASEDFT** operation in the same **CASE** group are never tested. The normal placement of the unconditional **CASEDFT** operation is after all other **CASE** operations.

See Also

BEGSR | CASE | ENDCS | EXSR

CATCH

You can use the **TRY...CATCH...FINALLY** statements for structured exception handling. This allows you to execute a particular block of statements if a specified exception occurs while your code is running. When this happens, the code is said to *throw* the exception, and you *catch* it with the appropriate **CATCH** command.

```
CATCH
Name (Variable name)
Type (Class filter type)
```

Parts

Name

Optional. Any variable name that is used to refer to the trapped exception object. The initial value is the value of the thrown error.

Type

Required. Specifies the type of class filter. If the value of **Name** is of the type specified by **Type** or of a derived type, the identifier becomes bound to the exception object.

Remarks

The **Try** statements are executed first. If no exception occurs, execution continues after the **ENDTRY**(*).

If an exception occurs, then the systems tries to match the exception to the exception types in the **CATCH** commands. The first one that matches causes the exception to be 'trapped' and the catch statements in the matched **CATCH** to be executed. If the exception is trapped by a **CATCH** then after the catch statements are executed, control goes to the next statement after the **ENDTRY** (*).

If a **FINALLY** command is present, then the finally statements will always be executed at the end of the **TRY/CATCH** sequence or **TRY**/untrapped exception, no matter what.

Rules

The **only** branching commands allowed inside a **TRY/CATCH** block are **LEAVE**, **ITERATE**, and **LEAVESR**. Any other branching command (**CAB**, **GOTO**, etc.) is not allowed in this context. It is also illegal to branch out of a **FINALLY** clause (i.e., you cannot use any of the branching commands (**ITERATE**, **CAB**, **LEAVESR**, **GOTO**, etc) in the body of a **FINALLY** clause.

The exception Type **MUST** derive from **System.Exception** (the mother of all exceptions). A **CATCH** with a **System.Exception** type will trap **all** exceptions.

If you have several **CATCH** commands, the **Name** must be different at this time.

Example

```
BEGSR TryCatch access( *Public )
  Dclfld myFld  Type( System.Windows.Forms.Button )
  Dclfld myInt  *integer 4
  TRY
    myFld.Show()
    myInt = myInt / 0
  CATCH name( z ) type( System.DivideByZeroException )
    msgbox msg( "by 0" ) Title( z.Message )
  CATCH name( e ) type( System.Exception )
    msgbox msg( "You can't do that!!!" ) Title( e.Message )
  FINALLY
    msgbox "Wow, this works..."
  ENDTRY
  msgbox "All is well!"
ENDSR
```

See Also

FINALLY | THROW | TRY | TRY...CATCH...FINALLY

CHAIN - Random Retrieval from a file

Retrieves a record from a full procedural file, and places the data into the input fields.

Please note that if you want to process a "Range" of records, you could use the **READRANGE** op code. This op code is especially useful in increasing performance and speed when you are working with large files. Also, if you will be accessing SQL files, you will definitely want to use **READRANGE**. The **READRANGE** op code still only reads 1 record like chain, but remembers the last position. **READRANGE** replaces **CHAIN**- **READE** loops.

```
CHAIN
From (File Name | Format Name)
Key (KeyList Name | Character Expression)
Access (*DFT |*NOLOCK )
NotFnd (Indicator Variable)
Err (Indicator Variable | *EXTENDED)
```


Parts

From

Required. **From** specifies the file or record format name that is to be read. A record format name is valid with an externally described file. If **From** is a record format name and access is by key, **CHAIN** retrieves the first record of the specified record type whose key matches the search argument. If no record is found, a No record Found condition exists, and the Not Found indicator is set On.

If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you must use the format name.

Key

Required. **Key** must contain the key or relative record number used to retrieve the record. If access is by key, **Key** can be any valid character expression. In addition, a DCL~~KL~~IST Name can be specified in **Key** for an externally described file. If access is by relative record number, **Key** must contain an integer literal or a numeric field with zero decimal positions. Refer to %KDS to use this built-in function to establish search arguments (Key) for Data Structures.

Access

Optional. **Access** determines if records are locked.

*DFT – (Default) If the parameter is not given, the default record locking behavior is taken. This means, if File is specified as INPUT, all records are read without locks. If File is specified as UPDATE, all records are locked if blank.

*NOLOCK – If this option is specified, then in no case will the record be locked when the **READRAND** operation is successful. The file specified in From is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record. If the **CHAIN** is not completed successfully, the file specified in From must be repositioned before a subsequent read operation can be done to that file.

NotFnd

Optional. Used to determine if the record was found while executing this command.

Note that you can also use the %FOUND function. However, be aware that %FOUND and the NotFnd indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. **Err** determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Example

/* The CHAIN operation retrieves the first record from the file CustMaster, that has a key field with the same value as the search argument CustNumber. Indicator 60 will come on (File record not found). */

```
CHAIN CustMaster CustNumber NotFnd(*In60)
If *In60 = *ON
    Subfile1AgentName = *Blanks
Else
    Subfile1AgentName = CustName
Endif
```


See Also

READ | READP | READRAND | READRANGE | %KDS

CHECK

Verifies that each character in a base string or array is among the characters in the comparator string.

```
CHECK
CmpStr (Character Expression)
BaseStr (Character Expression)
NoMatch (Numeric Variable | *NONE)
StartPos (Numeric Expression | 1)
Err (Indicator Variable | *EXTENDED)
Fnd (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
*FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
*ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

CmpStr

Required. Contains a character expression that is used to compare against the **BaseStr**.

BaseStr

Required. Contains a character expression that contains the string to compare with **CmpStr**.

NoMatch

Optional. Places an integer value in the result if a 'no match' condition is found.

StartPos

Optional. If the Start position is greater than 1, the result value is relative to the leftmost position in **BaseStr**, regardless of **StartPos**. Defaults to 1.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Fnd

Optional. Is set if base string characters are not found in the comparator string. This indicator must be present if the NoMatch parameter is not present. Note that you can also use the %FOUND function.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

Verification begins at the leftmost character of **BaseStr** and continues character by character left to right. Each character of **BaseStr** is compared with **CmpStr** characters. If a match is found in **BaseStr**, the next base string character is verified. If a 'no match' condition is found, an integer value is placed in **NoMatch** to indicate the position of the incorrect character.

CHECK stops checking when it finds the first incorrect character or when the end of **BaseStr** is reached. If no incorrect characters are found, **NoMatch** is set to zero.

Example

/* After the following Example, N=6 and the found indicator 90 is on. Because the start position is 2, the first nonnumeric character found is the '.'. */

```
Digits = '0123456789'  
Salary = '$2000.'  
Check CmpStr(Digits) BaseStr(Salary) StartPos(2) Fnd(*In90)
```

/* Because F1 is a blank, CHECK indicates the position of the first nonblank character. If STRING contains 'bbbASNA', NUM will contain the Value 4*/

```
STRING = // ASNA'  
Check CmpStr(' ') BaseStr(String) NoMatch(Num) Fnd(*In90)
```

/* The following Example checks that FIELD contains only the letters A to J. As a result, ARRAY=(136000) after the CHECK operation. Indicator 90 turns on.*/

```
DclArray Array 10 *ZONED Len(1,0)  
Letter = 'ABCDEFGHJIJ'  
Field = '1A=BC*'  
Check Letter Field Array Fnd(*In90)
```

/* In the following Example, because FIELD contains only the letters A to J, ARRAY=(000000). Indicator 90 turns off since all characters are valid.*/

```
Field = 'HIJHIJ'  
Check Letter Field Array Fnd(*In90)
```

See Also

CHECKR | %CHECK | %CHECKR

CHECKR

Verifies from right to left that each character in a base string or array is among the characters in the comparator string.

```
CHECKR
CmpStr (Character Expression)
BaseStr (Character Expression)
NoMatch (Numeric Variable) | *NONE)
StartPos (Numeric Expression) | 1)
Err (Indicator Variable | *EXTENDED)
End (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
        FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
        | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

CmpStr

Required. Verification begins at the rightmost character of **BaseStr** and continues character by character right to left. Each character of the base string is compared with **CmpStr** characters. If a match for a character in **BaseStr** exists in **CmpStr**, the next base string character is verified. If no match is found, an integer value is placed in **NoMatch** to indicate the position of the incorrect character. The position placed in **NoMatch** will be relative to the left, although checking is done from the right. **CmpStr** must be of the same type as **BaseStr**.

BaseStr

Required. Must contain either the base string or the base string, followed by a colon, followed by the start location. The base string portion of **BaseStr** can contain: a field name, array element, named constant, data structure name, data structure subfield name, literal, or table name. The start location portion of **BaseStr** must be numeric with no decimal positions, and can be a named constant, array element, field name, literal, or table name. If no start location is specified, the length of the string is used. **BaseStr** must be of the same type as **CmpStr**.

NoMatch

Optional. If not specified, you must specify the **Found Indicator**. It can be a numeric variable, numeric array element, numeric table name, or numeric array. Define the field or array specified with no decimal positions.

If **NoMatch** is an array, the operation continues checking after the first incorrect character is found for as many occurrences as there are elements in the array. If there are more array elements than incorrect characters, all of the remaining elements are set to zeros. **CmpStr** must be a string and can contain one of: a field name, array element, named constant, data structure name, data structure subfield, literal, or table name.

Figurative constants cannot be used in the **CmpStr**, **BaseStr**, or **NoMatch**. No overlapping is allowed in a data structure for **CmpStr** and **NoMatch**, or for **BaseStr** and **NoMatch**.

StartPos

Optional. Defaults to the length of the string.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Fnd

Optional. Is set if base string characters are not found in the comparator string. This indicator must be present if the **NoMatch** parameter is not present. Note that you can also use the %FOUND function.

Type

Optional. Can be any of the types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Example

```
/* Because Factor 1 is a blank character, CHECKR indicates the position of the first nonblank character.
This use of CHECKR allows you to determine the length of a string. If String contains 'ABCDEF ', Num
will contain the value 6. */
```

```
Checkr Cmpstr(' ') Basestr(String) NoMatch(Num)
```

```
/* After the following Example, N=1 and the found indicator 90 is on. Because the start position is 5, the
operation begins with the rightmost 0 and the first nonnumeric found is the '$'. */
```

```
DclConst Digits '0123456789'
Salary = '$2000.'
Checkr Cmpstr(Digits) Basestr(Salary) NoMatch(N) Startpos(5) +
Fnd(*In90)
```

```
/* The following Example checks that field contains only the letters A to J. As a result, array=(876310)
after the CHECKR operation. Indicator 90 turns on. */
```

```
DclArray Array DCLFLD(6) Len(1,0)
DclConst Letter 'ABCDEFGHJIJ'
DclFld Field *CHAR Len(8)
Field = '1A=BC***'
Checkr Cmpstr(Letter) Basestr(Field) NoMatch(Array) Err (*In90)
```

See Also

CHECK | %CHECK | %CHECKR

CLEAR

Sets elements in a structure (record format, data structure, array, table or a form) or a variable (field, subfield, property or indicator), to zero, blank or '0', depending on field type (numeric, character, or indicator). It allows you to clear structures on a global basis, as well as element by element, during run-time.

```
CLEAR
Result (Value)
Exclude (*NONE | *NOKEY)
Occurrence (*CURRENT | *ALL | *RECORDS)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
*FLOAT4 | *FLOAT8 | *IND |
*INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *ONECHAR | *PACKED |
*ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

Result

Required. Contains the elements to clear.

Exclude

Optional. Must be *NONE or not given unless Occurrence contains a DISK record format name; in which case, it can contain *NOKEY to indicate that all fields except key fields are to be cleared.

Occurrence

Optional. Valid only if the **Result** is a table, a record format, or a multiple occurrence data structure. *CURRENT is the default.

If *CURRENT (or nothing) is given, only the current occurrence will be cleared.

If *ALL is given, all occurrences will be cleared.

If you have partially overlapping fields of different definitions, data that is not valid could exist in numeric fields.

If you specify an array name, the entire array is cleared. If you specify an array element (including indicators) in factor 2 using an array index, only the element specified is cleared.

Type

Optional. Can be any of the types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

Time/Date/TimeStamp fields will be set to a default value depending on their type equal to 0001-01-01.00.00.00.000000.

Any property can be cleared, and any control that has a default property can be cleared. The default values work as follows:

Numeric properties: set to 0.
String properties: set to the empty string.
Picture properties: set to no picture.
Boolean properties: set to False (*OFF).
Date properties: set to 0001-01-01-00.00.00.000000

See Also

WRITE

CLOSE

Allows the user to close a file manually instead of when the program goes to end-of-job.

```
CLOSE  
File (*ALL| file name )  
Err (Indicator Variable | *EXTENDED)
```

Parts

File

Required. Will either contain the name of the file to be closed, or *ALL. **CLOSE** *ALL will only close those files that were implicitly and explicitly opened in that program as noted below. However, if all files were opened by the Startup program and a SHOWed form executed a **CLOSE** *ALL, no files would be closed.

When the **CLOSE** *ALL is issued from a shared member, only shared files in the class are closed.

When the **CLOSE** *ALL is issued from an instanced member, only instanced files in the object are closed.

Only Disk and Printer files apply. Memory files and subfiles are not included.

You cannot specify an array or table file in **File**. You can specify a resulting indicator to be set on if the **CLOSE** operation is not completed successfully.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Once a file has been closed, you cannot perform any operations on it unless you reopen it with an **OPEN** statement.

The explicit **CLOSE** operation closes one or more files or devices and disconnects them from the program. The file cannot be used again in the program unless you specify an explicit **OPEN** for that file.

An error will not occur if a **CLOSE** operation is performed to an already closed file.

Example

```
// The explicit CLOSE operation closes ItemMaster.

Close ItemMaster

/* The CLOSE *ALL operation closes all files in the program. You must specify an explicit OPEN for any
File that you wish to use again. If the CLOSE operation is not completed successfully, indicator 99 is set
on and an error subroutine is performed. */

Close *All
If *In99
Exsr ErrorOnClose
Endif
```

See Also

DCLDISKFILE | OPEN

COMMIT

COMMIT performs the following functions:

Makes all the changes to your files that have been specified in output operations since the previous **COMMIT** or **ROLLBACK** operation.

Releases all the record locks for files you have under commitment control.

```
COMMIT
Db (Database Object)
Marker (*NONE | Boundary)
Err (Indicator Variable | *EXTENDED)
```

Parts

Db

Required. The database connection for which transaction is to be committed. Database Objects are defined with the DCLDB command.

Marker

Optional. Allows you to specify a literal, named constant, array element, table name, data structure, or data structure subfield to identify the boundary between the changes made by this **COMMIT** operation and subsequent changes. If you leave Marker blank (or use default of *NONE), the identifier is null.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

The file changes and the record-lock releases apply to all the files you have under commitment control for the specified database connection.

COMMIT and **ROLLBACK** operations are supported for the iSeries DB2/400 and Windows SQL Server database engines. The **COMMIT/ROLLBACK** operations only apply to database files opened with the **COMMIT (*YES)** parameter.

See Also

DCLDB | ENDTPM | ROLLBACK | STARTTPM | Using Commitment Control

COMPARE

Compares **F1** with **F2** and places the result of the comparison in the resulting indicators.

COMPARE
F1 (Operand)
F2 (Operand)
Hi (Indicator Variable)
Lo (Indicator Variable)
Eq (Indicator Variable)

Parts

F1

Required. Can contain any valid Operand.

F2

Required. Can contain any valid Operand.

Hi

Optional. As a result of the comparison, the **Hi** indicator is set on if **F1** is greater than **F2**.

Lo

Optional. As a result of the comparison, the **Lo** indicator is set on if **F1** is less than **F2**.

Eq

Optional. As a result of the comparison, the **Eq** indicator is set on if **F1** is equal to **F2**.

Remarks

You must specify at least one resulting indicator. Do not specify the same indicator for all three conditions. When specified, the resulting indicators are set On or Off (for each cycle) to reflect the results of the compare.

Example

```
/* Initial values are :

ShirtsLong = 100
ShirtsShort = 105

PullOvers = 100

OnOrder = MasterOnOrder

ItemSold = MasterSold */

//Indicator 12 is set on. Indicators 11 and 13 are set off.

Compare ShirtsLong ShirtsShort Hi(*In11) Lo(*In12) Eq(*In13)
```



```
// Indicator 15 is set on; indicator 14 is set off.
Compare ShirtsLong ShirtsShort Hi(*In14) Lo(*In15) Eq(*In14)

// Indicator 18 is set on; indicator 17 is set off.
Compare ShirtsLong PullOvers Hi(*In17) Lo(*In17) Eq(*In18)

// Indicator 21 is set on; indicators 20 and 22 are set off.
Compare OnOrder ItemSold Hi(*In20) Lo(*In21) Eq(*In22)
```

See Also

Operands

CONCAT

Adds to the end of the character field specified by **Pre** (Prefix) the character field specified by **Suf** (Suffix). The concatenated field is placed in **Target**. Indicators are optional.

CONCAT
Pre (*NONE Character Expression)
Suf (String:Blanks)
Target (Character Expression)
BLANK (Numeric Expression)
Fill (*NOPAD *PAD)
Type (*NONE *BINARY *BOOLEAN *BYTE *CHAR *DECIMAL *FLOAT *FLOAT4 *FLOAT8 *IND *INTEGER *INTEGER2 *INTEGER4 *INTEGER8 *ONECHAR *PACKED *ZONED)
Len (Length Integer, [Decimal Integer])

Parts

Pre

Optional. When used, must contain a valid character expression.

If **Pre** is not specified, **Suf** is concatenated to the end of **Target**. **Pre** can contain a string, which can be one of: a field name, array element, named constant, data structure name, table name, or literal.

If **Pre** is not specified, **Target** is used. The following references to **Pre** apply to **Target** if **Pre** is not specified.

Suf

Required. Must contain a string that may contain the number of blanks to be inserted between the concatenated strings. Its format is the string, followed by a colon, followed by the number of blanks.

If graphic strings are being concatenated, the blanks are double-byte blanks.

The string portion can contain one of: a field name, array element, named constant, data structure name, table name, literal, or data structure subfield name.

The number of blanks portion must be numeric with zero decimal positions, and can contain one of: a named constant, array element, literal, table name, or field name. If a colon is specified, the number of blanks must be specified. If no colon is specified, concatenation occurs with the trailing blanks, if any, in **Pre**, or **Target** if **Pre** is not specified. If the number of blanks, N, is specified, **Pre** is copied to **Target** left-justified. Use the **Blank** () parameter to specify the number of blanks. The number of blanks must be numeric with zero decimal places, and can be a literal, an array element, or field name. Leading blanks are never dropped.

Target

Required. Must be a string and can contain one of: a field name, array element, data structure name, or table name. It is always left justified and its length is the length of **Pre** and **Suf** combined, plus any intervening blanks. If it is not, truncation occurs from the right.

BLANK

Optional. Indicates that **Target** should be padded on the right with blanks after the concatenation occurs, if **Target** is longer than the result of the operation. Only the left-most part of the field is affected if padding is not specified.

Resulting indicators are optional. If specified, they are set just as if a COMPARE operation had executed, whether or not the condition in the operation code was satisfied. Avoid using indicators in this way, however, as they detract from the structure and readability of the program. This capability is provided only for compatibility with the AS/400 and System/38.

Fill

Optional. Specifies if you want to fill, or pad **Target** (Target) with blanks. The default is *NOPAD.

*NOPAD indicates that **Target** will not be padded with blanks.

*PAD indicates that **Target** will be padded with blanks.

Type

Optional. Can be any of the types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

The CONCAT operation concatenates the string specified in **Suf** to the end of the string specified in **Pre** and places it in **Target**. The source and target strings must all be of the same type, either all character or all graphic.

See Also

+ Operator | COMPARE

CONNECT - Open a Connection to a Database

Opens a connection to the specified Database object.

```
CONNECT  
Db (Database Object)  
Err (Indicator Variable | *EXTENDED)
```

Parts**Db**

Required. A Database Object defined with the DCLDB command.

Err

Optional. **Err** determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

To close the connection to the database, use DISCONNECT.

See Also

DCLDB | DCLDISKFILE | DISCONNECT | %ERROR

DCLALIAS - Declare an Alias

DCLALIAS is used to 'alias' a variable name to an arbitrary expression.

```
DCLALIAS
Name (Field name)
Exp (Expression)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
        *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 |
        *INTEGER8 | *OBJECT | *ONECHAR | *PACKED | *STRING | *TIME |
        *TIMESTAMP | *ZONED | Field, Array, Array Element, table name, String
        Literal)
Len (Integer Literal, Integer Literal)
TimFmt (*PGMDFT | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
        *YMD)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
```

Parts**Name**

Required. A **Name** to be associated with an expression.

Exp

Required. An expression to be used instead of the name. An expression can refer to an arbitrary expression regardless of whether that expression could be on the left hand sign of an assignment. However, if it is used in such a manner, the compiler will give an error on the offending line.

Type

Optional. A **Type** can be provided as part of **DCLALIAS**, so the **Name** doesn't have to exist prior to the definition. However, if there is a field defined with the same name AND a Type is provided, then the Type must match the type of the field.

Len

Optional. **Len** specifies the length of the field. The length of a *BOOLEAN, *FORM, *IND, *OBJECT or *STRING type is implicit. The length of a *INTEGER Type is 2- or 4-bytes. The length of a *FLOAT Type is 4- or 8-bytes.

TimFmt

Optional. **TimFmt** specifies the format for time, date, and timestamp variables. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

You may encounter a date numeric variable from a system function or OCX that has a format similar to dddddd.nnnnnnnnn, where values to the left of the decimal represent date information while values to the right of the decimal represent time. For Example 36167.383645833 is 09:12:27 on 07 January, 1999. To isolate the date and time, simply assign this value to an OLE date variable and OLE time variable, respectively.

Access

Optional. The type of access to the field. ***PRIVATE** is the default.

***PRIVATE** (default) - access to the field is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

***PUBLIC** - access to the field is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

***PROTECTED** - access to the field is accessible only from within the same class, or from a class derived from this class.

***INTERNAL** - access to the field is accessible from within the same project, but not from outside the project.

Remarks

If there is a field defined with the same name AND any of the optional fields are provided, the optional fields must match.

Example

The following shows two Examples of using the **DclAlias** command. The first set of Examples shows creating the variable name with the **Type** with **DclAlias**. The second set of Examples shows creating the variable names with **DclFld**.

```
DclFld   Msg      Type(*Char) Len(7)
DclAlias MsgID Exp(%Subst(MSG, 1, 3)) Type(*Char) Len(3)
DclAlias MsgNo    %Subst(MSG, 4, 4)  Type(*Char) Len(4)
```

```
DclFld   FieldA Type(*Char) Len(20)
DclFld   FieldB Type(*Char) Len(20)
DclFld   x38    Type(*Char) Len(38)
```

```
DclAlias FieldA Exp(%Subst(X38, 5, 7))
DclAlias FieldB Exp(X38)
```

```
/* DclAlias FieldB Exp(X38) Type(*Char) Len(5) // This Declaration of
'FieldB' would conflict with the previous line.*/
```

An alias can refer to an arbitrary expression regardless of whether that expression could be on the left hand sign of an assignment; however, if it used in such a manner, the compiler will give an error on the offending line. For Example given the following declarations:

```
DclFld   Cost      Type(*Zoned) Len(5,0)
DclAlias DoubleCost Exp(Cost*2) Type(*Zoned) Len(6,0)
```

```
SalePrice = DoubleCost * .8 // 80% of twice the cost
DoubleCost = 99             <<<< Error : Cannot assign to -- it is read only
The compiler will complain when the value of DoubleCost is being assigned a
99.
```

See Also

DCLALIASGROUP | DCLFLD

DCLALIASGROUP

DCLALIASGROUP is used to 'alias' a **field** name to an **array** element. This command is especially useful when a group of fields are desired to be referenced as consecutive elements of an **array**.

```
DCLALIASGROUP
Array (Declared ArrayName)
Flds (Fld1, Fld2, Fld3, ...)
Fmt (Form name)
Count (Numeric Expression)
StartIndex (0 or number)
```

Parts

Array

Required. The name of the **array** of the same type that has been previously declared using DCLARRAY.

Note The array must be declared in the top of the program (not in a subroutine) prior to using DCLALIASGROUP.

Flds

Required. **Flds** can be assigned in two ways. In any event, the fields' type and length **must** be the same as the type of the array in the **Array** keyword.

Each field name is aliased to its corresponding element in the array (maximum of 255). For Example, Flds(Banana, Apple, Cherry) will map to the consecutive elements of the aliased array beginning at the value of **StartIndex**.

Fields from a record format (**Fmt** parameter) are aliased to the array beginning with the field name given. For Example, **Flds**(CSSales01) when aliased to a 12 element array will map 12 consecutive fields in the format starting with CSSales01. If the Count (**Count** parameter) is not given, then the count of fields defined in the format for aliasing cannot be fewer than the number of array elements available to be mapped. In other words, the length of the array less the **StartIndex** must be less than or equal to the number of fields to be aliased.

Fmt

Optional. Contains the name of the record format containing the aliased fields. When the format name is given, the **Flds** parameter must name only one field of the format.

Count

Optional. Contains the number of fields to be aliased to the array.

If the **Flds** parameter is a list, this parameter can be omitted.

If mapping fields from a record format, the elements of the array can be larger than the available number of alias fields if the correct alias count is given.

StartIndex

Optional. **StartIndex** (if used) should contain a number which is used as the starting index into the alias array. The number of fields in the **Flds** keyword plus the index in **StartIndex** must be equal or less than the size of the array. The default is 0.

Remarks

Array should be an array of size sufficient to alias **all the fields** in the **Flds** keyword.

For Example, the fields January, February, March etc.... can be aliased to Array[0], Array[1], Array[2] etc...

Example

The following shows 2 different ways to use DclAliasGroup. Also note the when using DclArray, the "Rank" keyword "is not allowed for the Array.

```
DclArray MySales      Type(*Packed) Len( 11, 2 ) Dim( 12 )
DclAliasGroup MySales Flds( CSSales01, CSSales02, CSSales03, CSSales04,
CSSales05, CSSales06,
                        CSSales07, CSSales08, CSSales09, CSSales10,
CSSales11, CSSales12 )
-OR -
DclAliasGroup MySales Flds( CSSales01 )  Fmt( RCSMastL1 )  Count( 12 )
```

See Also

DCLARRAY | DCLOVERLAYGROUP

DCLARRAY

Declares an n-dimensional array.

```
DCLARRAY
Name (name)
Dim (Integer Literal, Integer Literal, ...)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
*FLOAT4 | *FLOAT8 |
      *FORM | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *ONECHAR
      | *OBJECT | *PACKED | *STRING | *TIME | *TIMESTAMP | *ZONED)
Len (Length Integer, [Decimal Positions Integer])
TimFmt (*PGMDFT | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
*YMD)
Rank (IntegerLiteral)
Shared (*NO | *YES)
Access (*PRIVATE | *INTERNAL | *PUBLIC | *PROTECTED)
Static (*NO | *YES)
Table (*NAMESPEC | *YES | *NO)
Seq (*NONE | *ASCEND | *DESCEND)
New (*NOOBJECT | *DFT | Character Expression List)
Inz (Constant | Constant Expression)
Attributes (Attribute 1, Attribute 2, ...)
Like (Field name)
```

Parts

Name

Required. The Name parameter can contain either:

The name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

a valid program id registered on your system at compile time.

a valid class id registered on your system at compile time.

a valid interface name registered on your system at compile time.

A name of the form Library.Object, where Library is the name of one of the libraries selected in the new up and coming object libraries dialog, and Object is valid object name within that library.

Dim

Optional. The number of dimensions of a multidimensional array. Each Integer Literal on the **Dim** parameter specifies a dimension for the array. Individual data structures can be accessed with the OCCUR command or with the [] operator.

Type

Optional. **Type** can be any of Special Value (i.e., *CHAR, *BINARY, etc.) or it can be a class name. If Type is *BOOLEAN, you can use the *TRUE or *ON and *FALSE or *OFF special values to indicate **True** or **False**.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer given and *ZONED is assumed if there are two integers.

See Type Parameter for more information.

Len

Optional. **Len** defines the length of the field. If Len is decimal, e.g. Len (3,1), *ZONED is assigned as the default. If **Len** is character, et. Len (3), *CHAR is assigned as the default.

TimFmt

Optional. **TimFmt** specifies the format for time. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Rank

Optional. Rank declares an n-dimensional array of indefinite elements. The value given for Rank defines the number of dimensions for the array.

Shared

Optional. **Shared** specifies whether or not this array is to be shared. *NO is the default.

*NO (default) - this array will not be shared.

*YES - this array will be shared.

Access

Optional. The type of access to the array. *PRIVATE is the default.

*PRIVATE (default) - access to the array is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the array is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the array is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the array is accessible from within the same project, but not from outside the project.

Static

Optional. **Static** indicates that the declared array will remain in existence and retain its latest values after termination of the procedure in which they are declared. The default is ***NO**.

You can use **Static** only at procedure level. This means you can declare static arrays inside a procedure, but not at a class level.

***NO** (default) - the declared array will not remain in existence and will not retain its latest values after termination of the procedure in which it are declared.

***YES** - the declared array will remain in existence and retain its latest values after termination of the procedure in which its are declared.

Table

Optional. The number of dimensions of a multidimensional array parameter determines whether an array should behave like an RPG table. Tables have an implicit index, which can be manipulated with the **LOOKUP** op code and the **%TLOOKUP** function.

***NAMESPEC** specifies that arrays that have a name beginning with **"TAB"** are considered tables.

***YES** specifies the array is a table regardless of the name.

***NO** specifies that it is **not** a table, regardless of the name.

Seq

Optional. The **Seq** parameter determines the presumed sequence of the array. The programmer must ensure that entries are actually in the correct sequence. This aids the **LOOKUP** command to be able to search the array faster, because not every entry in the array has to necessarily be searched. ***ASCEND** or ***DESCEND** must be specified to use **%LOOKUPLT**, **%LOOKUPLE**, **%LOOKUPGT**, or **%LOOKUPGE**.

New

Optional. **New** allows you to specify a value when a field is of a class type (like **System.Text.StringBuilder**). You specify either a list of expressions or ***DFT** to use the parameterless constructor of the class. ***NOOBJECT** is the default.

***NOOBJECT** (default) allows you to use the parameterless constructor of the class.

***DFT** allows you to use the parameterless constructor of the class.

Example: `DclFld Address type(System.Text.StringBuilder) New(*Dft)`

// Constructs a **StringBuilder** object with the parameterless constructor. See the help for **StringBuilder**.

Character Expression List - Specify a list of expressions for the default class value.

Example: `DclFld Address type(System.Text.StringBuilder) New("This is the initial value")`

// Constructs a **StringBuilder** object with initial value "This is the initial value". See the help for **StringBuilder**.

Inz

Optional. **Inz** allows you to specify a default value when a field is of a value type (like ***Zoned**, ***Integer**, etc). The value should be a constant (or a constant expression, such as **%DecPos ()**), of the appropriate type.

Attributes

Optional. Allows you to provide one or more attributes that apply to this class. To specify more than one attribute, simply separate them with commas.

Like

Optional. Like must be the name of a field already defined in the class. **Type** cannot be a named constant or a literal. In this case, the field will be defined based on the attributes of another field. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the field being defined. The field length entry is allowed only for field definitions requiring a length.

Remarks

DCLARRAY declares a multidimensional array of simple variables where **Dim** (required) is the number of dimensions. The array can be accessed by using the **[]** operators.

Use **DCLDSFLD** to declare an array within a data structure.

For DIM arrays:

Depending on the type of the array elements, one of the two keywords will contain information for initialization:

If the element is scalar (i.e. a numeric type, a char type, a string, etc) then **INZ** contains the initial value.

// this initializes each element of myIntArray to the value 9

```
DclArray myIntArray DIM(8) type(*integer) len(4) INZ(9)
```

If the element type is a class (e.g. `System.Text.StringBuilder`, or `System.Windows.Forms.Form`), then the **NEW** keyword contains the arguments to the constructor. Each element contains its own instance, even though they are all obtained with the same constructor arguments.

// this initializes each element of mySBArray // to *new System.Text.StringBuilder("Hello World")

```
DclArray mySBArray DIM(10) type(System.Text.StringBuilder) NEW("Hello World")
```

For RANK arrays:

The **INZ** keyword is not allowed, it is an error to use **RANK** and **INZ** in the same **DclArray**. The **NEW** keyword, if given, must contain a comma separated list of indices to give to the array constructor. The list of indices must match the **RANK** of the array. The array will be initialized to the size specified, and its elements will be initialized to the default value of the element type.

// this creates myIntArray with 45 (two-dimensional 9 X 5) elements initialized to 0 (the default value for integers)

```
DclArray myIntArray RANK(2) type(*integer) len(4) NEW(9,5)
```

// this creates myCharArray with 10 elements initialized to " " (8 blanks, the default value for *char 8)

```
DclArray myCharArray RANK(1) type(*char) len(8) NEW(10)
```

// this creates mySBArray with 5 elements initialized to *nothing (the default value for class objects)

```
DclArray mySBArray RANK(2) type(System.Text.StringBuilder) NEW(5)
```

You can initialize the number of elements and their content when the array is instance using the following syntax:

```
Dclarray MyStrArr type(*string) rank(1)

// below will create array of 5 elements

MyStrArr = *new *string[] {"The", "quick", "brown", "fox", "jumped"}
Dclarray MyIntArr type(*integer4) rank(1)

// below will create array of 3 elements

MyIntArr = *new *integer4[] {10, 100, 1000}
```

Example

```
DCLFLD Name($X) Type(*ZONED) Len(2,0)
DCLFLD Name($Y) Type(*ZONED) Len(2,0)
//Declare an 8 x 8 two-dimensional array of characters within a data
structure called MyDs.
```

```
DclDs Name(MyDs)
DclDsFld Name(Z8by8) DCLFLD(8,8) Type(*char) Len(5)

// Initialize the array to all *'s.
X = *zeros
  Do 1 8 $X
    $Y = *zeros
    Do 1 8 $Y
      Z8by8[$X,$Y] = '*****'
    Enddo
  Enddo

// Declare an 8 x 8 two-dimensional array of characters.
DCLARRAY Name(ARR1) DIM(8,8) Type(*CHAR) Len(10)

// Initialize the array to all *'s.

$X=0
DOUNTIL ($X=8)
  $Y=0
  $X=$X+1
  DOUNTIL ($Y=8)
    $Y=$Y+1
    ARR1[$X,$Y] = '*****'
  ENDDO
ENDDO
```

See Also

[Arrays Overview](#) | [Custom Attributes](#) | [DCLDS](#) | [DCLDSFLD](#) | [DCOVERLAYGROUP](#) | [DCLFLD](#) | [LOOKUP](#) | [%LOOKUP](#) | [%TLOOKUP](#)

DCLCONST - Declare a Constant

Declares a named constant. By definition, constants are shared members and as such can only be declared at the class level and cannot be declared locally in subroutines or functions.

```
DCLCONST
Name (Character Expression)
Value (Constant Expression)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
```

Parts

Name

Required. The name of the constant.

Value

Required. The expression of the constant.

Access

Optional. The type of access to the constant. *PRIVATE is the default.

*PRIVATE (default) - access to the constant is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the constant is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the constant is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the constant is accessible from within the same project, but not from outside the project.

See Also

Constant Expression | DCLFLD | Literals

DCLDB - Declare a Database

Declares a database. The **Name** parameter can then be used on DCLDISKFILE's **Db** parameter and DCLMEMORYFILE's **DbDesc** parameter.

```
DCLDB
Name (Database name in code)
DbName (Actual Database name)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
Shared (*NO | *YES)
```

Parts

Name

Required. A character expression containing the name to call your database in code.

DbName

Optional. A character expression containing the name of the actual database.

Access

Optional. The type of access to the file. *PRIVATE is the default.

*PRIVATE (default) - access to the database is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the database is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the database is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the database is accessible from within the same project, but not from outside the project.

Shared

Optional. Shared specifies whether or not this database is to be shared. *NO is the default.

*NO (default) - this database will not be shared.

*YES - this database will be shared.

Remarks

You must specify the **Name** parameter, which is the name to call your database in code.

The **DbName** parameter is optional and is the name of the actual database you are declaring.

Use **CONNECT** to create a connection to a database, and **DISCONNECT** to close a database connection.

To 'load' the contents of the registry for a particular database name, set the **DbName** property of the database object declared with **DCLDB**. If a database name exists with the name, values such as server, label, and user, etc., become current and available through properties of the object.

You can use the **CreateName** method to take all the current values of the database name, except for the name itself, and creates a new 'Database Name' in the registry with the name given in **NewName**.

You can use the **UpdateName** method to write the current values of the database name object as the new values of the database name in the registry.

There are **two methods** which are Shared. These methods don't affect the current values of the database object.

GetNames Method (*Boolean Public Databases) - Returns an array of string with all the names found in the registry. The boolean parameter indicates whether to get **PublicDatabase (True)** or those private to the current user (**False**).

DeleteName Method (*String ExistingName) - This method deletes from the registry an existing database Name.

Because **ASNA.DataGate.Client** is a managed object, you should never refer to a **dcIDB** object in **Finalize**.

See Also

[CONNECT](#) | [CreateName Method](#) | [DbName Property](#) | [DeleteName Method](#) | [DISCONNECT](#) | [UpdateName Method](#) | [GetNames Method](#) | [Database Class](#) | [Database Class Members](#)

DCLDELEGATE

Declares a delegate.

```
DCLDELEGATE
Name (name)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
        *FLOAT4 | *FLOAT8 | *FORM | *IND | *INTEGER | *INTEGER2 |
        *INTEGER4 | *INTEGER8 | *ONECHAR | *OBJECT | *PACKED | *STRING |
        *TIME | *TIMESTAMP | *ZONED)
Len (Length Integer, [Decimal Positions Integer])
TimFmt (*PGMDFT | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
        *YMD)
Rank (IntegerLiteral)
Access (*PRIVATE | *INTERNAL | *PUBLIC | *PROTECTED)
Attributes (Attribute 1, Attribute 2, ...)
```

Parts

Name

Required. Name of the delegate type; follows standard variable naming conventions.

Type

Optional. **Type** can be any of Special Value (i.e., *CHAR, *BINARY, etc.) or it can be a class name. If **Type** is *BOOLEAN, you can use the *TRUE or *ON and *FALSE or *OFF special values to indicate **True** or **False**.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer given and *ZONED is assumed if there are two integers.

Len

Optional. **Len** defines the length of the delegate. If **Len** is decimal, e.g. Len (3,1), *ZONED is assigned as the default. If **Len** is character, et. Len (3), *CHAR is assigned as the default.

TimFmt

Optional. **TimFmt** specifies the format for time. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Rank

Optional. **Rank** declares an n-dimensional array of indefinite elements. The value given for **Rank** defines the number of dimensions for the array.

Access

Optional. The type of access to the delegate. *PRIVATE is the default.

*PRIVATE (default) - access to the delegate is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the delegate is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the delegate is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the delegate is accessible from within the same project, but not from outside the project.

Attributes

Optional. List of variables representing arguments that are passed to the procedure when it is called.

Remarks

This command declares a delegate. A delegate is a **type** that describes a method. Having the type allows programmers to pass methods as arguments, to declare fields that hold methods, and to declare events using the delegate as the event type.

See Also

[Arrays Overview](#) | [Custom Attributes](#) | [DCLDS](#) | [DCLDSFLD](#) | [DCLFLD](#) | [LOOKUP](#)

DCLDISKFILE - Declare a Disk File

Declares an externally described disk file for processing in a program.

```
DCLDISKFILE
Name (file name)
Type (*INPUT | *OUTPUT | *UPDATE)
Org (*ARRIVAL | *INDEXED)
Db (Database Object)
File (*NAME | String Literal)
Mbr (*FIRST | *FILE | String Literal)
Addrec (*NO | *YES)
ImpOpen (*YES | *NO)
Rnmfmt (*NONE | Old Format name, New Format, ...)
NetBlockFactor (*CALC | *NONE | Number of Records to Block)
CacheWrites (*NO | *YES | Indicator Expression)
ShareType (*FILE | *EXCLUSIVE | *EXCLUSIVEREAD | *SHAREUPDATE |
             SHARENOUPDATE | *SHAREREAD | Character Expression)
ChkFmtId (*YES | *NO | *SAME | *IGNORE | *MAP | Indicator Expression)
Prefix (prefix name, [Length])
Commit (*NO | *YES | Boolean/Indicator Expression)
QrySelect (*NONE | String Literal)
QryKeyFlds (*FILE | *NONE | String Literal)
QryFileName (*NAME | String Literal)
WaitRec (*FILE | *IMMEDIATE | *NOMAX)
Random (*YES | *NO)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
Shared (*NO | *YES)
ServerCursor (*DFT | *STATIC)
```

Parts

Name

Required. Name can contain either:

The name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

a valid program id registered on your system at compile time.

a valid class id registered on your system at compile time.

a valid interface name registered on your system at compile time.

A name of the form Library.Object, where Library is the name of one of the libraries selected in the new up and coming object libraries dialog, and Object is valid object name within that library.

Note: Using the DclDiskFile *name*, you can then get to its **fields** and **properties**. Please note that methods are reserved for use by the ASNA.Visual RPG compiler, so use the corresponding Visual RPG command if applicable. See DBFile Class Members for a listing of available members.

Type

Required. Type determines the accesses that will be permitted on the file. *Note that the **Type** parameter determines the **NetBlockFactor** default.* Input and Output files can employ network blocking as long the appropriate keyword is also specified. See **NetBlockFactor** parameter for more information.

Unless NetBlockFactor is *NONE, network blocking is employed for input and output files as described below.

*INPUT - will allow only read operations (i.e. READ, READP, CHAIN) to be performed on a file). The default for NetBlockFactor is *CALC - The system will calculate an efficient number of records to block.

*OUTPUT - only allows WRITE operations to the file. The default for NetBlockFactor is *NONE - No blocking will occur on the file.

*UPDATE - allows read, update and delete operations to a file. The default for NetBlockFactor is *NONE - No blocking will occur on the file.

Org

Optional. Specifies if the file is a Sequential (*ARRIVAL) or Indexed (*INDEXED) file. The default is *ARRIVAL.

*ARRIVAL (default) - the file is a sequential file.

*INDEXED - the file is an indexed file.

Db

Required. Db specifies the Database Object declaration whose DBName specifies where the compiler will acquire the file description. Db specifies the Database Connection to use when opening the file at run-time. Database Objects are defined with the DCLDB command.

File

Optional. File specifies the Library and File to use when opening the file at run-time.

If the File parameter is not given, the same file that was used to retrieve the description will be opened at run-time.

If a file name is given on the **File** parameter, that file will be accessed and must have the same description as the file accessed in the **Name** parameter unless **ChkFmtID** parameter is other than *SAME. The File parameter accepts literal as a parameter. To override the file at run-time, set the FilePath property for the file object in the Name parameter prior to opening the file.

Mbr

Optional. Mbr specifies the file member to use when opening the file at runtime. The **Mbr** parameter allows you to specify the name of the member you want to access or *FIRST. *FIRST is the default.

*FIRST (default) - will access the member with the oldest creation date.

*FILE - accesses the member name with the same name as the file name.

If a member name is given on the **Mbr** parameter, that member will be accessed when the file is opened. The **Mbr** parameter accepts literal as a parameter. To override the Member at run-time, set the Membername property for the file object in the **Name** parameter prior to opening the file.

Addrec

Optional. Addrec specifies whether you want to add a record to the file. The default is *DFT. You must use the Addrec (*YES) option if you wish to add a record to a *INPUT or *UPDATE file. *OUTPUT files automatically allow you to add records, and thus the Addrec parameter is not necessary.

To utilize **network blocking**, (**NetBlockFactor**) which significantly reduces the network burden of transferring records between server and client machines, use Addrec (*NO) with *INPUT files. When an input file is "blocked", the buffered data resides on the client PC. This allows successive READ and READE statements to execute quickly. Be aware that should records be updated in other definitions of the file, those changed are not correspondingly updated in the buffer. For Netblockfactor, either use *CALC (the system will calculate the number of records to block), or enter the maximum number of records to include in the block.

*DFT (default) - will look at the file type and add a record to a *INPUT or *UPDATE file.

*NO - with *INPUT file will enable network blocking, which significantly reduces the network burden of transferring records between server and client machines

*YES - will add a record to a *INPUT or *UPDATE file.

ImpOpen

Optional. ImpOpen specifies if the file will be implicitly opened. The default is *YES.

*YES (default) - specifies that the file will automatically be opened when the program is first run. If the implicit open is unsuccessful, a run-time error will occur.

*NO - specifies that the file will not automatically be opened when the program is first run. The RPG IV USROPN keyword is supported in Visual RPG by using **ImpOpen(*NO)**.

Rnmfmt

Optional. Rnmfmt renames the external formats to new, internal formats to avoid clashes with duplicate format names. *NONE is the default.

For **multi-format** files, you enter the external and internal format names in pairs. All names are separated by commas.

For **single** format files, you can either specify the internal format name as the only format name, or enter both the external and internal format names separated by a comma.

```
Rnmfmt ( RCMMaster, CustInfo) // single format
Rnmfmt ( CustInfo) // single format
Rnmfmt (RSubCust, CustSub, RSubSales, SalesSub) // multi-format
```

NetBlockFactor

Optional. NetBlockFactor determines if and optionally how many database records will be blocked across the Network for better performance when reading or writing groups of records. Network blocking significantly reduces the network burden of transferring records between server and client machines, and this is demonstrated by the substantial increase of data throughput.

Unless NetBlockFactor is *NONE, network blocking is employed for input and output files as described below.

INPUT - The file is declared with **Type(*INPUT)** and **AddRec(*NO)**. When an input file is "blocked", the buffered data resides on the client PC. This allows successive **READ** and **READE** statements to execute quickly. Be aware that should records be updated in other definitions of the file, those changed are not correspondingly updated in the buffer.

OUTPUT - The disk file is declared with **Type(*OUTPUT)** and **CacheWrites(*YES)**. When an output file is "blocked", the buffered data resides on the client PC until the block has been filled or the file is closed. Then the block is sent to the server in a single transaction. Be aware that I/O errors at the server are returned only during processing of the block. In other words, if the 33rd record in a block of 100 records had a duplicate key, your application would not be aware of it until after writing the 100th record. Therefore, the use of network blocking for output files should be employed only when your application can be restarted.

I/O operations that will reset the buffer are **SETLL**, **SETGT**, **SETRANGE**, **CHAIN** (READRAND) and **READRANGE**. *CALC - The system will calculate the number of records to block. The blocking factor is 6000 divided by the file's record length. If the record length is greater than 3000 (1/2 the block size),

*CALC (default) - will compute a record blocking Factor of 1 (one).

*NONE - no blocking will occur on the file. This is the default when the Type is other than *INPUT.

Record Count - the maximum number of records to include in the block. Note, the number of records in the block for SETRANGE with READ operations, and for READRANGE operations will be less than the record count if the set of records in the range is smaller than the maximum block size. Because of this dynamic blocking, the use of SETRANGE and READRANGE can realize even better performance than SETLL and READE, for Example.

Note: Record blocking requires additional memory to work. Blocking reserves memory for the complete block of each blocked file. For Example, a record length of 2560 and a blocking factor of 100 will require an additional 256K of memory.

CacheWrites

Optional. CacheWrites specifies whether data is written to disk immediately after an operation, or whether data is saved in a buffer, then written to disk when the Operating System has time. *NO is the default.

*NO (default) indicates that data will be written to disk at the time the operation is performed. Note that this option is slower than saving the data to a buffer.

*YES indicates that data operations will be saved to a buffer, and written to disk when the Operating System has time. Note that this option is faster. However, if the system fails with data in the buffer that has not been written to disk, the information will be lost.

When a disk file is declared with Type(*OUTPUT), Cachewrites(*YES) and NetblockFactor (*CALC or value), output records are buffered at the client before being sent as a block to the server database rather than being transmitted one record at a time.

ShareType

Optional. ShareType defines values on how a file will be shared. The default is *FILE.

*File (default) - Maintains the same share type specified when the file was created.

*Exclusive - Open for update, exclusive lock.

*ExclusiveRead - Open for update, other jobs may read.

*ShareUpdate - Open for update, other jobs may read or update.

*ShareNoUpdate - Open for read, other jobs may read.

*ShareRead - Open for read, other jobs may read or update.

ChkFmtId

Optional. ChkFmtId checks if the format ID of the file at run-time compares to the format ID of the file at compile-time. Visual RPG stores the format of file(s) when a program is first compiled. At run-time, Visual RPG compares the format ID that was stored during the compilation with the current format ID. The comparing of format ID's can be especially useful when checking if new fields have been added to a file used by a program that has been compiled months ago.

***SAME** is the default, and indicates that the run-time and compile-time format ID's **will be compared** to verify that they are the same. (This is exactly equivalent to *YES in previous releases).

***IGNORE** indicates that the run-time and compile-time format ID's **will not be compared** to verify that they are the same. Visual RPG will use the compile-time format ID. (This is exactly equivalent to *NO in previous releases).

Note Please be extremely careful in using *IGNORE, as it has the risk of the corruption of data files and misbehavior of your application!!.

***MAP** indicates that the run-time and compile-time format ID's will be compared, and Visual RPG will accommodate non-key fields added and changed **regardless** of where they are defined in the new record format. When the modified file is opened, Visual RPG will rebuild, or **map** its compiled record format at run-time to equal that of the new record format.

Restrictions apply to non-key fields depending upon the mode of file processing and the changes made to the file. The rule of thumb is that Visual RPG will process the file provided there is **no loss of data**.

Any change in the count, order or type of Key fields is not allowed, except that a numeric key may be changed from packed to zoned and vice versa, as long as the precision remains unchanged.

Prefix

Optional. Prefix allows the specification of a string which is to be prefixed to the field names of an externally described record format. The characters specified as (prefix name) are prefixed of all fields defined in all record formats of the file.

You can also optionally specify a numeric value (length) to indicate the number of leading characters in the existing name to be replaced. If a length is not specified or is zero, the string is prepended to the name.

The total Length of a name after applying the prefix must not exceed the maximum length of an RPG field name.

If a Length is specified, it must be a numeric constant containing a value between 0 and 13 with no decimal places. Specifying a value of zero is the same as not specifying a length at all.

The Length parameter must not be greater than or equal to number of characters in the name to be prefixed. That is, the resulting name cannot be the same as the prefix string after the prefix is applied.

Commit

Optional. Commit allows you to control if commitment control will be enabled for this file. You can also change at runtime by specifying a Boolean/Indicator expression. The default is *NO.

The parameter must be set prior to opening the file. If the file is opened explicitly using the OPEN operation in the calculation specifications, the parameter can be set prior to the OPEN operation.

Use the COMMIT and ROLLBACK operation codes to group changes to this file and other files currently under commitment control, so that all changes happen together, or do not happen at all.

*NO (default) - indicates that commitment control is not enabled.

*YES - indicates that commitment control has been enabled.

Boolean/Indicator Expression - allows commitment control to be specified or changed at runtime.

QrySelect

Optional. QrySelect specifies the selection expression used to determine which records are to be made available for processing. The select expression syntax is the same as for a **Select/Omit** expression in DataGate.

QryKeyFids can be used in conjunction with **QrySelect** parameters to specify ordering of the selected records. Use **QryFileName** to specify the name of the temporary query file created.

When using QrySelect, please note:

*NONE (default) - there is no selection expression available for processing.

The parameter must be string literal (use DBFile.QrySelect property to set as a character expression at runtime).

If there are any errors in the character expression, they will not be caught until the file is opened at run-time.

Selection expressions are evaluated during file open time. Therefore, a runtime error will be generated during the opening of the file when a selection expression error is encountered.

QryKeyFids

Optional. The **QryKeyFids** parameter is used to specify the name of one or more key fields used to arrange the query records. Each key field is composed of a field name followed by optional attributes that determine the sorting order (ascending descending, or absolute value for numeric fields).

*FILE - is the default, which specifies that the original keys in the file should be used.

*NONE - specifies that the file has no keys.

String Literal - contains a string containing key field name(s) and corresponding sort attributes. (Use DBFile.QryKeyFids property to set as a character expression at runtime).

To establish the query key fields at runtime, set the QryKeyFids property for the file object in the Name parameter prior to opening the file.

When specifying QryKeyFids with key fields that do not match the original file's key definition, you must also specify **Random(*NO)**. In addition, in order to process the file in the sequence specified by the **QryKeyFids** parameter, you must specify **Org (*INDEXED)**.

Valid input operations for **Random(*NO)** files are limited to: **READ**, **READP** and **SETLL** when used with the new special values ***START** and ***END**.

Syntax: The syntax for the QryKeyFids parameter is:

```
(KeyFieldName Attributes ...)
```

Key Field Name must be a valid field name in the file.

Attributes consist of the following: ***ASCEND**, ***DESCEND**, or ***ABSVAL**. You can have as many attributes as you want after a field name as long as they conform to the rules below.

Indicates that you can repeat the sequence for as many key fields as you want.

When using QryKeyFids, please note:

Each key field name and attribute must be separated by at least one blank.

*ASCEND and *DESCEND cannot be applied to the same field.

*ABSVAL only applies to numeric fields.

The parameter can be a character literal.

If there are any errors in the key fields' expression, they will not be caught until the file is opened at run-time.

To establish the query selection at runtime, set the **QrySelect** property for the file object in the **Name** parameter prior to opening the file.

Use **QrySelect** in conjunction with **QryKeyFlds** to specify the selection criteria indicating which records to select from a file. Use **QryFileName** to specify the name of the temporary query file created. Selection expressions are evaluated during file open time. Therefore, a runtime error will be generated during the opening of the file when a selection expression error is encountered. Example: **QryKeyFlds** ('CustNo *ASCEND ORDERDATE *DESCEND LINENO TOTAL *ABSVAL *DESCEND') For more information on **QryKeyFlds** Examples and using open query files in Visual RPG, refer to the following link: [Using Open Query File in Visual RPG](#).

QryFileName

Optional. **QryFileName** specifies the name of the query file created for the **QrySelect** and **QryKeyFlds** parameters. If *Name is specified, which is the default, the file name given on the DCLDISKFILE command is used.

If the database access with this file is an AS/400 database, then the **QryFileName** is used both on the OVRDBF and the OPNQRYF commands used when opening the file.

For DataGate databases, a temporary logical file is created in QTemp with the **QryFileName** as its name. You can use the special name '*UNIQUE' (don't forget the quotes) to guarantee that there will be no conflicts with other files residing in QTemp. You will receive errors if there are duplicate file names in QTEMP.

It is recommended the use of **QryFileName('*UNIQUE')** when working with Web Applications. Web Applications run under one user, the user that is specified in DataGate Work with Database Names dialogue. Web Applications usually store Runtime Errors and MsgBoxes in \$\$Visual RPGErr.TXT.

*Name (default) - the file name given on the DCLDISKFILE's **Name** parameter is used.

To establish the query file name at runtime, set the **QryFileName** property for the file object in the **Name** parameter prior to opening the file.

For more information on **QryKeyFlds** and using open query files in Visual RPG, refer to the following link: [Using Open Query File](#).

WaitRec

Optional. **WaitRec** specifies the waiting period for a record. *FILE is the default.

*FILE (default) - indicates that the properties of the file are used to determine how long it waits for the record.

*IMMEDIATE - it won't wait for a record.

*NOMAX - it will wait indefinitely.

Random

Optional. Random specifies whether random database processing using Keys and RRNs is allowed. *YES is the default, which determines the random database processing using keys and RRNs is allowed.

*YES - (default) specifies that random database processing using Keys and RRNs on op codes such as SETLL, CHAIN, etc., is allowed.

*NO - specifies that random database processing using Keys and RRNs is not allowed. This option is commonly used when the Keys look different at runtime from the compile time. The common scenario is when using **QryKeyFids** and under this scenario *NO option must be used to avoid a runtime exception.

The only I/O operations allowed are:

Input - READ, READP and SETLL with *START or *END as the Keys. Any other operations throw compiler error - "Random Access Operation Not Allowed For File". The results of READ and READP depend on the **Org** option and if *INDEXED read them sequentially based on the Key of the file and *ARRIVAL reads in arrival sequence in the file.

Output - Write and Update.

Access

Optional. The type of access to the file. *PRIVATE is the default.

*PRIVATE (default) - access to the file is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the file is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the file is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the file is accessible from within the same project, but not from outside the project.

Shared

Optional. Shared specifies whether or not this file is to be shared. *NO is the default.

*NO (default) - this file will not be shared.

*YES - this file will be shared.

ServerCursor

Optional. For SQL Server files only; ServerCursor allows you to gain access to a table or view that isn't supported by Dynamic cursors . *Dft is the default, which relates to **Dynamic** cursors. **Please refer to your SQL Server documentation for information on Dynamic vs. Static cursors.**

*DFT (default) - Defines the server cursor at **Dynamic** .

*STATIC - Defines the server cursor as **Static**.

Remarks

Because ASNA.DataGate.Client is a managed object, you should never refer to a **dclDiskfile** object in **Finalize**.

Example

```

dclDB Name( DBLocal ) DBName ("DG Net Local")
dclDB Name( ProdDB ) DBName ("DG Net iSeries")
dclDB Name( DBSQL ) DBName ("DG SQL Server")

// Indexed file for input. File is "*LIBL/CMASTNEWL2"
dclDiskFile Name(Cust2) Type(*input) Org(*indexed) DB(ProdDB)
File("CMASTNEWL2")

// Sequential file for output, but do not open it until an OPEN operation is
used.
// File description is "*LIBL/SKUMAST"
dclDiskFile Name(SKUMAST) Type(*output) Org(*arrival) DB(DBLocal)
ImpOpen(*NO)

// Indexed file for input, and rename format to CUST1.
// File description is "*LIBL/CSMASTERL1"
dclDiskFile Name(SalesMast) Type(*input) Org(*indexed) DB(DBSQL) +
File("*LIBL/CSMASTERL1") RnmFmt(CUST1)

// Indexed file for update and addition. File description is
"*LIBL/CustMast"
dclDiskFile Name(CustMast) Type(*update) Org(*indexed) DB(DBSQL)
AddRec(*Yes)

// Indexed file for input and block 100 records. File description is
"TestLib/CMASTNEWL1"
dclDiskFile Name(CustFile) Type(*input) Org(*indexed) DB(DB400) +
NetBlockFactor(100) ImpOpen(*NO) File ("TestLib/CMASTNEWL1")

```

See Also

[CLOSE](#) | [COMMIT](#) | [DCLDS](#) | [DCLDSFLD](#) | [DCLFLD](#) | [DCLPRINTFILE](#) | [LOOKUP](#) | [OPEN](#) | [ROLLBACK](#)
[DBFile Class Members](#) | [DBFile Class](#)

DCLDS - Declare a Data Structure

Declares an internal or external data structure.

DCLDS
Name (Character Expression)
Type (Class name)
Dim (IntegerLiteral, IntegerLiteral, ...)
FldScope (*GLOBAL *LOCAL *PGMDFT)
ExtDesc (*NO (YES)
DbDesc (*NONE Database Object Character Expression)
FileDesc (*DSNAME Character Literal)
Prefix (prefix name, [Length])
Shared (*NO *YES)
Access (*PRIVATE *PUBLIC *PROTECTED *INTERNAL)
Attributes (Attribute 1, Attribute 2, ...)

Parts**Name**

Required. The **name** of the data structure.

Type

Optional. Provide a fully-qualified Class name to be used as the basis for the field descriptions of the data structure. All the public fields and properties of the class become the fields of the datastructure.

Dim

Optional. **Dim** allows you to create a multiple occurrence data structure. This can be multi-dimensional. Each Integer Literal on the **Dim** parameter specifies a dimension for the array. Individual data structures can be accessed with the OCCUR command or with the [] operator.

FldScope

Optional. **FldScope** defines whether the subfields will be accessible directly in the program (*GLOBAL) or will require you to use the syntax: DSName.SubfieldName (*LOCAL). The *LOCAL option gives you the ability to create subfields without having to worry about name conflicts with other variables, or with fields on externally described files. The program default is *GLOBAL.

ExtDesc

Optional. **ExtDesc** specifies whether the data structure will be externally described. The default is *NO.

*NO – (Default). The data structure is not externally described.

*YES - The data structure is externally described. Note that to define the data structure externally, you must also have valid entries for the DbDesc and FileDesc parameters.

DbDesc

Optional. Database description. Specifies the database to use during compile-time to access the File Description.

*NONE – (Default).

Database Object declaration whose DBName specifies where the compiler will acquire the file description. Used in conjunction with **FileDesc**. Database Objects are defined with the DCLDB command.

Character expression - the Database Name to get the description from. Used in conjunction with **FileDesc**.

FileDesc

Optional. **FileDesc** specifies the name of the file in which to find the description of the data structure. This parameter is only used in conjunction with the **ExtDesc** *YES parameter to define the data structure as externally described.

*DSName is the default.

If (EXTDESC *YES) is specified, you need to specify the file name in which to find the data structure description. If a file name is not specified, then the data structure name becomes the same as the file name.

Prefix

Optional. Prefix is used to partially rename the fields in an externally-described file. The characters specified in the **Prefix Name** are a "String" and are prefixed to the names of all fields defined in all records of the file specified. In addition, you can optionally specify a numeric value to indicate the number of characters, if any, in the existing name to be Replaced. If the length (Len) is not specified, then the name string is attached to the **beginning** of the name.

If the "Length" is specified, it must be a numeric constant containing a value between 0 and 9, with no decimal places. For Example, the specification PREFIX(YE, 3) would change the field name 'YTDTOTAL' to 'YETOTAL'.

Specifying a value of zero is the same as not specifying "Length" at all.

Shared

Optional. **Shared** indicates whether or not this data structure is shared, or whether it is associated with a specific instance of a class or structure. The default is *NO. You can access a shared data structure by qualifying it either with the class or structure name, or with the data structure name of a specific instance of the class or structure.

You can use **Shared** only at module, namespace, or file level. This means you can declare shared data structures in a source file or inside a module, class, or structure, but not inside a procedure.

*NO (default) - specifies that this data structure is not shared, and is associated with a specific instance of a class or structure.

*YES - specifies that this data structure is shared, and is not associated with a specific instance of a class or structure. You can then access a shared variable by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.

Access

Optional. The type of access to the data structure. *PRIVATE is the default.

*PRIVATE (default) - access to the data structure is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the data structure is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the data structure is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the data structure is accessible from within the same project, but not from outside the project.

Attributes

Optional. Allows you to provide one or more attributes that apply to this class. To specify more than one attribute, simply separate them with commas.

Remarks

One or more subfields must be declared using the DCLDSFLD command after the **DCLDS** command for internally defined data structure.

To externally define the data structure, **ExtDesc** must be *YES, in addition to valid entries for the **DbDesc** and **FileDesc** parameters.

Example

```
// Declare a simple data structure with two subfields of type *CHAR.
DCLDS Name(DS1)
DCLDSFLD Name(DSFLD1) Type(*CHAR) Len(5)
DCLDSFLD Name(DSFLD2) Type(*CHAR) Len(10)

// Declare a 5 occurrence data structure with two subfields of type *CHAR.
DCLDS Name(DS2) DIM(5)
DCLDSFLD Name(DSFLD3) Type(*CHAR) Len(5)
DCLDSFLD Name(DSFLD4) Type(*CHAR) Len(10)

// Declare a simple data structure with two subfields of type *CHAR, which
// are unique to this data structure.
DCLDS Name(DS3) FLDScope(*LOCAL)
DCLDSFLD Name(DSFLD1) Type(*CHAR) Len(5)
DCLDSFLD Name(DSFLD2) Type(*CHAR) Len(10)
```



```
// Set up DS1 to contain the literal 'ABCDEFGHIJKLMNO'.
DSFLD1 = 'ABCDE'
DSFLD2 = 'FGHIJKLMNO'

// Set up occurrence 2 of DS2 to contain the literal '123451234567890'.
OCCUR DS(DS2) NEWINDEX(2)
DSFLD3 = '12345'
DSFLD4 = '1234567890'

// Set up occurrence 4 of DS2 to contain the literal 'AAAAABBBBBBBBBB'.
DS2[4].DSFLD3 = 'AAAAA'
DS2[4].DSFLD4 = 'BBBBBBBBBB'

// Set up DS3 to contain the literal 'IIIIJJJJJJJJ'. The FLDScope of this
// data structure is *LOCAL to avoid confusing DSFLD1 from DS1 and DSFLD1
// from DS3.
DS3.DSFLD1 = 'IIIII'
DS3.DSFLD2 = 'JJJJJJJJ'
```

Example Using Type

```
BeginClass Class1
  DCLDS myLocalDS
    DclDSfld myfld1 type(*char) len(10)
  DCLDS myLocalDS2 Dim(10)
    DclDSfld mylocalfld1 type(*char) len(2)
  DCLDS myTypeDS TYPE( fullyQualifiedTypeName )
  DCLDS myTypeDS2 TYPE( fullQualifiedTypeName2 ) Dim(10)
EndClass

BeginClass fullyQaulifiedTypeName
  DclFld mydsfld1 type(*char) len(10) access(*public)
  DclFld mydsfld2 type(*zoned) len(10,2) access(*public)
EndClass

BeginClass fullyQaulifiedTypeName2 Implements (ASNA.VisualRPG.Runtime.IDS )
  DclFld mydsfld1_2 type(*char) len(10) access(*public)
  DclFld mydsfld2_2 type(*zoned) len(10,2) access(*public)
EndClass
```

The Above Code

Generates a nested class myLocalDS_DS in Class1 for myLocalDS. This is a public class that can be used as a TYPE on other DS Declarations.

Generates a nested class myLocalDS2_MODS in Class1. myLocalDS2_MODS class contains a nested class myLocalDS2_DS. Also contains private members, an array of the _DS class to match the dimensions, an array to maintain the current occurrence and a rank field.

Generates two properties mydsfld1 and mydsfld2 that get and set fullyQualifiedTypeName.mydsfld1 and fullyQualifiedTypeName.mydsfld2.

Generates a nested class myTypeDS2_MODS for myTypeDS2 in Class1.

See Also

DCLDS | DCLALIASGROUP | DCLOVERLAYGROUP | MOVEARR | Time, Date, and Timestamp Variables | Data Structure Changes

DCLDSFLD - Declare a Subfield for a Data Structure

Declares a subfield for a data structure.

```
DCLDSFLD
Name (name)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
      *FLOAT4 |
      *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *TIME | *TIMESTAMP | *ZONED)
Len (Integer Literal, Integer Literal)
Dim (Integer Literal, Integer Literal, ...)
TimFmt *PGMDFT | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
      *YMD)
Like (Field name)
Overlay (Overlaid name {, pos})
```

Parts

Name

Required. The name of the data structure subfield.

Type

Optional. **Type** can be any of Special Value (i.e., *CHAR, *BINARY, etc.) or it can be a class name. If **Type** is *BOOLEAN or *IND, you can use *ON, *TRUE, *ON and *FALSE figurative constants to indicate true and false and "1" and "0", respectively.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer given and *ZONED is assumed if there are two integers.

Len

Optional. **Len** specifies the length of the field. If **Len** is decimal e.g. Len (3,1) *ZONED is assigned as the default. If **Len** is character e.g. Len (3), *CHAR is assigned as the default.

If **Type** is packed, then the length is not the number of bytes the packed field fits into, but rather the logical digit length of the field. For Example, if you want a 7 digit packed with 2 decimals, code Len (7,2).

Dim

Optional. When **Dim** is specified, the **DCLDSFLD** is treated as an array with all the properties associated with a DCLARRAY statement.

TimFmt

Optional. **TimFmt** specifies the format for time. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Like

Optional. **Like** must be the name of a field already defined in the class. **Type** cannot be a named constant or a literal. In this case, the field will be defined based on the attributes of another field. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the field being defined. The field length entry is allowed only for field definitions requiring a length.

Overlay

Optional. For **Type** *CHAR and *ZONED only. The **OVERLAY** keyword overlays the storage of one subfield with that of another subfield, or with that of the data structure itself.

This subfield will overlay the storage specified by the *overlaid name* parameter at the position specified by the *pos* parameter. If *pos* is not specified, it defaults to 1.

Note The *pos* parameter is in units of bytes, regardless of the types of the subfields.

The following rules apply to keyword **OVERLAY**:

The *overlaid name* parameter must be the name of a subfield defined previously in the current data structure, or the name of the current data structure.

The *pos* parameter (if specified) must be a value greater than 0 with no decimal positions. It can be a numeric literal, a built-in function returning a numeric value, or a numeric constant. If *pos* is a named constant, it must be defined prior to this specification.

If the *overlaid name* parameter is a subfield, the subfield being defined must be contained completely within the subfield specified by the *overlaid name* parameter.

Alignment of subfields defined using the **OVERLAY** keyword must be done manually. If they are not correctly aligned, a warning message is issued.

If the subfield specified by *overlaid name* is an array, the **OVERLAY** keyword applies to each element of the array.

That is, the field being defined is defined as an array with the same number of elements. The first element of this array overlays the first element of the overlaid array, the second element of this array overlays the second element of the overlaid array, and so on. No array keywords may be specified for the subfield with the **OVERLAY** keyword in this situation.

If the subfield specified by *overlaid name* is an array and its element length is longer than the length of the subfield being defined, the array elements of the subfield being defined are not stored contiguously.

If a subfield with overlaying subfields is not otherwise defined, the subfield is implicitly defined as follows:

The start position is the first available position in the data structure.

The length is the minimum length that can contain all overlaying subfields. If the subfield is defined as an array, the length will be increased to ensure proper alignment of all overlaying subfields.

Example

```
// Declare a simple data structure with two subfields with one overlay of
the data structure
DCLDS Name(DS1)
DCLDSFLD Name(DSFLD1)      Type(*CHAR) Len(5)
DCLDSFLD Name(DSFLD2)      Type(*CHAR) Len(10)
DCLDSFLD Name(OVERDSFLD) Type(*CHAR) Len(15) Overlay(DS1)

// Declare a simple data structure with two subfields with one overlay of
the 1st five characters of DSFLD2
DCLDS Name(DS1)
DCLDSFLD Name(DSFLD1)      Type(*CHAR) Len(5)
DCLDSFLD Name(DSFLD2)      Type(*CHAR) Len(10)
DCLDSFLD Name(OVERDSFLD) Type(*CHAR) Len(5) Overlay(DSFLD2,1)
```

See Also

DCLDS | DCLALIASGROUP | DCLOVERLAYGROUP | MOVEARR | Time, Date, and Timestamp variables

DCLENUMFLD

Declares the members of an Enumeration (BEGENUM/ENDENUM).

```
DCLENUMFLD
Name (field name)
Value (constant expression)
```

Parts

Name

Required. The name of the enumeration definition.

Value

Optional. If this is the first **DclEnumFld** in the Enum declaration, the default value is 0. Otherwise, the default value is the value of the previous **DclEnumFld** plus 1.

Remarks

Enums are used to describe enumeration constants. Their only members are **DclEnumFld** fields.

Example

```
BegEnum myEnum basetype( *byte )

    DclEnumFld q4 value(4)      // <--- the default value is 0

    DclEnumFld q5              // <--- this will be 5, which is 1 plus the
previous one

EndEnum
```

See Also

BEGENUM | ENDENUM | Enumeration Definition

DCLEVENT - Declare an Event

DCLEVENT creates an event for an ActiveX Control or ActiveX DLL.

```
DCLEVENT
Name (Character Expression)
Type (Delegate type)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
Shared (*NO | *YES)
Modifier (*NONE | *MUSTOVERRIDE | *NOTOVERRIDABLE | *OVERRIDABLE |
*OVERRIDES)
Implements (Interface name list)
Attributes (Attribute 1, Attribute 2, ...)
```

Parts

Name

Required. The name of the event.

Type

Optional. A delegate type, i.e. a type that derives from (or extends) `System.Multicast.Delegate`.

Access

Optional. The type of access to the event. *PRIVATE is the default.

*PRIVATE (default) - access to the event is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the event is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the event is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the event is accessible from within the same project, but not from outside the project.

Shared

Optional. **Shared** specifies whether or not this event is to be shared. *NO is the default.

*NO (default) - this event will not be shared.

*YES - this event will be shared.

Modifier

Optional. **Modifier** controls how properties and methods are overridden in derived classes. *NONE is the default.

*NONE (default) - specifies that none of the override options are available for properties and methods.

*MUSTOVERRIDE - specifies that a property or procedure in a base class must be overridden in a derived class before it can be used.

*NOTOVERRIDEABLE - specifies that a property or procedure cannot be overridden in a derived class.

*OVERRIDEABLE - specifies that a property or method can be overridden in a derived class.

*OVERRIDES - specifies that a property or method overrides a member inherited from a base class.

Implements

Optional. Indicates that this class implements the members of an interface. An interface defines a set of properties, events and methods, but does not provide an implementation of them, just a definition. You must implement every member of every interface you specify in this keyword.

Attributes

Optional. Allows you to provide one or more attributes that apply to this class. To specify more than one attribute, simply separate them with commas.

Remarks

Use **DCLEVENT** when creating an event for an ActiveX control or ActiveX DLL. It can be performed anywhere within calculations.

Parameters may be declared by using **DCLSRPARM** commands right after a **DCLEVENT**. See **DCLSRPARM** for more information.

See Also

Custom Attributes | **DCLSRPARM**

DCLFILECONT

Declares an externally described disk file for processing in a program.

```
DCLFILECONT
Db (Database Object)
File (*FILEDESC | Character Expression)
DbDesc (Database Object | Character Expression)
FileDesc (*NAME | Character Expression)
XmlDesc (Character Expression)
RnmFmt (*NONE | Old Format Name, New Format, ...)
ShareType (*FILE | *EXCLUSIVE | *EXCLUSIVEREAD | *SHAREUPDATE |
             *SHARENOUPDATE | *SHAREREAD | Character Expression)
ChkFmtId (*YES | *NO | *SAME | *IGNORE | *MAP | Indicator Expression)
Prefix (prefix name, [Length])
Mbr (*FIRST | *FILE | Character Expression)
NetBlockFactor (*CALC | *NONE | Number of Records to Block)
CacheWrites (*NO | *YES | Indicator Expression)
Commit (*NO | *YES | Boolean/Indicator Expression)
QrySelect (*NONE | Character Expression)
QryKeyFlds (*NONE | Character Expression)
QryFileName (*Name | Character Expression)
WaitRec (*FILE | *IMMEDIATE | *NOMAX | Time to wait)
Random (*NO | *YES)
```

Parts

Db

Required. **Db** specifies the Database Object declaration whose **DBName** specifies where the compiler will acquire the file description. **Db** specifies the Database Connection to use when opening the file at run-time. Database Objects are defined with the DCLDB command.

File

Optional. **File** specifies the Library and **File** to use when opening the file at run-time.

If the **File** parameter is not given, the same file that was used to retrieve the description will be opened at run-time.

If a file name is given on the **File** parameter, that file will be accessed and must have the same description as the file accessed in the **Name** parameter, otherwise unpredictable results may occur. The **File** parameter accepts a variable as a parameter, so that the file accessed can be overridden at run-time.

DbDesc

Optional. Database description. Specifies the database to use during compile-time to access the File Description.

Database Object declaration whose **DBName** specifies where the compiler will acquire the file description. Database Objects are defined with the DCLDB command.

Character expression - the Database Name to get the description from. Used in conjunction with **FileDesc**.

FileDesc

Optional. File description. Takes the description using the **Name** parameter to access a local Acceler8DB file on your PC. Use this parameter if the name of an Acceler8DB file uses characters not valid within RPG.

If you want to retrieve the description from a specific library or directory, use the FileDesc parameter to point to the particular file you want. This parameter must be a character literal, so the file description can be accessed at compile time.

*NAME (default) - takes the description using the Name parameter to access a local Acceler8DB file on your PC.

Character expression - the description of the Acceler8DB file.

XmlDesc

Optional. Takes the description using the Name parameter to access a local XML file on your PC. This parameter must be a character literal, so the file description can be accessed at compile time.

RnmFmt

Optional. Rename Format. Renames the old format to a new format, to avoid clashes with duplicate format names. *NONE is the default.

For **single format files**, you can specify **RnmFmt (New Format Name)** - which will automatically rename the only format in the file to the **New Format Name**. If you specify the syntax "Old format Name, New Format Name for single format files, the Visual RPG compiler will ignore the **Old Format Name**.

ShareType

Optional. **ShareType** defines values on how a file will be shared. The default is *FILE.

*File (default) - Maintains the same share type specified when the file was created.

*Exclusive - Open for update, exclusive lock.

*ExclusiveRead - Open for update, other jobs may read.

*ShareUpdate - Open for update, other jobs may read or update.

*ShareNoUpdate - Open for read, other jobs may read.

*ShareRead - Open for read, other jobs may read or update.

ChkFmtId

Optional. **ChkFmtId** checks if the format ID of the file at run-time compares to the format ID of the file at compile-time. Visual RPG stores the format of file(s) when a program is first compiled. At run-time, Visual RPG compares the format ID that was stored during the compilation with the current format ID. The comparing of format ID's can be especially useful when checking if new fields have been added to a file used by a program that has been compiled months ago.

***SAME** is the default, and indicates that the run-time and compile-time format ID's **will be compared** to verify that they are the same. *(This is exactly equivalent to *YES in previous releases).*

***IGNORE** indicates that the run-time and compile-time format ID's **will not be compared** to verify that they are the same. Visual RPG will use the compile-time format ID. *(This is exactly equivalent to *NO in previous releases). Note Please be extremely careful in using *IGNORE, as it has the risk of the corruption of data files and misbehavior of your application!!.*

***MAP** indicates that the run-time and compile-time format ID's **will be compared**, and Visual RPG will accommodate non-key fields added and changed **regardless** of where they are defined in the new record format. When the modified file is opened, Visual RPG will **rebuild, or map** its compiled record format at run-time to equal that of the new record format. Restrictions apply to non-key fields depending upon the mode of file processing and the changes made to the file. The rule of thumb is that Visual RPG will process the file provided there is **no loss of data**. Any change in the count, order or type of Key fields is not allowed, except that a numeric key may be changed from packed to zoned and vice versa, as long as the precision remains unchanged.

Prefix

Optional. **Prefix** allows the specification of a string which is to be prefixed to the field names of an externally described record format. The characters specified as (prefix name) are prefixed of all fields defined in all record formats of the file.

You can also optionally specify a numeric value (length) to indicate the number of leading characters in the existing name to be replaced. If a length is not specified or is zero, the string is prepended to the name.

The total Length of a name after applying the prefix must not exceed the maximum length of an RPG field name.

If a Length is specified, it must be a numeric constant containing a value between 0 and 13 with no decimal places. Specifying a value of zero is the same as not specifying a length at all.

The Length parameter must not be greater than or equal to number of characters in the name to be prefixed. That is, the resulting name cannot be the same as the prefix string after the prefix is applied.

Mbr

Optional. **Mbr** specifies the file member to use when opening the file at runtime. The **Mbr** parameter allows you to specify the name of the member you want to access or *FIRST. *FIRST is the default.

*FIRST (default) - will access the member with the oldest creation date.

*FILE - accesses the member name with the same name as the file name.

NetBlockFactor

Optional. **NetBlockFactor** determines if and optionally how many database records will be blocked across the Network for better performance when reading or writing groups of records. Network blocking significantly reduces the network burden of transferring records between server and client machines, and this is demonstrated by the substantial increase of data throughput.

Unless **NetBlockFactor** is *NONE, network blocking is employed for input and output files as described below.

INPUT - The file is declared with Type(*INPUT) and AddRec(*NO). When an input file is "blocked", the buffered data resides on the client PC. This allows successive READ and READE statements to execute quickly. Be aware that should records be updated in other definitions of the file, those changed are not correspondingly updated in the buffer.

OUTPUT - The disk file is declared with Type(*OUTPUT) and Cachewrites(*YES). When an output file is "blocked", the buffered data resides on the client PC until the block has been filled or the file is closed. Then the block is sent to the server in a single transaction. Be aware that I/O errors at the server are returned only during processing of the block. In other words, if the 33rd record in a block of 100 records had a duplicate key, your application would not be aware of it until after writing the 100th record. Therefore, the use of network blocking for output files should be employed only when your application can be restarted.

I/O operations that will reset the buffer are SETLL, SETGT, SETRANGE, CHAIN (READRAND) and READRANGE. *CALC - The system will calculate the number of records to block. The blocking factor is 6000 divided by the file's record length. If the record length is greater than 3000 (1/2 the block size),

*CALC (default) will compute a record blocking Factor of 1 (one).

*NONE - No blocking will occur on the file. This is the default when the Type is other than *INPUT.

Record Count - The maximum number of records to include in the block. Note, the number of records in the block for SETRANGE with READ operations, and for READRANGE operations will be less than the record count if the set of records in the range is smaller than the maximum block size. Because of this dynamic blocking, the use of SETRANGE and READRANGE can realize even better performance than SETLL and READE, for Example.

Note Record blocking requires additional memory to work. Blocking reserves memory for the complete block of each blocked file. For Example, a record length of 2560 and a blocking factor of 100 will require an additional 256K of memory.

CacheWrites

Optional. **CacheWrites** specifies whether data is written to disk immediately after an operation, or whether data is saved in a buffer, then written to disk when the Operating System has time. *NO is the default.

*NO (default) indicates that data will be written to disk at the time the operation is performed. Note that this option is slower than saving the data to a buffer.

*YES indicates that data operations will be saved to a buffer, and written to disk when the Operating System has time. Note that this option is faster. However, if the system fails with data in the buffer that has not been written to disk, the information will be lost.

When a disk file is declared with Type(*OUTPUT), Cachewrites(*YES) and NetblockFactor (*CALC or value), output records are buffered at the client before being sent as a block to the server database rather than being transmitted one record at a time.

Commit

Optional. **Commit** allows you to control if commitment control will be enabled for this file. You can also change at runtime by specifying a Boolean/Indicator expression. The default is *NO.

The parameter must be set prior to opening the file. If the file is opened explicitly using the OPEN operation in the calculation specifications, the parameter can be set prior to the OPEN operation.

Use the COMMIT and ROLLBACK operation codes to group changes to this file and other files currently under commitment control, so that all changes happen together, or do not happen at all.

*NO (default) - indicates that commitment control is not enabled.

*YES - indicates that commitment control has been enabled.

Boolean/Indicator Expression - allows commitment control to be specified or changed at runtime.

QrySelect

Optional. **QrySelect** specifies the selection expression used to determine which records are to be made available for processing. The select expression syntax is the same as for a Select/Omit expression in Acceler8DB.

QryKeyFids can be used in conjunction with QrySelect parameters to specify ordering of the selected records. Use QryFileName to specify the name of the temporary query file created.

When using QrySelect, please Note

*NONE (default) - there is no selection expression available for processing.

The parameter can be a character literal or a character variable.

If there are any errors in the character expression, they will not be caught until the file is opened at run-time.

Selection expressions are evaluated during file open time. Therefore, a runtime error will be generated during the opening of the file when a selection expression error is encountered.

QryKeyFlds

Optional. The **QryKeyFlds** parameter is used to specify the name of one or more key fields used to arrange the query records. Each key field is composed of a field name followed by optional attributes that determine the sorting order (ascending descending, or absolute value for numeric fields).

*FILE is the default, which specifies that the original keys in the file should be used.

*NONE specifies that the file has no keys.

Character Expression contains a string containing a keyfieldname and it's attributes, as described in the Syntax section below.

When specifying **QRYKeyFlds** with key fields that do not match the original file's key definition, you must also specify Random(*NO). In addition, in order to process the file in the sequence specified by the QRYKeyFlds parameter, you must specify ORG (*Indexed).

Valid input operations for RANDOM(*NO) files are limited to: READ, READP and SETLL when used with the new special values *START and *END.

Syntax: The syntax for the **QryKeyFlds** parameter is: (KeyFieldName Attributes ...)

KeyFieldName Must be a valid field name in the file.

Attributes Consist of the following: *ASCEND, *DESCEND, or *ABSVAL. You can have as many attributes as you want after a field name as long as they conform to the rules below.

... Indicates that you can repeat the sequence for as many key fields as you want.

When using **QryKeyFlds**, please Note

Each key field name and attribute must be separated by at least one blank.

*ASCEND and *DESCEND cannot be applied to the same field.

*ABSVAL only applies to numeric fields.

The parameter can be a character literal or a character variable.

If there are any errors in the key fields expression, they will not be caught until the file is opened at run-time.

Use QrySelect in conjunction with **QryKeyFlds** to specify the selection criteria indicating which records to select from a file. Use QryFileName to specify the name of the temporary query file created. Selection expressions are evaluated during file open time. Therefore, a runtime error will be generated during the opening of the file when a selection expression error is encountered. Example: QryKeyFlds ('CustNo *ASCEND ORDERDATE *DESCEND LINENO TOTAL *ABSVAL *DESCEND') For more information on QryKeyFlds Examples and using open query files in Visual RPG, refer to the following link: [Using Open Query File in Visual RPG](#).

QryFileName

Optional. QryFileName specifies the name of the query file created for the QrySelect and QryKeyFids parameters. If *Name is specified, which is the default, the file name given on the F-spec or DCLFILECONT command is used.

If the database access with this file is an AS/400 database, then the QryFileName is used both on the OVRDBF and the OPNQRYF commands used when opening the file.

For Acceler8DB databases, a temporary logical file is created in QTemp with the QryFileName as its name. You can use the special name '*UNIQUE' (don't forget the quotes) to guarantee that there will be no conflicts with other files residing in QTemp. You will receive errors if there are duplicate file names in QTEMP.

It is recommended the use of QryFileName('*UNIQUE') when working with Web Applications. Web Applications run under one user, the user that is specified in Acceler8DB Work with Database Names dialogue. Web Applications usually store Runtime Errors and MsgBoxes in \$\$Visual RPGErr.TXT.

*Name (default) - the file name given on the DCLFILECONT's Name parameter is used.

For more information on QryKeyFids and using open query files in Visual RPG, refer to the following link: [Using Open Query File](#).

WaitRec

Optional. WaitRec specifies the waiting period for a record. *File is the default.

*File (default) - indicates that the properties of the file are used to determine how long it waits for the record.

*Immediate - it won't wait for a record.

*NoMax - it will wait indefinitely.

Random

Optional. Random specifies whether random database processing using Keys and RRNs is allowed.

*YES is the default, which determines the random database processing using keys and RRNs is allowed.

*YES - (default) specifies that random database processing using Keys and RRNs on op codes such as SETLL, CHAIN, etc., is allowed.

*NO - specifies that random database processing using Keys and RRNs is not allowed. This option is commonly used when the Keys look different at runtime from the compile time. The common scenario is when using QryKeyFids and under this scenario *NO option must be used to avoid a runtime exception.

The only I/O operations allowed are:

Input - Read, ReadP and Setll with *Start or *End as the Keys. Any other operations throw compiler error - "Random Access Operation Not Allowed For File". The results of Read and ReadP depend on the ORG option and if *INDEXED read them sequentially based on the Key of the file and *ARRIVAL reads in arrival sequence in the file.

Output - Write and Update.

See Also

CLOSE | COMMIT | DCLDB | DCLDS | DCLDSFLD | DCLFLD | LOOKUP | DCLPRINTFILE | OPEN | ROLLBACK

DCLFLD - Declare a Field

DCLFLD declares a working field in the current scope. **DCLFLD** also creates a property if the **Access** parameter is ***PUBLIC**.

```
DCLFLD
Name (Field name)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *OBJECT | *ONECHAR | *PACKED | *STRING | *TIME | *TIMESTAMP | *ZONED
      | Field, Array, Array Element, table name, String Literal)
Len (Integer Literal, Integer Literal)
TimFmt (*PGMDFT | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
      *YMD)
Shared (*NO | *YES)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
Static (*NO | *YES)
WithEvents (*NO | *YES)
New (*NOOBJECT | *DFT | Expression List)
Inz (Constant | Constant Expression)
Attributes (Attribute 1, Attribute 2, ...)
Like (Field name)
```

Parts

Name

Required. **Name** is the name of the field to be declared. Name cannot be a named constant or a literal.

Local variables inside a subroutine can have the same name as global variables or other local variables in other subroutines, and they will not clash. Storage will be reserved locally for each instance of a local variable.

Type

Optional. **Type** can be any of Special Value (i.e., *CHAR, *BINARY, etc.) or it can be a class name. If **Type** is *BOOLEAN, you can use the *TRUE or *ON and *FALSE or *OFF special values to indicate **True** or **False**.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer given and *ZONED is assumed if there are two integers.

Len

Optional. **Len** specifies the length of the field. The length of a *BOOLEAN, *FORM, *IND, *OBJECT or *STRING type is implicit. The length of a *INTEGER Type is 2- or 4-bytes. The length of a *FLOAT Type is 4- or 8-bytes.

TimFmt

Optional. **TimFmt** specifies the format for time, date, and timestamp variables. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

You may encounter a date numeric variable from a system function or OCX that has a format similar to dddddd.nnnnnnnnnn, where values to the left of the decimal represent date information while values to the right of the decimal represent time. For Example 36167.383645833 is 09:12:27 on 07 January, 1999. To isolate the date and time, simply assign this value to an OLE date variable and OLE time variable, respectively.

Shared

Optional. **Shared** indicates whether or not this field is shared, or whether it is associated with a specific instance of a class or structure. The default is *NO. You can access a shared field by qualifying it either with the class or structure name, or with the field name of a specific instance of the class or structure.

You can use **Shared** only at module, namespace, or file level. This means you can declare shared fields in a source file or inside a module, class, or structure, but not inside a procedure.

*NO (default) - specifies that this field is not shared, and is associated with a specific instance of a class or structure.

*YES - specifies that this field is shared, and is not associated with a specific instance of a class or structure. You can then access a shared variable by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.

Access

Optional. The type of access to the field. *PRIVATE is the default.

*PRIVATE (default) - access to the field is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the field is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the field is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the field is accessible from within the same project, but not from outside the project.

Static

Optional. **Static** indicates that the declared field will remain in existence and retain its latest values after termination of the procedure in which they are declared. The default is *NO.

You can use Static only at procedure level. This means you can declare static fields inside a procedure, but not at a class level.

*NO (default) - the declared field will not remain in existence and will not retain its latest values after termination of the procedure in which it are declared.

*YES - the declared field will remain in existence and retain its latest values after termination of the procedure in which its are declared.

WithEvents

Optional. **WithEvents** allows you to reference and access events that have already been defined in another object. *NO is the default.

*NO (default) specifies that even though the object specified may have events, that this program will not be accessing those events.

*YES is only allowed on object types that are referenced using Type (Library name.Object name). If Type (*OBJECT) is used, a compiler error will occur. Note that the Library name is actually the project name, and the Object name is the name specified with the Name property. The Type parameter to use can also be found by using Object Browser on the object to reference.

New

Optional. Use the **New** parameter when you want to specify a value to a field that is of a **class** type (like `System.Text.StringBuilder`). You specify either a list of expressions or `*DFT` to use the parameterless constructor of the class. `*NOOBJECT` is the default when the **New** parameter is not specified, indicating that this field will contain its default initial value, which for objects is `*Nothing`.

`*NOOBJECT` (default) - the default when the **New** parameter is not specified, indicating that this field contains the default initial value; which for objects is `*Nothing`.

`*DFT` - constructs an object with a parameterless constructor of the class.

Expression list - Specify a list of expressions for the default class value.

Example: `DclFld Address type(System.Text.StringBuilder) New(*Dft)`

// Constructs a `StringBuilder` object with the parameterless constructor. See the help for `StringBuilder`.

Example: `DclFld Address type(System.Text.StringBuilder) New("This is the initial value")`

// Constructs a `StringBuilder` object with initial value "This is the initial value". See the help for `StringBuilder`.

nz

Optional. **Inz** allows you to specify a default value when a field is of a value type (like `*ZONED`, `*Integer`, etc). The value should be a constant (or a constant expression, such as `%DECPos ( )`), of the appropriate type.

Attributes

Optional. Allows you to provide one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

Attributes use required, and optional arguments. The required parameters are provided in positional fashion. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For Example, the following code calls the **Developer** constructor of an attribute with two values: `BegClass MyClass Attributes(Developer("Joe Cool", 17))`

You can also use named arguments, which are optional, to directly set the value of properties and fields of the attributes. You specify a named argument by appending `:=` to the name of the argument, followed by the value being specified. Unlike positional arguments, named arguments can be specified in any order. For Example, the following code sets the value `*True` for the **Reviewed** property of the **Developer** Attribute, and **False** for the **Keep** field:

`BegClass MyClass Attributes(Developer("Joe Cool", 17, Reviewed:=*True)`

Note You must use positional arguments for arguments that are used with the attribute's constructor. Named arguments can only be used to set the values of fields and properties of attribute classes.

Like

Optional. **Like** must be the name of a field already defined in the class. **Type** cannot be a named constant or a literal. In this case, the field will be defined based on the attributes of another field. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the field being defined. The field length entry is allowed only for field definitions requiring a length.

Remarks

Note You should code your declaration statements at the beginning of the code region of the class or routine to which it applies (i.e., `BegClass`, `BegConstructor`, `BegFunc`, `BegGet`, `BegSet`, `BegSr`). If a declaration statement initializes the value of a variable via `INZ` or `New`, it will be initialized before any other statement makes reference to it.

Variables declared with **DCLFLD** are available to all code within the region containing the **DCLFLD** code. If they are declared in a program, class, or structure, but outside any procedure, they can be accessed from anywhere within that program, class, or structure. To specify their accessibility in more detail, include the **Access** parameter options of `*PRIVATE`, `*PUBLIC`, `*PROTECTED` and `*INTERNAL`.

Variables declared inside a procedure or a block are accessible only from within that procedure or block. You cannot use any of the accessibility keywords for such variables.

DCLFLD can declare the data type of a variable and initialize its contents. The declaration statements in the following Example declare an **Integer** variable, a **Boolean** variable, and an object variable. These are initialized, respectively, to 20, **True**, and a newly created instance of the **Label** class in the **System.Windows.Forms** namespace.

```
DCLFLD Name (Quantity) Type (*Integer) Len (2) Inz (20)
DCLFLD Name (FirstTry) Type (*Boolean) Inz (*True)
DCLFLD Name (MyLabel) New (System.Windows.Forms.Label() )
```

If you do not specify an initialization value for a variable, Visual RPG initializes it to the default value for its data type. The default initialization values are as follows:

0 for all numeric types (including `*Byte`).

Binary 0 for `*Char`.

***Nothing** for all reference types (including `*Object`, `*String`, and all arrays). Don't forget that attempting to reference these types before you have set them to a value will result in a "object not instanced" runtime error.

`*False` for `Boolean`.

12:00 AM of January 1 of the year 1 for `*Date`.

Each element of a structure or array is initialized as if it were a separate variable.

You can initialize the values of an array by surrounding the initialization values with brackets (`[]`). The following Example creates an array with four elements:

```
DCLFLD Name (Array1) Type (*Integer) Inz [0, 1, 2, 3]
```

A variable with ***Public** access can be either early bound or late bound. Access to `Protected`, `Internal`, and `Private` variables cannot be late bound. This means you cannot access members on a late-bound object variable unless it is declared as ***Public**. If you attempt to do this, the compiler does not generate an error, but an error occurs at run-time. A static variable has a longer lifetime than that of the procedure in which it is declared. The boundaries of the variable's lifetime depend upon where the procedure is declared and whether or not it is **Shared**.

If the procedure is declared in a module, its static variables are initialized the first time the procedure is called, and they continue to exist until your program terminates execution.

If the procedure is declared as **Shared** within a class or structure, its static variables are initialized the first time the procedure is called either on a specific instance or on the class or structure itself. They remain in existence until the program terminates.

If the procedure is declared within a class or structure and is not **Shared**, its static variables are associated with each individual instance of the class or structure. They are initialized the first time the procedure is called in that instance, and they continue to exist until that instance is released.

There are 3 ways to use **DCLFLD** to create objects in Visual RPG:

```
DCLFLD myWord1 Type(Word.Application) // Type is LibraryName.ObjectName
DCLFLD myWord2 Type('{000209FF-0000-0000-C000-000000000046}') // Type is
ClassId String Literal
DCLFLD myWord3 Type('Word.Application.8') //Type is Prog ID
```

The First way uses the **LibraryName.Objectname**. For this form to work, the Object has to be referenced from the IDE. This means the user has to add an **explicit** reference to this Library by using the References Dialog box. If not, this line would cause a compile error.

The Second and third way uses the ClassID string and the ProgId string respectively. If one knows the ClassID or the ProgId of the object he wants to use, he can use these forms. The object doesn't have to be referenced by using the References dialog to use an object this way.

If Microsoft Word is installed on your machine, then you could add a Reference to it from the References dialog and use Form1.

If by looking at the Object Browser, or another tool, you know the ClassID or ProgId of Word, then you could use Examples 2 or 3.

In the case of Form2 and Form3, if the ClassId or ProgId was not resolved (i.e., the compiler could not find it in the registry) then the Property names and Method names will be resolved at run-time. Otherwise, they will be resolved at compile-time and error messages will be given for errors in Property names and methods.

Example

```
// Declare a character field, 10 bytes long, and move data into it.
DCLFLD Name(CHAR10) Type(*CHAR) Len(10)
CHAR10 = 'ABCDEFGHJIJ'

// Declare a packed numeric field, 7 long, 2 decimals, and initialize it to
12345.67.
DCLFLD Name(PACK72) Type(*PACKED) Len(7,2)
PACK72 = 12345.67

// Declare a zoned numeric field, 7 long, 2 decimals, and initialize it to
76543.21.
DCLFLD Name(ZONE72) Type(*ZONED) Len(7,2)
ZONE72 = 76543.21

// Declare a binary data field, 11 long, 2 decimals, and assign a value to
it with an expression.
DCLFLD Name(BIN112) Type(*BINARY) Len(9,2)

BIN92 = PACK72 + ZONE72
```

See Also

Custom Attributes | DCLKFLD | DCLALIAS

DCLKFLD - Declare a Key Field

Indicates that a field is part of a search argument identified by a DCLKLIST name.

```
DCLKFLD
Name (KeyField name)
```

Parts

Name

Required. Name must contain the field name that is to be part of the search argument, and cannot contain an array or table name.

Each **DCLKFLD** must correspond in length, data type, and decimal position with the corresponding field in the composite key of the record or file.

The **DCLKFLD** order specified in the **DCLKLIST** determines which **DCLKFLD** is associated with a field in the composite key. For Example, the first **DCLKFLD** following a **DCLKLIST** is associated with the left-most (or high-order) field of the composite key.

See Also

DCLKLIST

DCLKLIST - Declare a Key List

Gives a name to a list of DCLKFLDS. This list can be used to retrieve records from files that have a composite key.

```
DCLKLIST
Name (KeyField name)
```

Parts

Name

Required. Name must contain a unique name.

A **DCLKLIST** must be followed immediately by at least one **DCLKFLD**. A **DCLKLIST** and its associated DCLKFLD fields must appear in the global declaration part of a class.

A **DCLKLIST** name can be specified as a search argument only for externally described files.

You must specify a **DCLKLIST** with multiple **DCLKFLDS** if a search argument contains more than one field.

A **DCLKLIST** can appear as a factor of the CHAIN, DELETE, READE, READPE, SETGT, or SETLL operations.

See Also

CHAIN | DCLKFLD | DELETE | READE | READPE | SETGT | SETLL

DCLMEMORYFILE - Declare Memory File as ADO.NET Dataset

Declares a memory file that results in an ADO.NET Dataset object. The file can be created and accessed similar to an externally described database file even though it is cached in memory. Currently, the memory file description is externally described as an iSeries/400 or DataGate Windows database file.

```
DCLMEMORYFILE
Name (MemoryFile name)
Org (*ARRIVAL )
DbDesc (Database Object | Character Expression)
FileDesc (*NAME | Character Expression)
XmlDesc (Character Expression)
AddRec (*NO | *YES)
ImpOpen (*YES | *NO)
RnmFmt (*NONE | Old Format name, New Format name, ...)
Subfile(*NONE | Format name, RRN name, ...)
Prefix (prefix name, [Length])
Random (*NO | *YES)
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
Shared (*NO | *YES)
```

Parts

Name

Required. Name of a memory file. Must be a valid Visual RPG name, such as:

The name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

- a valid program id registered on your system at compile time.

- a valid class id registered on your system at compile time.

- a valid interface name registered on your system at compile time.

A name of the form **Library.Object**, where Library is the name of one of the libraries selected in the new object libraries dialog, and Object is valid object name within that library.

Note: Using the MemoryFile *name*, you can then get to its **fields** and **properties**. Please note that methods are reserved for use by the ASNA Visual RPG compiler, so use the corresponding Visual RPG command if applicable.

Org

Optional. Specifies if the file is a Sequential (*Arrival) file. The default is *ARRIVAL.

*ARRIVAL (default) - the file is a Sequential file.

DbDesc

Optional. Database description. Specifies the database to use during compile-time to access the File Description.

Database Object declaration whose *DbName* specifies where the compiler will acquire the file description. Database Objects are defined with the DCLDB command.

Character expression - the Database Name to get the description from. Used in conjunction with *FileDesc*.

FileDesc

Optional. File description. Takes the description of the named file as the bases to layout the memory file. Used in conjunction with **DbDesc**.

This parameter must be a character literal, so the file description can be accessed at compile time.

*NAME (default) - uses the parameter **Name** to find the file to obtain the description of the file's layout.

Character expression - the description of the database file.

XmlDesc

Optional. This parameter can be either an XML string containing the **schema layout** for the memory file or alternatively it can have a **local PC file** name containing the schema. This parameter must be a character literal, so the file description can be accessed at compile time.

If the character literal begins with a '<', then the string is interpreted to be the **XML Schema**, otherwise it is interpreted to be a **reference** to a text file containing the schema.

When referencing a text file, you can provide either an **absolute** or a **relative** path to the file.

An absolute path uses the standard Uniform Resource Identifier syntax (like the one used to address web pages).

A relative path starts with a period followed by the list of folders leading to the file location. The 'current directory' referenced by the period is considered to be the same as the one where the source file containing the DclMemoryFile resides.

Examples:

```
DclMemoryFile Rel . . . XmlDesc("./mySchema.xsd")
DclMemoryFile Abs . . . XmlDesc("file:///C:/Def/mySchema.xsd")
DclMemoryFile Ftp . . .
  XmlDesc("ftp://ftp.myCompany.com/Def/mySchema.xsd")
```

AddRec

Optional. You must use the AddRec (*YES) option if you wish to add a record to the file.

ImpOpen

Optional. Implicit Open. Specifies whether the file will automatically be opened when the program is first run. If the implicit open is unsuccessful, a run-time error will occur. The default is *YES.

*YES (default) - the file will automatically be opened when the program is first run. If the implicit open is unsuccessful, a run-time error will occur.

*NO - the file will not automatically be opened when the program is first run.

RnmFmt

Optional. Rename Format. Renames the old format to a new format to avoid clashes with duplicate format names. *NONE is the default.

For **single format files**, you can specify **RNMFMF (New Format name)** - which will automatically rename the only format in the file to the **New Format name**. If you specify the syntax "Old format name, New Format name for single format files, the Visual RPG compiler will ignore the **Old Format name.Prefix**.

Subfile

Optional. The **Subfile** keyword is used to internally define the subfiles that are specified in the memory file. The *Format name* identifies the name of the subfile record format and the *RRN name* identifies the name of the numeric field to be associated with the subfile to communicate the relative-record-number. Each subfile *format name* with its corresponding *RRN name*, must be listed in the single **Subfile** keyword.

The relative-record number of any record retrieved by a **READC** or **CHAIN** operation is placed into the field identified by the RRN parameter. This field is also used to specify the record number used in **WRITE** operations to the subfile. The field name specified as the rrn field parameter must be defined as numeric with zero decimal positions.

Prefix

Optional. Allows the specification of a string which is to be prefixed to the field names of an externally described record format. The characters specified (prefix name) are prefixed to all fields defined in all record formats of the file.

You can also optionally specify a numeric value (length) to indicate the number of leading characters in the existing name to be replaced. If a length is not specified or is zero, the string is prepended to the name.

If a length is specified, it must be a numeric constant containing a value between 0 and 13 with no decimal places. Specifying a value of zero is the same as not specifying a length at all.

The **length** parameter must not be greater than or equal to number of characters in the name to be prefixed. That is, the resulting name cannot be the same as the prefix string after the prefix is applied.

Random

Optional. **Random** specifies whether random database processing using Keys and RRNs is allowed. *YES is the default, which determines the random database processing using keys and RRNs is allowed.

*YES - (default) specifies that Random database processing using Keys and RRNs on op codes such as SETLL, CHAIN, etc., is allowed.

*NO - specifies that Random database processing using Keys and RRNs is not allowed. This option is commonly used when the Keys look different at runtime from the compile time. The common scenario is when using Query Key Fields and under this scenario *NO option must be used to avoid a runtime exception.

The only I/O operations allowed are:

Input - Read, ReadP and Setll with *Start or *End as the Keys. Any other operations throw compiler error - "Random Access Operation Not Allowed For File". The results of Read and ReadP depend on the ORG option and if *INDEXED read them sequentially based on the Key of the file and *ARRIVAL reads in arrival sequence in the file.

Output - Write and Update.

Access

Optional. The type of access to the file. The default is *PRIVATE.

PRIVATE (default) - access to the file is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

*PUBLIC - access to the file is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

*PROTECTED - access to the file is accessible only from within the same class, or from a class derived from this class.

*INTERNAL - access to the file is accessible from within the same project, but not from outside the project.

Shared

Optional. Indicates that one or more declared programming elements are shared. Shared elements are not associated with a specific instance of a class or structure. You can access them by qualifying them either with the class or structure name, or with the variable name of a specific instance of the class or structure. *NO is the default, indicating that the file will not be shared.

*NO (default) - this file will not be shared.

*YES - this file will be shared.

Remarks

The following is a listing of the fields and properties that you can use by specifying the "name" of the memory file as declared with **DCLMEMORYFILE Name (MemoryFile Name)**.

See Also

MemoryFile Members | DCLDB

DCLNAMESPACE - Declare a Namespace

Declares a namespace, which organizes the objects defined in an assembly.

```
DCLNAMESPACE
Name (Character Expression)
```

Parts

Name

Required. The namespace to allocate for this application.

Remarks

At a top level, *namespaces* are entities that organize other entities, such as nested namespaces and types. *Types* are entities that describe values and define executable code. Types may contain nested types and type members. *Type members* are constants, variables, methods, properties, events, enumeration values, and constructors. The declaration space of a namespace is "open ended," so two namespace declarations with the same fully qualified name contribute to the same declaration space.

See Also

Namespace Declarations | Namespaces (Visual RPG Language Concepts)

DCOVERLAYGROUP - Declare an Overlay of a Data Structures' Subfields

Declares an overlay of a Data Structures' subfields for mapping the field to an element of the array.

```
DCOVERLAYGROUP
Name (name)
Dim (Integer Literal)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *TIME | *TIMESTAMP | *ZONED)
Len (Integer Literal, Integer Literal)
StartField (field-name)
Like (Field name)
```

Parts

Name

Required. The name of the overlay group.

Dim

Required. The number of elements in the single-dimension array. **Dim**, together with the **StartField** parameter, defines how many consecutive fields of the DS will be part of the mapping.

Type

Optional. **Type** can be any of Special Value (i.e., *CHAR, *BINARY, etc.) or it can be a class name. If **Type** is *BOOLEAN or *IND, you can use *ON, *TRUE, *ON and *FALSE figurative constants to indicate true and false and "1" and "0", respectively.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer given and *ZONED is assumed if there are two integers. The **Type** of array must be the same as the declared data structure subfields it's mapping to.

Len

Optional. **Len** specifies the length of the field. If **Len** is decimal e.g. Len (3,1) *ZONED is assigned as the default. If **Len** is character e.g. Len (3), *CHAR is assigned as the default.

If **Type** is packed, then the length is not the number of bytes the packed field fits into, but rather the logical digit length of the field. For Example, if you want a 7 digit packed with 2 decimals, code Len (7,2). The **Len** must be the same as the declared data structure subfields it's mapping to.

StartField

Required. **StartField** contains the name of the subfield within the declared data structure at which the overlay group starts. This must be the name of a subfield defined previously in the current data structure. **StartField**, together with the **Dim** parameter, defines how many consecutive fields of the DS will be part of the mapping.

Like

Optional. **Like** must be the name of a field already defined in the class. **Type** cannot be a named constant or a literal. In this case, the field will be defined based on the attributes of another field. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the field being defined. The field length entry is allowed only for field definitions requiring a length.

Example

```
// Declare a simple data structure with two subfields with an overlay group
of the data structure
DCLDS myDS
  DCLDSFLD CSSales01    Type(*Packed) Len(11,2)
  DCLDSFLD CSSales02    Type(*Packed) Len(11,2)
  DCLOVERLAYGROUP SalesArr Dim(2) Type(*Packed) Len(11,2)
StartField(CSSales01)
// Declare a simple data structure with 6 subfields with one overlay of the
first AND last 3 subfields
DCLDS myDS
  DCLDSFLD CSSalesJan    Type(*Packed) Len(11,2)
  DCLDSFLD CSSalesFeb    Type(*Packed) Len(11,2)
  DCLDSFLD CSSalesMar    Type(*Packed) Len(11,2)
  DCLDSFLD CSSalesApr    Type(*Packed) Len(11,2)
  DCLDSFLD CSSalesMay    Type(*Packed) Len(11,2)
  DCLDSFLD CSSalesJun    Type(*Packed) Len(11,2)
  DCLOVERLAYGROUP SalesFirstQtr Dim(3) Type(*Packed) Len(11,2)
StartField(CSSalesJan)
  DCLOVERLAYGROUP SalesLastQtr Dim(3) Type(*Packed) Len(11,2)
StartField(CSSalesApr)
```

See Also

DCLDS | DCLDSFLD | DCLALIASGROUP | OCCUR | MOVEARR

DCLPARM - Identify Parameters

Defines the parameters that compose a parameter list (DCLPLIST) that will be passed to a subprogram when a CALL command is issued, or will be passed to a subroutine when an EXSR command is issued.

DCLPARM

```
Name (Variable name)
CpyFrom (*NONE | Variable | Literal)
CpyTo (*NONE | Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *FLOAT | *FLOAT4 |
      *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
DbDirection (*BOTH | *INPUT | *OUTPUT)
```

Parts

Name

Required. Name contains the name of the parameter field to pass.

Visual RPG allows you to name a control property as a parameter field, provided the property conforms to valid **DCLPARM** types. However, you must not assume the default property for the control. Instead, you must specifically code the desired property, as shown below.

```
DCLPARM Name ( ioField1.value )
DCLPARM Name ( checkbox1.value )
```

CpyFrom

Optional. **CpyFrom** contains the value of the parameter. *NONE is the default. See Remarks section for more information.

CpyTo

Optional. **CpyTo** contains the Parm values are copied from **CpyFrom**. *NONE is the default.

Type

Optional. **Type** can be any of the types listed (Character, Packed or Zoned). *NONE is the default. See Type Parameter for further information.

Len

Optional. **Len** defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

DbDirection

Optional. **DbDirection** specifies whether the parameter to a remote program call is an input-only parameter (*Input), an output-only parameter (*Output) or an input and output parameter (*Both). When a large amount of data is passed in only one direction - either to the called program, or from the called program, **DbDirection** allows you to reduce the call time by eliminating the transmission of unused data. *Both is the default.

The direction of the parameter is from the perspective of the program being called. That is, *Input means the called program is expecting data in the parameter, but doesn't need to return it to the caller. *Output means the called program will send data back in the parameter to the caller, but is not expecting data in the parameter as input to it.

DbDirection has specific meaning depending upon the type of the Database to which the Call is being made.

Calling AS/400 programs - AS/400 programs do not specify whether the parameter is used by the program for input or output, so the real value of DbDirection is in time savings when the parameter is large. Specifying the proper direction when data is only used in one direction saves time because the data in the parameter is only sent to the AS/400 when the parameter is marked as *Input and only transmitted back to the client if the parameter is marked for *Output.

Calling SQL Server stored procedures - SQL Server defines two kinds of parameters, regular parameters and 'output' parameters. SQL Server regular parameters are only received by the procedure and so must have a **DbDirection** of ***Input**. What SQL Server means by an 'output' parameter is really a parameter that is optionally received but always sent back to the client. Therefore, a SQL Server 'output' parameter may have a **DbDirection** of ***Output** if it is certain the procedure does not use the received value. However, if ***Output** is specified and the procedure does depend on a received value, unpredictable results will occur. It is never wrong to specify a **DbDirection** of ***Both** for a SQL Server 'output' parameter even if the stored procedure does not use the received value.

DBDirection Example:

If an SQL Server Stored procedure looks like the following:

```
CREATE PROCEDURE CountNewOrders
  @WarehouseId as decimal(8,0) ,
  @OrderCount as numeric (5,0) output
AS
  select @OrderCount = count(distinct(OrderID)) from [Orders] where WId=@
  WarehouseId
GO
```

Then the Visual RPG Call should be:

```
CALL PGM ('CountNewOrders') DB ('SQL')
DCLPARM OrderNum DbDirection (*Input)
DCLPARM OrderCount DbDirection (*BOTH)
```

Remarks

The **DCLPARM** command comes after a **CALL**, **DCLPLIST**, or **EXSR** command, and defines the parameters that will be passed to the subprogram/subroutine when it is called. One or more **DCLPARM** commands can come after a **CALL** or **DCLPLIST** command.

A **DCLPARM** command that does not come after a **CALL**, **DCLPLIST** or another **DCLPARM** command will cause a compiler error.

When a program is called, any parameters specified are assigned the value of the operand specified in **CpyFrom**. The parameter is then passed to the program by reference, meaning that the program may change the value of the Parm. Finally, the Parm values are copied to any **CpyTo** operands specified on the **CpyTo** keyword.

The data type and length can be specified.

You can pass a data structure as a parameter to an AS/400 program when doing a remote call. There are, however, some restrictions as noted below, on the format of the data structure due to the fact that these fields need to be converted between PC and AS/400 formats (things like ASCII to EBCDIC, byte order, etc.). Note that the following restrictions don't apply for calling programs within the same project.

All of the space taken by the data structure should be described. If an area of the data structure is not occupied by a field, the AS/400 program will not get the contents of that area. If the AS/400 program were to modify that area, then the PC program will not get the modification.

Data structure fields cannot be arrays.

Example

```
// Call program GetDate passing it a named parameter list.
CALL PGM('GetDate') PARMLIST(IoDateValues)
// Declare the parameter list for the above call. The value of DayYear is
// copied to IoDayYear before the CALL is processed, and the return value is
// then placed in field IoYearMonthDay. These fields are declared elsewhere. */
DCLPLIST Name(IoDateValues)
```



```

DCLPARM Name (IoMonth)
DCLPARM Name (IoCompleteDate) CPYFrom(DayYear) CPYTO(IoYearMonthDay)
// The following call is identical in function to the above call, but passes
the parameters directly, rather than using a names parameter list. */
CALL PGM('GetDate')
DCLPARM Name (IoMonth)
DCLPARM Name (IoCompleteDate) CPYFrom(DayYear) CPYTO(IoYearMonthDay)
// The following is an Example of calling a subroutine and passing
parameters.
EXSR ComputeTax
DCLPARM Name (TaxRateCode)
DCLPARM Name (Value)

```

See Also

CALL | DCLPLIST | DCLSRPARM | EXSR

DCLPLIST - Declare a Parm List

Begins the declaration of a list of parameters to be passed to a program, or be received from a caller if the name given is *ENTRY.

```

DCLPLIST
Name (ParmListName)

```

Parts

Name

Required. **Name** must contain the name of the parameter list.

A parameter list is ended when an operation other than **DCLPARM** is encountered.

See Also

DCLPARM

DCLPRINTFILE - Declare a Print File

Declares a print file for printing in a program.

```

DCLPRINTFILE
Name (Printerfile name)
Db (Database Object)
File (*FILEDESC | Character Expression)
ImpOpen (*YES | *NO)
RnmFmt (*NONE | Old Format name, New Format name, ...)
ShareType (*FILE | *EXCLUSIVE | *EXCLUSIVEREAD | *SHAREUPDATE |
*SHARENOUPDATE | *SHAREREAD |
Character Expression)
ChkFmtId (*YES | *NO | *SAME | *IGNORE | *MAP | Indicator Expression)
Prefix (prefix name, [Length])
OverFlowInd (Indicator-name)
IndDS (Indicator DS Name)

```

Parts

Name

Required. **Name** can contain either:

The name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

a valid program id registered on your system at compile time.

a valid class id registered on your system at compile time.

a valid interface name registered on your system at compile time.

A name of the form **Library.Object**, where Library is the name of one of the libraries selected in the new object libraries dialog, and Object is valid object name within that library.

Note: Using the DclDiskFile *name*, you can then get to its **fields** and **properties**. Please note that methods are reserved for use by the ASNA.Visual RPG compiler, so use the corresponding Visual RPG command if applicable. See PrintFile Class Members for a listing of available members.

Db

Required. **Db** specifies the Database Object declaration whose DBName specifies where the compiler will acquire the file description. **DB** specifies the Database Connection to use when opening the file at run-time. Database Objects are defined with the DCLDB command.

File

Optional. **File** specifies the Library and File to use when opening the file at run-time.

If the **File** parameter is not given, the same file that was used to retrieve the description will be opened at run-time.

If a file name is given on the **File** parameter, that file will be accessed and must have the same description as the file accessed in the **Name** parameter, otherwise unpredictable results may occur. The **File** parameter accepts a variable as a parameter, so that the file accessed can be overridden at run-time.

ImpOpen

Optional. **ImpOpen** specifies if the file will be implicitly opened. The default is *YES.

*YES (default) - specifies that the file will automatically be opened when the program is first run. If the implicit open is unsuccessful, a run-time error will occur.

*NO - specifies that the file will not automatically be opened when the program is first run. The RPG IV USROPN keyword is supported in Visual RPG by using Impopen(*NO).

RnmFmt

Optional. **Rename Format**. Renames the old format to a new format, to avoid clashes with duplicate format names. *NONE is the default.

For single format files, you can specify **RnmFmt** (*New Format Name*) - which will automatically rename the only format in the file to the *New Format Name*. If you specify the syntax "*Old format Name, New Format Name*" for single format files, the Visual RPG compiler will ignore the *Old Format Name*.

ShareType

Optional. **ShareType** defines values on how a file will be shared. The default is *FILE.

*File (default) - Maintains the same share type specified when the file was created.

*Exclusive - Open for update, exclusive lock.

*ExclusiveRead - Open for update, other jobs may read.

*ShareUpdate - Open for update, other jobs may read or update.

*ShareNoUpdate - Open for read, other jobs may read.

*ShareRead - Open for read, other jobs may read or update.

ChkFmtId

Optional. **ChkFmtId** checks if the format ID of the file at run-time compares to the format ID of the file at compile-time. Visual RPG stores the format of file(s) when a program is first compiled. At run-time, Visual RPG compares the format ID that was stored during the compilation with the current format ID. The comparing of format ID's can be especially useful when checking if new fields have been added to a file used by a program that has been compiled months ago.

***SAME** is the default, and indicates that the run-time and compile-time format ID's **will be compared** to verify that they are the same. (This is exactly equivalent to *YES in previous releases).

***IGNORE** indicates that the run-time and compile-time format ID's **will not be compared** to verify that they are the same. Visual RPG will use the compile-time format ID. (This is exactly equivalent to *NO in previous releases).

Note Please be extremely careful in using *IGNORE, as it has the risk of the corruption of data files and misbehavior of your application!!

***MAP** indicates that the run-time and compile-time format ID's **will be compared**, and Visual RPG will accommodate non-key fields added and changed **regardless** of where they are defined in the new record format. When the modified file is opened, Visual RPG will **rebuild, or map** its compiled record format at run-time to equal that of the new record format.

Restrictions apply to non-key fields depending upon the mode of file processing and the changes made to the file as illustrated by the **table below**. The rule of thumb is that Visual RPG will process the file provided there is **no loss of data**.

Any change in the count, order or type of Key fields is not allowed, except that a numeric key may be changed from packed to zoned and vice versa, as long as the precision remains unchanged.

Prefix

Optional. **Prefix** allows the specification of a string which is to be prefixed to the field names of an externally described record format. The characters specified as (prefix name) are prefixed of all fields defined in all record formats of the file.

You can also optionally specify a numeric value (length) to indicate the number of leading characters in the existing name to be replaced. If a length is not specified or is zero, the string is prepended to the name.

The total Length of a name after applying the prefix must not exceed the maximum length of an RPG field name.

If a Length is specified, it must be a numeric constant containing a value between 0 and 13 with no decimal places. Specifying a value of zero is the same as not specifying a length at all.

The Length parameter must not be greater than or equal to number of characters in the name to be prefixed. That is, the resulting name cannot be the same as the prefix string after the prefix is applied.

OverflowInd

Optional. **OverflowInd** contains the name of an indicator used to specify if the print file has reached the Overflow area. The indicator-name must be a previous declared field of Type (*Ind). After each 'write' to the print file, the cursor is positioned in the overflow area (line number as defined by the .NET Print file). The indicator named will then be set on (*ON), and the program may then test for the indicator specified to Write to other formats, such as Page Headers. Page Headers, or other formats should include properties to start a new page before printing, causing the cursor to be positioned outside the overflow area, ready for the next Write. For the next 'write' that is not within the overflow area, the indicator named will be turned off (*OFF).

IndDS

Optional. The name of the indicator data structure to be associated with the print file. The indicator data structure contains the conditioning and response indicators passed to and from data management for the file. Some basic considerations are:

The data structure named may be associated with more than one file.

The data structure name must be defined by a DclDS and can be a multiple-occurrence data structure.

The length of the indicator data structure is always 99.

The data structure will be initialized to ***Blanks**. To get the same behavior as the iSeries, you need to manually initialize the structure to (all) ***Zeros**.

Note Monarch generated code accomplishes this by adding a Move with ***Zeros** to the structure's largest ***Char** field.

Only ***Char** and ***Ind** type member fields may be declared.

The indicator data structure is passed into the called application, so that the global conditioning and response indicators are not affected. Upon return, the indicator data structure fields will have the response indicators values modified. referring to the third Example shown below, if the application changed response indicator *IN03, the **Exit** field value is affected instead of the global *IN03; if *IN33 was changed, the **Change** field value is affected, and so forth.

Remarks

DCLPRINTFILE declares a DataGate printfile for printing at run-time. At compile time, the description of the file is read in with its record format. The file name and each record format name must be unique.

The **Name** parameter is **required** and contains the name of the print file. Using the printerfile *name*, you can then get to its **fields** and **properties**. Please note that **methods** are reserved for use by the ASNA.Visual RPG compiler, so use the corresponding Visual RPG command when applicable. See PrintFile Class Members for a listing of available members.

Print Files are always opened for sequential output operation.

The actual file/member that is accessed at run-time is determined by the **Db** and **File** parameters. They accept a variable as a parameter, so that the file accessed can be overridden at run-time.

Example

```
DclPrintFile Name(SalesRept) ImpOpen(*NO) File("LIBL/CustReport")
DB(ProdDB)
// File description is "LIBL/CustReport"
DclPrintFile Name(CustReport) ImpOpen(*NO) DB(DBSQL)
// File description is "TestLib/InvReport" and the Indicator Data Structure
is dspind
DclPrintFile Name(Inventory) ImpOpen(*NO) DB(DBSQL)
File("TestLib/InvReport") IndDS( dspind )
. . .
DclDS dspind
  DclDsFld dspindIndicators Type( *Char ) Len( 99 )
  DclDsFld Exit              Type( *Ind ) Overlay( dspindIndicators,3 )
  DclDsFld Search            Type( *Ind ) Overlay( dspindIndicators,4 )
  DclDsFld Previous         Type( *Ind ) Overlay( dspindIndicators,12 )
  . . .
  DclDsFld Change           Type( *Ind ) Overlay( dspindIndicators,33 )
  . . .
  DclDsFld Protect_Cmp      Type( *Ind ) Overlay( dspindIndicators,99)
```

See Also

DCLDB | DCLDISKFILE | DCLFILECONT | PrintFile Class Members

DCLSRPARM - Declare Subroutine Parameter

Defines the parameters that will be received by BEGSR, BEGPROP, BEGFUNC or BEGCONSTRUCTOR. Parameters can be passed by value or by reference.

```
DCLSRPARM
Name (*NONE | Variable name)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *DECIMAL | *FLOAT4 | *FLOAT8 | *FORM | *IND |
      *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *OBJECT | *ONECHAR |
      *PACKED | *STRING | *TIME | *TIMESTAMP | *ZONED |
      Form type, Field, Array, Array Element, Table name, String Literal,
      Library.Object)
Len (Length Integer, [Decimal Integer])
By (*VALUE | *REFERENCE)
TimFmt (*PGMDFT | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
      *YMD)
Rank (IntegerLiteral)
Attributes (Attribute 1, Attribute 2, ...)
Like (Function name)
```

Parts

Name

Required. **Name** can contain either:

- *NONE (default). If a specified name is not defined, this default prevents a warning if the parameter is not used.

- The name of the variable already defined.

- A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:
 - *NONE (default). If a specified name is not defined, this default prevents a warning if the parameter is not used.
 - The name of the variable already defined.
 - A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

- a valid program id registered on your system at compile time.

- a valid class id registered on your system at compile time.

- A valid interface name registered on your system at compile time.

Type

Optional. **Type** can be any of the "" Types (*CHAR, *BINARY, etc.) or it can be the name of a field already defined in the program.

Type cannot be a named constant or a literal. In this case, the field will be defined based on the attributes of another field.

If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field.

Type can also contain a form type, which is specified by setting the Nameproperty on a form. For more information on using form variables, see Form Variables.

If **Type** is *BOOLEAN, you can use the *TRUE or *ON and *FALSE or *OFF special values to indicate True or False.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, ***CHAR** is assumed if there is one integer, and ***ZONED** is assumed if there are two integers.

Len

Optional. **Len** specifies the length of the variable. If **Len** is specified without a **Type** parameter, ***CHAR** is assumed if there is one integer, and ***ZONED** is assumed if there are two integers.

By

Optional. **By** specifies whether the parameter will be passed to the routine by value or by reference.

A referenced parm is one where the pointer memory location of the variable is passed to the subroutine. Any change made to the parameter data will change the data of the parameter variable in the parameter. When passing parameters by reference, you **must** use the ***ByRef** operator.

***Value** passes the parameter data to the subroutine. If the subroutine modifies the data of a ***value** SRPARM, the original parm data is unaffected.

TimFmt

Optional. **TimFmt** specifies the format for time. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Rank

Optional. **Rank** declares an n-dimensional array of indefinite elements. The value given for **Rank** defines the number of dimensions for the array.

Attributes

Optional. Allows you to provide one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

Attributes use required, and optional arguments. The required parameters are provided in positional fashion. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For Example, the following code calls the **Developer** constructor of an attribute with two values:

```
BegClass MyClass Attributes(Developer("Joe Cool", 17))
```

You can also use named arguments, which are optional, to directly set the value of properties and fields of the attributes. You specify a named argument by appending "!=" to the name of the argument, followed by the value being specified. Unlike positional arguments, named arguments can be specified in any order. For Example, the following code sets the value ***True** for the Reviewed property of the Developer Attribute, and **False** for the Keep field:

```
BegClass MyClass Attributes(Developer("Joe Cool", 17, Reviewed:!=True)
```

Note You must use positional arguments for arguments that are used with the attribute's constructor. Named arguments can only be used to set the values of fields and properties of attribute classes.

Like

Optional. **Like** must be the name of a property already defined in the class. **Type** cannot be a named constant or a literal. In this case, the property will be defined based on the attributes of another property. If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field. You can use **Len** parameter to make the declared field longer or shorter than the **Like** parameter entry.

A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. You cannot change the number of decimal positions for the property being defined. The length entry is allowed only for property definitions requiring a length.

Remarks

The **DCLSRPARM** command comes after a **BEGSR**, **BEGFUNC** or **BEGPROP** command, and defines the parameters that will be received by the subroutine when it is called. One or more **DCLSRPARM** commands can come after a **BEGSR** command. A **DCLSRPARM** command that does not come immediately after a **BEGSR**, **BEGFUNC** or **BEGPROP** or another **DCLSRPARM** command will cause a compiler error.

The data type (**Type**) and length (**Len**) can be specified.

Subroutine parameters are declared as local variables to your subroutine. Thus, multiple subroutines can declare the same parameter name, and the names won't clash.

When parameters are passed on the **EXSR**, they don't necessarily need to be the same type as specified on the **DCLSRPARM** statement. Visual RPG will coerce the parameter to the type you give. For Example, many parameters are four-byte integers, but if you declare the parm as zoned, the value will be converted/possibly truncated accordingly. Likewise, if you specify character, the parm will be copied to the character as if you were doing a **MOVE** with **Pad**. The programmer is responsible to make sure that values are not truncated because the receiving parameter is too small.

Some ActiveX Control events have parameters that tell you more specific information about the event. **MouseDown** for Example, passes the X and Y coordinates, and a code that tells which mouse button was pressed. See Event Subroutine Parameters for more information.

See Also

CALL | DCLPLIST | EXSR | *ByRef Operator

DCLSUBFILE - Declare a Subfile

Declares a subfile control file.

```
DCLSUBFILE
Name (file name)
FormatName (format name)
RRNName (field name)
ExtDesc (*NO | (YES)
DbDesc (Database Object | Character Expression)
FileDesc (*NAME | Character Expression)
Access (*PRIVATE |*PUBLIC |*PROTECTED |*INTERNAL)
Parent (*this | parent-control-name])
```

Parts

Name

Required. Name can contain either:

- The name of the variable already defined.

- A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

- a valid program id registered on your system at compile time.

- a valid class id registered on your system at compile time.

- a valid interface name registered on your system at compile time.

- A name of the form Library.Object, where Library is the name of one of the libraries selected in the new up and coming object libraries dialog, and Object is valid object name within that library.

Note: Using the **DclSubfile name**, you can then get to its **methods**, **events**, and **properties**.

FormatName

Required. Identifies the name of the subfile record format. See **FormatName** property.

RRNName

Required. Identifies the name of the numeric field to be associated with the subfile to communicate the relative-record-number. See **RRRName** property.

ExtDesc

Optional. **ExtDesc** specifies whether the data structure will be externally described. The default is ***NO**.

***NO** – (Default). The data structure is not externally described.

***YES** - The data structure is externally described. Note that to define the data structure externally, you must also have valid entries for the **DbDesc** and **FileDesc** parameters.

DbDesc

Optional. Database description. Specifies the database to use during compile-time to access the File Description.

Database Object declaration whose *DbName* specifies where the compiler will acquire the file description. Database Objects are defined with the **DCLDB** command.

Character expression - the Database Name to get the description from. Used in conjunction with **FileDesc**.

FileDesc

Optional. File description. Takes the description of the named file as the bases to layout the memory file. Used in conjunction with **DbDesc**.

This parameter must be a character literal, so the file description can be accessed at compile time.

***NAME** (default) - uses the parameter **Name** to find the file to obtain the description of the file's layout.

Character expression - the description of the database file.

Access

Optional. The type of access to the subfile. ***PRIVATE** is the default.

***PRIVATE** (default) - is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

***PUBLIC** - is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

***PROTECTED** - is accessible only from within the same class, or from a class derived from this class.

***INTERNAL** - is accessible from within the same project, but not from outside the project.

Parent

Optional. The name of the parent control that this subfile is a child of. Defaults to ***this**. If used, the parent-control-name is the name of another control declared in a prior **DCLSUBFILE** command that is to contain this control.

Remarks

One or more subfields must be declared using the DCLSUBFILEFLD command after the **DCLSUBFILE** command.

See Also

DCLDB | DCLSUBFILEFLD | SubfileControl Overview | FormatName Property | RRRName Property

DCLSUBFILEFLD - Declare a Subfield for a SubfileControl

Declares a subfield for a SubfileControl.

```
DCLSUBFILEFLD
Name (name)
Type (*BINARY | *BOOLEAN | *CHAR | *PACKED | *ZONED)
Len (Integer Literal, Integer Literal)
Heading (string literal)
Width (*CALC | Integer)
EditWord (*String)
EditCode (*None | *EditCodeOne | *EditCodeTwo | *EditCodeThree |
*EditCodeFour |
*EditCodeA | *EditCodeB | *EditCodeC | *EditCodeD | *EditCodeJ |
*EditCodeK | *EditCodeL |
*EditCodeM | *EditCodeN | *EditCodeO | *EditCodeP | *EditCodeQ |
*EditCodeY | *EditCodeZ)
Hidden (*False | *True)
Browse (*False | *True)
UpperCaseOnly (*False | *True)
Alignment (*Left | *Right | *Center)
DropDown (*None | *SimpleList | *DropDownList | *DropDownCombo)
```

Parts

Name

Required. **Name** is the field name of the field being declared.

Type

Required. **Type** can be any of Special Value (i.e., *CHAR, *BINARY, etc.) or it can be a class name. If **Type** is *BOOLEAN, you can use *TRUE or *ON and *FALSE or *OFF special values to indicate True and False.

Len

Required. **Len** specifies the length of the field. If **Len** is decimal e.g. Len (3,1) *ZONED is assigned as the default. If **Len** is character e.g. Len (3), *CHAR is assigned as the default.

If **Type** is packed, then the length is not the number of bytes the packed field fits into, but rather the logical digit length of the field. For Example, if you want a 7 digit packed with 2 decimals, code **Len** (7,2).

Heading

Optional. The string that will be displayed as the column heading in the grid for the field. If set to blanks, the **Heading** defaults to the **Name** parameter. See Also the FieldColumnHeading property.

Width

Optional. *CALC or integer value containing the width of the column. *CALC is the default, which automatically calculates the column width based upon **Len** or **Heading**, whichever is greater. See Also the FieldColumnWidth property.

EditWord

Optional. An edit word allows you to control the display of blanks, negative sign, suppression of unwanted zeroes, commas and decimal points, and named constants. See Edit Word usage for a description of setting this parameter.

EditCode

Optional. *None, *EditCodeOne, *EditCodeTwo, *EditCodeThree, *EditCodeFour, *EditCodeA, *EditCodeB, *EditCodeC, *EditCodeD, *EditCodeJ, *EditCodeK, *EditCodeL, *EditCodeM, *EditCodeN, *EditCodeO, *EditCodeP, *EditCodeQ, *EditCodeY, or *EditCodeZ. *None is the default. An EditCode allows you to punctuate numeric fields, including \$ signs, commas, periods, minus sign, and floating minus according to the standard RPG edit code rules. See Edit Code usage for a description of setting this parameter.

Hidden

Optional. *True, *False, or figurative constant containing *ON or *OFF. *False is the default. *ON/*True indicates the contents of the field are hidden, otherwise *OFF/*False. See Also the FieldHidden property.

Browse

Optional. *True, *False, or figurative constant containing *ON or *OFF. *False is the default. *ON/*True indicates the field contents can not be edited by the user, otherwise *OFF/*False. See Also the FieldBrowse property.

UpperCaseOnly

Optional. *True indicates the contents of the field are uppercase, otherwise *False. *False is the default. See Also the UpperCaseOnly property.

Alignment

Optional. *Left (left justified), *Right (right justified), *Center (centered) to indicate the alignment of data within the field. *Left is the default.

DropDown

Optional. *None, *SimpleList, *DropDownList, or *DropDownListCombo. *None is the default.

*None indicates this is not a dropdown field. This is the default.

*SimpleList indicates a list and an edit area that is always displayed. The user can select from the list or type in the edit area.

*DropDownList indicates a drop-down list from which the user can make a selection.

*DropDownCombo indicates a drop-down list and an edit area. The user can select from the list or type in the edit area.

Remarks

See SubfileField Collection Editor to set these properties at design-time.

See Also

DCLSUBFILE | Subfile Control Overview | Subfile Control Members | Edit Word Usage | Edit Code Usage

DCLWORKSTNFILE

Declares an externally described workstation file for processing in a program.

```
DCLWORKSTNFILE
Name (name)
DspFile (*NAME | Character Expression)
ImpOpen(*YES | *NO)
RnmFmt (*NONE | Old Format name, New Format name, ...)
Subfile(*NONE | Format name, RRN name, ...)
Prefix (prefix name, [ Length ] )
Access (*PRIVATE | *PUBLIC | *PROTECTED | *INTERNAL)
Shared (*NO | *YES)
ShareOpen (*NO | *YES)
IndDS (Indicator DS Name)
```

Parts

Name

Required. Name of the workstation file.

DspFile

Optional. For ASP.Net based display files, the **DspFile** keyword names the '.aspx' file containing the web form. The character expression may contain an absolute or relative path to where the file is located; for relative paths, the location of the program containing the **DclWorkStnFile** is considered the 'current' folder.

ImpOpen

Optional. **ImpOpen** specifies if the file will be implicitly opened. The default is *YES.

*YES (default) - specifies that the file will automatically be opened when the program is first run. If the implicit open is unsuccessful, a run-time error will occur.

*NO - specifies that the file will not automatically be opened when the program is first run. The RPG IV USROPN keyword is supported in Visual RPG by using **ImpOpen**(*NO).

RnmFmt

Optional. **Rnmfmt** renames the external formats to new, internal formats to avoid clashes with duplicate format names. *NONE is the default.

For **multi-format** files, you enter the external and internal format names in pairs. All names are separated by commas.

For **single** format files, you can either specify the internal format name as the only format name, or enter both the external and internal format names separated by a comma.

```
Rnmfmt ( RCMMaster, CustInfo)           // single format
Rnmfmt ( CustInfo)                       // single format
Rnmfmt (RSubCust, CustSub, RSubSales, SalesSub) // multi-format Subfile
```

Optional. The **Subfile** keyword is used to internally define the subfiles that are specified in the workstation file. The *Format name* identifies the name of the subfile record format and the *RRN name* identifies the name of the numeric field to be associated with the subfile to communicate the relative-record-number. Each subfile *format name*, with its corresponding *RRN name*, must be listed in the single **Subfile** keyword.

The relative-record number of any record retrieved by a **READC** or **CHAIN** operation is placed into the field identified by the **RRN** parameter. This field is also used to specify the record number used in **WRITE** operations to the subfile. The field name specified as the **rrn** field parameter must be defined as numeric with zero decimal positions.

Prefix

Optional. Prefix allows the specification of a string which is to be prefixed to the field names of an externally described record format. The characters specified as (prefix name) are prefixed of all fields defined in all record formats of the file.

You can also optionally specify a numeric value (length) to indicate the number of leading characters in the existing name to be replaced. If a length is not specified or is zero, the string is prepended to the name.

Access

Optional. The type of access to the workstation file. ***PRIVATE** is the default.

***PRIVATE** (default) - access to the workstation file is accessible only from within their declaration context, including from members of any nested types, for Example from within a nested procedure or from an assignment expression in a nested enumeration.

***PUBLIC** - access to the workstation file is accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project.

***PROTECTED** - access to the workstation file is accessible only from within the same class, or from a class derived from this class.

***INTERNAL** - access to the workstation file is accessible from within the same project, but not from outside the project.

Shared

Optional. Shared specifies whether or not this workstation file is to be shared. ***NO** is the default.

***NO** (default) - this workstation file will not be shared.

***YES** - this workstation file will be shared.

ShareOpen

Optional. ShareOpen specifies if the display file (active server page) used by this **DclWorkStnFile** allows any other to refer to the same display file. Valid values are ***Yes** and ***No**. ***No** is the default.

***NO** (default) - this display file can not be shared.

***YES** - When specifying yes, two or more programs can share the same instance of the display file, that is, one program can write records to the file, and the other can read the data provided by the user.

IndDS

Optional. The name of the indicator data structure to be associated with the workstation file. The indicator data structure contains the conditioning and response indicators passed to and from data management for the file. Some basic considerations are:

The data structure named may be associated with more than one file.

The data structure name must be defined by a **DclDs** and can be a multiple-occurrence data structure.

The length of the indicator data structure is always 99.

The data structure will be initialized to ***Blanks**. To get the same behavior as the **iSeries**, you need to manually initialize the structure to (all) ***Zeros**.

Note Monarch generated code accomplishes this by adding a Move with ***Zeros** to the structure's

largest ***Char** field.

Only ***Char** and ***Ind** type member fields may be declared.

With **ExFmt** *MyWorkStnFile* or **Read** *MyWorkStnFile*, the indicator data structure is passed in, so that the global conditioning and response indicators are not affected. Upon return from **ExFmt** or **Read** (from the *WorkStnFile*), the indicator data structure fields will have the response indicators values modified by the workstation file. In the Example below, if the workstation file changed response indicator ***IN03**, the **Exit** field value is affected instead of the global ***IN03**; if ***IN33** was changed, the **Change** field value is affected, and so forth.

Remarks

This command declares an externally described workstation file for processing in a program.

Example

```
DclWorkStnFile MyWorkStnFile DspFile("./MyWorkStnFile.aspx") Subfile(
CustSfl, CustRrn ) IndDS( dspind )
DclDs dspind
  DclDsFld dspindIndicators Type( *Char ) Len( 99 )
  DclDsFld Exit              Type( *Ind ) Overlay( dspindIndicators,3 )
  DclDsFld Search            Type( *Ind ) Overlay( dspindIndicators,4 )
  DclDsFld Previous          Type( *Ind ) Overlay( dspindIndicators,12 )
  ...
  DclDsFld Change            Type( *Ind ) Overlay( dspindIndicators,33 )
  ...
  DclDsFld Protect_Cmp       Type( *Ind ) Overlay( dspindIndicators,99)
```

DELETE

Database file: Physically deletes a record from a file. The file must be an update file.

SubfileControl : Removes the record from the subfile (equivalent to the *RemoveItem* method).

```
DELETE
From (file Name | Format Name)
Key (*CURRENT | Operand)
NotFnd (Indicator Variable)
Err (Indicator Variable | *EXTENDED)
```

Parts

From

Required. The name of a record format, file or subfile.

Key

Optional. The search argument to use when deleting a file. ***CURRENT** is the default. Refer to **%KDS** to use this built-in function to establish search arguments (Key) for Data Structures.

NotFnd

Optional. Used to determine if the record was found while executing this command.

Note that you can also use the **%FOUND** function. However, be aware that **%FOUND** and the **NotFnd** indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Database File - Unless a search argument (Key) is specified, the last record read from the file is deleted. If there has been no record read, the error indicator will be set.

SubfileControl - The record referenced by the subfile's RRN variable (specified in RRNFidName property) is deleted. The record does not have to be previously read. If the RRN value is invalid, the error indicator will be set.

Once a record has been deleted, it cannot be retrieved by any I/O operation. Note that a record deleted from an AS/400 or Acceler8DB database still occupies space in the file.

Example

/* In this Example, the file, MyFile, is retrieved in update mode by key MyKey. The record is found but the user does not have authority to delete this record. Error indicator 42 is turned on and the user issued the message. */

```
ReadRand MfFile MyKey NotFnd(*In42)
  If *In41 = *Off
    Delete MyFile Err (*In42)
    If *In42
      MSGBOX 'You do not have authority to perform this function'
    Endif
  Endif
```

/* In this Example, the file, MyFile, is retrieved in update mode by key MyKey and deleted all in the same step. Error indicator 42 is turned on if there was an error and indicator 41 is turned on if the record was not found.*/

```
Delete MyFile MyKey NotFnd(*In41) Err (*In42)
  If *In41
    Exsr RecordNotFound
  Endif  If *In42
    Exsr ErrorOnAuthority
  Endif
```

See Also

%KDS | %FOUND | %KDS

DELETERANGE

Positions the file to the desired range of records constrained by the FirstKey and the LastKey parameters and deletes those records. The range is determined by the combination of the parameters of the command. For a detailed explanation of properly marking a range, please refer to the Example provided within SETRANGE.

This operation is valid only for files opened for indexed access (Use ORG (*Indexed) option on DCLDISKFILE).

```
DELETERANGE
File (FileName | FormatName)
FirstKey (Key List Name | Variable Operand)
LastKey (*FIRSTKEY | Key List Name | Variable Operand)
IncludeFirstKey (*YES | *NO)
IncludeLastKey (*YES | *NO)
NotFnd (Indicator Variable)
Err (Indicator Variable | *EXTENDED)
NotAll (Indicator Variable)
```

Parts

File

Required. The name of a file or a format name. The range operation works on this file or format name.

FirstKey

Required. Can be any valid variable operand or a Key List name that is used as the search argument for the top of the range in positioning a file. If you want to exclude from the range the records whose key is equal to the first key, use IncludeFirstKey (*No), which positions the file to the first record whose key is greater than the **FirstKey**.

LastKey

Optional. Defaults to *FirstKey (meaning **LastKey** is the same as the **FirstKey**). This parameter can be any valid variable operand or a Key List name that is used to mark the bottom of the range. If you want to exclude, from the range, the records whose key is equal to the **LastKey**, use IncludeLastKey (*No), which marks the last record whose key is less than the **LastKey** parameter as the bottom of the range.

IncludeFirstKey

Optional. Specifies whether to include the very first record found within the range when deleting records. The default is *Yes.

*YES – (Default) If the operation finds a record whose key matches the FirstKey, then the file is positioned at that record. If the record that matches the key is not found, then the file is positioned at the first record whose key is greater than the **FirstKey**.

*NO – The file is positioned on the first record whose key is greater than the **FirstKey**.

IncludeLastKey

Optional. Specifies whether to mark the last record found that matches the **LastKey** as the bottom of the range. The default is *Yes.

*YES – (Default) If the operation finds a record whose key matches the **LastKey**, then the bottom of the range is marked at that record. If the record that matches the key is not found, then the last record whose key is less than the **LastKey** is marked as the bottom of the range.

*NO – The last record whose key is less than the **LastKey** is marked as the bottom of the range.

NotFnd

Optional. Used to determine if the record was found while executing this command.

Note that you can also use the %FOUND function. However, be aware that %FOUND and the NotFnd indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. Will be set if any error condition is found with the file. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

NotAll

Optional. Is set on when all of the records specified in the range to be deleted were not deleted.

Remarks

Use **DELETERANGE**, in conjunction with the **SETRANGE** and **READRANGE** op codes to optimize SQL Server processing and to enhance client/server performance with all supported database engines (iSeries/400, Acceler8DB and SQL Server) with dynamic Network Blocking.

`DELETERANGE` replaces a `Read(PE)` - Delete loop.

See Also

DCLDISKFILE | READRANGE | SETRANGE

DISCONNECT - Close a Connection to a Database

Closes a connection to a database.

```
DISCONNECT
Db (Database Object)
Err (Indicator Variable | *EXTENDED)
```

Parts**Db**

Required. A Database Object defined with the DCLDB command.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Operand) - the name of the err variable.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

To open a connection to a database, use **CONNECT**.

See Also

CONNECT

DIV

Divides **F1** by **F2** if **F1** is specified. If not, it divides the **Result** by **F2** and places the quotient in **Result**.

```

DIV
F1 (*NONE | Numeric Expression)
F2 (Numeric Expression)
Result (Numeric Expression)
Rem (*NONE | Numeric Variable)
Adj (*TRUNC | *RNDUP)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
        *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
        | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])

```

Parts

F1

Optional. Must be numeric and be any valid expression. When **F1** is not specified, the dividend (**Result**) is divided by **F2** (divisor), and the result (quotient) is placed in **Result**.

F2

Required. Must be a numeric expression. If it is zero, an error occurs at run-time.

Result

Required. Must be a numeric expression.

Rem

Optional. The remainder is lost unless the **Rem** parameter is specified, or the MOVEREM (Move Remainder) operation is specified as the next operation. If **Rem** is not used, and MOVEREM is not processed directly after the **DIV** operation, undesirable results will occur.

Adj

Optional. Indicates half-adjust. Use **Adj** (*RNDUP) if you wish to round up the value, or use **Adj** (*TRUNC) if you wish to truncate the value of the **Result** Field. *TRUNC is the default.

Pos

Optional. Will be turned on if the value of **Result** is a positive number.

Neg

Optional. Will be turned on if the value of **Result** is a negative number.

Zero

Optional. Will be turned on if the value of **Result** is zero.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of ***CHAR**, ***PACKED**, ***ZONED** will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Example

When using positional values, (i.e. **F1** is not used) the Keywords **F2** and **Result** must be used unless ***NONE** occupies the null position.

In these Examples the following are set values.

```
DclFld Name (A) Len(5,0) Inz (515)
DclFld Name (B) Len(5,0) Inz (280)
DclFld Name (C) Len(7,2) Inz (0)
DclFld Name (D) Len(9,3) Inz (0)
DclFld Name (E) Len(7,2) Inz (35)
```

Divide A by B giving C. If ***RNDUP** is specified, the **rem** value will be zero and should be set to ***none**. The value of C will be 1.84.

```
If B <> *Zeros
  Div A B C rem(*none) adj(*rndup)
Endif
```

Divide A by B giving C. **Adj(*RNDUP)** is not specified, and the **REM** parameter will be used. The value of C will be 1.83 and the value of D will be 2.6.

```
If B <> *Zeros
  Div A B C rem(D)
Endif
```

Divide E by B giving E. If ***RNDUP** is specified, the **rem** value will be zero and should be set to ***none**. The value of E will be 0.13 and there is no remainder. When all values (Factor 1, Factor 2 and result) are not used, the keywords **F2** and **Result** must be specified.

```
If B <> *Zeros
  Div F2(B) Result(E) rem(*none) adj(*rndup)
Endif
```

Divide E by B giving E. **Adj(*RNDUP)** is not specified, and the **REM** parameter will be used. The value of E will be 0.12 and the remainder will be 1.4. When all values (Factor 1, Factor 2 and result) are not used, the keywords **F2** and **Result** must be specified.

```
If B <> *Zeros
  Div F2(B) Result(E) rem(D)
Endif
```

DO

Begins a group of operations and indicates the number of times the group will be processed.

```
DO
FromVal (Numeric Variable)
ToVal (Numeric Variable)
Index (Character Expression)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *FLOAT | *FLOAT4 |
*FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *ONECHAR |
*PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

FromVal

Optional. Specify a starting value with zero decimal positions, using a numeric literal, named constant, or field name. If you do not specify this parameter, the starting value is 1.

ToVal

Optional. Specify the limit value with zero decimal positions, using a numeric field name, literal, or named constant. If you do not specify this parameter, the limit value is 1.

Index

Optional. Specify a numeric field name that will contain the current index value. The result must be large enough to contain the limit value plus the increment. If you do not specify an index field, one is generated for internal use. Any value in the index field is replaced by **FromVal** when the **DO** operation begins.

Type

Optional. Can be any of the types listed (Character, Packed or Zoned). *NONE is the default. See Type Parameter for further information.

Len

Optional. The **Len** parameter defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, TYPES of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

The **DO** operation begins a group of operations and indicates the number of times the group will be processed. To indicate the number of times the group of operations is to be processed, specify an index field (**Index**), a starting value (**FromVal**), and a limit value (**ToVal**). An associated ENDDO statement marks the end of the group.

The **DO** operation follows these 7 steps:

If the conditioning indicators on the **DO** statement line are satisfied, the **DO** operation is processed (step 2). If the indicators are not satisfied, control passes to the next operation to be processed following the associated ENDDO statement (step 7).

The starting value (**FromVal**) is moved to the **Index** field when the **DO** operation begins.

If the index value is greater than the limit value, control passes to the calculation operation following the associated ENDDO statement (step 7). Otherwise, control passes to the first operation after the **DO** statement (step 4).

Each of the operations in the **DO** group is processed.

If the conditioning indicators on the ENDDO statement are not satisfied, control passes to the calculation operation following the associated ENDDO statement (step 7). Otherwise, the ENDDO operation is processed (step 6).

The ENDDO operation is processed by adding the increment to the index field. Control passes to step 3. **Note** the conditioning indicators on the **DO** statement are not tested again (step 1) when control passes to step 3.)

The statement after the ENDDO statement is processed when the index value is greater than the limit value (step 3).

Example

```
/* The DO group is processed 10 times when indicator 17 is on; it stops running when the index value in
field X, the Index, is greater than the limit value (10) in ToVal. When the DO group stops running, control
passes to the operation immediately following the ENDDO operation. Because FromVal in the DO
operation is not specified, the starting value is 1. Because ToVal of the ENDDO operation is not
specified, the incrementing value is 1. */
```

```
  If *In17 = *On
```

```

Do 1 10 X
  Y = Y + 1
Enddo
Endif

```

/* The DO group can be processed 10 times. The DO group stops running when the index value in field X is greater than the limit value (20) in ToVal. The starting value of 2 is specified in FromVal of the DO operation, and the incrementing value of 2 is specified in ToVal of the ENDDO operation. */

```

Do 2 20 X
  Y = Y + 1
Enddo 2

```

See Also

DOUNTIL | ENDDO

DOEVENTS

Forces the event cycle to be run to process any pending events.

```
DOEVENTS
```

Remarks

DOEVENTS passes control to the event cycle, and control is not returned until the event cycle has finished processing the pending events in its queue.

Use **DOEVENTS** periodically to relinquish control to the operating system if Parts of your code take up a lot of processor time, so that events, such as keyboard input and mouse clicks, can be processed without significant delay.

In an executable, you have control of the CPU for that process. If you have one or more forms running that are awaiting events and you are running some code anywhere in the executable, the events will be held in the message queue until you stop processing or until you execute a **DOEVENTS**.

DOUNTIL

Begins a group of operations you want to process until a condition is True. The ENDDO command marks the end of the group.

```

DOUNTIL
Cond (Comparison Expression)

```

Parts

Cond

Required. The comparison expression is evaluated at the end of the execution of the group of commands between the **DOUNTIL** and the **ENDDO**. If the expression is False, the group of commands is executed again.

Remarks

DOUNTIL first evaluates the complete condition. If the condition is False, the group of statements up to the **ENDDO** will be executed and control passes back to the **DOUNTIL** command again. If the condition is True, control passes to the next operations to be processed following the associated ENDDO command.

See Also

DO | DOWHILE | ENDDO | ITERATE | LEAVE

DOWHILE

Begins a group of operations you want to process while the relationship 'xx' exists between the two operands.

```
DOWHILE  
Cond (Comparison Expression)
```

Parts

Cond

Required. The comparison expression is evaluated at the end of the execution of the group of commands between the **DOWHILE** and the **ENDDO**. If the expression is True, the group of commands is executed again.

Remarks

DOWHILE first evaluates the complete condition. If the condition is True, the group of statements up to the **ENDDO** will be executed and control passes back to the **DOWHILE** command again.

If the conditioning indicators on the **DOWHILE** command line are satisfied, the **DOWHILE** operation is processed. If the indicators are not satisfied, control passes to the next operation to be processed following the associated **ENDDO** command.

Example

/* The DOWHILE operation allows the operation within the DO group to be processed only if EarnedPremium is less than TotalPremium. If EarnedPremium is not less than TotalPremium, the program branches to the operation immediately following the ENDDO operation. If EarnedPremium is less than TotalPremium, the operation within the DO group is processed. */

```
DoWhile EarnedPremium < TotalPremium
```

/* The ENDDO operation causes the program to branch to the first DOWHILE operation, where a test is made to determine whether EarnedPremium is less than TotalPremium. This loop continues processing until EarnedPremium is equal to or greater than TotalPremium; then the program branches to the operation immediately following the ENDDO operation. */

```
EarnedPremium = DailyAvgPremium + EarnedPremium  
Read CustMaster Eof(*In99)  
Enddo
```

/* In this Example, multiple conditions are tested. The combined DOWHILE operation allows the operation within the DO group to be processed only while *IN99 is off due to "Not End of file" or EarnedPremium is less than TotalPremium or DrugFreePremium. If neither specified condition exists, the program branches to the operation immediately following the ENDDO operation. If either of the specified conditions exists, the operation after the ORLT operation is processed. */

```
Read CustMaster Eof(*In99)  
DoWhile *In99 = *Off *or EarnedPremium < TotalPremium +  
*or EarnedPremium < DrugFreePremium
```

/* The ENDDO operation causes the program to branch to the second DOWHILE operation where a test determines whether specified conditions exist. This loop continues until EarnedPremium is equal to or greater than TotalPremium and DrugFreePremium; then the program branches to the operation immediately following the ENDDO operation. */

```
EarnedPremium = DailyAvgPremium + EarnedPremium  
DrugFreePremium = DailyAvgPremium + DrugFreePremium  
Read CustMaster Eof(*In99)  
Enddo
```

/* The DOWHILE operation allows the operation within the DO group to be processed only if EarnedPremium is less than TotalPremium. If EarnedPremium is not less than TotalPremium, the program branches to the operation immediately following the ENDDO operation. If EarnedPremium is less than TotalPremium, the operation within the DO group is processed. */


```
DoWhile EarnedPremium < TotalPremium
```

/* The ENDDO operation causes the program to branch to the first DOWHILE operation, where a test is made to determine whether EarnedPremium is less than TotalPremium. This loop continues processing until EarnedPremium is equal to or greater than TotalPremium; then the program branches to the operation immediately following the ENDDO operation. */

```
EarnedPremium = DailyAvgPremium + EarnedPremium  
Read CustMaster Eof(*In99)  
Enddo
```

/* In this Example, multiple conditions are tested. The combined DOWHILE operation allows the operation within the DO group to be processed only while *IN99 is off due to "Not End of file" or EarnedPremium is less than TotalPremium or DrugFreePremium. If neither specified condition exists, the program branches to the operation immediately following the ENDDO operation. If either of the specified conditions exists, the operation after the ORLT operation is processed. */

```
Read CustMaster Eof(*In99)  
DoWhile *In99 = *Off *or EarnedPremium < TotalPremium +  
*or EarnedPremium < DrugFreePremium
```

/* The ENDDO operation causes the program to branch to the second DOWHILE operation where a test determines whether specified conditions exist. This loop continues until EarnedPremium is equal to or greater than TotalPremium and DrugFreePremium; then the program branches to the operation immediately following the ENDDO operation. */

```
EarnedPremium = DailyAvgPremium + EarnedPremium  
DrugFreePremium = DailyAvgPremium + DrugFreePremium  
Read CustMaster Eof(*In99)  
Enddo
```

See Also

DO | DOUNTIL | ENDDO

DSDUMP

Dumps the contents of the fields of the data structure assembled into a string.

```
DSDUMP  
Name (Data structure name)  
Target (Data structure contents)
```

Parts

Name

Optional. The name of the data structure to dump the contents from.

Target

Required. The string in which you will store the contents of the data structure.

Remarks

Make sure that all the numeric fields are positive and all characters are padded with the right amount of blanks.

The **Target** field length may be longer or shorter than the sum of the fields in the data structure. If the Target length is greater than the data structure length, the remaining target data is cleared to blanks. If the Target length is less than the data structure length, DSDump will complete without error when the Target field is filled.

Example

```

DclDs myDs AllowLoadDump(*yes)
DclDsFld myDSChar *Char Len(6)
DclDSFld myDSZoned *Zoned len(5, 2)
DclDSFld myDSIndicator *Ind
    myDSChar = "Hello"
    myDSZoned = 987.65
    myDSIndicator = *Off
DSDump Name(myDS) target(myString)
Result: myString := "Hello 987650"

```

See Also

DSLOAD

DSLOAD

Populates the fields of the **data structure** specified in **Name** using the **Source** parameter.

```

DSLOAD
Name (DS name)
Source (String)

```

Parts**Name**

Required. The name of the data structure in which to populate.

Source

Required. The **Source** parameter is parsed to match the lengths of each field in the data structure. The length of the string should match the total length of the fields in data structure.

Remarks

DSLOAD populates the fields of the data structure specified in **Name** using the **Source** parameter. The data structure should only have fixed length fields (*Char, *Zoned, *Date, *Time, *TimeStamp, *OneChar, *Indicator).

The Source field length must be at least as long as the sum of the fields in the data structure. If the Source length is greater than the data structure length, the excess data is ignored. If the Source length is less than the data structure length, an exception is thrown and the DSLOAD is unsuccessful.

Character fields should be padded with the proper amount of spaces and zoned fields should be padded with zeros to match the size of the data structure field. Make sure there are no special characters in the string for numbers (e.x. + or - etc...).

Example

```

DclDs myDs AllowLoadDump(*Yes)
DclDsFld myDSChar *Char Len(6)
DclDSFld myDSZoned *Zoned len(5, 2)
DclDSFld myDSIndicator *Ind
DUMPDS Name( myDS ) Source( "ABCDEF123451" )

```

Result:

```

myDSChar := ABCDEF
myDSZoned := 123.45
myDSIndicator := *On

```

See Also

DSDUMP

ELSE

Marks the beginning of a group of statements to be executed if the condition on a prior **IF** was false.

ELSE

Remarks

If the **IF** comparison is met, the calculations before **ELSE** are processed; otherwise, the calculations after **ELSE** are processed.

Use the **ENDIF** operation to end an **IF/ELSE** block of commands.

Conditioning indicator entries are not allowed.

Example

// If indicator 41 is On, process subroutine ProcessIfOn, else process subroutine ProcessIfOff.

```
If *In41 = *On
    Exsr ProcessIfOn
Else
    Exsr ProcessIfOff
Endif
```

// If Acode is equal to A and indicator 50 is On, the expression and Write operations are processed.

```
If Acode = 'A' *And *In50 = *On
    AcRec = 'A'
    Write Rcrsn
Endif
```

/* If the previous conditions were not met but Acode is equal to A, indicator 50 is off, and AcRec is equal to D, the following expression is processed. */

```
Else
    If Acode = 'A' *And *In50 = *On *And AcRec = 'D'
        AcRec = 'A'
    Endif
```

See Also

ENDIF | IF

ELSEIF

ELSEIF is similar to **ELSE**, except that it allows a condition to be tested if the previous conditional test (**IF** or **ELSEIF**) has failed. **ELSEIF** provides a more succinct way of doing nested **IFs**.

ELSEIF
Cond (Condition Expression)

Parts

Cond

Required. The condition to test.

Remarks

An **ELSEIF** can appear only within the context of an **IF/ENDIF**. If the previous **IF/ELSEIF** comparison is met, the calculations before **ELSEIF** are processed; otherwise, if the **ELSEIF** condition is met, the calculations after **ELSEIF** (and before the next **ELSEIF**, **ELSE** or **ENDIF**) are processed.

Conditioning indicator entries are not allowed.

Example

```

If ACode = 'A'
    ...
ElseIf ACode = 'B'
    ...
ElseIf AcRec = 'D'
    ...
Else
    ...
EndIf

```

See Also

ELSE | IF

END

Ends a CASE, DO, DOUNTIL, DOWHILE, IF, SELECT group.

```
END
```

Remarks

An **END** statement is required for each statement in a nested IF group.

A corresponding **END** or ENDIF statement is required for each IF-type statement.

A corresponding **END** or ENDDO statement is required for each DO-type statement.

See Also

CASE | DO | DOUNTIL | ENDIF | ENDDO | IF | SELECT

ENDCLASS

Marks the **end** of a **class** declaration.

```
ENDCLASS
```

Remarks

ENDCLASS must be the last statement in the class body.

For more information on classes, see Visual Studio's topics on Understanding Classes.

Example

See BEGCLASS Example.

See Also

BEGCLASS | Understanding Classes

ENDCONSTRUCTOR

Defines the **end** of a **constructor** declaration.

```
ENDCONSTRUCTOR
```

Remarks

ENDCONSTRUCTOR indicates the end of a constructor routine started with BEGCONSTRUCTOR.

For more conceptual information on using constructors in Visual Basic, see Constructors, Instance Constructors and Shared Constructors in the Visual Basic Language Specification.

Example

See BEGCONSTRUCTOR Example.

See Also

BEGCONSTRUCTOR | Using Constructors

ENDCS

Ends a CASE operation.

```
ENDCS
```

Remarks

Conditioning and resulting indicators are not allowed.

Example

```
Case (OnHand = OnOrder) Sr(CutOrder)
  CaseDft Sr(ReviewOrder)
  //The Endcs operation denotes the end of the Case group.
Endcs
```

ENDDO

Ends a DO, DOUNTIL or DOWHILE group.

```
ENDDO
```

```
Incr (Numeric Expression)
```

Parts

Incr

Optional. Only valid if ending a **DO** loop. The loop index will be incremented by the increment value each time the group of commands is executed. The **DO** group will never end if a value is negative.

Remarks

Conditioning indicators are optional.

Resulting indicators are not allowed.

If you do increment the loop with a negative value, keep in mind that **INCR** of the **DO** operation does not work in the opposite way it does with a positive value. That is, you can't have a larger number in the first value (say 45), and 1 in the second, and expect the loop to stop.

In a decrementing loop, the second value must be equal to or larger than the first. After the loop starts, the second value is no longer a "factor" in the loop, which is why you must check your index value and stop the loop manually when the index reaches the desired value.

Example

/* The Do group can be processed 10 times. The Do group stops running when the index value in field X is greater than the limit value (20) in Factor 2. The starting value of 2 is specified in the Do operation, and the incrementing value of 2 is specified in the Enddo operation. */

```
Do 2 20 X
  Y = Y + 1
Enddo 2
```

// In this Example of DoUntil, one of several conditions are met with the Enddo.

```
DoUntil *In99 = *On *and ManualPremium = TotalPremium +
  *Or EarnedPremium = 1500
  ManualPremium = ManualPremium + AvgDailyPremium
  EarnedPremium = EarnedPremium + ManualPremium
  Read CustMaster Eof(*In99)
Enddo
```

// In this Example of DoWhile, one of several conditions are met with the Enddo.

```
DoWhile *In99 = *Off *or EarnedPremium < TotalPremium +
  *or EarnedPremium < DrugFreePremium
  EarnedPremium = DailyAvgPremium + EarnedPremium
  DrugFreePremium = DailyAvgPremium + DrugFreePremium
  Read CustMaster Eof(*In99)
Enddo
```

See Also

DO | DOUNTIL | DOWHILE | END

ENDENUM

Finishes an Enumeration definition (BEGENUM).

```
ENDENUM
```

Remarks

An **ENDENUM** finishes an Enumeration definition.

Example

```
BegEnum myEnum basetype(*byte )
  DclEnumFld q4 value(4) // <--- the default value is 0
  DclEnumFld q5 // <--- this will be 5, which is 1 plus the
previous one
EndEnum
```

See Also

BEGENUM | Enumeration Definition

ENDFOR

Ends a FOR/ENDFOR or FOREACH/ENDFOR loop.

```
ENDFOR
```

Remarks

FOR/ENDFOR blocks mimic the RPG ILE free-format **FOR** opcode. The **FOR** operation is performed as follows:

If specified, the initial value is assigned to the index name. Otherwise, the index name retains the same value it had before the start of the loop.

If specified, the limit value is evaluated and compared to the index name. If no limit value is specified, the loop repeats indefinitely until it encounters a statement that exits the loop (such as a LEAVE or GOTO) or that ends the program or procedure (such as a LEAVESR).

If the **To** clause is specified and the index name value is greater than the limit value, control passes to the first statement following the **ENDFOR** command. If **DownTo** is specified and the index name is less than the limit value, control passes to the first statement after the **ENDFOR**.

The operations in the FOR group are processed.

If the increment value is specified(*By*), it is evaluated. Otherwise, it defaults to 1.

The increment value is either added to (for *To*) or subtracted from (for *DownTo*) the index name. Control passes to step 2.

The statement after the END or **ENDFOR** command is processed when the index value is greater than (for *To*) or less than (for *DownTo*) the limit value (step 2), or when the index value overflows.

Note: If the **FOR** loop is performed *n* times, the limit value is evaluated *n*+1 times and the increment value is evaluated *n* times. This can be important if the limit value or increment value is complex and time-consuming to evaluate, or if the limit value or increment value contains calls to subprocedures with side-effects. If multiple evaluation of the limit or increment is not desired, calculate the values in temporaries before the **FOR** loop and use the temporaries in the **FOR** loop.

See Also

FOR | FOREACH

ENDFUNC

ENDFUNC defines the end of a function.

```
ENDFUNC
```

Remarks

ENDFUNC must be the last statement within a subroutine that is preceded with a BEGFUNC and also contains a LEAVESR.

See Also

BEGFUNC | LEAVESR

ENDGET - End a GET Procedure

Terminates a BEGGET procedure.

```
ENDGET
```

Remarks

An **ENDGET** routine must begin with a **BEGGET**.

Example

```
BEGPROP Prop1 Type(*Integer) Len(4) Access(*Public)
  BEGGET
    LeaveSR 5
  ENDGET
  BEGSET NewValName (MyName)
    C = A * B
    zero[1] = D
  ENDSET
ENDPROP
```

See Also

BEGGET | BEGSET | ENDSET

ENDIF

Ends an IF, or ELSE group.

```
ENDIF
```

Remarks

ENDIF must be the last statement within a subroutine that is preceded with an **IF**, or **ELSE** group.

Example

```
If *In17 = *On
  Do 1 10 X
    Y = Y + 1
  Enddo
Endif
```

See Also

ELSE | IF

ENDINTERFACE - End a BEGININTERFACE Procedure

Ends a BEGININTERFACE interface definition.

```
ENDINTERFACE
```

Remarks

An **ENDINTERFACE** marks the end of a **BEGININTERFACE** interface definition.

Example

```
//Define an Interface
BeginInterface myInterface
  BegSr Customer
    DclSrParm Name *integer4
  EndSr
  BegFunc Function Type( *integer4 )
  EndFunc
  DclEvent EVENT type( System.EventHandler )
EndInterface

// Interface Usage
  BegClass Overloaded Access (*Public) Implements( myInterface )
  BegSr First Access (*Public) implements ( myInterface.Customer )
    DclSrParm theParm *integer4
  EndSr
```

See Also

BEGININTERFACE

ENDPROP

ENDPROP identifies the ending of a property routine.

ENDPROP

Remarks

ENDPROP must be the last statement within a property routine that is preceded with a **BEGPROP**.

Note that if the **BEGPROP** is a (*GET) routine, then your subroutine must also contain a **LEAVESR** before the **ENDPROP**, as shown in the Example below.

Example

```
DclFld_Firstname    *string    INZ('')
DclFld_Lastname     *string    INZ(' ')
BegProp Firstname   *string    Access( *public )
  BegSet
    _FirstName = *Propval
  EndSet
  BegGet
    LeaveSr _FirstName
  EndGet
EndProp
BegProp Lastname    *string    Access( *public )
  BegSet NewValname( Lname )
    _LastName = Lname
  EndSet
  BegGet
    LeaveSr _LastName
  EndGet
EndProp
// Read only property
BegProp FullName Type( *string ) Access( *public )
  BegGet
    LeaveSR Value( _FirstName + " " + _LastName )
  EndGet
EndProp
```

See Also

BEGPROP | LEAVESR | ENDGET | ENDSET

ENDSET - End a BEGSET Procedure

Terminates a **BEGSET** property procedure.

ENDSET

Remarks

An **ENDSET** routine must begin with a **BEGSET**.

Example

```
DclFld_Firstname    *string    INZ('')
DclFld_Lastname     *string    INZ(' ')
BegProp Firstname   *string    Access( *public )
  BegSet
    _FirstName = *Propval
  EndSet
  BegGet
    LeaveSr _FirstName
  EndGet
EndProp
```

```

BegProp Lastname  *string  Access( *public )
  BegSet NewValname( Lname )
    _LastName = Lname
  EndSet
  BegGet
    LeaveSr _LastName
  EndGet
EndProp
// Read only property
BegProp FullName Type( *string )  Access( *public )
  BegGet
    LeaveSR Value( FirstName + " " + LastName )
  EndGet
EndProp

```

See Also

BEGGET | BEGSET | ENDGET

ENDSL

Ends a SELECT group.

ENDSL

Remarks

Conditioning indicators are not allowed.

Example

```

Select
  When X = 1
    B = A + B
  Exsr UpdateMaster
    When X < 10 *and Y = 2
      B = B - A
    Other
      Exsr ErrorRoutine
EndSL

```

See Also

SELECT | WHEN

ENDSR

Defines the end of a subroutine or event-handler.

ENDSR

Remarks

ENDSR must be the last statement in the subroutine.

Conditioning indicator entries are not allowed.

The **ENDSR** causes a subroutine to branch back to the statement immediately following the EXSR operation, unless the subroutine is an event handler, in which case control goes back to the event cycle.

Example

//Subroutine to get customer details record with EndsR used to define an exit.

```
BegSR GetDetail
ReadRand CustMaster CustNumber NotFnd(*IN99)
  If *IN99
    CustNbr = 0
    CustName = ''
  Endif
EndSR
```

See Also

BEGSR | EXSR | HIDE | SHOW | UNLOAD

ENDTPM

ENDTPM is used to end the transaction processing monitor on the specified data source. If there are uncommitted changes, the changes are rolled back.

```
ENDTPM
Db (Database Object)
Err (Indicator Variable | *EXTENDED)
```

Parts

Db

Required. The database connection for which transaction processing is to be ended. Database Objects are defined with the DCLDB command.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

See Also

COMMIT | DCLDB | ROLLBACK | STARTTPM | Using Commitment Control

ENDTRY - End a SET Procedure

Terminates the **TRY...CATCH...FINALLY** procedure.

```
ENDTRY
```

Remarks

An **ENDTRY** routine must begin with a TRY.

Example

```
BEGSR TryCatch access(*public)
  Dclfld myFld Type(System.Windows.Forms.Button)
  Dclfld myInt *integer 4
  TRY
    myFld.Show()
```

```

        myInt = myInt / 0
    CATCH name( z ) type( System.DivideByZeroException )
        msgbox msg( "by 0" ) Title( z.Message )      CATCH name( e ) type(
System.Exception )      msgbox msg( "You can't do that!!!" ) Title(
e.Message )      FINALLY
        msgbox msg( "wow, this works...")
    ENDTRY
    msgbox "All is well!"

ENDSR

```

See Also

CATCH | FINALLY | TRY | TRY...CATCH...FINALLY

ENDWITH

Ends a WITH block.

ENDWITH

Remarks

An **ENDWITH** marks the end of a **WITH** block.

Example

// You used to code like this:

```

txtDBNme.Text = My_Production_Database.DBname
chkIsOpen.Checked = My_Production_Database.Isopen
txtLabel.Text = My_Production_Database.Label
txtServer.Text = My_Production_Database.Server
txtPort.Text = My_Production_Database.Port
txtUserId.Text = My_Production_Database.User
txtPlatform.Text = My_Production_Database.Platform

```

// Now you can code the following using WITH.

```

With My_Production_Database
txtDBNme.Text = .DBname
    chkIsOpen.Checked = .Isopen
    txtLabel.Text = .Label
    txtServer.Text = .Server
    txtPort.Text = .Port
    txtUserId.Text = .User
    txtPlatform.Text = .Platform
EndWith

```

See Also

WITH

ENTERLOCK

Locks an object for serializing access in multithreaded applications.

ENTERLOCK

Object (expression)

Parts

Object

Optional. If given, Object must be an expression that evaluates to an object (an instance of a reference type). Instances of value types are not allowed. The default value depends on where in the program the **EnterLock** appears.

- If it's in the body of an instance method, then the default value is ***This**.
- If it's in the body of a shared method, then the default value is ***typeof(<current class>)**. This ensures serialization at a coarse level.

Remarks

ENTERLOCK routines allow a programmer to protect segments of code for serial access in multithreaded applications.

An **ENTERLOCK** routines must end with an EXITLOCK.

See Also

EXITLOCK

EVAL

Evaluates an expression. Is useful for trapping errors.

```
EVAL
F2 (Expression, Assignment Expression)
Adj (*TRUNC | *RNDUP)
Err (Indicator Variable | *EXTENDED)
```

Parts

F2

Required. Must contain an expression or assignment expression.

Adj

Optional. Indicates half-adjust for numeric expressions. Use **Adj** (*RNDUP) if you wish to round up or **Adj** (*TRUNC) if you wish to truncate the value of the Result. *TRUNC is the default.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Use **EVAL** to capture errors in an expression and be notified by the setting of the error indicator, such as:

```
Eval F2(object.method(x, y, z)) Err (*IN88)
```

To report or analyze the error, use the *Err built-in object to retrieve the Error Number, Error Type, Description and Source. For Example, after your **EVAL**, you could add:

```
msgbox msg(*err.Description + " " + *err.Source)
```

Examples of errors are:

Divide by zero.

Error on a method call.

Example

```

EVAL X=(A+B)/C 05 // Evaluate an expression to determine if the user pressed
Cancel
Eval F2 (MAPIMessages1.SHOW(0)) Err (*IN01)
  IF *IN01
    MsgBox ('User pressed Cancel')
  EndIf

```

EXFMT

In AVR, **EXFMT** shows the form instance as a modal dialog box.

In Monarch, EXFMT redirects an application to a workstation format web page.

```

EXFMT
Fmt (Form Name)
Err (Indicator Variable | *EXTENDED)

```

Parts

Fmt

Required. In AVR, this contains the name of the form to display. In Monarch, this contains the name of the web page to be redirected to.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

In AVR, You can use this command to display a modal Form in your application. When this command is executed, the code following it is not executed until after the Form is closed or Hidden.

In Monarch, you can use this execute format command to redirect an application to a workstation format web page.

AVR Example

```

BeginClass Form1 Extends(System.Windows.Forms.Form) Access(*Public)
  DeclFld F2 Type( Form2 )
  BeginSr EXFMTExample Access(*Private) Event(*this.btnEXFMT.Click)
    DeclSrParm sender Type(*Object)
    DeclSrParm e Type(System.EventArgs)
    F2 = *new Form2() // create instance of Form2
    Exfmt F2 // same as F2.ShowDialog()
  EndSr
EndClass

```

Monarch Example

```

DeclWorkStnFile MyWorkStnFile DspFile("./MyWorkStnFile.aspx") ImpOpen( *No )
Subfile( sfl,r )
ExFmt sftctl // Executes the stlctl format as described by
MyWorkStnFile.aspx file.

```

See Also

Control.Hide method | Form.ShowDialog method | Control.Show method | HIDE | SHOW |
DclWorkStnFile

EXITAPP

EXITAPP exits the program.

```
ExitAPP  
ExitCode (Numeric Expression)
```

Parts

ExitCode

Optional. Specifies an exit code. If **ExitCode** is not specified, the default exit code is 0.

Remarks

EXITAPP causes the program to terminate.

EXITLOCK - End an ENTERLOCK Procedure

Ends the ENTERLOCK procedure.

```
EXITLOCK
```

Remarks

An **EXITLOCK** routine must begin with an **ENTERLOCK**.

See Also

ENTERLOCK

EXSR

Executes a subroutine in the local program or in the main program.

```
EXSR  
Sr (Subroutine Name)  
ParmList (*NONE | Parameter Name List)
```

Parts

Sr

Required. Specifies the name of the subroutine. If the subroutine name specified is defined in the current program, the current program's subroutine will be called. Otherwise, if a subroutine is found in the main program, the main program's subroutine will be called.

ParmList

Optional. Allows you to declare parameters by specifying the name of a parameter list. The default is *NONE. You can also define parameters as noted in the paragraph below.

Remarks

If the subroutine has parameters (declared by using the DCLSRPARM command after the BEGSR command), you **must** pass the correct number of parameters by using the DCLPARM commands after the **EXSR**. See **DCLPARM** for an Example of this.

Example

```

DoWhile *In03 = *Off
  Exfmt frmListName
  Select
    When Option = Add
      Exsr AddRecord
    When Option = Delete
      Exsr DeleteRecord
  Endsl
Enddo

```

See Also

BEGSR | CAB | DCLSRPARM | DOUNTIL | DOWHILE | IF | WHEN | Comparison Operators

EXTRACT

The **EXTRACT** operation code will return one of the following:

The hours, minutes or seconds part of a time or timestamp field.

The year, month or day part of a date or timestamp field.

The microseconds part of the timestamp field to the field specified in **Result**.

EXTRACT
F2 (Date Time Timestamp Array Array element)
F2Fmt (*DAYS, *MONTHS, *YEARS, *MSECONDS, *SECONDS, *MINUTES, *HOURS)
Result (Numeric Variable)
Type (*NONE *BINARY *BOOLEAN *BYTE *CHAR *DECIMAL *FLOAT *FLOAT4 *FLOAT8 *IND *INTEGER *INTEGER2 *INTEGER4 *INTEGER8 *ONECHAR *PACKED *ZONED)
Len (Length Integer, [Decimal Integer])
Err (Indicator Variable *EXTENDED)

Parts

F2

Required. Contains the Date, Time or Timestamp from which the information is required, followed by the Duration Code (**F2FMT**).

F2Fmt

Required. Can be a field, subfield, table element, or array element. The duration code must be consistent with the data type of **F2**.

Result

Required. Can be any numeric or character field, subfield, array/table element. **Result** is cleared before the extracted data is assigned. For a character **Result**, the data is put left adjusted into **Result**.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Please note that **EXTRACT** does not extract the century, so you would have to hardcode a value for the century. When using the **EXTRACT** operation with a Julian Date (format *JUL), specifying a duration code of *D will return the day of the month, specifying *M will return the month of the year. If you require the day and month to be in the 3-digit format, you can use a data structure to overlay the field, and extract the three digits.

Example

```
EXTRACT StartDate: *M StartMonth 2 0
EXTRACT StartDate: *D StartDay 2
```

See Also

ADDDUR | Duration Codes | SUBDUR

FEOD

Causes a file to start printing, mainly print files.

```
FEOD
File (file name)
Err (Indicator Variable | *EXTENDED)
Reset (Indicator Expression | *NO | *YES)
```

Parts

File

Required. Specifies the name of the file to print.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.;

Reset

Optional. Only valid for print files. If **Reset** is *ON, *YES, or has an expression that evaluates to *ON, the printer parameters are reset using the variables and values specified in DCLFILECONT for this file. Changing those values during **FEOD** is faster than closing and re-opening the file. One possible application is changing the number of copies of the next document sent to the printer.

With **Reset** you may only change the following DCLPRINTFILE parameters: Copies, ReportName, Duplex, Scale, Quality, UseColor, and Y-Resolution (See PrintFile Members).

Remarks

FEOD causes a document to start printing by first flushing the current page, then the document starts printing immediately without closing unless as noted below.

If the PrintFile.Preview is "PrintPreview", the print driver commands are saved to a temporary file on the client machine. When the **FEOD** command is issued, a preview of the printed output is displayed (nothing is yet sent to the printer). While previewing, the user has the option to then send the output to the target printer.

See Also

%ERROR | CLOSE | DCLFILECONT | DCLPRINTFILE | PrintFile Class | PrintFile Members | PrintFile.Preview Property

FINALLY

You can use the TRY...CATCH...FINALLY statements for structured exception handling. This allows you to execute a particular block of statements if a specified exception occurs while your code is running. When this happens, the code is said to THROW the exception, and you *catch* it with the appropriate **CATCH** command.

FINALLY

Remarks

The **TRY** statements are executed first. If no exception occurs, execution continues after the **ENDTRY**.

If an exception occurs, then the systems tries to match the exception to the exception types in the **FINALLY** commands. The first one that matches causes the exception to be 'trapped' and the catch statements in the matched **FINALLY** to be executed. If the exception is trapped by a **FINALLY** then after the catch statements are executed, control goes to the next statement after the **ENDTRY**.

If a **FINALLY** command is present, then the finally statements will always be executed at the end of the **TRY/CATCH** sequence or **TRY**/untrapped exception, no matter what.

Rules

The **only** branching commands allowed inside a **TRY/CATCH** block are **ITERATE**, and **LEAVESR**. Any other branching command (**CAB**, **GOTO**, etc.) is not allowed in this context. It is also illegal to branch out of a **FINALLY** clause (i.e., you cannot use any of the branching commands (**ITERATE**, **CAB**, **LEAVESR**, **GOTO**, etc) in the body of a **FINALLY** clause.

The exception Type **MUST** derive from **System.Exception** (the mother of all exceptions). A **CATCH** with a **System.Exception** type will trap **all** exceptions.

If you have several **CATCH** commands, the **Name** must be different at this time.

Example

```
BEGSR TryCatch access(*public)
  Dclfld myFld Type(System.Windows.Forms.Button)
  Dclfld myInt *integer 4
  TRY
    myFld.Show()
    myInt = myInt / 0
  CATCH name( z ) type( System.DivideByZeroException )
    msgbox msg( "by 0" ) Title( z.Message )
  CATCH name( e ) type( System.Exception )
    msgbox msg( "You can't do that!!!" ) Title( e.Message )
  FINALLY
    msgbox "Wow, this works..."
  ENDTRY
  msgbox "All is well!"
ENDSR
```

See Also

CATCH | ENDTRY

FOR

Starts a FOR/ENDFOR loop.

```
FOR
Index (index expression)
To (Expression)
DownTo (Expression)
By (Expression)
```

Parts

Index

Required. The index expression is either an assignment expression (<field> = <expression>), or a field (<field>). <field> represents a valid numeric field with 0 decimal positions (any of the integral types is allowed). <expression>, if given, represents the initial value of the field.

To

Optional. Represents the upper limit of an upward-going iteration.

DownTo

Optional. Represents the lower limit of a downward-going iteration.

By

Optional. Represents the amount to be added or subtracted from the field given in the INDEX keyword. It defaults to 1.

Remarks

FOR/ENDFOR blocks mimic the RPG ILE free-format **FOR** opcode. The **FOR** operation is performed as follows:

If specified, the initial value is assigned to the index name. Otherwise, the index name retains the same value it had before the start of the loop.

If specified, the limit value is evaluated and compared to the index name. If no limit value is specified, the loop repeats indefinitely until it encounters a statement that exits the loop (such as a LEAVE or GOTO) or that ends the program or procedure (such as a LEAVESR).

If the **To** clause is specified and the index name value is greater than the limit value, control passes to the first statement following the **ENDFOR** command. If **DownTo** is specified and the index name is less than the limit value, control passes to the first statement after the **ENDFOR**.

The operations in the **FOR** group are processed.

If the increment value is specified, it is evaluated. Otherwise, it defaults to 1.

The increment value is either added to (for **To**) or subtracted from (for **DownTo**) the index name. Control passes to step 2.

The statement after the **END** or **ENDFOR** command is processed when the index value is greater than (for **To**) or less than (for **DownTo**) the limit value (step 2), or when the index value overflows.

Note: If the **FOR** loop is performed n times, the limit value is evaluated $n+1$ times and the increment value is evaluated n times. This can be important if the limit value or increment value is complex and time-consuming to evaluate, or if the limit value or increment value contains calls to subprocedures with side-effects. If multiple evaluation of the limit or increment is not desired, calculate the values in temporaries before the **FOR** loop and use the temporaries in the **FOR** loop.

See Also

ENDFOR

FOREACH - Repeat a Group of Statements

Iterates over a group of statements, once for each element in an array or a collection. **FOREACH** is used to iterate through the array or collection to get the desired information, but should not be used to change the contents of the array or collection to avoid unpredictable side effects. The statements that constitute the body of the **FOREACH** are those that follow the **FOREACH** declaration up to the first **ENDFOR** found.

```
FOREACH
Name (Field name)
Collection (Expression)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
*FLOAT4 | *FLOAT8 | *FORM | *IND |
    *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *OBJECT | *ONECHAR |
*PACKED | *STRING | *TIME | *TIMESTAMP | *ZONED)
Len (Integer Literal, Integer Literal)
```

Parts

Name

Required. The name of the field that holds the current iteration element of the array or collection. Its type must match the types of the elements in the array or collection.

Collection

Required. Expression that evaluates to a collection (an object whose type implements **IEnumerable**) or an array.

Type

Optional. **Type** can be any of the ******* Types list. If **Type** is not specified, the **Len** specified determines the default.

Type cannot be a named constant or a literal.

If **Type** is an array, array element, or a table name, the attributes of an element of the array or table are used to define the field.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer, and *ZONED is assumed if there are two integers.

Len

Optional. **Len** specifies the length of the field. If **Len** is decimal e.g. Len (3,1) *ZONED is assigned as the default. If **Len** is character e.g. Len (3), *CHAR is assigned as the default.

If **Type** is packed, then the length is not the number of bytes the packed field fits into, but rather the logical digit length of the field. For Example, if you want a 7 digit packed with 2 decimals, code **Len** (7,2).

Remarks

The embedded statements continue to execute for each element in the array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the **FOREACH/ENDFOR** block.

Example

In this Example, we iterate over the Controls collection of a GroupBox control, finding all the buttons and setting their captions.

```
ForEach name( mybtn ) collection( groupBox1.Controls ) type( Button )  
    myBtn.Text = "This is Button " + count.ToString()  
    count = count + 1  
EndFor
```

See Also

DCLARRAY | ENDFOR

GOTO

Allows commands to be skipped by instructing the program to go to another command within the program.

```
GOTO  
Label (label name)
```

Parts

Label

Required. Names the destination of a **GOTO** operation.

Remarks

The **GOTO** operation allows calculation operations to be skipped by instructing the program to go to (or branch to) another calculation operation in the program. A TAG operation names the destination of a **GOTO** operation.

Use a GOTO operation to specify a branch:

- To a previous or a succeeding specification line.
- From a detail calculation line to another detail calculation line.
- From a total calculation line to another total calculation line.
- From a detail calculation line to a total calculation line.
- From a subroutine to a detail calculation line or to a total calculation line.

A **GOTO** operation outside a subroutine cannot specify a branch to a **TAG** or **ENDSR** operation within that subroutine.

The **GOTO** command is **not** allowed within a TRY/CATCH block.

See Also

ENDSR | Labels | TAG

HIDE

Hides a form from the users' view. The form and its associated code are still active. Although it cannot respond to user events while hidden, it continues to respond to hardware events such as a Timer.

```
HIDE
Form (form variable)
```

Parts

Form

Required. Contains the name of the form to hide.

Remarks

If a form is loaded, **HIDE** will hide a form from the users' view, if not already hidden. All files declared by the program views remain open, and all global variables remain intact. Although it cannot respond to user events while hidden, it continues to respond to hardware events such as timer. A hidden form is reactivated with the SHOW or EXFMT opcodes.

If the form is not loaded, **HIDE** will load the form, causing the main-line code of the program to be executed, and then the form will be hidden.

When a form (from **EXFMT** or **CALL**) is hidden with the **HIDE** op-code, it will return control to the statement following the **EXFMT** or **CALL** to that form. If the form was **EXFMTed**, it will continue running. If the form was **CALLED**, it will be unloaded.

The form may "flash" if the Visible property is true at design-time. To avoid a "flash", set the Visible property to false in the property window.

See Also

*This | CALL | EXFMT | SHOW | UNLOAD

IF

IF processes a group of calculations if a comparison expression evaluation is True.

```
IF
Cond (Comparison Expression)
```

Parts

Cond

Required. The comparison expression is evaluated at the end of the execution of the group of commands between the ENDIF, and END. If the expression is true, the group of commands is executed again.

Remarks

An **ENDIF**, or **END** is required to end the group of commands after the **IF** command.

In Visual RPG, **IF** was expanded to include an **ELSEIF**.

Example

```
If ACode = 'A'
...
ElseIf ACode = 'B'
...
ElseIf AcRec = 'D'
...
Else
...
EndIf
```

See Also

ELSEIF | END | ENDIF

ITERATE

ITERATE transfers control from within a loop to the **END** statement of the loop.

```
ITERATE
```

Remarks

ITERATE can be used in DO, DOUNTIL, and DOWHILE loops to transfer control to a loop **END** statement.

See Also

DO | DOUNTIL | DOWHILE | LEAVE | END

LEAVE

Control is transferred from within a loop to the statement following the **END** operation.

```
LEAVE
```

Remarks

LEAVE causes control to transfer "outwards" by one level only in nested loops. **LEAVE** is not allowed without a **DO** group.

LEAVE is similar to the **ITERATE** operation, however **ITERATE** transfers control to the loop's **END** command.

See Also

DO | DOUNTIL | DOWHILE | END | ITERATE

LEAVESR

LEAVESR transfers out of the current subroutine, property or function.

```
LEAVESR
Value (value to return)
```

Parts**Value**

Optional. However, the parameter is required when used with **BEGFUNC** and also when used with **BEGPROP**'s **BEGGET**.

Remarks

LEAVESR causes control to transfer out of the current procedure.

LEAVESR is allowed inside of a TRY/CATCH block.

LEAVESR must be contained within the *GET routine of the BEGPROP/BEGFUNC and ENDPROP/ENDFUNC groups.

Visual RPG allows you to name a control property for the **LeaveSR Value** parameter. However, you must not assume the default property for the control. Instead, you must specifically code the desired property.

Failure to specify the property can cause your application to crash.

The only branching commands allowed inside a TRY/CATCH block are LEAVE, ITERATE, and LEAVESR. Any other branching command (CAB, GOTO, etc.) is not allowed in this context. It is also illegal to branch out of a FINALLY clause (i.e., you cannot use any of the branching commands (ITERATE, CAB, LEAVESR, GOTO, etc) in the body of a FINALLY clause.

If you do not code a specific property for the control, Visual RPG returns the control's IDispatch regardless of the data type of the **BEGPROP** or **BEGFUNC**.

Example

Examples within BegFunc:

```
BegFunc myFunction type(*string)
  LeaveSR Value ( Label1.Caption )
EndFunc
```

When a BegFunc or BegProp type is *object, then you are required to omit the property.

```
BegFunc myFunction type(*Object)
  LeaveSR Value ( Label1 )
EndFunc
```

Example within BegProp:

```
BegProp Name (MyProp) event (*GET) type (*PACKED) Len (4,0)
  LeaveSR ValueSet
Endprop
```

See Also

BEGFUNC | BEGPROP | BEGSR | CATCH | ENDFUNC | ENDPROP | ENDSR | TRY

LOADPICTURE

Loads a graphic into the **Image** property of a **PictureBox** control.

```
LOADPICTURE
File (Character Expression)
Target (Character Expression)
Err (Indicator Variable | *Extended)
```

Parts

File

Required. Must contain the path of the graphic to load, i.e.,

```
FILE ('C:\program Files\Graphics\Icon.ICO').
```

Target

Required. Must be an **Image** property of a control.

If you are loading a picture into a particular panel on a **Panel**, you must also specify the index of the panel in which the picture is to be loaded, i.e.,

```
TARGET (Panel1.Panels[0].Image) .
```

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - the specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Graphic files include bitmap (.BMP, *DIB) files, icon (.ICO, .CUR) files, metafile (.WMF, .EMF) and .JPG files.

Example

```
// Loads a picture into an image control.
LoadPicture File('C:\Program Files\Graphics\Icon.ICO')
Target(PictureBox1.Image) Err(*IN01)

// Loads a picture into the first panel of a Panel control.
LoadPicture File('C:\Program Files\Asna\Avr35\Diskette1.bmp')
Target(Panel1.Panels[0].Picture)
```

LOOKUP

Searches for a particular element in an array or table.

LOOKUP
Source (Character Expression)
Table (table name Array)
Result (Character Expression)
EndIndex (Numeric Variable)
Hi (Indicator Variable)
Lo (Indicator Variable)
Eq (Indicator Variable)

Parts**Source**

Required. Contains the search argument or data for which you want to find a match in the single-dimension array or table named. If **Source** is a table name, the search argument used is the element of the table last selected in a **LOOKUP** operation. The array or table to be searched is specified in **Table**.

Table

Required. Specifies the array or table to be searched. If the table has not been selected before, the first element in the table is used.

Result

Optional. Must be a table name if **Table** contains a table name. Otherwise, the result should be left blank.

FndIndex

Optional. Returns the index value of the element located. The value is relative to zero. The index is set equal to zero if the search is unsuccessful.

Hi, Lo, Eq

Optional. At least one conditional indicator must be set or an error message will occur. In addition, the **Hi** and **Lo** indicators cannot be specified simultaneously. The compiler assumes a sorted sequenced array or table when a high or low indicator is specified for the LOOKUP operation. The **LOOKUP** operation searches for an entry that satisfies the low/equal or high/equal condition with equal given priority.

Hi - Instructs the program to find the entry that is nearest to, yet higher in sequence than, the search argument. When a higher entry is found, the high indicator is set on. For Example, if an ascending array contains the values A B C C C D E and the search argument is B, then the first C will satisfy the search. If a descending array contains E D C C C B A and the search argument is B, then the last C will satisfy the search. If an entry higher than the search argument is not found in the array or table, the search is unsuccessful.

Lo - Instructs the program to find the entry that is nearest to, yet lower in sequence than, the search argument. When a lower entry is found, the low indicator is set on. For Example, if an ascending array contains the values A B C C C D E and the search argument is D, then the last C will satisfy the search. If a descending array contains E D C C C B A and the search argument is D, then the first C will satisfy the search. If an entry lower than the search argument is not found in the array or table, the search is unsuccessful.

Eq - Instructs the program to find the entry equal to the search argument. The first equal entry found sets the equal indicator on. If an entry equal to the search argument is not found, the search is unsuccessful.

See Also

%LOOKUP | %TLOOKUP

MOVE

Transfers characters from **Source** to the **Target**, right-adjusted with optional blanks padded on the left side of the **Target**.

```

MOVE
Source (Character Expression)
Target (Character Expression)
Fill (*NOPAD | *PAD)
Pos (Indicator Variable)
Neg (Indicator Variable)
Boz (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
TimFmt (*NONE | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
        *YMD)

```

Parts

Source

Required. If the **Source** parameter is longer than the **Target**, the excess leftmost characters of **Source** are not moved. If the **Target** is longer than **Source**, the excess leftmost characters are left untouched, unless padding is specified.

Target

Required. Resulting indicators cannot be specified if the **Target** is an array. They can be specified if the **Target** is a non-array field or array element.

Fill

Optional. Specifies if you want to fill, or pad the **Target** (Target) with blanks. The default is *NOPAD.

*NOPAD indicates that the **Target** will not be padded with blanks.

*PAD indicates that the **Target** will be padded with blanks.

Pos

Optional. Turned on if the value of **Target** is a positive number.

Neg

Optional. Turned on if the value of **Target** is a negative number.

Boz

Optional. Turned on if the value of **Target** is a zero or blank number.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Type *CHAR, *PACKED, and *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

TimFmt

Optional. Specifies the default time format for time fields. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Remarks

Moving Text from a Control to a Field

When moving text from a control into a field, be aware that the contents is a null-terminated string and does not assume the field length. **MOVE** and **MOVEL** operations will move only the number of characters in the string into the field, which may result in extraneous data remaining in the field. Be sure to use Fill(*Pad) in the operation to prevent this.

Moving Date, Time or Timestamp fields

TimFmt must be blank unless either the source or the target is a character or numeric field. Otherwise, **TimFmt** contains the date or time format compatible with the character or numeric field that is the source or target of the operation.

If Source is *DATE and the result is a Date field, **TimFmt** is not required. If **TimFmt** contains a date format it must be compatible with the format of *DATE. See Moving Date-Time Data for more information.

Moving a Character Field to a Date-Time Field

Factor-1#	Factor-2#	Result-Field#	
Entry#	(Character)#	Value#	DTZType#
*MDY#	11/19/75#	75/323#	D(*JUL)#
*JUL#	92/114#	23/04/92#	D(*DMY)#
*YMD#	14/01/28#	01/28/2014#	D(*USA)#
*USA#	12/31/9999#	31.12.9999#	D(*EUR)#
*ISO#	2036-05-21#	21/05/36#	D(*DMY)#
*JUL#	45/333#	11/29/1945#	D(*USA)#
*USA#	1:00-PM#	13.00.00#	T(*EUR)#
*EUR#	1.10.07#	11:10:07#	T(*JIS)#
*JIS#	14:16:18#	14.16.18#	T(*HMS.)#
*ISO#	24.00.00#	12:00-AM#	T(*USA)#
Blank#	1991-09-14-13.12.56.123456#	1991-09-14-13.12.56.123456#	Z(*ISO)#
*ISO#	1991-09-14-13.12.56.123456#	1991-09-14-13.12.56.123456#	Z(*ISO)#

Moving a Numeric Field to a Date-Time Field

Factor-1#	Factor-2#	Result-Field#	
Entry#	(Numeric)#	Value#	DTZType#
*MDY#	111975#	75/323#	D(*JUL)#
*JUL#	92114#	23/04/92#	D(*DMY)#
*YMD#	140128#	01/28/2014#	D(*USA)#
*USA#	12319999#	31.12.9999#	D(*EUR)#
*ISO#	20360521#	21/05/36#	D(*DMY)#
*JUL#	45333#	11/29/1945#	D(*USA)#
*MDY#	030533#	03.05.33#	D(*MDY)#
*USA#	*DATE:(092195)#	1995-09-21#	D(*JIS)#
Blank#	*DATE:(092195)#	1995-09-21#	D(*JIS)#
*MDY#	UDATE:(092195)#	21.09.1995#	D(*EUR)#
*HMS#	231256#	23.12.56#	T(*ISO)#
*EUR#	111007#	11:10:07#	T(*JIS)#
*JIS#	141618#	14.16.18#	T(*HMS)#
*ISO#	240000#	12:00-AM#	T(*USA)#
Blank#	19910914131256123456#	1991-09-14-13.12.56.123456#	Z(*ISO)#

Moving a Date-Time Field to a Character Field.

Factor-1#	Factor-2#	Result-Field#	
Entry#	Value#	DTZType#	(Character)#
*JUL#	11/19/75#	D(*MDY)#	75/323#
*USA#	14.01.28#	D(*YMD)#	01/28/2014#
*EUR#	12/31/9999#	D(*USA)#	31.12.9999#
*USA#	45/333#	D(*JUL)#	11/29/1945#
*ISO#	24.12.56#	T(*HMS)#	23.12.56#
*EUR#	11:00-AM#	T(*USA)#	11:00:00#
*JIS#	11:10:07#	T(*EUR)#	11.10.07#
*USA#	24.00.00#	T(*ISO)#	12:00-AM#
Blank#	2045-10-27-23.34.59.123456#	Z(*ISO)#	2045-10-27-23.34.59.123456#

Moving a Date-Time Field to a Numeric Field.

Factor-1#	Factor-2#	Result-Field#	
Entry#	Value#	DTZType#	(Character)#
*JUL#	11/19/75#	D(*MDY)#	75/323#
*USA#	14.01.28#	D(*YMD)#	01282014#
*EUR#	12/31/9999#	D(*USA)#	31129999#
*USA#	45/333#	D(*JUL)#	11291945#
*ISO#	241256#	T(*HMS)#	231256#
*EUR#	11:00-AM#	T(*USA)#	110000#
*JIS#	11.10.07#	T(*EUR)#	111007#
*USA#	45/333#	T(*JUL)#	11291945#

Example

Source is shorter than Result.

Character to Character

```
Value1 = 'ASNA4ME'
Value2 = '123456784'
Move Value1 Value2

Value2 now = '12ASNA4ME'.
```

Character to Numeric

```
Value1 = 'ASNA4ME'
Value2 = 123456784
Move Value1 Value2

Value2 now contains invalid Data for a numeric field.
```

Numeric to Numeric

```
Value1 = 1278425
Value2 = 123456789
Move Value1 Value2

Value2 now = 121278425.
```

Numeric to Character

```
Value1 = 1278425
Value2 = 'ACFGPH4SN'
Move Value1 Value2

Value2 now = 'AC1278425'< BR >
```

Source is longer than Result.

```
Character to Character
Value1 = 'ASNA4ME'
Value2 = 'PH4SN'
Move Value1 Value2

Value2 now = 'NA4ME'.
```

Character to Numeric

```
Value1 = 'ASNA4ME'  
Value2 = 56784  
Move Value1 Value2
```

Value2 now contains invalid Data for a numeric field.

Numeric to Numeric

```
Value1 = 1278425  
Value2 = 56748  
Move Value1 Value2
```

Value2 now = 78425.

Numeric to Character

```
Value1 = 1278425  
Value2 = 'ASNA4'  
Move Value1 Value2
```

Value2 now = '78425'.

Source is shorter than Result with *Pad extension.

```
Character to Character  
Value1 = 'ASNA4ME'  
Value2 = '123456784'  
Move Value1 Value2 Fill(*Pad)
```

Value2 now = // ASNA4ME'.

Character to Numeric

```
Value1 = 'ASNA4ME'  
Value2 = 123456784  
Move Value1 Value2 Fill(*Pad)
```

Value2 now contains invalid Data for a numeric field.

Numeric to Numeric

```
Value1 = 1278425  
Value2 = 123456789  
Move Value1 Value2 Fill(*Pad)
```

Value2 is now = 1278425.

Numeric to Character

```
Value1 = 1278425  
Value2 = 'ACFGPH4SN'  
Move Value1 Value2 Fill(*Pad)
```

Value2 now = // 1278425'.

Source and Result are the same length.

```
Character to Character  
Value1 = 'ASNA4ME'  
Value2 = '1234567'  
Move Value1 Value2
```

Value2 now = 'ASNA4ME'.

Character to Numeric

```
Value1 = 'ASNA4ME'
Value2 = 1234567
Move Value1 Value2
```

Value2 now contains invalid Data for a numeric field.

Numeric to Numeric

```
Value1 = 1278425
Value2 = 1234567
Move Value1 Value2
```

Value2 now = 1278425.

Numeric to Character

```
Value1 = 1278425
Value2 = 'ACFGPH4'
Move Value1 Value2
```

Value2 now = '1278425'

See Also

MOVEL | Moving Date-Time Data

MOVEARR

Transfers character or numeric values from **Source** to the **Target**. **Source** and **Target** *must* be an array.

```
MOVEARR
Source (Expression)
Target (Character Expression)
Fill (*NOPAD | *PAD)
Pos (Indicator Variable)
Neg (Indicator Variable)
Boz (Indicator Variable)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR |
      *PACKED | *TIME | *TIMESTAMP | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts**Source**

Required. If **Source** is longer than the **Target**, the excess rightmost characters are not moved. If the **Target** is longer than **Source**, the excess characters are left untouched, unless padding is specified.

If **Source** is Character, then the **Target** also needs to be Character type. If **Source** is numeric, then the **Target** must also be numeric with the same format.

Target

Required. If **Source** is longer than the **Target**, the excess rightmost characters are not moved. If the **Target** is longer than **Source**, the excess characters are left untouched unless padding is specified.

If **Source** is Character, then the **Target** also needs to be Character type. If **Source** is numeric, then the **Target** must also be numeric with the same format.

Fill

Optional. Specifies if you want to fill, or pad the **Target** with blanks. The default is *NOPAD.

*NOPAD indicates that the Target will not be padded with blanks.

*PAD indicates that the Target will be padded with blanks.

Pos

Optional. Turned on if the value of **Target** is a positive number.

Neg

Optional. Turned on if the value of **Target** is a negative number.

Boz

Optional.

Type

Optional. Can be any of the Types listed (Character, Packed or Zoned). *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, TYPES of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

Data is moved starting with the first element of an array if the array is not indexed; or, with the element specified if the array is indexed. Data stops moving when the last array element is moved or filled.

A **P** specified in the operation extender position causes the **Target** to be padded on the right after the move occurs, if Source is shorter than the **Target**.

Example

```
DclArray Arrx 5 *Char 2
DclArray Arry 6 *Char 2
DclArray Arrz 4 *Char 3
```

Array-to-array move. No indexing; different length array, same element length.

```
ARRX ARRY
|12|34|56|78|90| Before |AA|BB|CC|DD|EE|FF|
MoveArr Source(Arrx) Target(Arry)
|12|34|56|78|90| After |12|34|56|78|90|FF|
```

Array-to-array move with index Result Field.

```
ARRX ARRY
|12|34|56|78|90| Before |AA|BB|CC|DD|EE|
MoveArr Arrx Array[3]
|12|34|56|78|90| After |AA|BB|12|34|56|
```

Array-to-array move, no indexing and different length array elements.

```
ARRX ARRY
|12|34|56|78|90| Before |AA|AB|BB|CC|CD|DD|
MoveArr Arrx Array
|12|34|56|78|90| After |12|34|56|78|90|DD|
```

Array-to-array move, index Factor 2 with different length array elements.

```
ARRX ARRZ
|12|34|56|78|90| Before |AAA|BBB|CCC|DDD|
MoveArr Arrx[4] Array
|12|34|56|78|90| After |789|0BB|CCC|DDD|
```

Field-to-array move, no indexing on array.

```
FIELDA ARRY
|1|2|3|4|5|6|7| Before |98|65|43|21|0A|BC|
MoveArr Fielda Array)
|1|2|3|4|5|6|7| After |12|34|56|71|0A|BC|
```

In the following Example, N=3. Array-to-field move with variable indexing.

```
ARRX FIELD
|01|0A|02|0B|03|0C| Before |0|1|0|A|
MoveArr Arrx[N] Field
|01|0A|02|0B|03|0C| After |0|2|0|B|
An array-to-array move showing numeric elements.
DclArray Arrb 4 *Packed Len(2,1)
DclArray Arrc 5 *Packed Len(2,1)
|1.0 |1.1 |1.2 |1.3| Before |2.0 |3.0 |4.0 |5.0 |6.0|
MoveArr Arrb Arrc
|1.0 |1.1 |1.2 |1.3| After |1.0 |1.1 |1.2 |1.3 |6.0|
```

See Also

DCLARRAY | MOVE | MOVEL

MOVEL

Transfers characters from **Source** to the **Target**, left-adjusted with optional padding on the right.

```
MOVEL
Source (Expression)
Target (Character Expression)
Fill (*NOPAD | *PAD)
Pos (Indicator Variable)
Neg (Indicator Variable)
Boz (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND |
      *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *ONECHAR | *PACKED |
      *ZONED)
Len (length integer, decimal integer)
TimFmt (*NONE | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA |
        *YMD)
```

Parts

Source

Required. If **Source** is longer than the **Target**, the excess rightmost characters of **Source** are not moved. If the **Target** is longer than **Source**, the rightmost excess characters of **Target** are left untouched, unless padding is specified (Fill (*PAD)).

Resulting indicators cannot be specified if the **Target** is an array. They can be specified if the **Target** is a non-array field or array element.

When Data is moved to a numeric field, if **Source** is as long or longer than the **Target**, the sign (+ or -) will not be retained.

Target

Required. If **Source** is longer than the **Target**, the excess rightmost characters of **Source** are not moved. If the **Target** is longer than **Source**, the rightmost excess characters of **Target** are left untouched, unless padding is specified (Fill (*PAD)).

Resulting indicators cannot be specified if the **Target** is an array. They can be specified if the **Target** is a non-array field or array element.

When Data is moved to a numeric field, if **Source** is as long or longer than the **Target**, the sign (+ or -) will not be retained.

Fill

Optional. Specifies if padding will be specified on the right of the **Target**. *NOPAD is the default.

*NOPAD indicates that the Target will not be padded with blanks.

*PAD indicates that the Target will be padded from the right with blanks.

Pos

Optional. Turned on if the value of **Target** is a positive number.

Neg

Optional. Turned on if the value of **Target** is a negative number.

Boz

Optional.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

TimFmt

Optional. Specifies the format for time, date, and timestamp variables. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Remarks

Moving Text from a Control to a Field

When moving text from a control into a field, be aware that the contents is a null-terminated string and does not assume the field length. **MOVE** and **MOVEL** operations will move only the number of characters in the string into the field, which may result in extraneous data remaining in the field. Be sure to use **Fill(*Pad)** in the operation to prevent this.

When moving Date, Time or Timestamp fields

TimFmt must be blank unless either the source or the target is a character or numeric field. Otherwise, **TimFmt** contains the date or time format compatible with the character or numeric field that is the source or target of the operation.

If Source is *DATE and the result is a Date field, **TimFmt** is not required. If **TimFmt** contains a date format it must be compatible with the format of *DATE. See Moving Date-Time Data for more information.

Example

Source and Result are the same length.

Numeric to Numeric

```
Value1 = 78425
Value2 = 56784
Move1 Value1 Value2
Value2 now = 78425.
```

Numeric to Character

```
Value1 = 78425
Value2 = 'ASNA4'
Move1 Value1 Value2
Value2 now = '78425'.
```

Character to Numeric

```
Value1 = 'ASNA4'
Value2 = 78425
Move1 Value1 Value2
Value2 is no longer a valid field.
```

Character to Character

```
Value1 = 'ASNA4'
Value2 = 'MENOW'
Move1 Value1 Value2
Value2 now = 'ASNA4'
```

Source Longer than Result.

```
Numeric to Numeric
Value1 = 000258425
Value2 = 56784
Move1 Value1 Value2
Value2 now = 00025.
```

Numeric to Character

```
Value1 = 903178425
Value2 = 'ASNA4'
Move1 Value1 Value2
Value2 now = 90317.
```

Character to Numeric

```
Value1 = 'ASNA4MEEE'  
Value2 = '56784'  
Move1 Value1 Value2  
Value2 is no longer a valid field.
```

Character to Character

```
Value1 = 'ASNA4MEE'E  
Value2 = 'OK4EU'  
Move1 Value1 Value2  
Value2 now = 'ASNA4'.
```

Source is shorter than Result.**Numeric to Numeric**

```
Value1 = 78425  
Value2 = 130943210  
Move1 Value1 Value2  
Value2 now = 784253210.
```

Character to Numeric

```
Value1 = 'ASNA4'  
Value2 = 130943210  
Move1 Value1 Value2  
Value2 is no longer a valid field.
```

Numeric to Character

```
Value1 = 78425  
Value2 = 'ASNA4MEEE'  
Move1 Value1 Value2  
Value2 now = '78425MEEE'.
```

Character to Character

```
Value1 = 'ASNA4'  
Value2 = 'NOMORERPG'  
Move1 Value1 Value2  
Value2 now = 'ASNA4ERPG'.
```

Source is shorter than Result with Fill(*Pad).**Numeric to Numeric**

```
Value1 = 78425  
Value2 = 130943210  
Move1 Value1 Value2 Fill(*Pad)  
Value2 now = 784250000.
```

Character to Numeric

```
Value1 = 'ASNA4'  
Value2 = 130943210  
Move1 Value1 Value2 Fill(*Pad)  
Value2 is no longer a valid field.
```

Numeric to Character

```
Value1 = 78425  
Value2 = 'ASNA4MEEE'  
Move1 Value1 Value2 Fill(*Pad)  
Value2 now = '78425 '
```

Character to Character

```
Value1 = 'ASNA4'
Value2 = 'NOMORERPG'
Move1 Value1 Value2 Fill(*Pad)
Value2 now = 'ASNA4 '
```

See Also

MOVE | Moving Date-Time Data

MOVEREM

Moves the remainder from a previous DIV operation to a separate field named in the Result Field.

```
MOVEREM
Target (Numeric Variable)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
        *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
        | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

Target

Required. The maximum number of whole number positions in the remainder is equal to the whole number of positions in F2 of the previous **DIV** operation.

Pos

Optional. Turned on if the value of **Target** is a positive number.

Neg

Optional. Turned on if the value of **Target** is a negative number.

Zero

Optional. Turned on if the value of **Target** is zero.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

The MOVEREM operation must immediately follow the DIV operation (if the **REM** keyword was not used with the **DIV** operation) or undesirable results may occur. Note that you must use the **DIV** op code, and cannot use the divide sign "/" in it's place.

Note that you can also use the REM parameter in the previous DIV operation instead of the **MOVEREM** op code.

See Also

DIV

MSGBOX

Displays a message in a dialog box, waits for the user to choose a button, and returns a value indicating which button the user has chosen. It contains an application-defined message and title, plus any combination of icons and buttons.

MSGBOX

```
Msg (Character Expression)
Title (Character Expression)
Buttons (*OK | *ABORTRETRYIGNORE | *OKCANCEL | *RETRYCANCEL | *YESNO |
        *YESNOCANCEL)
RtnCode (Variable)
Icon (*NONE | *EXCLAMATION | *INFORMATION | *QUESTION | *STOP)
DftButton (1 | 2 | 3 | 4)
Justify (*LEFT | *RIGHT)
RtlReading (*NO | *YES)
TopMost (*NO | *YES)
```

Parts

Msg

Required. Contains the message to be displayed in the message box.

Title

Optional. Contains a character expression for the title that will display in the title bar of the dialog box, as shown below.



Buttons

Optional. Specifies the buttons that will be included in the message box. These buttons allow the user to respond to the message. The buttons will display in the order in which they are listed, from left to right. The OK button will display by default.

*OK - The message box contains the OK button only. This is the default.

*AbortRetryIgnore - The message box contains 3 buttons: Abort, Retry and Ignore.

*OKCancel - The message box contains 2 buttons: OK and Cancel.

*RetryCancel - The message box contains 2 buttons: Retry and Cancel.

*YesNo - The message box contains 2 buttons: Yes and No.

*YesNoCancel - The message box contains 3 buttons: Yes, No and Cancel.

RtnCode

Optional. Returns a value indicating which button was selected by the user. The variable must be defined as numeric. The valid return codes and the button selected are:

1 = OK

2 = Cancel

3 = Abort

4 = Retry

5 = Ignore

6 = Yes

7 = No

Icon

Optional. Specifies if an icon will display in the left side of the message box. The default is *NONE. The following options can be specified, displaying the following icons in the message box.

*Exclamation - An exclamation point within a yellow triangle. 

*Information - A lowercase letter i in a circle. 

*Question - A question mark in a circle. 

*Stop - An "X" within a red circle. 

DftButton

Optional. Indicates which button in the message box will be the default button. There can be only one default button and it will be noted with a dotted line inside of the button, as shown below.



The default is 1, for the first button. Enter a 2 to indicate the second button, 3 for the third, and 4 for the fourth button. If a number higher than the number of buttons indicated is specified, the first button will be the default button.

Justify

Optional. Specifies if the text in the message box is left or right-justified. The default is left-justified.

RtlReading

Optional. Displays a message and a caption text using right-to-left reading order on Hebrew and Arabic systems.

TopMost

Optional. Indicates whether the dialog box will display on top when another application is open when *YES is specified. The default is *NO. This parameter is only supported on Windows NT 4.0.

Remarks

When a Message Box is displayed, it is always on top of all other windows. Thus, input is inhibited from any other form, so the user must respond to the message box before processing can continue.

MULT

F1 (if specified) is multiplied by **F2** and the product is placed in **Result**.

```

MULT
F1 (Numeric Expression | *NONE)
F2 (Numeric Expression)
Result (Numeric Variable)
Adj (*TRUNC | *RNDUP)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
        *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
        | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])

```

Parts

F1

Optional. If **F1** is not specified, **F2** is multiplied by the **Result** and the product is placed in **Result**.

F2

Required. Contains a numeric expression to multiply.

Result

Required. Must be large enough to hold the product of **F1** and **F2**. The length of **Result** equals the length of **F1** plus the length of **F2**.

Adj

Optional. Indicates half-adjust. Use **Adj** (*RNDUP) if you wish to round up the value, or use **Adj** (*TRUNC) if you wish to truncate the value of the **Result** Field. *TRUNC is the default.

Pos

Optional. Turned on if the value of the **Result** is a positive number.

Neg

Optional. Turned on if the value of the **Result** is a negative number.

Zero

Optional. Turned on if the value of the **Result** is zero.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Example

```
/* Multiply the value of ManualPremium by the value of DrugFreeCredit and place Result in
ExperienceModify.
```

```
  Mult ManualPremium DrugFreeCredit ExperienceModify
```

```
/* Multiply the value of ManualPremium by the value of DrugFreeCredit and place Result in
ManualPremium with half adjust. */
```

```
  Mult *NONE DrugFreeCredit ManualPremium *RNDUP
```

OR

```
  Mult ManualPremium DrugFreeCredit ManualPremium *RNDUP
```

See Also

* Operator | Assignment

OCCUR

Sets and/or gets the current occurrence of a data structure that is to be used for the next access to the data structure or data structure subfield.

```
OCCUR
Ds (Data Structure)
NewIndex (*NONE | Numeric Expression, Numeric Expression,...)
OldIndex (*NONE | Character Expression)
Err (Indicator Variable | *EXTENDED)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *FLOAT | *FLOAT4 |
      *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

Ds

Required. Contains the name of the Data Structure. If the data structure has multiple occurrences or a subfield of that data structure is specified in an operation, the first occurrence of the data structure is used until an **OCCUR** operation is specified. After **OCCUR** is specified, the occurrence of the data structure that was established by **OCCUR** is used.

NewIndex

Required only if **OldIndex** is not specified; optional if **OldIndex** is specified. Use to specify a new current occurrence for the data structure. The current occurrence is relative to zero.

Please note that either the **NewIndex** or **OldIndex** parameter is required.

OldIndex

Required only if **NewIndex** is not specified; optional if **NewIndex** is specified. Use as a receiver of the current index before a new index is set. If the data structure has multiple dimensions, a value must be provided for each dimension of both the old and new occurrence values. The current occurrence is relative to zero.

An alternate way to access occurrences is by using the [] operators. See Multiple Occurrence Data Structures.

Please note that either the **OldIndex** or **NewIndex** parameter is required.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - the specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Type

Optional. Can be any of the Types listed (Character, Packed or Zoned). *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

All data structures (and arrays) in Visual RPG are **zero-based**.

Example

```
DclDs LastYear DCLFLD(50)
DclDsFld SalesLy Len(10)
DclDsFld ReturnsLy Len(10)

DclDs ThisYear DCLFLD(50)
DclDsFld SalesTy Len(10)
DclDsFld ReturnsTy Len(10)
```

LASTYEAR is set to the third occurrence. The subfields SALES LY and RETURNS LY of the third occurrence can now be used. The ZADD operations change the contents of SALES LY and RETURNS LY, respectively, in the third occurrence of LASTYEAR.

```
Occur LastYear 3
SalesLy = TotalSales
ReturnsLy = TotalReturns
```

LASTYEAR is set to the fourth occurrence. Using the values in SALES LY and RETURNS LY of the fourth occurrence of LASTYEAR, the ZADD operation places the contents of SALES LY in the Result Field, SALES, and the ZADD operation places the contents of RETURNS LY in the Result Field, RETURNS.

```
Occur LastYear 4
Sales = SalesLy
Returns = ReturnsLy
```

LASTYEAR is set to the occurrence specified in field Index. For Example, if Index = 10, LASTYEAR is set to the tenth occurrence.

```
Occur LastYear Index
```

LASTYEAR is set to the current occurrence of THISYEAR. For Example, if the current occurrence of THISYEAR is the twelfth occurrence, LASTYEAR is set to the twelfth occurrence.

```
Occur LastYear ThisYear
```

The value of the current occurrence of LASTYEAR is placed in the Result Field, Z. Field Z must be numeric with zero decimal positions. For Example, if the current occurrence of LASTYEAR is 15, field Z contains the value 15.

```
Occur Z LastYear
```

LASTYEAR is set to the current occurrence of THISYEAR. The value of the current occurrence of LASTYEAR is then moved to the Result Field, Z. For Example, if the current occurrence of THISYEAR is the fifth occurrence, LASTYEAR is set to the fifth occurrence. The Result Field, Z, contains the value 5.

```
Occur DS (LastYear) NewIndex (ThisYear) OldIndex (Z) Err (*In20)
```

LASTYEAR is set to the current occurrence of Index. For Example, if Index = 15, LASTYEAR is set to the fifteenth occurrence. If Index equals 0 or is greater than 50, an error occurs, and indicator 20 is set on. If indicator 20 is on, the LR indicator is set on.

```
Occur Ds (LastYear) NewIndex (Index) OldIndex (Z) Err (*In20)
If *In20
*InLr = *On
Endif
DclDs LastYear DCLFLD(50)
DclDsFld SalesLy Len(10)
DclDsFld SalesByMonthLy DCLFLD(12) Len(10,0)
DclDsFld ReturnsLy len(10)
DclDsFld ReturnsByMonthLy DCLFLD(12) Len(10,0)
DclDs ThisYear DCLFLD(50)
DclDsFld SalesTy Len(10)
DclDsFld SalesByMonthTy DCLFLD(12) Len(10,0)
DclDsFld ReturnsTy len(10)
DclDsFld ReturnsByMonthTy DCLFLD(12) Len(10,0)
```

LASTYEAR is set to the third occurrence. The subfields SALES LY and RETURNS LY of the third occurrence can now be used. The ZADD operations change the contents of SALES LY and RETURNS LY, respectively, in the third occurrence of LASTYEAR. Also the monthly elements of both Sales and Returns are moved in an array of 12 each.

```
Occur LastYear 3
SalesLy = TotalSales
ReturnsLy = TotalReturns
SalesMonthLyArray = SalesByMonthLy
ReturnMonthLyArray = ReturnsByMonthLy
```

See Also

DCLDS | DCLDSFLD | Multiple Occurrence Data Structures | %OCCUR

OPEN

Opens the file named in **File** for processing.

```
OPEN
File (Name of file)
FilePath (*FILENAME | Character Variable)
Mbr (*FIRST | *FILE | Character Variable)
Db (Database Object)
Err (*EXTENDED | Indicator Variable)
```

Parts

File

Required. Cannot be designated as a primary, secondary, or table file.

FilePath

Optional. Contains the path to the **File**.

Mbr

Optional. Specifies the file member to use when opening the file at runtime. The **Mbr** parameter allows you to specify the name of the member you want to access or ***FIRST**. ***FIRST** is the default.

The following rules apply:

***FIRST** (default) - will access the member with the oldest creation date.

***FILE** - accesses the member name with the same name as the file name.

Db

Optional. Specifies the Database Connection to use when opening the file at run-time.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - the specified Indicator is set if an error occurred.

Err (***EXTENDED**) - a program status flag is set whose status can be checked using the **%ERROR** built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Open the file specified in **File** for the first time in a program with an explicit **OPEN** operation. If a file is opened and later closed by the **CLOSE** operation in the program, you can reopen the file with the **OPEN** operation. Multiple **open** operations in a program to the same file are valid as long as the **File** is closed when the **OPEN** operation is issued.

Example

```
DCLDISKFILE ItemDetail *INPUT ORG(*INDEXED) ADDRREC(*NO) Impopen(*NO)
DCLDISKFILE ItemOut *OutPut
```

// The explicit **OPEN** operation opens the **ItemDetail** file for processing if indicator 97 is on and indicator 98 is off. Note that the **ItemDetail** file on the file description specifications has the **Impopen** keyword specified. Indicator 99 will be set on if the **OPEN** operation fails.

```
If *In97 = *On *And *In98 = *Off
  Open ItemDetail Err (*In99)
If *In99 = *On
  Write ErrorRecord
Endif
Endif
```

See Also

CLOSE | **DCLDB** | **DCLDISKFILE** | **DCLMEMORYFILE** | **DCLPRINTFILE**

OSEXEC

The **OSEXEC** command opens or prints a specified file. The file can be an executable file or a file that is associated with a program.

```

OSEXEC
CmdLine (Character Expression)
ShowType (*SHOWNORMAL | *HIDE | *MINIMIZED | *RESTORE | *SHOW |
          *SHOWMAXIMIZED | *SHOWMINIMIZED | *SHOWNA | *SHOWNOACTIVE |
          *SHOWNORMAL)
RtnCode (Numeric Field)
Err (*EXTENDED | Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *FLOAT | *FLOAT4 |
      *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Positions Integer])
Operation (*OPEN | *PRINT | *EXPLORE)
Directory (Character Expression)
Wait (*NO | *YES | Indicator Expression)
ExitCode (Numeric Variable)

```

Parts

CmdLine

Required. Specifies the file to open or print, or the folder to open or explore. The function can be either an executable file or a document file. It can also print a document file. **CmdLine** must contain the complete path of the application to execute.

If the application is being called from a folder that has an embedded blank in the path name, it must be contained within double quotes. There are several ways in which this can be coded, for Example:

```

OSEXEC "C:\bin\Sysmeter.exe Parm1 Parm2"
OSEXEC "C:\Program Files\Examples\Calculator.EXE"
OSEXEC "'C:\Program Files\Examples\Calculator.EXE'"
OSEXEC '''Program Files\Examples\Calculator.EXE'''
OSEXEC ""Program Files\Examples\Calculator.EXE""

```

ShowType

Optional. Specifies how the application is to be shown when it is open. The default for **ShowType**, if none is specified, is *ShowNormal. The parameter can be one of the following values:

- *ShowNormal - Activates and displays a window. If the window is minimized or maximized, windows restores it to its original size and position.
- *Hide - Hides the window and activates another window.
- *Maximize - Maximizes the specified window.
- *Minimize - Minimizes the specified window, and activates the next top-level window in the system's list.
- *Restore - Activates and displays a window. If the window is minimized or maximized, windows restores it to its original size and position.
- *Show - Activates a window and displays it in its current size and position.
- *ShowDefault - Sets the initial show state of its main window.
- *ShowMaximized - Activates a window and displays it as a maximized window.
- *ShowMinimized - Activates a window and displays it as a minimized window.
- *ShowMinNoActive - Displays a window as a minimized window. The active window remains active.
- *ShowNA - Displays a window in its current state. The active window remains active.
- *ShowNoActivate - Displays a window in its most recent size and position. The active window

remains active.

If **CmdLine** specifies a document file, **ShowType** should be 0.

RtnCode

Optional. Returns a value greater than 32 if successful, or an error value that is less than or equal to 32 otherwise. The following table lists the error values. The value of the return code is set by Windows. If there was no error, the contents of **RtnCode** are without meaning. For further information, see <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellicc/platform/shell/reference/functions/shellexecute.asp>.

0 = System is either out of memory or resources.

2 = The specified file was not found.

3 = The specified path was not found.

5 = The operating system denied access to the specified file.

8 = Insufficient memory to start the application.

11 = The executable file was invalid, either it was not a windows applications, or

26 = A sharing violation has occurred.

27 = The filename association is incomplete or invalid.

28 = The DDE transaction could not be completed because the request timed-out.

29 = The DDE transaction failed.

30 = The DDE transaction could not be completed because other DDE transactions were

31 = There is no application associated with the extension given for the filename.

32 = The specified dynamic link library was not found.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Type

Optional. Can be any of the ******* Types (*CHAR, *BINARY, etc.) or it can be the name of a field already defined in the program, a string literal or a library object, as described below. See Type Parameter for further information.

Name is the name of the variable already defined.

A string literal, which defines an *OBJECT variable of a particular type. The string literal must represent:

- a valid program id registered on your system at compile time
- a valid class id registered on your system at compile time.
- a valid interface name registered on your system at compile time.

A name of the form Library.Object, where Library is the name of one of the libraries selected in the new up and coming object libraries dialog, and Object is valid object name within that library.

If a **Type** parameter is not given, the **Len** parameter is required. In this case, *CHAR is assumed if there is one integer, and *ZONED is assumed if there are two integers.

Len

Optional. Defines the length of the field. If Len is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *BINARY, *CHAR, *PACKED, *ZONED, *DECIMAL, *FLOAT, *INTEGER will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Operation

Optional. Specifies what type of operation is to be performed on the file. You can either open the file (*OPEN), print the file, (*PRINT) or explore the folder (*EXPLORE) specified by CmdLine.

*OPEN opens the file specified by **CmdLine**. The file can be an executable file or a document file. The file can be a folder to open. *OPEN is the default.

*PRINT prints the file specified by **CmdLine**. The file should be a document file. If the file is an executable file, *PRINT merely opens the file, as if *OPEN had been specified.

*EXPLORE explores the folder specified by **CmdLine**.

Directory

Optional. Contains the string that specifies the default directory. For Example, C:\Program Files\My Programs\Customer.EXE

Wait

Optional. Determines whether the program will wait for the launched application to end before continuing execution. The **Wait** parameter can be an expression that returns an Indicator, in addition to the constants *YES and *NO.

ExitCode

Optional. Determines the variable that will receive the exit code of the launched application after waiting for it to finish. To be able to use **ExitCode**, **Wait** must not be the constant *NO. If **Wait** is an expression that evaluates to *NO, the variable in **ExitCode** will receive the value 0. If an error occurs when launching the application, the **ExitCode** variable receives 0. Verify that the **Err** indicator is *OFF before using the value returned in **ExitCode**.

Remarks

A good way to test your **OSEXEC** command is to confirm that it can be run from the Windows Start Menu "Run" command. You can do anything with **OSEXEC** that you can do from the Run command line.

The **OSEXEC** command can execute a file that has been associated with an application. For Example, if the execute file is a Word document, Word will be initiated. The advantage is that if Word is already active, then the file comes up much faster. See your Windows documentation for more information on associating files with an application.

When passing parameters to an .EXE from the command line, the following rules apply:

Arguments are delimited by white space, which is either a space or a tab.

The caret character (^) is not recognized as an escape character or delimiter. The character is handled completely by the command-line parser in the operating system before being passed to the program.

A string surrounded by double quotation marks ("string") is interpreted as a single parameter (argument), regardless of white space contained within. A quoted string can be embedded in a parameter.

A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").

Backslashes are interpreted literally, unless they immediately precede a double quotation mark.

If an even number of backslashes is followed by a double quotation mark, one backslash is placed in the parameter for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.

If an odd number of backslashes is followed by a double quotation mark, one backslash is placed in the parameter for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash, causing a literal double quotation mark (") to be placed in the parameter.

OTHER

Defines the start of the default group of statements for a **SELECT** group. These are the statements that will be executed if all of the **WHEN** conditions are False.

OTHER

Remarks

The group of statements following the **OTHER** command must be terminated with an **END**, or **ENDSL** command.

In a **SELECT** group, the **OTHER** operation is optional. Additionally, only one **OTHER** operation can be used. The order of calculation operations in the **OTHER** group can be vacant.

A **WHEN** operation cannot be specified after an **OTHER** operation in the same **SELECT** group.

Example

// Example of a **SELECT** group with **WHxx** and **OTHER**. If **SALESType** equals 1, do the operations in sequence 1; if **SALESType** does not equal 1 and **PROD_Type** equals 2, do the operations in sequence 2. If neither condition is true, do the operations in sequence 3.

```
SELECT
  When SALESType = 1
  :
  When PROD_Type = 2
  :
  OTHER
  :
```

```

:
ENDSL

// In the following Example, if SALESType equals 1, do the operations in sequence 1 ; if SALESType
does NOT equal 1, and if PROD_Type=2 and SALESType<10, do the operations in sequence 2. If
neither condition is true, do the operations in sequence 3.

```

```

SELECT
  WHEN SALESType = 1
    TotalSales = Sales1996
    YearSales = '1996'
    NameOfValue = 'Sales'
  :
  WHEN PROD_Type = 2 *And SALESType < 10
  :
  OTHER
  :
  :
ENDSL

```

See Also

END | ENDSL | SELECT | WHEN

READ

Reads the currently pointed-to record from a full procedural file.

```

READ
From (File Name | Format Name)
Access (*DFT | *NOLOCK)
Err (*EXTENDED | Indicator Variable)
Eof (Indicator Variable)

```

Parts

From

Required. Must contain a file name or a record format name. If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you **must** use the format name.

Access

Optional. Determines if records are locked.

***DFT** – (Default) - if the parameter is not given, the default record locking behavior is taken. This means, if File is specified as INPUT, all records are read without locks. If File is specified as **UPDATE**, all records are locked if blank.

***NOLOCK** – if this option is specified, then in no case will the record be locked when the **READ** operation is successful. The file specified in From is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - the specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function or the Files' IsError property.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Eof

Optional. Use the **Eof** (End-of-File) indicator, the %EOF function, or the Files' IsEof property to determine if the End-Of-File was reached while executing the command.

If any other error occurs, the error indicator will be set. The End of File indicator is also set when no more records exist with the same search argument.

Remarks

If the **READ** operation is successful, the file is positioned at the next record that satisfies the read.

Example

/ Read retrieves the next record from the file CustMast, which must be a full procedural file. Indicator 61 is set on if End-Of-File occurs on Read, or if End-Of-File has occurred previously and the file has not been repositioned. When indicator 61 is set on, the program will perform its End-of-Job routine. If there is an error on the Read, indicator 99 will be turned on. */*

```
Read CustMast Eof(*In61) Err(*In99)
If *In61 = *On
  Exsr EndOfJob
Endif
```

```
  If *In99
    Exsr ErrorOnRead
  Endif
```

/ Read retrieves the next record of the type CustRecd from an externally described file (CustRecd is a record format name.) Indicator 61 is set on if End-Of-File occurs on Read, or if it has occurred previously and the file has not been repositioned. When indicator 61 is set on, the program will perform its End-of-Job routine. Access(*NoLock) indicates that the record is not locked. */*

```
Read CustRecd Access(*noLock) Eof(*In61) Err(*In99)
If *In61 = *On
  Exsr EndOfJob
Endif
```

```
  If *In99
    Exsr ErrorOnRead
  Endif
```

/ Read retrieves the next record from the file CustMast, which must be a full procedural file. Indicator 61 is set on if End-Of-File occurs on Read, or if End-Of-File has occurred previously and the file has not been repositioned. When indicator 61 is set on, the program will perform its End-of-Job routine. If there is an error on the Read, indicator 99 will be turned on. */*

```
Read CustMast Eof(*In61) Err(*In99)
If *In61 = *On
  Exsr EndOfJob
Endif
```

```
  If *In99
    Exsr ErrorOnRead
  Endif
```

/* Read retrieves the next record of the type CustRecd from an externally described file (CustRecd is a record format name.) Indicator 61 is set on if End-Of-File occurs on Read, or if it has occurred previously and the file has not been repositioned. When indicator 61 is set on, the program will perform its End-of-Job routine. Access(*NoLock) indicates that the record is not locked. */

```
Read CustRecd Access(*noLock) Eof(*In61) Err(*In99)
If *In61 = *On
  Exsr EndOfJob
Endif

If *In99
  Exsr ErrorOnRead
Endif
```

See Also

CHAIN | READE | READP | READPE | READC | %EOF | %ERROR | IsEof Property | IsError Property

READC

Reads the next record that has been changed in a Subfile.

```
READC
From (SubfileControl Name)
Err (*EXTENDED | Indicator Variable)
Eof (Indicator Variable)
```

Parts

From

Required. Must contain the name of the SubfileControl.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function or the Files' IsError property.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Eof

Optional. Use the **Eof** (End-of-File) indicator, the %EOF function, or the Files' IsEof property to determine if the End-Of-File was reached while executing the command.

If any other error occurs, the error indicator will be set. The End of File indicator is also set when no more records exist with the same search argument.

Remarks

READC retains the current cursor position as the starting point when looking for the next changed record. In order for a record format to be used in a **READC** operation, it has to be listed in the *Subfile* keyword of its corresponding file. See DCLMEMORYFILE and DCLWORKSTNFILE.

See Also

CHAIN | DCLMEMORYFILE | DCLWORKSTNFILE | READ | READE | READP | READPE | %EOF | %ERROR | IsEof Property | IsError Property

READE

Reads the next sequential record in a full procedural file that has the same search argument.

```
READE
From (File Name | Format Name)
Key (*CURRENT | KeyList Name | Character Expression)
Access (*DFT | *NOLOCK)
Err (*EXTENDED | Indicator Variable)
Eof (Indicator Variable)
```

Parts

From

Required. Must contain a file name or a record format name. If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you must use the format name.

Key

Optional. Determines the key in which to search the file by. Refer to %KDS to use this built-in function to establish search arguments (Key) for Data Structures.

Access

Optional. Determines if records are locked.

*DFT – (Default) if the parameter is not given, the default record locking behavior is taken. This means, if File is specified as INPUT, all records are read without locks; if File is specified as **UPDATE**, all records are locked if blank.

*NOLOCK – if this option is specified then in no case will the record be locked when the **READE** operation is successful. The file specified in **F2** is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function or the Files' IsError property.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Eof

Optional. Use the **Eof** (End-of-File) indicator, the %EOF function, or the Files' IsEof property to determine if the End-Of-File was reached while executing the command.

If any other error occurs, the error indicator will be set. The End of File indicator is also set when no more records exist with the same search argument.

Remarks

If the **READ** operation is successful, the file is positioned at the record prior to the one just read.

Example

/* With Key specified, the ReadE operation retrieves the next record from the file CustMast and compares its key to the search argument, CustNumb. Indicator 55 is set on if CustNumb is not equal to the key of the record read or if End-Of-File is encountered. Indicator 99 is turned on if any error occurred during ReadE operation. */

```
ReadE CustMast CustNumb Eof(*In55) Err(*In99)
If *In55 = *On
    Exsr EndOfJob
Endif
If *In99
    Exsr ErrorOnRead
Endif
```

/* With Key specified, the ReadE operation retrieves the next record of the type CustRecd from an externally described file and compares the key of the record read to the search argument, CustNumb (CustRecd is a record format name). Indicator 61 is set on if CustNumb is not equal to the key of the record read or if End-Of-File is encountered. Indicator 99 is turned on if any error occurred during ReadE operation. The record is also unlocked after the read for an update described file. */

```
ReadE CustRecd CustNumb Access(*noLock) Eof(*In61) Err(*In99)
If *In61 = *On
    Exsr EndOfJob
Endif
If *In99
    Exsr ErrorOnRead
Endif
```

/* With no Key specified, the ReadE operation retrieves the next record in the access path from the file CustMast if the key value is equal to the key value of the record at the current cursor position. If the key values are not equal, indicator 55 is set on. Indicator 99 is turned on if any error occurred during ReadE operation. */

```
ReadE CustMast Eof(*In55) Err(*In99)
If *In55 = *On
    Exsr EndOfJob
Endif
If *In99
    Exsr ErrorOnRead
Endif
```

/* With no Key specified, the ReadE operation retrieves the next record in the access path from the file CustMast if the key value equals the key value of the record at the current position. CustRecd is a record format name. Indicator 61 is set on if the key values are unequal. Indicator 99 is turned on if any error occurred during ReadE operation. The record is also unlocked after the read for an update described file. */

```
ReadE CustRecd Access(*noLock) Eof(*In61) Err(*In99)
If *In61 = *On
    Exsr EndOfJob
Endif
If *In99
    Exsr ErrorOnRead
Endif
```

See Also

CHAIN | READ | READC | READP | READPE | %EOF | %ERROR | %KDS | IsEof Property | IsError Property

READP

Reads the prior record from a full procedural file.

```
READP
Fr om (File Name | Format Name)
Access (*DFT | *NOLOCK)
Err (*EXTENDED | Indicator Variable)
Bof (Indicator Variable)
```

Parts

From

Required. Must contain a file name, or a record format name.

Access

Optional. Determines if records are locked.

***DFT** – (Default) if the parameter is not given, the default record locking behavior is taken. This means, if File is specified as INPUT, all records are read without locks. If File is specified as UPDATE, all records are locked if blank.

***NOLOCK** – if this option is specified, then in no case will the record be locked when the **READP** operation is successful. The file specified in F2 is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - the specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you must use the format name.

Bof

Optional. The **Bof** parameter, or the %EOF function or Eof Parameter, can be used to determine if the Beginning-Of-File was reached.

Indicator Variable - The specified indicator is set if the beginning of file is reached.

Remarks

If the **READP** operation is successful, the file is positioned at the record prior to the one just read.

Example

/* The READP operation reads the prior record from CustMast. Indicator 71 is set on if Beginning-of-file is encountered. When indicator 71 is set on, the program executes the subroutine BeginFile specified in the indicator conditioned instructions. If there is an error on the READP, indicator 99 will be turned on. */

```
ReadP CustMast Bof( *In71 ) Err ( *In99 )
  If *In71 = *On
    Exsr BeginFile
  Endif

  If *In99
```



```

        Exsr ErrorOnRead
    Endif

/* The READP operation reads the next prior record of the type CustRecd from an externally described
file (CustRecd is a record format name). Indicator 72 is set on if Beginning-of-file is encountered during
processing of the READP operation. When indicator 72 is set on, the program executes the subroutine
BeginFile specified in the indicator conditioned instructions. Access(*NoLock) indicates that the record is
not locked. */

ReadP CustRecd Access( *NoLock ) Bof(* In72 ) Err ( *In99 )
    If *In72 = *On
        Exsr BeginFile
    Endif
    If *In99
        Exsr ErrorOnRead
    Endif

```

See Also

CHAIN | DCLDISKFILE | READ | READE | READPE | %EOF

READPE

Reads the next prior sequential record from a full procedural file that has the same search argument.

```

READPE
From (File Name | Format Name)
Key (*CURRENT | KeyList Name | Character Expression)
Access (*DFT | *NOLOCK)
Err (*EXTENDED | Indicator Variable)
Bof (Indicator Variable)
Eof (Indicator Variable)

```

Parts

From

Required. Must contain a file name or a record format name.

Key

Optional. Determines the key in which to search the file by. Refer to %KDS to use this built-in function to establish search arguments (Key) for Data Structures.

Access

Optional. Determines if records are locked.

***DFT** – (Default) If the parameter is not given, the default record locking behavior is taken. This means, if File is specified as INPUT, all records are read without locks and the extender must be blank. If File is specified as UPDATE, all records are locked if blank.

***NOLOCK** – If this option is specified, then in no case will the record be locked when the **READPE** operation is successful. The file specified in **From** is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function or the Files' IsError property

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file Name, you must use the format name.

Bof

Optional. The Bof indicator will be set if the Beginning-Of-File was reached.

Eof

Optional. Use the **Eof** (End-of-File) indicator, the %EOF function, or the Files' IsEof property to determine if the End-Of-File was reached while executing the command.

If any other error occurs, the error indicator will be set. The End of File indicator is also set when no more records exist with the same search argument.

Remarks

If the READ operation is successful, the file is positioned at the record prior to the one just read.

Example

/* With Key specified, the ReadPE operation retrieves the previous record from the file CustMast and compares its key to the search argument, CustNumb. Indicator 55 is set on if CustNumb is not equal to the key of the record read or if beginning of file is encountered. Indicator 99 is turned on if any error occurred during ReadPE operation. */

```
ReadPE CustMast CustNumb Bof(*In55) Err (*In99)
  If *In55 = *On
    Exsr BeginningOfFile
  Endif

  If *In99
    Exsr ErrorOnRead
  Endif
```

/* With Key specified, the ReadPE operation retrieves the previous record of the type CustRecd from an externally described file and compares the key of the record read to the search argument, CustNumb (CustRecd is a record format name). Indicator 55 is set on if CustNumb is not equal to the key of the record read or if Beginning-of-File is encountered. Indicator 99 is turned on if any error occurred during ReadPE operation. The record is also unlocked after the read for an update described file. */

```
ReadPE CustRecd CustNumb Access(*noLock) Bof(*In61) Err (*In99)
  If *In61 = *On
    Exsr BeginningOfFile
  Endif

  If *In99
    Exsr ErrorOnRead
  Endif
```

/* With no Key specified, the ReadPE operation retrieves the previous record in the access path from the file CustMast if the key value is equal to the key value of the record at the current cursor position. If the key values are not equal, indicator 55 is set on. Indicator 55 is also set on if the beginning of file is reached. Indicator 99 is turned on if any error occurred during the ReadPE operation. */

```
ReadPE CustMast Bof(*In55) Err (*In99)
  If *In55 = *On
    Exsr BeginningOfFile
  Endif

  If *In99
    Exsr ErrorOnRead
```

```

Endif

/* With no Key specified, the ReadPE operation retrieves the previous record in the access path from the
file CustMast if the key value equals the key value of the record at the current position. CustRecd is a
record format name. Indicator 61 is set on if the key values are unequal. Indicator 61 is also set on if the
Beginning-of-File is reached. Indicator 99 is turned on if any error occurred during ReadPE operation.
The record is also unlocked after the read for an update described file. */

ReadPE CustRecd Access(*noLock) Bof(*In61) Err (*In99)
  If *In61 = *On
    Exsr BeginningOfFile
  Endif

  If *In99
    Exsr ErrorOnRead
  Endif

```

See Also

CHAIN | READ | READE | READP | %EOF | %ERROR | %KDS | IsEof Property | IsError Property

READRAND

Retrieves a record from a full procedural file, and places the data into the input fields.

```

READRAND
From (File Name | Format Name)
Key (KeyList Name | (Character Expression)
Access (*DFT | *NOLOCK)
NotFnd (Indicator Variable)
Err (*EXTENDED | Indicator Variable)

```

Parts

From

Required. Specifies the file or record format name that is to be read. A record format name is valid with an externally described file. If **From** is a record format name and access is by key, **READRAND** retrieves the first record of the specified record type whose key matches the search argument. If no record is found, a No record Found condition exists, and the Not Found indicator is set On.

If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you must use the format name.

Key

Optional. Must contain the key or relative record number used to retrieve the record. If access is by key, **From** can be any valid character expression. In addition, a DCLKLIST Name can be specified in **Key** for an externally described file. If access is by relative record number, **Key** must contain an integer literal or a numeric field with zero decimal positions.

If the **Access** parameter is not given, the default record locking behavior is taken. This means, if **File** is specified as INPUT, all records are read without locks and the extender must be blank. If **File** is specified as UPDATE, all records are locked if blank.

Access

Optional. Determines if records are locked.

***DFT – (Default)** If the parameter is not given, the default record locking behavior is taken. This means, if **File** is specified as INPUT, all records are read without locks and the extender must be blank. If **File** is specified as UPDATE, all records are locked if blank.

***NOLOCK** – If this option is specified, then in no case will the record be locked when the **READPE**

operation is successful. The file specified in **From** is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record. If **Access** (*NOLOCK) is specified, then in no case will the record be locked when the **READRAND** operation is successful. The file specified in **Key** is positioned so a subsequent read operation retrieves the next sequential record following the retrieved record. If the **READRAND** is not completed successfully, the file specified in **From** must be repositioned before a subsequent read operation can be done to that file.

If **READRAND** operation for file is successful, the **RRNFldName** property is updated with current **RRN**. If it is not successful, a value of zero is assigned to **RRNFLDName**.

NotFnd

Optional. Used to determine if the record was found while executing this command.

Note that you can also use the **%FOUND** function. However, be aware that **%FOUND** and the **NotFnd** indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the **%ERROR** built-in function or the Files' **IsError** property.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

CHAIN is exactly the same as **READRAND**, and can be used interchangeably.

Example

/ The ReadRand operation retrieves the first record from the file CustMaster, that has a key field with the same value as the search argument CustNumber (Factor 1). Indicator 60 will come on (File record not found). */*

```
ReadRand CustMaster CustNumber NotFnd(*In60)
  If *In60 = *ON
Subfile1AgentName = *Blanks
  Else
    Subfile1AgentName = CustName
  Endif
```

See Also

CHAIN | **DCLKLIST** | **READ** | **READP**

READRANGE

READRANGE positions the file to the desired range of records constrained by the **FirstKey** and the **LastKey** parameters and reads the first record of the range. The range is determined by the combination of the parameters of the command.

This operation is valid only for files opened for indexed access. Use the **ORG (*Indexed)** option on **DCLDISKFILE**.

```
READRANGE
File (FileName | FormatName)
FirstKey (Key List Name | Variable Operand)
LastKey (*FIRSTKEY | Key List Name | Variable Operand)
StartAt (*FIRST | *LAST)
IncludeFirstKey (*YES | *NO)
IncludeLastKey (*YES | *NO)
Access (*DFT | *NOLOCK)
NotFnd (Indicator Variable)
Err (*EXTENDED | Indicator Variable)
Eof (Indicator Variable)
```

Parts

File

Required. Must be the name of a file or a format name. The range operation works on this file or format name.

FirstKey

Required. Can be any valid variable operand or a Key List name that is used as the search argument for the top of the range in positioning a file. If you want to exclude from the range the records whose key is equal to the first key, use **IncludeFirstKey (*No)**, which positions the file to the first record whose key is greater than the **FirstKey**.

LastKey

Optional. Defaults to ***FirstKey** (meaning **LastKey** is the same as the **FirstKey**). This parameter can be any valid variable operand or a Key List name that is used to mark the bottom of the range. If you want to exclude from the range the records whose key is equal to the **LastKey**, use **IncludeLastKey (*No)**, which marks the last record whose key is less than **LastKey** parameter as the bottom of the range.

StartAt

Optional. Specifies the direction the records are read after the Set (either starting with the first record or the last record in the range). ***First** is the default.

***First** – The records read will start at the first record found in the range, using the first key, and continue reading until the last record in the range is read.

***Last** – The records read will start at the last record found in the range, using the last key, and continue reading until the first record in the range is read.

IncludeFirstKey

Optional. Specifies whether to include the very first record found within the range when deleting records. The default is ***Yes**.

***YES** – (Default) If the operation finds a record whose key matches the **FirstKey**, then the file is positioned at that record. If the record that matches the key is not found, then the file is positioned at the first record whose key is greater than the **FirstKey**.

***NO** – The file is positioned on the first record whose key is greater than the **FirstKey**.

IncludeLastKey

Optional. Specifies whether to mark the last record found that matches the **LastKey** as the bottom of the range. The default is *Yes.

*YES – (Default) If the operation finds a record whose key matches the **LastKey**, then the bottom of the range is marked at that record. If the record that matches the key is not found, then the last record whose key is less than the **LastKey** is marked as the bottom of the range.

*NO – The last record whose key is less than the **LastKey** is marked as the bottom of the range.

Access

Optional. Specifies the record locking behavior. If the parameter is not specified (*DFT), the default record locking behavior is taken. See below for a description of the parameters.

*DFT – (Default) If the parameter is not given, the default record locking behavior is taken. This means, if the File is specified as INPUT, all records are read without locks. If File is specified as UPDATE, all records are locked if blank.

*NOLOCK – If this option is specified, then in no case will the record be locked when the **READRANGE** operation is successful.

NotFnd

Optional. Used to determine if the record was found while executing this command.

Note that you can also use the %FOUND function. However, be aware that %FOUND and the **NotFnd** indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. Will be set if any error condition is found with the file. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - a program status flag is set whose status can be checked using the %ERROR built-in function or the Files' IsError property.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Eof

Optional. Use the **Eof** (End-of-File) indicator, the %EOF function, or the Files' IsEof property to determine if the End-Of-File was reached while executing the command.

If any other error occurs, the error indicator will be set. The End of File indicator is also set when no more records exist with the same search argument.

Remarks

Use **READRANGE**, in conjunction with the **SETRANGE** and **DELETERANGE** op codes to optimize SQL Server processing and to enhance client/server performance with all supported database engines (iSeries/400, Acceler8DB and SQL Server) with dynamic Network Blocking.

Use a **READRANGE** – Read(P) loop instead of Chain – Read(P)E loop.

To use a **READRANGE** in existing code, replace the **CHAIN** op code that exists outside of the **CHAIN/READ** loop with a **READRANGE**. Then replace the **READ** op code inside the loop with a **READP**. The **READRANGE** establishes the range of records to be read and proceeds to read the first record within the range. Subsequent uses of the read op code will only retrieve records that fall within that range. Once the last record within the range has been read, a subsequent use of the read op code will set on Eof.

See Also

DELETERANGE | DCLDISKFILE | SETRANGE | %EOF | %ERROR | IsEof Property | IsError Property

REMOVEHANDLER

Removes a previously attached event handler from an object's event.

```
REMOVEHANDLER
SourceObject (Object)
SourceEvent (Name)
HandlerObject (Object)
HandlerSr (Name)
```

Parts

SourceObject

Required. The name of the object firing the event that you do not want to handle any more.

SourceEvent

Required. The name of the event that you do not want to handle.

HandlerObject

Optional. Object containing the current event handler subroutine. The default is *This class.

HandlerSr

Required. Name of the subroutine that currently handles the event.

Remarks

The ADDHANDLER command associates a subroutine as the event handler for an object's event.

REMOVEHANDLER breaks the association so that the subroutine will stop being invoked when the event fires.

See Also

ADDHANDLER

RNMFLDS - Rename Externally Described Fields

RNMFLDS allows fields from an externally described record format to be individually renamed to an internal field name.

```
RNMFLDS
Fmt (Format name)
Flds (External field name, Internal field name, ...)
```

Parts

Fmt

Required. **Fmt** must contain the name of the record format containing the fields to be renamed.

Flds

Required. **Flds** must contain matched pairs of external field and internal field names. For readability, a colon may be used to separate pairs of field names.

Example

```

RnmFlds Fmt ( RCMasterL2) Flds ( CM CustNo, myCust#,          +
                                CMName, myName,             +
                                CMAddr2, myAddress )
RnmFlds Fmt ( myFormat ) Flds ( Fieldone, myFld1, FieldTwo, myFld2,
Fieldthree, myFld3 )
RnmFlds Fmt ( myFormat ) Flds ( Field1, myFld1 : Field2, myFld2 : Field3,
myFld3 )

```

See Also

DCLDISKFILE | DCLPRINTILE

ROLLBACK

ROLLBACK performs several functions:

Cancels all the changes to your files that have been specified in output operations to files under commitment control since the previous **COMMIT** or **ROLLBACK** operation.

Repositions the file to its position at the time of the previous **COMMIT** operation (or at the time of the file **OPEN**, if there has been no previous **COMMIT** operation.)

Releases all record locks for the files you have under commitment control.

```

ROLLBACK
Db (Database Object)
Err (Indicator Variable | *EXTENDED)

```

Parts**Db**

Required. The database connection for which transaction is to be rolled back. Database Objects are defined with the DCLDB command.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

The file changes are eliminated to all the file you have under commitment control for the specified database connection.

COMMIT and **ROLLBACK** operations are supported for the iSeries DB2/400 and Windows SQL Server database engines. The **COMMIT/ROLLBACK** operations only apply to database files opened with the **COMMIT** (*YES) parameter.

See Also

COMMIT | DCLDB | ENDTPM | OPEN | STARTTPM | Using Commitment Control

SCAN

Scans a base string for an occurrence of the compare string. A starting position within the base string can be specified as well as the length of base string to be compared.

```
SCAN
CmpStr (Character Expression)
CmpLen (*ALL | Numeric Expression)
BaseStr (Character Expression)
StartPos (Numeric Expression | 1)
FndPos (*NONE | Numeric Expression)
Err (Indicator Variable | *EXTENDED)
Fnd (Indicator Variable)
```

Parts

CmpStr

Required. Must contain the string to be compared to **BaseStr** and be character type. It can be a field name, array element, named constant, data structure name, literal, or table name. Leading, trailing, or embedded blanks specified in the compare string are included in the **SCAN** operation.

CmpLen

Optional. Must be numeric with no decimal positions. It can be a field name, named constant, array element, literal, or table name. The default value is the length of **CmpStr**.

BaseStr

Required. Must contain the base string to be compared to. It must be character type. It can be a field name, array element, named constant, data structure name, literal, or table name.

StartPos

Optional. Must be numeric type with no decimal positions and can be a named constant, array element, field name, literal, or table name. The value can not be greater than the length of **BaseStr**. If no start location is specified, a value of 1 is used.

FndPos

Optional. This parameter must be specified if **Fnd** is not. If specific, it must be a numeric field with zero decimal positions and can contain a field name, array element, array name, or table name and will contain the numeric value of the first occurrence (from left) of **CmpStr** in **BaseStr**, if found.

FndPos is set to 0 if the string is not found. If this contains an array, each occurrence of the compare string is placed in the array with the leftmost occurrence in element 1. The array elements following the element containing the rightmost occurrence are all zero. The resulting array should be as large as the field length of the base string.

Note the following:

The strings are indexed from position 1.

If the start position is greater than 1, the result contains the position relative to the beginning of **BaseStr**, not relative to the start position.

Figurative constants cannot be used in the **CmpStr**, **BaseStr**, or **FndPos**.

No overlapping within data structures is allowed for **CmpStr** and the result or **StartPos** and the result.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Find

Optional. **Find** (or FD indicator) will be set if the string being scanned for is found. This parameter must be specified if **FindPos** is not. Note that you can also use the %FOUND function.

Remarks

SCAN begins at the leftmost character of **BaseStr** at **StartPos** and continues character by character, from left to right, comparing the characters in **BaseStr** to those in **CmpStr** for the **CmpLen** specified. If the result is not an array, the **SCAN** operation will locate only the first occurrence of the compare string. To continue scanning beyond the first occurrence, use the result from the previous **SCAN** operation to calculate the starting position of the next **SCAN**. If the result is a numeric array, as many occurrences as there are elements in the array are noted.

If no occurrences are found, **FindPos** is set to zero. If **FindPos** is an array, all its elements are set to zero.

The **SCAN** operation is case-sensitive. A compare string specified in lowercase will not be found in a base string specified in uppercase.

Example

/* The SCAN operation finds the substring 'RPG' starting in position 5 in BaseStr; 5 is placed in the FindPos(Result). Indicator 90 is set on because the string is found. Because no starting position is specified, the default of 1 is used. */

```
Scan CmpStr('RPG') BaseStr('ASNARPG') FindPos(Result) Find(*In90)
```

/* This SCAN operation scans the string in Field1(F2) for an occurrence of the string in Field2 (F1) starting at position 3. The 'Y' in position 1 of the base string is ignored because the scan operation starts from position 3. The operation places the values 5 and 6 in the first and second elements of the MyArray array. Indicator 90 is set on. */

```
DclArray Name(MyArray) DCLFLD(10) Type(*zoned) Len(1,0)
Field1 = 'YARRY'
Field2 = 'Y'
Scan Field2 BaseStr(Field1) StartPos(3) FindPos(MyArray) Find(*In90)
```

/* This SCAN operation scans the string in Field1 (F2), starting at position 2, for an occurrence of the string in Field2(F1) for a length of 3. Because 'RPG' is not found in FIELD1, INT90 is set to zero and indicator 90 is set off. */

```
Field1 = 'Visual'
Index = 2
Field2 = 'RPG'
Scan Field2 3 Field1 Index Int90 Find(*In90)
```

See Also

Character Expression | Numeric Expression

SELECT

Conditionally processes one of several alternative sequences of operations. It consists of a **SELECT** statement, zero or more WHEN groups, an optional OTHER group, and an END or ENDSL statement.

SELECT

Remarks

The **SELECT** group can be specified anywhere in calculations. It can be nested within IF and DO select groups. The **IF** and **DO** groups can also be nested within **SELECT** groups.

Example

/* In this Example, if X equals 1, do the operations in sequence 1 (note that no **END** operation is needed before the next WHEN); if X does NOT equal 1, and if Y=2 and X<10, do the operations in sequence 2. If neither condition is true, do the operations in sequence 3. */

```
SELECT
  When X = 1
    B = A + B
    ExSR UpdateMaster
  When X < 10 *and Y = 2
    B = B - A
  Other
    ExSR ErrorRoutine
EndSL
```

/* This Example shows a **SELECT** group with conditioning indicators. After the ReadRand operation, if indicator 10 is on, then control passes to the ADD operation. If indicator 10 is off, then the **SELECT** group is processed. */

```
ReadRand ItemFile ItemKey NotFnd(*In10)
If *In10 = *Off
  SELECT
    When X = 1
      B = B + Q
      ExSR Update NewOrder
    When X = 2
      ExSR NewOrderForm
  EndSL
EndIf
```

See Also

END | ENDSL | IF | OTHER

SET

The SET command allows for a reference assignment onto a data structure.

```
SET
F2 (expression)
Result (Data structure>)
```

Parts

F2

Required. The expression for the data structure. It must evaluate to a data structure object of the same type as the *Result* data structure.

Result

Optional. Contains a character variable to contain the result of the SET.

Remarks

F2 must evaluate to a data structure object of the same type as the *Result* data structure.

See Also

DCLDS | DCLDSFLD

SETASSEMBLYATTRIBUTE

Allows you to set an attribute on an assembly.

```
SETASSEMBLYATTRIBUTE
Attribute (Argument)
```

Parts

Attribute

Required. Specifies the attribute to be set.

Remarks

To specify more than one attribute for the assembly use multiple **SETASSEMBLYATTRIBUTE** commands. This command can be coded in any of your classes, but must come **before** the **BEGCLASS** and after any **USING** commands.

Typically, custom attributes' names end up with the word 'Attribute', so the Attribute part of the name can be omitted.

It is common to create a dedicated source file called AssemblyInfo.vr where all assembly attributes are grouped. This file usually lacks a class definition, i.e.: it contains no **BEGCLASS** whatsoever.

Example

Here is an Example of an AssemblyInfo.vr using System.Reflection

```
SetAssemblyAttribute Attribute (AssemblyDescription("AllParms_Test")
SetAssemblyAttribute AssemblyCompany("ASNA")
SetAssemblyAttribute AssemblyVersionAttribute("1.0.0")
SetAssemblyAttribute AssemblyTitle("THE ALLPARMS PROGRAM")
SetAssemblyAttribute AssemblyTrademark("ASNA")
```

See Also

Custom Attributes

SETGT

Positions a file at the next record with a key or relative record number that is greater than the key or relative record number specified in Key.

```
SETGT
File (file name | Format Name)
Key (KeyList Name | Character Expression | *START | *END)
NotFnd (Indicator Variable)
Err (*EXTENDED | Indicator Variable)
```

Parts

File

Required. Must be a file name or format name.

Key

Required. **Key** can be a field name, a named constant, a figurative constant, or a literal that is used as the search argument in positioning a file. You can also specify *START or *END to position the file at the beginning or end of a file. Refer to %KDS to use this built-in function to establish search arguments (Key) for Data Structures.

*START - positions the file at the beginning of file, so the **READ** returns the 1st record.

*END - positions the file at the end of the file, so the **READP** returns the last record of the file.

NotFnd

Optional. Indicator variable used to determine if the record was found while executing this command.

Note that you can also use the %FOUND function. However, be aware that %FOUND and the NotFnd indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you must use the format name.

Following the **SETGT** operation, a file is positioned so that it is immediately before the first record whose key or relative record number is greater than the search argument specified in **Key**. Read the file to retrieve this record. However, before you read the file, records may be deleted from the file by another job or through another file in your job. Thus, you may not get the expected record.

Example

/* This Example shows how to position the file so READ will read the next record. The search argument, ITEMNUMBER, specified for the SETGT operation has a value of 98; therefore, SETGT positions the file before the first record of file format ITEMMASTER that has a key field value greater than 98. The file is positioned before the first record with a key value of 100. The READ operation reads the record that has a value of 100 in its key field. */

```
DclFld NotFound *Ind
DclFld ErrorOnSet *Ind
DclFld EndOfFile *Ind
SETGT ItemMaster ItemNumber Err (ErrorOnSet) NotFnd(NotFound)
If NotFound = *Off *And ErrorOnSet = *Off
    Read ItemMaster Eof(EndOfFile)
Endif
```

/* This Example shows how to read the last record of a group of records with the same key value and format from a program Described file. The search argument, ITEMNUMBER, specified for the SETGT operation positions the file before the first record of file PARTMASTER that has a key field value greater than 70. The file is positioned before the first record with a key value of 80. The READP operation reads the last record that has a value of 70 in its key field. */

```
DclFld NotFound *Ind
DclFld ErrorOnSet *Ind
DclFld EndOfFile *Ind
SETGT PartMaster ItemNumber Err (ErrorOnSet) NotFnd(NotFound)
If NotFound = *Off *And ErrorOnSet = *Off
    Read PartMaster Eof(EndOfFile)
Endif
```

key field		key field
Values		Values
97		50 /FONT>
97		60
97		70
97	(READP)	70
98	ITEMMASTER (SETGT)	80 PARTMASTER
(SETGT)(READ) 100		80
100		80
100		90
101		90
101		91

/*This Example shows the use of *LOVAL. The SETLLOperation positions the file before the first record of a file in ascending order. The READ operation reads the first record (key value 97)./*

```
DclFld NotFound *Ind
DclFld ErrorOnSet *Ind
DclFld EndOfFile *Ind
SETLL PartMaster *LoVal Err (ErrorOnSet) NotFnd(NotFound)
If NotFound = *Off *And ErrorOnSet = *Off
    Read PartMaster Eof(EndOfFile)
Endif
```

/*This Example shows the use of *HIVAL. The SETGT operation positions the file after the last record of a file in ascending order. The READP operation reads the last record (key value 91)./*

```
DclFld NotFound *Ind
DclFld ErrorOnSet *Ind
DclFld EndOfFile *Ind
SETGT ItemMaster *HiVal Err (ErrorOnSet) NotFnd(NotFound)
If NotFound = *Off *And ErrorOnSet = *Off
```

```
ReadP ItemMaster Eof(EndOfFile)
Endif
```

	key		key
	field		field
	Values		Values
(SetLL) (Read)	97		50
	97		60
	97		70
	97		70
	98	PARTRECORD	80
		record format	ITEMRECORD
			D
			record format
	100		80
	100		80
	100		90
	101		90
	101	(READP)	91
		(SETGT)	

See Also

*HIVAL | *LOVAL | CHAIN | Figurative Constants | SETLL | *START | *END | %KDS

SETLL

Positions a file at the next record that has a key or relative record number that is less than or equal to the search argument.

```
SETLL
File (file name | Format Name)
Key (KeyList Name | Character Expression | *START | *END)
NotFnd (Indicator Variable)
Err (*EXTENDED | Indicator Variable)
KeyEq (Indicator Variable)
```

Parts

File

Required. Must be a file name or format name.

Key

Required. **Key** can be a field name, a named constant, a figurative constant, or a literal that is used as the search argument in positioning a file. You can also specify *START or *END to position the file at the beginning or end of a file. Refer to %KDS to use this built-in function to establish search arguments (Key) for Data Structures.

*START - positions the file at the beginning of file, so the **READ** returns the 1st record.

*END - positions the file at the end of the file, so the **READP** returns the last record of the file.

NotFnd

Optional. Indicator variable used to determine if the record was found while executing this command.

Note that you can also use the %FOUND function.

However, be aware that %FOUND and the **NotFnd** indicator have **opposite** meanings: one will be **False** when the other one is **True** and vice versa.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

KeyEq

Optional. This indicator is set on when a record is present whose key or relative record number is equal to the search argument. This information can also be obtained from the %EQUAL built-in function, which returns **True** if an exact match is found.

If the file is a multiformat file accessed by RRN, you cannot perform the operation using the file name, you must use the format name.

Remarks

If you want to process a "Range" of records, you could use the SETRANGE op code. This op code is especially useful in increasing performance and speed when you are working with large files. Also, if you will be accessing SQL files, you definitely want to use **SETRANGE**. The **SETRANGE** op code allows you to specify a range of records to process by specifying a first and last key. However, the last key defaults to the same as the FirstKey, so if you are only searching 1 key, you basically just change the **SETLL**.

Example

/* In this Example, the file OrderMasterFile contains order records. The key field is the order number (OrderNumber) field. There are multiple records for each order. OrderMasterFile looks like this. */

OrderMasterFile

Order	Other fields
100	1st record of 100
100	2nd record of 100


```
(SETLL)      3rd record of 100

100

101          1st record of 101

101          2nd record of 101

101          3rd record of 101

101          4th record of 101

102          1st record of 102
```

/* All the 101 records in OrderMasterFile are to be printed. The value 101 has previously been placed in OrderNumber. The SETLL operation positions the file at the first record with the key value 101 and indicator 55 is set on.*/

```
SetLL OrderMasterFile OrderNumber NotFnd(*In55) Err (*In99)
```

/* The following DO loop processes all the records that have the same key value. Indicator 55 is used for conditioning the DO loop.*/

```
*If *In55 = *Off
Dountil *In56 = *On
Read OrderMasterFile OrderNumber Eof(*In56)
Write DetailPrintLine
Enddo
Endif
```

The READE operation reads the second, third, and fourth 101 records in the same manner as the first 101 record was read. After the fourth 101 record is read, the READE operation is attempted. Because the 102 record is not of the same group, indicator 56 is set on, the Write operation is bypassed and the Dountil loop ends.

See Also

CHAIN | READE | READRANGE | SETGT | SETRANGE | *START | *END | %ERROR | %EQUAL | %FOUND | %KDS

SETMOUSEPTR

Sets the shape of the mouse pointer for an entire application.

```
SETMOUSEPTR
Icon (*DFT | *APPSTARTING | *ARROW | *CROSS | *IBEAM | *ICON | *NODROP |
      *SIZE | *SIZENESW | *SIZENS | *SIZENWSE | *SIZEWE | *UPARROW | HOURLASS
      | Character Expression)
Err (Indicator Variable | *EXTENDED)
```

Parts

Icon

Required. Must contain a special value or a character expression. In character expression, you can enter the icon file name, or assign a picture property to a control e.g. (Image1.Picture). The following is a list of available special values.

*Dft - The mouse pointer of the window in which the mouse is on.

*Arrow - Arrow

*Cross - Cross Hair

*IBeam - I-Beam
*Size - Four-pointed arrow pointing North, South, East, West
*SizeNESW - Double-arrow pointing NorthEast and SouthWest
*SizeNS - Double-arrow pointing NorthEast and SouthWest
*SizeNWSE - Double-arrow pointing NorthWest and SouthEast
*SizeWE - Double-arrow pointing West and East
*UpArrow - Up arrow
*Hourglass - Hour glass
*NoDrop - No drop
*AppStarting - application starting

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

The MousePointer property on the form controls the shape of the mouse pointer when the mouse is over that form. However, if moved to another form, you would get the mouse pointer that is specified for that form. The **SETMOUSEPTR** op code will specify one mouse pointer to be used for all forms within the application.

SETOFF

Sets the indicators specified in the resulting indicators OFF.

```
SETOFF  
Hi (Indicator Variable)  
Lo (Indicator Variable)  
Eq (Indicator Variable)  
IndList (Indicator Variable, Indicator Variable, ...)
```

Parts

Hi

Optional. Sets off the Hi resulting indicator variable.

Lo

Optional. Sets off the Lo resulting indicator variable.

Eq

Optional. Sets off the Eq resulting indicator variable.

IndList

Optional. Allows you to specify all of the resulting indicators, separated by commas.

Remarks

You can set off any combination of the resulting indicators, or you can specify all of them using the **IndList** parameter.

Example

```
DclFld NotFoundIndicator *Ind
SETOFF *In50
SETOFF NotFoundIndicator
-or-
*In50 = *Off
NotFoundIndicator = *Off
```

See Also

SETON

SETON

Sets the indicators specified in the resulting indicators ON.

```
SETON
Hi (Indicator Variable)
Lo (Indicator Variable)
Eq (Indicator Variable)
IndList (Indicator Variable, Indicator Variable, ...)
```

Parts**Hi**

Optional. Sets on the Hi resulting indicator variable.

Lo

Optional. Sets on the Lo resulting indicator variable.

Eq

Optional. Sets on the Eq resulting indicator variable.

IndList

Optional. Allows you to specify all of the resulting indicators, separated by commas.

Remarks

You can set on any combination of the resulting indicators, or you can specify all of them using the **IndList** parameter.

Example

```

DclFld NotFoundIndicator *Ind
Seton *In50
SETON NotFoundIndicator

-or-

*In50 = *On
NotFoundIndicator = *On

```

See Also

SETOFF

SETRANGE

SETRANGE positions the file to the desired range of records constrained by the FirstKey and the LastKey parameters. The range is determined by the combination of the parameters of the command.

```

SETRANGE
File (file name | Format Name)
FirstKey (Key List Name | Variable Operand)
LastKey (*FIRSTKEY | Key List Name | Variable Operand)
StartAt (*FIRST | *LAST)
IncludeFirstKey (*YES | *NO)
IncludeLastKey (*YES | *NO)
Err (*EXTENDED | Indicator Variable)

```

Parts**File**

Required. Must be the name of a file or a format name. The range operation works on this file or format name.

FirstKey

Required. If the file is accessed by key, this parameter can be a field name, a named constant, a figurative constant, or a literal that is used as the search argument for the top of the range in positioning a file. If you want to ignore the top first record of the specified range to be deleted, use IncludeFirstKey (*NO).

LastKey

Optional. Defaults to *FirstKey (meaning LastKey is the same as the FirstKey). This parameter can be a field name, a named constant, a figurative constant, or a literal that is used as the search argument for the bottom of the range in positioning a file. If you want to ignore the bottom last record of the specified range to be read, use IncludeLastKey (*NO).

StartAt

Optional. Specifies the direction the records are read after the Set (either starting with the first record or the last record in the range). *First is the default.

*FIRST- (Default) The records read will start at the first record found in the range, using the first key, and continue reading until the last record in the range is read.

*LAST - The records read will start at the last record found in the range, using the last key, and continue reading until the first record in the range is read.

IncludeFirstKey

Optional. Specifies whether to include the very first record found within the range. The default is *YES.

*YES – (Default) If the operation finds a record whose key matches the **FirstKey**, then the file is positioned at that record. If the record that matches the key is not found, then the file is positioned at the first record whose key is greater than the **FirstKey**.

*NO – The file is positioned on the first record whose key is greater than the **FirstKey**.

IncludeLastKey

Optional. Specifies whether to mark the last record found that matches the **LastKey** as the bottom of the range. The default is *Yes.

*YES –(Default) If the operation finds a record whose key matches the **LastKey**, then the bottom of the range is marked at that record. If the record that matches the key is not found, then the last record whose key is less than the **LastKey** is marked as the bottom of the range.

*NO – The last record whose key is less than the **LastKey** is marked as the bottom of the range.

Err

Optional. Will be set if any error condition is found with the file. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

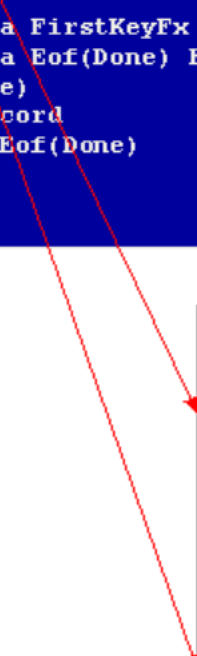
Remarks

Use **SETRANGE**, in conjunction with the **READRANGE** and **DELETERANGE** op codes to optimize SQL Server processing and to enhance client/server performance with all supported database engines (iSeries/400, Acceler8DB and SQL Server) with dynamic Network Blocking.

Use a **SETRANGE – Read(P)** loop instead of a **SETLT/SETGT – Read(P)E** loop.

Example

```
FirstKeyFx = 'D'  
LastKeyFx = 'F'  
SetRange SomeData FirstKeyFx LastKey(LastKeyFx)  
Read      SomeData Eof(Done) Err (Done)  
DoWhile (Not Done)  
  ExSr ProcessRecord  
  Read SomeData Eof(Done)  
EndDo
```

B

Fx	Fy	Fz
B	1	Ask
D	2	Not
D	3	What
D	4	Your
F	5	Country
F	6	Can
K	7	Do
R	8	For
S	9	You

If IncludeFirstKey (*No)

```

FirstKeyFx = 'D'
LastKeyFx = 'F'
SetRange SomeData FirstKeyFx LastKey(LastKeyFx)
      IncludeFirstKey(*No)
Read      SomeData Eof(Done) Err (Done)
DoWhile (Not Done)
  ExSr ProcessRecord
  Read SomeData Eof(Done)
EndDo

```

B

F _x	F _y	F _z
B	1	Ask
D	2	Not
D	3	What
D	4	Your
F	5	Country
F	6	Can
K	7	Do
R	8	For
S	9	You

If IncludeLastKey (*No)

```

FirstKeyFx = 'D'
LastKeyFx = 'F'
SetRange SomeData FirstKeyFx LastKey(LastKeyFx)
      IncludeLastKey(*No)
Read      SomeData Eof(Done) Err (Done)
DoWhile (Not Done)
  ExSr ProcessRecord
  Read SomeData Eof(Done)
EndDo

```

B

Fx	Fy	Fz
B	1	Ask
D	2	Not
D	3	What
D	4	Your
F	5	Country
F	6	Can
K	7	Do
R	8	For
S	9	You

See Also

DELETERANGE | DCLDISKFILE | READRANGE

SHOW

Shows the form instance as a modeless dialog box.

```

SHOW
Form (form variable)
Err (*EXTENDED | Indicator Variable)

```

Parts

Form

Required. Contains the name of the form to show.

Err

Optional. The **Err** parameter determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

You can use this method to display a modeless dialog box in your application. Showing the Form is equivalent to setting the **Visible** property to true. After the **SHOW** is executed, the **Visible** property returns a value of true until the **Hide** method is called or **HIDE** command is executed.

SHOW is equivalent to the **Show** method except that the Form from which the **SHOW** is executed is given ownership of the Form shown. The behavior of this ownership is when the owning Form is minimized, the shown Form is also minimized.

Example

```

BegClass Form1 Extends(System.Windows.Forms.Form) Access(*Public)
  DclFld F2 Type( Form2 )
  BegSr SHOWExample Access(*Private) Event(*this.btnSHOW.Click)
    DclSrParm sender Type(*Object)
    DclSrParm e Type(System.EventArgs)
    F2 = *new Form2()      // create instance of Form2
    SHOW F2                // same as F2.Show() but F2 is owned by Form1
  EndSr
EndClass

```

See Also

Control.Hide method | Form.ShowDialog method | Control.Show method | EXFMT | Form Variables | HIDE | UNLOAD

SLEEP

Sleep suspends execution of the current thread for the specified milliseconds.

```

SLEEP
Interval (Expression)

```

Parts

Interval

Required. Specifies the time, in milliseconds, for which to suspend execution.

Remarks

A thread can relinquish the remainder of its time slice by calling this function with a sleep time of zero milliseconds.

You have to be careful when using **SLEEP** and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **SLEEP** with infinite delay, the system will deadlock.

SORTARR

Sorts an array.

```

SORTARR
Arr (Array Name)

```

Parts

Arr

Required. Contains the name of an array to be sorted. The array is sorted in ascending or descending order, depending on the sequence specified for the array in the extension specifications. The default is ascending order.

Remarks

In a **SORTARR** operation, array *IN cannot be designated in **Arr**.

Sorting an array does not maintain the previous order.

Example

/* The SORTARR operation sorts MyArray into descending sequence because the array is defined as descending. If the unsorted MyArray contents were ASNARPGZ, the sorted MyArray contents would be ZSRPNGAA. */

```
DCLARRAY Name(MyArray) dim(8) Len(1) Seq(*Descend)
SORTARR MyArray
```

See Also

DCLARRAY

SQRT

Specifies the square root of **F2** and places it in **Result**.

```
SQRT
F2 (Numeric Expression | Character Expression)
Result (Numeric Variable | Character Variable)
Adj (*TRUNC | *RNDUP)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT | FLOAT4
      | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

F2

Required. Must be any valid numeric operand. It must be numeric, and can contain one of: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

Result

Required. Must be numeric, and can contain one of: an array, array element, subfield, or table element. An entire array can be used in a **SQRT** operation if **F2** and **Result** contain array names. The number of decimal positions in **Result** can be either less than or greater than the number of decimal positions in **F2**. However, **Result** should not have fewer than half the number of decimal positions in **F2**.

If the value of the **F2** field is zero, the **Result** value is also zero. However, if the value of **F2** is negative, this is invalid, and a run-time error will occur.

Adj

Optional. Specifies if the field will be adjusted by truncation (*TRUNC) or by rounding up the value (*RNDUP). *TRUNC is the default.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

The **SQRT** operation derives the square root of **F2** and places it in **Result**. This operation is done in floating point mathematics, so some truncation/roundup may occur.

Example

```
// When NumberDresses equals 144, the result, PerBoxCount, will be 12.
DclFld NumberDresses *ZONED len(5,0)
DclFld PerBoxCount *ZONED len(3,0)
Sqrt NumberDresses PerBoxCount
/* When an array is used, all elements are computed at once and placed in
the resulting array. In this case, each element in array MinGroup would be
12. Each element will be rounded to the nearest decimal value. */
DclFld I *ZONED Len(3,0)
DCLARRAY Name(MajGroup) DIM(20) Len(9,0)
DCLARRAY Name(MinGroup) DIM(20) Len(9,0)
Do 1 20 I
    MajGroup[I] = 144
    MinGroup[I] = 0
Enddo
Sqrt MajGroup MinGroup *RndUp
```

See Also

%SQRT

STARTTPM

The STARTTPM (Start Transaction Processing Monitor) op code is used to establish a commitment definition. This command specifies the level of record locking that occurs for the commitment definition to be started. You can specify some text that briefly describes the commitment definition to be started.

```
STARTTPM
Db (Database Object)
Level ( *LOW | *MEDIUM | *HIGH | *SERIAL)
Text (*NONE | Character Expression)
ExtraParms (Parm List)
Err (Indicator Variable | *EXTENDED)
```

Parts

Db

Required. The database connection for which transaction processing is to be started. Database Objects are defined with the DCLDB command.

Level

Optional. Specify the level of record locking that occurs for the commitment definition to be started.

*LOW - *CHG ReadUncommitted - Do not issue shared locks and do not honor exclusive locks. When this option is in effect, it is possible to read an uncommitted transaction or a set of pages that are rolled back in the middle of a read. Dirty reads are possible.

*MEDIUM - *CS ReadCommitted - Perform a scan with the same locking semantics as a transaction running at the READ COMMITTED isolation level. By default, SQL Server 2000 operates at this isolation level.

*HIGH - *ALL RepeatableRead - Perform a scan with the same locking semantics as a transaction running at the REPEATABLE READ isolation level.

*SERIAL - *ALL Serializable - Hold a shared lock until completion of the transaction instead of releasing the lock as soon as the required table, row, or data page is no longer required.

Text

Optional. Specify some text that briefly describes the commitment definition to be started.

ExtraParms

Optional. Specify some extra parameters to be used.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

See Also

COMMIT | DCLDB | ENDTPM | ROLLBACK | Using Commitment Control

SUB

F2 is subtracted from **F1** (if specified) and the difference is placed in **Result**.

```
SUB
F1 (*NONE | Numeric Expression)
F2 (Numeric Expression)
Result (Numeric Variable)
Adj (*TRUNC | *RNDUP)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

F1

Optional.

F2

Required. Contains a numeric value. The contents of **F2** is subtracted from the **Result** if **F1** is not specified.

Result

Optional. Contains a numeric variable to contain the result of the subtraction.

Adj

Optional. Indicates half-adjust. Use **ADJ** (*RNDUP) if you wish to round up the value, or use **ADJ** (*TRUNC) if you wish to truncate the value of the **Result**. *TRUNC is the default.

Pos

Optional. Turned on if the value of the **Result** is a positive number.

Neg

Optional. Turned on if the value of the **Result** is a negative number.

Zero

Optional. Turned on if the value of the **Result** is zero.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

F1, **F2** and **Result** must be numeric.

When **F1** is specified, **F2** is subtracted from **F1** with the difference placed in the **Result**. The contents of **F2** is subtracted from the **Result** if **F1** is not specified.

Example

```
B = 10.0; C = 32; D = -20; E = 6
DclFld W *PACKED Len(5,1)
E = 1 - E      // E = 5
W = C - B      // W = 22.0
W = C - D      // W = 52.0
```

See Also

ADD | DIV | MULT

SUBDUR

SUBDUR either subtracts the duration specified in **F2** from a date or time and places the resulting date, time or timestamp in the **Result**, or calculates a duration between two **date/time** values.

```
SUBDUR
F1 (Date | Time | Timestamp)
F2 (Date | Time | Timestamp | Numeric Duration)
F2Fmt (*NONE | *DAYS, *MONTHS, *YEARS, *MSECONDS, *SECONDS, *MINUTES, HOURS)
Result (Date | Time | Timestamp | Numeric Duration)
ResultFmt (*NONE | *DAYS, *MONTHS, *YEARS, *MSECONDS, *SECONDS, *MINUTES,
           *HOURS)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
Err (Indicator Variable | *Extended)
```

Parts

F1

Optional. May contain a date, time or timestamp field, array, array element, literal or constant.

If **F1** contains a field name, array or array element, then its data type must be the same type as the field specified in the **Result**.

F2

Required. A numeric field, array or constant with zero decimal positions. If the field is negative then the duration is added to the field.

F2Fmt

Optional. Must be a valid duration code indicating the type of duration. The duration code must be consistent with the **Result** data type. For Example, you can subtract a year, month or day duration but not a minute duration from a date field.

Result

Required. The name of a zero decimal numeric field, array or array element in which the result of the operation will be placed.

ResultFmt

Optional. Must be a date, time or timestamp data type field, array or array element.

If **F1** is blank, the duration is subtracted from the value in the **Result**.

If the **Result** is an array, the value in **F2** is subtracted from each element in the array.

If the **Result** is a time field, the result will always be a valid Time. For Example, subtracting 59 minutes from 00:58:59 would give -00:00:01. Since this time is not valid, the run-time adjusts it to 23:59:59.

Type

Optional. Can be any of the **Types** listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Err

Optional. Determines what the run-time will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

Subtracting a Duration:

Use **SUBDUR** to subtract a duration specified in **F2** from a field or constant specified in **F1** and place the resulting date, time or timestamp in the field specified in **Result**.

When subtracting a duration in months from a date, the general rule is that the month portion is decreased by the number of months in the duration, and the day portion is unchanged. The exception to this is when the resulting day portion would exceed the actual number of days in the resulting month. In this case, the resulting day portion is adjusted to the actual month end date. See Adding or Subtracting Dates for more information.

The system places a 18 digit limit on durations. Subtracting a Duration with more than 18 significant digits will cause errors or truncation. These problems can be avoided by limiting the first factor in **F2** to 18 digits or less.

Possible Error Situations:

If **F1** is blank and the value of the **Result** before the operation is invalid or if the result of the operation is invalid.

If the value of the Date, Time or Timestamp field in **F1** is invalid.

Calculating a Duration:

The **SUBDUR** operation can also be used to calculate a duration between:

Two times

A time and a timestamp

Two dates

A date and a timestamp

Two timestamps

The data types in **F1** and **F2** must be compatible types as specified above.

Possible Error Situations:

If **F1** is blank and the value of the **Result** before the operation is invalid or if the result of the operation is invalid.

If the value of the Date, Time or Timestamp field in **F1** is invalid.

Example

```
'97/07/21' SUBDUR2:*MONTH results in '97/05/21'
'97/07/21' SUBDUR1:*YEAR results in '96/07/21'
'97/07/21' SUBDUR5:*DAY results in '97/07/16'
SUBDUR F1(isodate2) F2(isodate) Result(duration) ResultFmt(*Days)
```

See Also

ADDDUR | Adding or Subtracting Dates | EXTRACT

SUBSTR (Return substring)

Returns a substring from **BaseStr**, starting at the location specified in **BaseStr** for the length specified in **SubLen**, and places it in the **Target**.

```

SUBSTR
BaseStr (Character Operand)
StartPos (Numeric Expression | 1)
SubLen (Numeric Expression | *ALL)
Target (Character Operand)
Fill (*NOPAD | *PAD)
Err (Indicator Variable | *EXTENDED)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
*FLOAT4 | *FLOAT8 |
      *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *ONECHAR |
*PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])

```

Parts

BaseStr

Required. Must contain either the base character string or the base character string followed by ":"; followed by the start location.

StartPos

Optional. Must be numeric type with no decimal positions and can be a named constant, array element, field name, literal, or table name. If no start location is specified, a value of 1 is used.

SubLen

Optional. Can contain the length value of the string to be extracted from the base string specified in **BaseStr**. **SubLen** must be numeric with no decimal positions and can contain either a field name, array element, table name, literal, or named constant.

Target

Required. The length of the **Target** should be at least as large as the length designated in **SubLen**. Should the substring be longer than the field specified in the **Target**, the substring will be truncated from the right.

The start location and the length of the substring to be extracted must be positive integers.

Fill

Optional. Specifies if you want to fill, or pad the **Target** with blanks. The default is *NOPAD.

*NOPAD indicates that the **Target** will not be padded with blanks.

*PAD indicates that the **Target** will be padded with blanks.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Type

Optional. Can be any of the **Types** listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

To be considered numeric, each character in the field, except the low-order character, must contain a hexadecimal 3 zone and a digit from 0 through 9. The number has a negative value when the low-order character contains a hexadecimal 7 zone, and a digit from 0 through 9 (alphabetic characters 'p' through 'y').

Example

/* The SubStr operation extracts the substring from **BaseStr** starting at position 3 for a length of 2. The value 'NA' is placed in **Target** MyTarget. Indicator 90 is not set on because no error occurred. */

```
Beginhere = 3
MyString = 'ASNARP'
SubStr BaseStr(MyString) StartPos(Beginhere) Sublen(2) Target(MyTarget) Err
(*IN90)
```

/* In this SubStr operation, the length is greater than the length of the string minus the start position plus 1. As a result, indicator 90 is set on and the **Target** is not changed. */

```
BeginHere = 4
MyString = 'ASNARP'
SubStr MyString BeginHere 4 MyTarget Err (*IN90)
```

/* In this SubStr operation, 3 characters are substringed starting at the fifth position of the base string. Because Fill(*Pad) is not specified, only the first 3 characters of MyTarget are changed. When the operation is completed, MyTarget contains '123XXXXX'. */

```
Length = 3
Beginhere = 5
MyString = 'ASNA123'
MyTarget = 'XXXXXXXXX'
SubStr MyString Beginhere Length MyTarget Err (*IN90)
```

/* This Example is the same as the previous one except that "Fill(*Pad) is specified, and the result is padded with blanks. When the operation is completed, MyTarget equals '123bbbbbb'. */

```
Length = 3
Beginhere = 5
MyString = 'ASNA123'
MyTarget = 'XXXXXXXXX'
SubStr MyString Beginhere Length MyTarget Err (*IN90) Fill(*Pad)
```

/* In the following Example, CityState contains the string 'San Antonio, Texas'. The Scan operation is used to locate the separating comma (position 12 in this illustration). SubStr without length to extract places the string starting at position 14 and continuing for the length of the string in field State. When the operation is completed, State contains 'Texas'. */

```
DclFld CommaPlace *PACKED Len(2,0)
Scan CmpStr(',') BaseStr(CityState) FndPos(CommaPlace)
CommaPlace = CommaPlace + 1
SubStr BaseStr(CityState) StartPos(CommaPlace) Target(State) Fill(*Pad)
```

/* Before the operations MyString = 'bbbAsnabbbbb'. MyResult is a 10 character field which contains 'ABCDEFGHJIJ'. The Check operation locates the first nonblank character and sets on indicator 10 if such a character exists. If *IN10 is on, the SubStr operation substrings MyString starting from the first non-blank to the end of MyString. Padding is used to ensure that nothing is left from the previous contents of the **Target**. After the operations MyResult = 'Asnabbbbb'. */

```
DclFld FirstBlank *PACKED Len(2,0)
Check CmpStr(' ') BaseStr(MyString) Startpos(FirstBlank) Fnd(*In10)
If *In10
SubStr MyString FirstBlank Target(MyResult) Fill(*Pad) Err (*In90)
Endif
```

SUMARR

Adds array elements together and places the sum into **Result**.

```
SUMARR
Arr (Array Name)
Result (Numeric Variable)
Adj (*RNDUP | *TRUNC)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
*FLOAT4 | *FLOAT8 | *IND |
*INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 | *ONECHAR | *PACKED |
*ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

Arr

Required. Specifies the name of the array.

Result

Required. The sum of the array elements.

Adj

Optional. Indicates half-adjust. Use **Adj** (*RNDUP) if you wish to round up the value, or use **Adj** (*TRUNC) if you wish to truncate the value of **Result**. *TRUNC is the default.

Pos

Optional. Turned on if the value of **Result** is a positive number.

Neg

Optional. Turned on if the value of **Result** is a negative number.

Zero

Optional. Turned on if the value of **Result** is zero.

Type;

Optional. Can be any of the **Types** listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

If the array is half-adjusted, all elements will be added before rounding occurs. If Result is an element of the array in ARR, the value of the element preceding the SUMARR operation is used to calculate the array.

XFOOT is exactly the same as **SUMARR**, and can be used interchangeably.

Example

```
// MyArray Values are 1.064, 1.703, -1.110.
DclArray MyArray 3 *PACKED Len(4,3)
DclFld MyArraySum *PACKED Len(5,2)
SumArr MyArray MyArraySum *RndUp
// Result 1.66xample
```

TAG

Declares a label that identifies the destination of a **GOTO** operation.

```
TAG
Name (Label name)
```

Parts

Name

Required. Must contain the name of the destination of a **GOTO** operation. This name must be a unique symbolic name, which is specified in the Label of a **GOTO** operation. The name can be used as a common point for multiple **GOTO** operations.

Remarks

The declarative TAG operation names the label that identifies the destination of a **GOTO** operation. It can be specified anywhere within calculations, including within total calculations.

A **GOTO** within a subroutine in the main procedure can be issued to a **TAG** within the same subroutine, detail calculations or total calculations. A **GOTO** within a subroutine in a subprocedure can be issued to a **TAG** within the same subroutine, or within the body of the subprocedure.

The **TAG** command is **not** allowed within a TRY/CATCH block.

See Also

GOTO | LABEL

TESTBITS

Compares the bits in **Mask** with the corresponding bits in the field named in **Fld**.

```
TESTBITS
Mask (Expression)
Fld (Character Expression)
Off (Indicator Variable)
On (Indicator Variable)
Eq (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

Mask

Required. Can contain:

Bit numbers 0-7. Enclose the bit numbers in apostrophes. For Example, to set bits 0, 3, 6 on, enter '036'.

A Character Expression.

Hexadecimal literal (x'01') for Example.

Named constant up to 8 positions long containing the bit numbers to be set on.

Fld

Required. Contains the result of the compare.

Off

Optional. Reflects the status of the result field bits. Assuming a Mask of X'E4' (1100 0100), Off is true if all of the masked bits are **Off** in the field. In this case, for **Off** to be true, the field would contain X'3B' (0011 1011).

On

Optional. Reflects the status of the result field bits. Assuming a Mask of X'E4' (1100 0100), On is true if some of the masked bits are **On** and some the masked bits are **Off** in the field. In this case, the field would contain anything but X'3B' or X'E4'.

Eq

Optional. Reflects the status of the result field bits. Assuming a Mask of X'E4' (1100 0100), Eq is true if all of the masked bits are **On** in the field. In this case, the field would contain X'E4' (1100 0100).

Type

Optional. Can be any of the **Types** listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, **Type** 's of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

When using Bit operations to format a character, you should use both **BITON** and **BITOFF**. **BITON** specifies the bits to be set on (= 1). **BITOFF** to specify the bits to be set off (= 0). You may not get the character you want unless you explicitly set all the bits in a character **On** or **Off**.

See Also

BITOFF | BITON

TESTNUM

Tests a field for the presence of zoned decimal digits and blanks.

```
TESTNUM
Fld (Character Expression)
Num (Indicator Variable)
BlankNum (Indicator Variable)
Blank (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED))
Len (Length Integer, [Decimal Integer])
```

Parts

Fld

Required. The **Fld** parameter is required and contains the character expression to test.

Num

Optional. Turned on if **Fld** contains all numeric characters.

BlankNum

Optional. Turned on if **Fld** contains both numeric characters and at least one leading blank. Fields with embedded blanks will be considered an invalid field.

Blank

Optional. Turned on if **Fld** contains all blanks.

Type

Optional. Can be any of the **Types** listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, TYPES of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

To be considered numeric, each character in the field, except the low-order character, must contain a hexadecimal 3 zone and a digit from 0 through 9. The number has a negative value when the low-order character contains a hexadecimal 7 zone, and a digit from 0 through 9 (alphabetic characters 'p' through 'y').

See Also

TESTBITS | Character Expression

TESTTIME

Allows users to test the validity of time, date, or timestamp fields prior to using them.

```
TESTTIME
TimFmt (*NONE | *DMY | *EUR | *HMS | *ISO | *JIS | *JUL | *MDY | *USA | YMD)
Result (Char | Numeric | Date | Time | Timestamp)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
Err (Indicator Variable | *EXTENDED)
ResultType(*NONE | *DATE | *TIME | *TIMESTAMP)
```

Parts

TimFmt

Optional. Specifies the format for time, date, and timestamp variables. If the **TimFmt** parameter is not specified, the format specified by the program is assumed.

Result

Required. Contains the type of field.

Type

Optional. Can be any of the **Types** listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Err

Required. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err(*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

ResultType

Optional. Unless the **Result** contains fields declared as character or numeric, then a **ResultType** must be specified, as indicated below. *NONE is the default.

If the **ResultType** is *DATE, **TimFmt** is optional and may contain any of the valid **Date** formats.

If the **ResultType** is *TIMESTAMP, **TimFmt** is optional and may contain *ISO.

If the **ResultType** is *TIME, **TimFmt** is optional and may contain any of the valid **Time**

formats.

The *USA time format is not allowed with the **ResultType** *TIME. The *USA time format has an AM/PM restriction that cannot be converted to numeric when a numeric **Result** is used.

Remarks

Numeric fields are tested for valid digit portions of a Date, Time or Timestamp value. Character fields are tested for both valid digits and separators.

See Also

ADD DUR | EXTRACT | SUBDUR | Time Formats

THROW

THROW is used to signal the occurrence of an anomalous situation (exception) during the program execution.

```
THROW
Exception (Type of class filter)
```

Parts

Exception

Required. The exception object. Provides information about the exception to be thrown.

Remarks

The Throw statement throws an exception that you can handle with structured exception-handling code (Try...Catch...Finally) or unstructured exception-handling code (If %Error).

Example

```
Throw *new System.Exception( "Invalid Property Value")
```

See Also

CATCH | FINALLY | TRY | TRY...CATCH...FINALLY

TIME

Accesses the system time of day at any time during program processing. The system time is based on the 24-hour clock.

```
TIME
Target (Numeric | Date | Time | Timestamp)
Type (*BINARY | *BOOLEAN | *BYTE | *CHAR | *DATE | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *TIME | *TIMESTAMP | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

Target

Required. Can specify one of the following into which the time of day or the time of day and the system date are written:

Target	Value Returned	Returned Format
6-digit	Numeric	Time hhmmss
12-digit	Numeric	Time and Date YYMMDD
14-digit	Numeric	Time and Date YYYYMMDD
Time	Time	Format of Result
Date	Date	Format of Result
Timestamp	Timestamp	Format of Result

If **Target** is a numeric field, to access the time of day only, specify **Target** as a 6-digit numeric field.

To access both the time of day and the system date, specify **Target** as a 12 (2-digit year portion) or 14-digit (4-digit year portion) numeric field. The time of day is always placed in the first six positions of **Target** in the following format:

```
hhmmss (hh=hours, mm=minutes, and ss=seconds)
```

If **Target** is a numeric field, then if the system date is included, it is placed in **Target**.

If **Target** is a Timestamp field, the last 3 digits in the microseconds-part is always 000.

The special field *DATE contains the job date and reflects the date when the program was started. These values are not updated when midnight is passed, or when the job date is changed during the running of the program.

Type

Optional. Can be any of the Types listed. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

Design-time controls (DTC's) are no longer supported.

TRY

You can use the **TRY...CATCH...FINALLY** statements for structured exception handling. This allows you to execute a particular block of statements if a specified exception occurs while your code is running. When this happens, the code is said to *throw* the exception, and you *catch* it with the appropriate **CATCH** command.

TRY

Remarks

The **TRY** statements are executed first. If no exception occurs, execution continues after the **ENDTRY**(*).

If an exception occurs, then the systems tries to match the exception to the exception types in the **TRY** commands. The first one that matches causes the exception to be 'trapped' and the catch statements in the matched **TRY** to be executed. If the exception is trapped by a **TRY** then after the catch statements are executed, control goes to the next statement after the **ENDTRY**(*).

If a **FINALLY** command is present, then the finally statements will always be executed at the end of the **TRY/CATCH** sequence or **TRY**/untrapped exception, no matter what.

Rules:

The only branching commands allowed inside a **TRY/CATCH** block are **LEAVE**, **ITERATE**, and **LEAVESR**. Any other branching command (**CAB**, **GOTO**, etc.) is not allowed in this context. It is also illegal to branch out of a **FINALLY** clause (i.e., you cannot use any of the branching commands (**ITERATE**, **CAB**, **LEAVESR**, **GOTO**, etc) in the body of a **FINALLY** clause.

The exception Type **MUST** derive from **System.Exception** (the mother of all exceptions). A **CATCH** with a **System.Exception** type will trap all exceptions.

If you have several **CATCH** commands, the Name must be different at this time.

Example

```
BEGSR TryCatch access(*public)
  Dclfld myFld Type(System.Windows.Forms.Button)
  Dclfld myInt *integer 4
  TRY
    myFld.Show()
    myInt = myInt / 0
  CATCH name( z ) type( System.DivideByZeroException )
    msgbox msg( "by 0" ) Title( z.Message )
  CATCH name( e ) type(
System.Exception )
    msgbox msg( "You can't do that!!!" ) Title(
e.Message )
  FINALLY
    msgbox msg( "wow, this works...")
  ENDTRY
  msgbox "All is well!"
ENDSR
```

See Also

CATCH | **FINALLY** | **THROW** | **TRY...CATCH...FINALLY** | **ENDTRY** | **LEAVE** | **ITERATE** | **LEAVESR** | **CAB** | **GOTO**

UPDATE

UPDATE is a special word that allows the programmer to supply the date at run-time. These special words give the date and time when the project was started. Use the TIME operation code to obtain the current time and date while the program is running.

Remarks

The user dates can be written at output time and are written in the format specified by the control specification.

These special words can be used in three different date formats:

Month/day/year

Year/month/day

Day/month/year

UPDATE gives a 6-character numeric date field.

UPDATE can be edited by using the Y edit code.

User date fields are not **date** data type fields, but are numeric fields.

Example

```
IOFIELD1 = UPDATE
```

See Also

UDAY | UMONTH | UYEAR | TIME

UDAY

UDAY is a special word that gives the **day** the project was started.

Remarks

UDAY (and **UMONTH** and **UYEAR**) will print a 2-position numeric date field when specified.

User date fields are not date data type fields but are numeric fields.

See Also

UPDATE | UMONTH | UYEAR

UMONTH

UMONTH is a special word that gives the month the project was started.

Remarks

UMONTH (and **UDAY** and **UYEAR**) will print a 2-position numeric date field when specified in the output specifications.

See Also

UDAY | UPDATE | UYEAR

UNLOAD

Unloads a form, if not already unloaded.

```
UNLOAD
Form (form variable)
```

Parts

Form

Required. Contains the form variable to unload. The **Form** Variable may also be generically specific, such as *THIS.

Remarks

If the form is currently visible to the user, the form will be removed from view. The program is terminated and all files will be closed that were declared in the form, and all memory allocated for the form program will be deallocated.

UNLOAD does not act immediately when executed from within the form that you're unloading. The **UNLOAD** will happen once the program has returned to the event cycle after executing the **ENDSR** statement.

You can abort an **UNLOAD** during the processing of the **Closing** event for the **Form** class by setting the *CancelEventArgs*'s property to **True**.

A form can also be closed by using **EXITAPP** followed by **LeaveSr**, **EndSr** etc. However, when **EXITAPP** is used to terminate a form, there is no **Closing** event.

See Also

EXFMT | ENDSR | EXITAPP | **LeaveSr** | **HIDE** | **SHOW** | *THIS

UNLOCK

Releases record locks for a file.

```
UNLOCK
Obj (File Name)
Err (Indicator Variable | *EXTENDED)
```

Parts

Obj

Required. Contains the name of the file.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

UNLOCK allows the most recently locked record to be unlocked for an update disk file.

See Also

DCLDISKFILE | READ | UPDATE

UPDATE

Modifies the last record retrieved for processing from an update disk file.

```
UPDATE
File (file name | Format Name)
Err (Indicator Variable | *EXTENDED)
```

Parts

File

Required. Must contain the file or record format to be updated. For subfiles, the file name is the SubfileControl name. The relative record number to write the record must be greater than zero and contained in the numeric variable contained in the RRNName property.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Remarks

If the file is a multifORMAT file, you cannot perform the operation using the file name, you must use the format name.

See Also

DCLDISKFILE | READ | UNLOCK

USING - Use a Namespace

Allocates memory for a file to be processed at run-time. At compile time, the description of the file is read in with its record format. The file name and each record format name must be unique.

```
USING
Namespace
```

Parts

Namespace

Required. The namespace to use for this application.

See Also

DCLNAMESPACE

UYEAR

UYEAR is a special word that allows the programmer to supply a year that the project was started. These special words access the job date that is specified in the job description.

Remarks

UYEAR will print a 2-position numeric date field.

User date fields are not date data type fields but are numeric fields.

See Also

UDAY | UMONTH | UDATE

WHEN

Determines where control passes after the **SELECT** operation is processed.

```
WHEN
Cond (Conditional Expression)
```

Parts

Cond

Required. Determines where control passes after the **SELECT** operation is processed.

Remarks

If the expression is the first one true on the list of **WHEN** commands after the select, the group of statements after the **WHEN** will be executed.

If no expressions or **WHEN** commands evaluate to true, the group of commands after the **OTHER** command will be executed (if specified).

Example

// The following Example shows nested **SELECT** groups. The employee type can be one of Casual, Retired, Regular, and Student. Depending on the employee type (EmpType), the number of days off per year (Days) will vary. */

// When the employee type is Regular, the days off depend also on the number of years of employment. The base number of days is 14. For less than 2 years, no extra days are added. Between 2 and 5 years, 5 extra days are added. Between 6 and 10 years, 10 extra days are added, and over 10 years, 20 extra days are added. */

```
Select
When EmpType = Casual *or EmpType = Retired
    DaysWorked = *Zeros
When EmpType = Regular
    DaysWorked = 14
```

//Nested select group.

```
Select
    When Yearsworked < 2
    When YearsWorked <= 5
        DaysWorked = DaysWorked + 5
    When YearsWorked <= 10
        DaysWorked = DaysWorked + 10
    Other
        DaysWorked = DaysWorked + 20
Endsl
```

```
//End of nested select group.
    When EmpType = Student
    DaysWorked = 5
Endsl
```

See Also

ENDSL | SELECT

WITH

Starts a WITH block.

```
WITH
Value (Expression)
```

Parts

Value

Required. Value consists of an expression that will be evaluated once, and its resulting value will be the 'default' for the unqualified dotted expressions in the body of the **WITH** block.

Remarks

WITH blocks allow a programmer to write a series of commands or statements on an object without having to specify the object each time. The way to write an expression using this object is with an unqualified dotted expression:

```
.<member>
```

WITH blocks can be nested, but only the innermost **WITH** qualifies the unqualified dotted expressions in its body, i.e. the outer **WITH** blocks are masked inside the innermost **WITH** block.

If the body of the **WITH** block is empty, <expression> will not be evaluated.

WITH blocks must be ended with the **ENDWITH** command.

Example

```
// You used to code like this:      txtDBNme.Text          =
My_Production_Database.DBname
    chkIsOpen.Checked              = My_Production_Database.Isopen
    txtLabel.Text                  = My_Production_Database.Label
    txtServer.Text                 = My_Production_Database.Server
    txtPort.Text                   = My_Production_Database.Port
    txtUserId.Text                 = My_Production_Database.User
    txtPlatform.Text               = My_Production_Database.Platform

// Now you can code the following using WITH. With My_Production_Database
    txtDBNme.Text                  = .DBname
    chkIsOpen.Checked              = .Isopen
    txtLabel.Text                  = .Label
    txtServer.Text                 = .Server
    txtPort.Text                   = .Port
    txtUserId.Text                 = .User
    txtPlatform.Text               = .Platform
EndWith
```

See Also

ENDWITH

WRITE

Writes a new record to a file.

```
WRITE
File (file name | Format Name)
Err (Indicator Variable | *EXTENDED)
At (integer)
Eof (Indicator Variable)
```

Parts

File

Required. Must contain a file name or record format name. The file must be declared to allow adding of records, otherwise a compile error will occur. For subfiles, the file name is the SubfileControl name. The relative record number to write the record must be greater than zero and contained in the numeric variable contained in the RRNName property.

If the file is a multiformat file, you cannot perform the operation using the file name, you must use the format name.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

At

Optional. An integer representing the position within the Memory file Subfile control where the new row/record is to be **inserted**. Indexing is relative to 0. For Example:

Write FName At(1) // inserts record between 1st and 2nd records

Write FName At(0) // inserts record as first in file/table Note If **At** row is greater than row count, the row is added at rowcount + 1.

Eof

Optional for ASNA Monarch™ Subfiles. Not allowed for database files. Use the EOF (End-of-File) indicator, the %EOF function, or the Subfile's IsEof property to determine if the End-Of-File was reached while executing the **Write** command.

Remarks

When writing records to a subfile where the *char data contains hex zeros (X'00'), these values will be displayed as a period. Hex 00 is a terminator character for a windows string.

See Also

DCLDISKFILE | IsEof Property | RRNName Property | %ERROR | RowCount Property

XFOOT

Adds array elements together and places the sum into **Result**.

```
XFOOT
Arr (Array Name)
Result (Numeric Variable)
Adj (*TRUNC | *RNDUP)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
      *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
      | *ONECHAR | *PACKED | *ZONED))
Len (Length Integer, [Decimal Integer])
```

Parts

Arr

Required. Specifies the name of the array.

Result

Required. The sum of the array elements.

Adj

Optional. Indicates half-adjust. Use **Adj** (*RNDUP) if you wish to round up the value, or use **Adj** (*TRUNC) if you wish to truncate the value of **Result**. *TRUNC is the default.

Pos

Optional. Turned on if the value of **Result** is a positive number.

Neg

Optional. Turned on if the value of **Result** is a negative number.

Zero

Optional. Turned on if the value of **Result** is zero.

Type

Optional. Can be any of the types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

If half-adjust is specified, the rounding occurs after all elements are summed and before the results are moved into the result field. If **Result** is an element of the array specified in **Arr**, the value of the element before the XFOOT operation is used to calculate the total of the array.

If the array is float, **XFOOT** will be performed as follows: When the array is in descending sequence, the elements will be added together in reverse order. Otherwise, the elements will be added together starting with the first elements of the array.

SUMARR is exactly the same as **XFOOT**, and can be used interchangeably.

Example

```
// MyArray Values are 1.064, 1.703, -1.110.
DclArray MyArray 3 *PACKED Len(4,3)
DclFld MyArraySum *PACKED Len(5,2)
XFoot MyArray MyArraySum *RndUp
// Result 1.66
```

See Also

SUMARR

XLATE

Translates characters in the source string according to the **MapFrom** and **MapTo** string and puts them into **Target**.

```
XLATE
MapFrom (Character Expression)
MapTo (Character Expression)
BaseStr (Character Expression)
StartPos Numeric Expression) | 1 )
Target (Character Expression)
Fill (*NOPAD | *PAD)
Err (Indicator Variable | *EXTENDED)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT |
        *FLOAT4 | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8
        | *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

MapFrom

Required. Must contain the **from** string, along with the **MapTo** string. If a character in the from string is duplicated, the first occurrence is used.

MapTo

Required. Must contain the **to** string.

BaseStr

Required. Must contain either the source string or the source string followed by a colon and the start location (**StartPos**).

StartPos

Optional. Start location.

Target

Required. Contains the **Target** string.

Fill

Optional. Specifies if you want to fill, or pad the **Target** with blanks. The default is *NoPad.

*NoPad indicates that **Target** will not be padded with blanks.

Pad indicates that **Target** will be padded with blanks.

Err

Optional. Determines what the runtime will do if an error occurs while executing this command. The following rules apply:

Err (Indicator Variable) - The specified Indicator is set if an error occurred.

Err (*EXTENDED) - A program status flag is set whose status can be checked using the %ERROR built-in function.

If the **Err** keyword is not given and an error occurs, a runtime exception will be thrown, terminating the program.

Type

Optional. Can be any of the Types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

XLATE starts translating the source at the location specified in **BaseStr** and continues character by character, from left to right. If a character of the source string exists in the **MapFrom** string, the corresponding character in the **MapTo** string is placed in **Target**. Any characters in the source field before the starting position are placed unchanged in **Target**.

See Also

%XLATE | %ERROR | Type Parameter

ZADD

The field specified in **F2** is added to a field of zeros and the sum is placed in **Result**.

```
ZADD
F2 (Numeric Expression)
Result (Numeric Variable)
Adj (*RNDUP | *TRUNC)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT | FLOAT4
      | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])
```

Parts

F2

Required. Must be numeric and can contain one of the following: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

Result

Required. Can contain one of the following: an array, array element, field, subfield, or table name.

Adj

Optional. Indicates half-adjust. Use **Adj** (*RNDUP) if you wish to round up the value, or use **Adj** (*TRUNC) if you wish to truncate the value of **Result**. *TRUNC is the default.

Pos

Optional. Turned on if the value of **Result** is a positive number.

Neg

Optional. Turned on if the value of **Result** is a negative number.

Zero

Optional. Turned on if the value of **Result** is zero.

Type

Optional. Can be any of the types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. The **Len** parameter defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default. If **Len** is character (3), *CHAR is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

F2 is added to a field of zeros. The sum is placed in **Result**. Half-adjust can be specified.

See Also

ZSUB

ZSUB

F2 is subtracted from a field of zeros and the sum is placed in **Result**.

```

ZSUB
F2 (Numeric Expression)
Result (Numeric Variable)
Adj (*RNDUP | *TRUNC)
Pos (Indicator Variable)
Neg (Indicator Variable)
Zero (Indicator Variable)
Type (*NONE | *BINARY | *BOOLEAN | *BYTE | *CHAR | *DECIMAL | *FLOAT | FLOAT4
      | *FLOAT8 | *IND | *INTEGER | *INTEGER2 | *INTEGER4 | *INTEGER8 |
      *ONECHAR | *PACKED | *ZONED)
Len (Length Integer, [Decimal Integer])

```

Parts

F2

Required. Must be numeric and can contain one of the following: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

Result

Required. Can contain one of the following: an array, array element, field, subfield, or table name.

Adj

Optional. Indicates half-adjust. Use **Adj** (*RNDUP) if you wish to round up the value, or use **Adj** (*TRUNC) if you wish to truncate the value of **Result**. *TRUNC is the default.

Pos

Optional. Turned on if the value of **Result** is a positive number.

Neg

Optional. Turned on if the value of **Result** is a negative number.

Zero

Optional. Turned on if the value of **Result** is zero.

Type

Optional. Can be any of the types listed. *NONE is the default. See Type Parameter for further information.

Len

Optional. Defines the length of the field. If **Len** is decimal (3,1) *ZONED is assigned as the default.

Depending upon the **Type** specified, the **Len** parameter may be required. For instance, Types of *CHAR, *PACKED, *ZONED will require a **Len**. A compiler message will display if the **Len** parameter is needed.

Remarks

F2 is subtracted from a field of zeros. The difference, which is the negative of **F2**, is placed in **Result**. You can use the operation to change the sign of a field. Half-adjust can be specified.

See Also

ZADD

Operators

+ Operator

The **+** **Operator** is used to sum two numbers. If the two operands are character, concatenation occurs.

```
result = operand1 + operand2
```

Remarks

The result is always a string. Otherwise the result is numeric and both operands will be 'promoted' to numeric types.

See Also

Operators | Operators and Their Precedence

* Operator

The ***** **Operator** is used to multiply two numbers.

```
result = operand1 * operand2
```

Remarks

The operands can be any numeric expression.

Example

```
StartSec = (Hour * 3600) + (Min * 60) + Sec
```

See Also

Operators | Operators and Their Precedence

/ Operator

The **/** **Operator** is used to divide two numbers.

```
result = operand1 / operand2
```

Remarks

The operands can be any numeric expression.

See Also

Operators | Operators and Their Precedence

** Operator

The **** (Exponent) Operator** is used to denote an exponential value.

```
result = operand1 ** operand2
```

Remarks

The operands can be any numeric expression.

This operation is done in floating point mathematics, so some truncation/roundup may occur.

See Also

Operators | Operators and Their Precedence

>> Operator

Performs an arithmetic right shift on a bit pattern.

```
result = pattern >> bitCount
```

Parts

result

Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of pattern.

pattern

Required. Integral numeric expression. The bit pattern to be shifted. The data type must be *Byte, *Integer.

bitCount

Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be *Integer.

Remarks

The shift is an arithmetic right shift, the bits shifted beyond the range of the result data type are discarded, and the leftmost bit (sign) is propagated into the bit positions vacated by the shift. This means that if pattern has a negative value, the vacated positions are set to one; otherwise they are set to zero.

Note that data type *Byte is considered unsigned, so there is no sign bit to propagate. If pattern is of type *Byte, the vacated positions are always set to zero.

To prevent shifting by more bits than the result can hold, AVR ensures that bitCount is less than the number of bits in the patter. If bitCount is zero, the value of result is identical to the value of pattern. If bitCount is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

Example

```
BegSr shiftTest
    DclFld Byte *Byte
    DclFld Int2 *Integer Len(2)
    DclFld Int4 *Integer Len(4)
    DclFld Int8 *Integer Len(8)
    DclFld string *String

    Byte = 7
    Int2 = 1
    Int4 = 7
    Int8 = 458752 // H'000070000'
    Byte = Byte << 3
    System.Console.WriteLine (Byte *as System.Int32)
    Byte = Byte << 9
```

```

        System.Console.WriteLine (Byte *as System.Int32)
        Byte = Byte << 1
        System.Console.WriteLine (Byte *as System.Int32)
        Byte = Byte >> 1
        System.Console.WriteLine (Byte *as System.Int32)
        Int2 = Int2 << Int4
        System.Console.WriteLine (Int2 *as System.Int32)
        Int4 = Int4 << Int2
        System.Console.WriteLine (Int4)
        Int8 = Int8 << 2
        System.Console.WriteLine (Int8)
        Int8 = Int4 << 94
        System.Console.WriteLine (Int8)
        Int2 = 17000
        Int2 = Int2 << 1
        System.Console.WriteLine (Int2 *as System.Int32)
        Int2 = Int2 >> 1
        System.Console.WriteLine (Int2 *as System.Int32)
    EndSub

```

See Also

<< Operator | Operators | Operators and Their Precedence

*As Operator

The ***AS** operator is used to perform conversions between compatible types. The ***AS** operator is used in an expression of the form:

```
expression as type
```

Parts

expression

An expression of a reference type. *type* A reference type.

Example

The ***AS** operator is similar to a cast, except that it yields null on conversion failure instead of raising an exception.

The ***AS** operator only performs reference conversions. The ***AS** operator cannot perform other conversions, such as user-defined conversions, which should instead be performed using cast expressions.

See Also

Operators | Operators and Their Precedence

*ASFLD Operator

The ***ASFLD** operator casts <expression> to the type of <field or property>. If <expression> cannot be cast to the desired type, ***ASFLD** evaluates to ***Nothing**.

```
<expression> *ASFLD <field or property>
```

Parts

expression

An expression of a reference type.

field or property

A field or property type.

See Also

Operators and Their Precedence

***BitAnd Operator**

Used to perform a logical conjunction on two Boolean expressions, or a bitwise conjunction on two integer values.

```
result = expression1 *BitAnd expression2
```

Parts**result**

Required. Any Boolean or numeric expression. For Boolean comparisons, the result is the logical conjunction of two expressions. For bitwise operations, the result is a numeric value resulting from the bitwise conjunction of two numeric expressions.

expression1

Required. Any Boolean or numeric expression.

expression2

Required. Any Boolean or numeric expression.

Remarks

For **Boolean** comparison, if both *expression1* or *expression2* evaluate to **True**, result is **True**. If *expression1* evaluates to **True**, and *expression2* evaluates to **False** the result is **False**. If *expression1* evaluates to **False**, and *expression2* evaluates to **True**, the result is **False**. The following table illustrates how result is determined:

If <i>expression1</i> is	And <i>expression2</i> is	Then <i>result</i> is
True	True	True
True	False	False
False	True	False
False	False	False

For **bitwise** operations, the **bitAnd* operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in <i>expression1</i> is	And bit in <i>expression2</i> is	Then <i>result</i> is
0	0	0
0	1	0
1	0	0

1

1

1

Note Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

If the operands consist of one **Boolean** expression and one numeric expression, the result **Boolean** expression will be converted to a numeric value (-1 For True, and 0 For False) and the bitwise operation will result.

Example

```
dclconst  Hex24  Value( H'24' )
dclfld   Byte1   Type( *byte )      INZ( H'46' )
dclfld   Byte2   Type( *byte )
dclfld   Int1    Type( *integer2 )  INZ( H'46' )
dclfld   Int2    Type( *integer2 )
dclfld   Bool1   Type( *Boolean )  INZ( *true )
dclfld   Bool2   Type( *Boolean )
dclfld   Str2    Type( *string )
Byte2 = Byte1 *BitAnd Hex24          // Byte2 = H'04'
Int2  = Int1  *BitAnd Hex24          // Int2 = 4 or H'0004'
Bool2 = Int2  *BitAnd H'11'          // Bool2 = False
Str2  = Int1  *BitAnd Hex24          // Str2 = "4"
Int2  = Bool1 *BitAnd H'0C00'        // Int2 = 4 or H'0004'
```

See Also

[*BitAnd Operator](#) | [*BitNot Operator](#) | [*BitOr Operator](#) | [Operators](#) | [Operators and Their Precedence](#)

*BitNot Operator

The ***BitNot** operator specifies the result of a bitwise NOT operation performed on a numeric value.

```
*BitNot (nExpression)
```

Parts

nExpression

Specifies the numeric value on which the bitwise NOT operation is performed. If *nExpression* is not an integer, it is converted to an integer before its bits are shifted.

Example

```
dclconst  Hex24  Value( H'24' )
dclfld   Byte1   Type( *byte )      INZ( H'46' )
dclfld   Byte2   Type( *byte )
dclfld   Int1    Type( *integer2 )  INZ( H'46' )
dclfld   Int2    Type( *integer2 )
dclfld   Bool1   Type( *Boolean )  INZ( *true )
dclfld   Bool2   Type( *Boolean )
dclfld   Str2    Type( *string )
Byte2 = Byte1 *BitOr ( *BitNot Hex24 )          // Byte2 = H'DF'
Int2  = *BitNot ( Int1 *BitAnd Hex24 )          // Int2 = -5 or H'FFFB'
Bool2 = ( *BitNot Int2 ) *BitXor H'11'          // Bool2 = True
Str2  = *BitNot ( Int1 *BitAnd Hex24 )          // Str2 = "-5"
Int2  = (*BitNot Bool1) *BitOr H'0C00'          // Int2 = -2 or H'FFFE'
```

See Also

[*BitAnd Operator](#) | [*BitOr Operator](#) | [Operators](#) | [Operators and Their Precedence](#)

*BitOr Operator

Used to perform a logical disjunction on two Boolean expressions, or a bitwise disjunction on two integer values.

```
result = expression1 *BitOr expression2
```

Parts

result

Required. Any Boolean or numeric expression. For Boolean comparisons, the result is the logical disjunction of two expressions. For bitwise operations the result is a numeric value resulting from the bitwise disjunction of two numeric expressions.

expression1

Required. Any Boolean or numeric expression.

expression2

Required. Any Boolean or numeric expression.

Remarks

For **Boolean** comparison, if either *expression1* or *expression2* evaluates to **True**, result is **True**. If *expression1* evaluates to **True**, and *expression2* evaluates to **False**, the result is **True**. The following table illustrates how the *result* is determined:

If <i>expression1</i> is	And <i>expression2</i> is	Then <i>result</i> is
True	True	True
True	False	True
False	True	True
False	False	False

For **bitwise** operations, the *bitOr operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in <i>expression1</i> is	And bit in <i>expression2</i> is	Then <i>result</i> is
0	0	0
0	1	1
1	0	1
1	1	1

Note Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

If the operands consist of one **Boolean** expression and one numeric expression, the result **Boolean** expression will be converted to a numeric value (-1 For True, and 0 For False) and the bitwise operation will result.

Example

```

dclconst    Hex24    Value( H'24' )
dclfld      Byte1     Type( *byte )      INZ( H'46' )
dclfld      Byte2     Type( *byte )
dclfld      Int1      Type( *integer2 )  INZ( H'46' )
dclfld      Int2      Type( *integer2 )
dclfld      Bool1     Type( *Boolean )   INZ( *true )
dclfld      Bool2     Type( *Boolean )
dclfld      Str2      Type( *string )
Byte2 = Byte1 *BitOr Hex24              // Byte2 = H'66'
Int2  = Int1 *BitOr Hex24              // Int2 = 102 or H'0066'
Bool2 = Int2 *BitOr H'11'              // Bool2 = True
Str2  = Int1 *BitOr Hex24              // Str2 = "102"
Int2  = Bool1 *BitOr H'0C00'          // Int2 = 3073 or H'0C01"

```

See Also

*BitAnd Operator | *BitNot Operator | *BitOr Operator | Operators | Operators and Their Precedence

*BitXOr Operator

Used to perform a logical exclusive OR on two Boolean expressions, or a bitwise exclusive OR on two integer values.

```
result = expression1 *BitXOr expression2
```

Parts

result

Required. Any Boolean or numeric expression. For Boolean comparison the result is the logical conjunction of two expressions. For bitwise operations the result is a numeric value resulting from the bitwise conjunction of two numeric expressions.

expression1

Required. Any Boolean or numeric expression. If not an integer, it is converted to an integer before the bitwise exclusive OR operation is performed.

expression2

Required. Any Boolean or numeric expression. If not an integer, it is converted to an integer before the bitwise exclusive OR operation is performed.

Remarks

For Boolean comparison, if both expression1 and expression2 evaluate to True, result is False. If expression1 evaluates to True and expression2 evaluates to False, result is True. If expression1 evaluates to False, and expression2 evaluates to True, the result is True. The following table illustrates how result is determined:

If expression1 is	And expression2 is	Then <i>result</i> is
True	True	False
True	False	True
False	True	True
False	False	False

For bitwise operations, the *bitXOr operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in **result** according to the following table:

If bit in <i>expression1</i> is	And bit in <i>expression2</i> is	Then <i>result</i> is
0	0	0
0	1	1
1	0	1
1	1	0

Note Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

If the operands consist of one Boolean expression and one numeric expression, the result **Boolean** expression will be converted to a numeric value (-1 For True, and 0 For False) and the bitwise operation will result.

Example

```
declconst  Hex24  Value( H'24' )
declfld   Byte1   Type( *byte )      INZ( H'46' )
declfld   Byte2   Type( *byte )
declfld   Int1     Type( *integer2 )  INZ( H'46' )
declfld   Int2     Type( *integer2 )
declfld   Bool1    Type( *Boolean )   INZ( *true )
declfld   Bool2    Type( *Boolean )
declfld   Str2     Type( *string )
Byte2 = Byte1 *BitXor Hex24           // Byte2 = H'04'
Int2  = Int1  *BitXor Hex24           // Int2 = 4 or H'0004'
Bool2 = Int2  *BitXor H'11'           // Bool2 = False
Str2  = Int1  *BitXor Hex24           // Str2 = "4"
Int2  = Bool1 *BitXor H'0C00'         // Int2 = 4 or H'0004'
```

See Also

Operators | Operators and Their Precedence

*ByRef Operator

The ***ByRef** operator is used to declare that the associated reference parameter is being passed by reference.

Example

```
myObject.Mymethod( Parm1, *ByRef Parm2, Parm3 ) // Parm2 is passed by
reference
```

See Also

Figurative Constants | Operators

*Is Operator

The ***Is** operator is used to check whether the run-time type of an object is compatible with a given type. The ***Is** operator is used in an expression of the form:

```
expression *Is type
```

Parts

expression

An expression of a reference type. *type* A type.

Example

An ***Is** expression evaluates to true if both of the following conditions are met:

expression is not **null**.

expression can be cast to *type*. That is, a cast expression of the form (type)(expression) will complete without throwing an exception.

A compile-time warning will be issued if the expression *expression* is type is known to always be **True** or always be **False**.

The ***Is** operator cannot be overloaded.

Note that the ***Is** operator only considers reference conversions. Other conversions, such as user-defined conversions, are not considered by the **is** operator.

See Also

Operators and Their Precedence

*ISFLD Operator

The ***ISFLD** operator evaluates to ***True** if <expression> can be cast to the type of <field or property>; otherwise it's ***False**.

```
expression *ISFLD <field or property>
```

Parts

expression

An expression of a reference type. *field or property* A field or property type.

See Also

Operators and Their Precedence

*New Operator

The ***New** operator creates a new object instance. The ***New** operator must specify a defined class from which the instance can be created. You can use ***New** in a declaration statement or an assignment statement. When the statement is executed, it calls the constructor of the specified class, passing any arguments you have supplied.

The common language runtime throws an **OutOfMemoryException** error if there is insufficient memory to create the new instance.

See Also

[Operators | Operators and Their Precedence](#)

*NOT

***NOT** or **~**, is an operator to be used in expressions to determine if the opposite condition is true.

Remarks

Any number of parentheses () can be used to force the precedence of any operator. The deepest levels of parentheses will be evaluated first, until the whole expression is evaluated.

All of the mathematical operands, such as -, +, /, *, and ** are valid only for numeric values, except for the '+'. The '+' will also do a concatenate if the two operands are character strings.

Example

```
IF (*NOT (A=B *AND <+D) )
```

See Also

[Operators and Their Precedence](#)

*TypeOf Operator

Returns a **Type** object for the specified type. The **Type** object can be used to retrieve information about the type such as its properties, methods, and events.

```
*Typeof (typename)
```

Parts

typename

The name of the type for which information is desired.

Remarks

The ***Typeof** operator is used to obtain the **System.Type** object for a type. If you want to get the type object of an object variable, use **Type.GetType** method.

Example

```
DclFld myType Type (Type)  
MyType = *typeof( system.Web.HttpWorkerRequest )
```

See Also

[Operators | Operators and Their Precedence](#)

Properties

Database Properties

Public Properties

 CurrentUserLibl	Sets the current library list for the current connection.
 DBName	Gets or sets the name of a database.
 InitialException	Gets the InitialException source object of a database.
 InitialLibl	Gets or sets the current database's initial library list.
 IsOpen	Gets a boolean value indicating if the file is currently open for I/O operations.
 Label	Gets or sets a string containing the label of the database to connect to.
 Password	The password to be used in conjunction with User for connection authentication.
 PlatformAttribute	Gets or sets a platform-specific attribute of the database engine.
 PoolingTimeOut	Gets the amount of time (in minutes) in which a connection will remain idle in the pool until it is closed and removed from the pool.
 Port	Gets or sets the TCP port number used by the database server for TCP/IP-based transport (default 5042).
 Server	Gets or sets the database server host machine name or address.
 Text	Gets or sets a description of the database name.
 User	Gets or sets the authenticated user profile that is used to authorize the current database connection.

See Also

ASNA.VisualRPG.Runtime | Database Class | Database Class Methods | Database Class Members | DCLDISKFILE

DBFile Properties

Public Properties

 FormatName	Gets the name of the record format last read or written.
 RecCount	Gets the current number of non-deleted records in the file.
 RecNum	Gets the current RRN for the last input operation to a file.

See Also

ASNA.VisualRPG.Runtime | DBFile Class | DCLDISKFILE

MemoryFile Properties

Public Properties

 DataSet	Gets an ASNA DataGate Client ADO.NET DataSet. This property is of type ASNA.DataGate.Client.AdgDataSet.
 SfiRRN	Gets an integer value containing the RRN number for the current subfile record held by the MemoryFile .

See Also

ASNA.VisualRPG.Runtime | MemoryFile Class | MemoryFile Members | DCLMEMORYFILE

PrintFile Properties

Public Properties

 Collate	Gets or sets whether collation will be used when printing multiple copies.
 Copies	Gets or sets the number of copies to print.
 CursorY	Gets the position, in Print Units, that the printer is from the top of the page.
 Duplex	Gets or sets whether the printer file is to print duplex (prints on both sides of the paper). This property is of type System.Drawing.Printing.Duplex.
 FooterSize	Gets the size of the page footer.
 FormName	Gets or sets the name of the customized page size to use.
 InOverflow	Gets the overflow state of the printer file. This property is of type *Boolean. When the value is True , the print file is within the overflow area, otherwise False .
 IsOpen	Gets the open state of the printer file. This property is of type *Boolean. When the value is *True, the file is Open. (Same as the %Open built-in function.)
 LastException	Gets the last exception thrown for this file. This property is of type ASNA.VisualRPG.Runtime.AvrException.
 ManuscriptPath	Gets or sets the file and path name for the manuscript.
 Orientation	Gets or sets the print orientation (Horizontal or Vertical). This property is of type ASNA.DataGate.Common.PaperOrientation .

 PageLength	OBSOLETE: This property is marked obsolete and is referenced only for backward compatibility. This has been replaced with PrintableLength property.
 PageWidth	OBSOLETE: This property is marked obsolete and is referenced only for backward compatibility. This has been replaced with PrintableWidth property.
 PaperLength	Gets or sets the physical length of the page. Valid only with a continuous-form printer.
 PaperSize	Gets or sets the paper size. This property is of type <code>System.Drawing.Printing.PaperKind</code> .
 PaperSource	Gets or sets the printer tray that will be used. This property is of type <code>System.Drawing.Printing.PaperSourceKind</code> .
 PaperWidth	Gets or sets the physical width of the paper. Valid only with a continuous-form printer.
 Preview	Gets or sets whether the print preview dialog is to be displayed before printing. This property is of type ASNA.DataGate.Common.AccessMode .
 PrintableLength	Gets the length of the printer's printable area.
 PrintableWidth	Gets the width of the printer's printable area.
 Printer	Gets or sets the full path name of the printer to be used.
 PrintSetup	Gets or sets whether the print setup dialog is to be displayed before printing.
 ProcessMacro	Gets or sets whether DataGate print engine is to search for page numbering "macros" and replace them with appropriate data before sending the output to the printer.
 Quality	Gets or sets the print quality. This property is of type <code>System.Drawing.Printing.PrinterResolutionKind</code> .
 ReportName	Gets or sets the name of the report as seen on the printer spool. If not assigned, the default name is the name given when the print file was declared.
 Scale	Gets or sets the percentage factor by which the printed document is to be scaled.
 TTOption	Gets or sets how true type fonts are to be printed. This property is of type ASNA.DataGate.Common.PrintTrueType .
 UseColor	Gets or sets whether to print in color when a color printer is specified.
 YResolution	Gets or sets the Y-axis resolution (in dot per inch) when printing graphics.

See Also

ASNA.VisualRPG.Runtime | PrintFile Class | PrintFile Members | DCLPRINTFILE

Special Values

*END

***END** positions the file at the end of the file, so the READP returns the last record of the file.

It is also used in conjunction with ***START**, which positions the file at the beginning of file, so the READ returns the 1st record.

***END/*START** is available with SETLL and SETGT.

See Also

***START** | **READ** | **READP** | **SETGT** | **SETLL**

*PropVal

The ***PropVal** special value contains the value of a property when it is set when using BEGPROP. See Example below with Remarks // Using *Propval.

You can also define a name for ***PropVal**, and then just reference that name using the NewValname parameter. See Example below with Remarks // Using NewValname.

Example

// Below is variable to maintain the value of MyProp property

```
DCLFLD MyPropVariable Type(*Zoned) Len(9,0) Access(*Public)
```

// Using *Propval

```
BEGPROP MyProp EVENT(*SET) Type(*Integer) Len(4) Access (*PUBLIC)
  MyPropVariable = *PropVal
ENDPROP
```

// Using NewValName

```
BEGPROP MyProp EVENT(*SET) Type(*Integer) Len(4) Access(*PUBLIC) NewValName
(MyPropVal)
  MyPropVariable = MyPropVal
ENDPROP
```

See Also

BEGPROP

*START

***START** positions the file at the beginning of a file, so the READ returns the 1st record.

It is also used in conjunction with ***END**, which positions the file at the end of the file, so the READP returns the last record of the file.

***START/*END** is available with SETLL and SETGT.

See Also

*END | READ | READP | SETGT | SETLL

Syntactical Elements

Commands

Every **command** name is considered a reserved word of the Visual RPG Language. Each command has its set of keywords that the command processor is aware of.

Parameters can be provided either plain or keyworded. Each parameter has a position assigned to it, even if it's optional. A parameter can always be keyworded, but can only be plain if it is provided in the position assigned to it.

For Example:

The command DELETE has 4 parameters as follows:

DELETE	From()	Key()	NotFnd()	Err ()
Position	1	2	3	4

The following are valid uses of the command:

DELETE	From(F)	Key(K)	NotFnd(N)	Err (E)
DELETE	F	K	N	E
DELETE	F	K	N	Err (E)
DELETE	From(F)	K	Err (E)	NotFnd(N)
DELETE	Key(K)	From(F)	N	E

Any time a list or expression is provided as a parameter value, it must be given in keyworded format. If an operand is provided in place of an expression, it can be provided in plain format, otherwise it must be keyworded.

There are certain commands, such as IF, DOWHILE, DOUNTIL which have some built-in parameters that have no keyword associated with them. These parameters must always be provided positionally, even if they are expressions.

There are 2 types of commands:

Declarative - Commands which are allowed anywhere.

Operational - Commands which are subdivided into 4 subgroups.

Most operational commands may be labeled and may have conditional indicators.

A Label consists of a valid RPG name followed by a colon (":") right before the command name.

i.e.: DoOpen: OPEN DISTMAST

Above, DoOpen is a valid target for a GOTO statement. Note that the TAG command provides an alternative way to specify a label in an RPG program.

Operational commands that have a **Result** or **Target** parameter, can define the receiver as a global variable by providing the following Type and Len parameters as the last parameters of the command:

```
Type (*NONE | *CHAR | *ZONED | *PACKED)
Len (Length, DecimalPositions)
```

See Also

DELETE | IF | DOWHILE | DOUNTIL | GOTO | TAG | Visual RPG Language Reference | Syntactical Elements

Expressions

An expression is a combination of variables, operators and functions that yield a numeric result or a string.

See Also

Visual RPG Language Reference | Syntactical Elements

Assignments

An **assignment** defines or reassigns the contents of a field. An **assignment** copies the value of an expression into a field left-adjusted with padding on the right with blanks if the field is a character field, and right-adjusted with a zero pad if the field is numeric.

For Example:

```
Array [2, x+1] = DataStruct.Field1 + control.property
Numeric = Binary-Packed[5,2]
```

See Also

Object Variables

Expression Operators and their Precedence

The following are the valid operators for expressions listed in order of precedence from highest to lowest.

	Expression	Description
1.	Unary -, +	- Makes a negative number + Provides an explicit sign
2.	**	Exponent
3.	*, /	Multiply, Divide
4.	+, -	Add, Subtract
5.	<, *LT	Less Than
	>, *GT	Greater Than
	<=, *LE	Less Than or Equal To
	>=, *GE	Greater Than or Equal To

	=, *EQ	Equal Condition
	<>, *NE	Not Equal Condition
6.	~, *NOT	Not Condition
7.	&, *AND	And Condition
8.	, *OR	Or Condition

Any number of parentheses () can be used to force the precedence of any operator. The deepest levels of parentheses will be evaluated first, until the whole expression is evaluated.

All of the mathematical operands, such as -, +, /, *, and ** are valid only for numeric values, except for the '+'. The '+' will also do a concatenate if the two operands are character strings.

See Also

Operators | Comparison Operators

Operands

The following are valid operands in an expression.

Array Elements

Control Properties

Data Structure Element

Fields

Multiple Occurrence Data Structures

See Also

Variables | Visual RPG Language Reference

Expression Overview with QrySelect

An expression to be used with a **QrySelect** within DCLDISKFILE is a logical expression consisting of one or more relational expressions optionally combined with the "&" (and) and "|" (or) operators. Since "&" has a higher priority than "|", you can use parentheses to force the order in which the expression is evaluated.

An **expression** is defined by a single or double quote, followed by any combination of characters, and terminated by a single or double quote. Note that a numeric field can be compared against numeric constants, and a character field must be compared against a quoted character constant.

Please note that if you want a single quote within a literal, you must include two quotes. For Example, O'Clock is coded as 'O"CLOCK'.

A **relational expression** consists of a field name, a relational operator and a constant or another field name.

The valid relational operators are:

>	GT	greater than
>=	GE	greater than or equal to
<	LT	less than
<=	LE	less than or equal to
=	EQ	equal to
!=	NE	not equal to
<>	NE	not equal to
&	AND	
	OR	

The Example below (Table 1) lists some Example expressions, showing that if you want the expression to equal a particular character value, you must put double quotes around the value. Note that you do not need to specify quotes around a numeric value.

The result of the expression is derived from the fields X, Y and Z as listed in table 2 below.

Table 1

Expression	Explanation	Result
'x = y'	The contents of field x = the contents of field y.	Record 2
'x = "y"'	The contents of field x = the character value Y.	Record 5
'Z = 4'	The contents of field Z = the value 4.	Record 4

Table 2

Field X	Field Y	Field Z
A	X	1
B	B	2
C	Y	3
X	C	4

Y	Z	5
Z	A	6

Simple Types

The following are the **simple types** allowed in the Visual RPG Language.

Character Types:

*CHAR Character string. The maximum character string is 65,523.

Numeric Types:

*BINARY Decimal binary number (2's complement). The maximum size for a 2-byte binary is four digits, and the maximum size for a 4-digit binary is 9 digits.

*PACKED Decimal packed number (X'7' as the rightmost nibble is negative, X'3' as the rightmost nibble is positive). The maximum number of digits can be 31 and the maximum number of decimal positions is 9.

*ZONED Decimal zoned number (X'3' as the rightmost zone is positive, X'7' as the rightmost zone is negative). The maximum number of digits can be 31 and the maximum number of decimal positions is 9.

Special Types:

*FORM Form program handle.

*IND Indicator (can only have the special values *ON and *OFF).

See Also

Language Elements | Visual RPG Language Reference | Syntactical Elements

Type Conversions

Indicator Conversions:

A single character '1' moved to an indicator variable sets the indicator to the *ON value, and a single character '0' moved to an indicator variable sets the indicator to the *OFF value. Likewise, a move from an indicator variable to a character variable sets the first byte of the character variable to a '1' or '0'.

Numeric Conversions:

Numeric operands are promoted to a zoned number before being used on any operation. If all the operands involved are binary with the same specification, the promotion may be virtual.

Implicit Conversions:

Implicit Conversions are done when two operands in an expression are not of the same type. These conversions can only be done implicitly if the operands are both character or indicator, or are both numerics. Character or indicator operands mixed with numeric operands will result in a compiler error.

See Also

Language Elements | Visual RPG Language Reference | Syntactical Elements

Variables

Array Element

An **array element** can be accessed by using the [] characters.

Example

```
Array [x+3] = 'A'  
Move "Hello" Greeting [2,3]
```

Related Sections

Character Expression

Global Variable Numeric

Variable Variable Names

See Also

[Control Properties](#) | [Data Structure Element](#) | [Fields](#) | [Multiple Occurrence Data Structures](#) | [Variables](#) | [Visual RPG Language Reference](#)

Character Expression

A character expression may be a variable name, literal, or an expression containing variables and literals.

Remarks

An indicator variable can be used in a character expression, and is implicitly converted to character using the implicit conversion rules.

Example

```
MyFieldName  
@Literal@  
  
(MyLibName + MyFileName)  
(@Literal1@ + @Literal2)  
(MyState + @ @ + MyZipCode)
```

See Also

[Variables](#)

Constant Expression

A **constant expression** is an expression made up entirely of literal or named constant values.

Remarks

No variables can be included in a constant expression.

Example

Character Constant Expression

```
'Hello' + x'09' + 'World'
```

Numeric Constant Expression

```
3 + 2.5 - 1.53
```

See Also

[Variables | Literals](#)

Data Structure Element

While traditional RPG allows you to access data structure elements without referring to the data structure, Visual RPG allows **data structure element** names to be "hidden" in the data structure, and accessed via the "." Character.

Remarks

If a **data structure element** is an array, or if a control's property is an array property, individual elements of these arrays can be accessed using the [] characters.

Example

```
MOVE      'A'      MyDS.Element1
MyDS.Element2= 3
```

You can also access the properties of a control.

```
MOVE      *ON      IOField1.FontBold
MOVE      *OFF     IOField2.FontItalic
```

See Also

[Variables | Array Elements | Control Properties | Fields | Multiple Occurrence Data Structures | Visual RPG Language Reference](#)

Field

A **field** is the simplest type of variable. It can be declared using the DCLFLD command or implicitly by giving the Type and Length on commands that have a result field.

See Also

[Variables | DCLFLD | Names | Reserved Names | Array Elements | Control Properties | Data Structure Element | Multiple Occurrence Data Structures | Visual RPG Language Reference](#)

Global Variables

Variables can be global to the program or local to a subroutine.

Global Variables

Any field defined with a DCLFLD, DCLDS, DCLDSFLD and DCLARRAY in the "mainline code" (not contained between BEGSR and ENDSR) is globally defined for the entire Program.

Any field defined implicitly in code anywhere in the program is globally defined for the entire program.

Global variables can be of most any type. For Example, you can have and access a global array data structure, file, or scalar (packed, char, or zoned field) all from a second program.

Any name used without a previous definition gets implicitly declared with a global scope.

Indicator

An **indicator** is any field declared using the *IND simple type. *ON, *OFF are indicator literals.

Remarks

In Visual RPG, an indicator array *IN is automatically defined for each program. Individual elements can be accessed using the array element syntax ([]) or by using the predefined names *IN01, *IN02, ... etc. The *IN array is 99 long.

*INLR is another special predefined indicator that is used to determine the program end, if the program should be unloaded.

See Also

[Variables](#)

Indicator Variable

An indicator variable is any indicator defined with the DCLFLD command, or any of the predefined indicator variables.

Example

```
*IN array
*INLR
*IN01, *IN02, ...
DCLFLD BadCredit *IND
```

See Also

[Variables](#) | [DCLFLD](#)

Integer Literal

An **integer literal** is a numeric literal that does not contain a decimal point.

Example

Integer literals:

```
3, 35, -20
```

Not integer literals:

```
2. and 30.5
```

See Also

[Variables](#)

Labels

A label is a name specified in a program that can allow commands to be the target of a GOTO command.

Remarks

Many operational commands allow a label to be placed before the command with a colon after it.

A label can be specified using the TAG command.

Example

```
Next: MOVE 'A' FLD
```

Use the **TAG** command

See Also

Variables | GOTO | TAG

Language Elements

Commands

Simple Types

Type Conversions

Variables

See Also

Variables

Lexical Elements

Continuation Lines

Literals

Names

Reserved Names

Special Values

See Also

Variables

Local Variables

Any field defined with a **DCLFLD**, **DCLDS**, **DCLDSFLD** and **DCLARRAY** within a subroutine (i.e., between **BEGSR** and **ENDSR**) is defined locally to that subroutine only.

Such definition strategies can make pre-written subroutines completely insular from the remaining code in a program. This allows subroutines written in this manner to be copied into a program without affecting declarations anywhere else in the program. If the subroutine uses indicators, it is only a simple matter to save the status of the indicators in locally defined fields upon entry to the subroutine, then restoring them to their original status upon exiting.

A local variable hides the global one. A local variable exists from the point of its definition to the end of the subroutine. To define a local variable, any of the following commands can be used: **DCLARRAY**,

DCLDS, DCLFLD.

Remarks

A local variable can not be defined in the equivalent of an old 'C' spec.

See Also

Variables | DCLARRAY | DCLDS | DCLDSFLD | DCLFLD | BEGSR | ENDSR

Numeric Expression

A numeric expression is any expression containing solely numeric variables and literals.

See Also

Variables | Numeric Variables | Literals

Numeric Variable

A **numeric variable** is a variable with a Numeric Type, such as *ZONED, *PACKED, and *BINARY.

Remarks

A numeric variable can be the receiver of a MOVE, ZADD or assignment operation.

See Also

Variables | MOVE | ZADD

Record Format

A **record format** is a name defined in a Acceler8DB file that refers to a given layout of data in a record read from the file.

Remarks

When a **record format** name is used in an input or output operation, data from the record is copied to fields in the Acceler8DB file.

See Also

Variables

Rules for User Date

Use the following rules when using user date special words:

User date fields are numeric fields, not Date Data Type fields.

For an interactive job or batch program, the user date special words are set to the value of the job date when the program starts running in the system. The value of the user date special words are not updated during program processing, even if the program runs past midnight or if the job date is changed. Use the TIME operation code to obtain the time and date while the program is running.

The user date fields cannot be modified. This means they cannot be used:

As input fields

In the result field of calculations

As F1 of DCLPARM operations

As Index of LOOKUP operations

See Also

Variables | Date Data Type | *Date | *Day | DCLPARM | Figurative Constants Rules | LOOKUP | *Month | TIME

Subroutine

A subroutine is a segment of code that can be executed by issuing the EXSR command, or because a user event occurs, and the subroutine has been designated to handle that event.

See Also

Variables | EXSR

User Date Special Words

User date special words allow the programmer to supply a date for the program at run-time.

*DATE / UDATE

*DAY / UDAY

*MONTH / UMONTH

*YEAR / UYEAR

See Also

Variables

Variable Operand

A variable operand is any operand that can receive new contents either by doing a MOVE, ZADD, or an Assignment.

Remarks

Variables include the following:

Arrays and array elements

Control and form properties

Data structures and data structure elements

Fields

Multiple occurrence data structures

See Also

Variables | MOVE | ZADD | Assignment