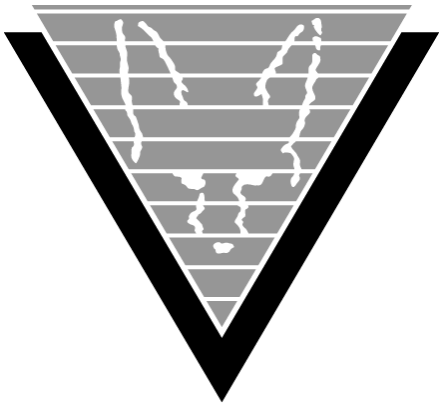

TRIMpl
Function Reference
5.5.3.1.x

December 7, 2016

Trifox Inc.
2959 S. Winchester Blvd
Campbell, CA 95008

www.trifox.com

Trademarks

Copyright

Contents

Preface

TRIMpl is a 4th generation language that you can use to write window-based and stand-alone applications, including triggers called by other applications.

We use the term *window* to refer generically to 16-bit and 32-bit Microsoft Windows and Java platforms. For details on how the same TRIMpl code supports all three display options, read the ***DesignVision Users Guide.***

This reference guide lists and describes the TRIMpl built-in functions in alphabetical order. Functions can be available in any of the following Trifox products:

- ***DVapp*** — windowing application.
- ***TRIMapp*** — character-based environment.
- ***DVreport, TRIMreport*** — report writer.
- ***TRIMrpc*** — remote procedure calls.
- ***Standalone*** — stand-alone TRIMpl applications.

Background

Trifox Inc. has been serving the relational database market since 1984 through consulting and the development of software products. In 1987, Trifox created SQL*QMX for Oracle. This easy-to-use, powerful querying and report writing tool, which is based on IBM's QMF, continues to be used at thousands of sites. In 1989, Trifox created TRIMtools, a family of application and reportwriting tools now known as DesignVision. DesignVision was developed in response to the OLTP requirements of several large application vendors.

Database Access

VORTEX is an integrated family of products that allows nearly any production application to access SQL data:

- On any or all of the major relational databases.
- Across networks.
- Across platforms.
- With a dramatic increase in the number of concurrent users.
- Without any additional hardware.

In a client/server or multi-tier configuration, VORTEX makes it possible for your SQL applications to access data on different platforms over one or more network configurations. Currently it supports only TCP/IP.

Inherent in this approach are services that allow production applications originally written for one relational database (such as Oracle) can access the same data on another database (such as Informix), even if it is spread across different databases.

VORTEX Precompilers for C and COBOL, as well as a variety of program interfaces, allow existing SQL programs to take full advantage of VORTEX services such as performance enhancement, transaction monitoring, and flat-file database access.

With VORTEXaccelerator in your configuration, you dramatically increase the number of concurrent users who can log on to a specific SQL production application. Your users experience faster performance and you won't have to change any programs or add any hardware.

Application and Report Development

DesignVision DVapp lets you design, generate, and maintain forms-based applications. You can easily port the pop-up windows, customizable menus and submenus, and custom keyboard assignments, in fact the entire application, to 16-bit Windows, 32-bit Windows, or Java with no extra effort.

The reportwriter, TRIMreport, lets you create simple reports quickly, or complex reports with absolute confidence in their power.

When you want to write stand-alone applications (including triggers) without a user interface, the TRIMpl 4GL language gives you the freedom you want. The procedural language has over 100 database-specific functions that help you write powerful applications in very little time.

Reaching Legacy Data

GENESISsql is a SQL processor that accesses low-level data sources such as ISAM, SDMS, ADABAS, RMS, and MicroFocus and makes the data accessible to VORTEX clients. You can add GENESIS data sources to a VORTEX system in a matter of days, simplifying what used to be an enormous task.

Conventions

Screen shots in this manual come from the Windows version of our software.

Trifox documentation uses the following conventions for communicating information:

Example	Describes
`CHOOSE REPORT > [F3] >`	Press [F3] on the CHOOSE REPORT menu and ...
Right-click	Clicking the right mouse button.
Left-click	Clicking the left mouse button.
`connect_string`	Replace italicized text with your own variable.
`vtxnetd`	Text in bold typewriter style represents strings that you type exactly as they appear in the manual.

Support

If you have a question about a TRIFOX product that is not answered in the documentation (paper or online), contact the Customer Support Services group at:

- support@trifox.com
- Trifox Customer Support Services
  2959 Winchester Boulevard
  Campbell, CA 95008
  U.S.A.
- 408-796-1590

5.0.2.1.2 Updates

Flag Operations

GUI object flags with the same name are now chained. This means that if you have a menu item and a push button with the same name, a single flag operation affects both objects.

Function Changes

This version has four new TRIMpl functions:

- `os_id()`. See "***os_id***" on page 195 for full details.
- `window_table()`. See "***window_table***" on page 254 for full details.
- `table_exec()`, documented on page 233.
- `record_exec()`, documented on page 209.

`edit_text()`(page 60) has been enhanced to display read-only text, such as help messages. See for full details.

Window Triggers

RECORD and TABLE are different from the WINDOW/UPDATE/QUERY/INIT triggers in that they are invoked via record_exec() and table_exec() rather than via window(). This allows for parameters to be passed in as well as allowing for an optional return value.

Calling a `record_exec()` and/or `table_exec()`, invokes the current window's RECORD and/or TABLE trigger. This makes it handy to call from user triggers. The RECORD trigger is read from the dictionary from the same TRIGGER set as WINDOW/UPDATE/etc.

The TABLE trigger is read from the dictionary based on the primary table (A.) used in the window.

For more information about triggers, refer to the ***DesignVision Users Guide***.

5.0.3.1.11 Updates

Improved DVSlave for Java

The new DVSlave for Java is faster and easier to use. Built with JDK1.1.8 and Swing1.1.1, it also runs with Java2.

Function Changes

This version has two new TRIMpl functions:

- `syslog()`. See "***sysinfo***" on page 230 for full details.
- `alarm()`. See "***alarm***" on page 14 for full details.

Initialization Parameters

This version has one new `trim/dv.ini` file parameter:

- busy_alarm

Additions for MVS

- TRIMrun (console) version now available. Read the component description in ***DesignVision Users Guide*** for more information.
- Support for `SET CURRENT SQLID ...`

  If a connect string has two parts (UID/DSN), then the SQLID is set to UID.
- Support for `SET CURRENT PATH ...`

  If the environment variable `VORTEX_DB2_PATH` is set, an `EXEC SQL SET CURRENT PATH=:hostvar` is performed where `:hostvar` is the string retrieved from `VORTEX_DB2_PATH`.

  `VORTEX_DB2_PATH` can be retrieved from either the `envfile` on MVS or passed in the connect string.

5.0.3.1.12 Updates

New Function sql_xlate

See the DesignVision manual for details on using SQL syntax translations, including the new `dv/trim.ini` keyword `sql_xlate_file,` the new TRIMgen option for compile-time translation. For information on the runtime translation function, see "***sql_xlate***" on page 225.

Improved documentation for `record_exec()` and `table_exec()` functions.

5.1.0.0.x Updates

Function changes

New function "***dialog_button***" on page 56.

Enhanced function "***instr***" on page 127.

All references to `gui.*` and `gcmnod.*` are updated to new filenames, `dv.*`.

5.1.0.1.x Updates

Function changes

Updated decode() information.

Added "edi" to the *control list* types for `list_open()`.

Documentation updates

- Added information for specifying filenames for all functions that operate on files. The complete text for the specifications `gui!`, `vortex!`, `net!`, and `dir!` appear in the description for list_open()and is referenced in all other functions that work on files.
- Corrected sys`log()` description (removed filename reference).

5.2.1.0.x Updates

Function changes

New function "***call_level***" on page 22.

New function"***crypt***" on page 34.

New function "***db_command***" on page 43.

New function "***input_screen***" on page 122.

New function "***list_index***" on page 152

New function "***list_refcnt***" on page 168.

New function "***timestamp***" on page 234.

New function "***tokenize***" on page 241.

Documentation updates

- Added delimiter option to list_file().
- Added XML option to list_file().

5.5.2.7.x Updates

Function changes

New function "***gui_winmod***" on page 118.

5.5.3.1.x Updates

Function changes

New function "***chr2***" on page 24

New function "***clipboard***" on page 25

New function "***convert***" on page 32

Enhanced function "***crypt***" on page 34

New function "***datadump***" on page 41

Enhanced function "***db_command***" on page 43

New function "***gui_canvas***" on page 100

New function "***gui_config***" on page 103

New function "***gui_grid***" on page 105

New function "***gui_info***" on page 109

New function"***gui_ipc***" on page 111

New function "***gui_listen***" on page 113

New function "***gui_spawn***" on page 114

New function "***gui_util***" on page 115

Enhanced function "***gui_winmod***" on page 118

New function "***list_copy2***" on page 140

New function "***list_treeview***" on page 175

New function "***regexp***" on page 212

New function "***replace***" on page 214

New function "***ret_freeheap***" on page 215

New function "***set_option***" on page 220

New function "***signal***" on page 221

New function "***signal_client***" on page 223

New function "***unicode***" on page 245

New function "***xml***" on page 258

Functions by Name

All TRIMpl functions are listed in this manual in alphabetical order.

The option types typically correspond to TRIMpl data types (see Chapter 4, Datatypes, of the ***DesignVision Users Guide*** for more complete information).

- ***int*** — TRIMpl's **int** behaves the same as C's except that it accepts NULLs. The variable's byte size is machine-dependent.
- ***block*** — Block of TRIMpl code, enclosed in braces ({ }).
- ***list*** — The **list** variable is a dynamic *m*x*n* matrix of data. Each cell of the matrix can contain any variable datatype including other lists. You manipulate columns in each row with (`list_*()`) functions.
- ***string & char*** — TRIMpl treats character variables as strings of characters. **char** and **string** keywords are interchangeable. The string size is the length of the longest string that can be accommodated by the variable. You cannot directly address individual characters in a string; instead, you must use string functions.
- ***ident*** — Unquoted series of characters that can be up to 30 characters long.
- ***expr*** — Expressions are TRIMpl's wildcards. Whenever an option is of type **expr**, you can use any type, or a statement that resolves to any type. The function simply passes the value.
- ***trigger*** — The **trigger** datatype, or variable, is a pointer to another block of code.
- ***keyword*** — TRIMgen reserved word. Reserved words include keywords and function names. Do not declare variables with these names.

Keywords

block	footer	parent	sibling
child	header	parm	update
count	initialize	query	
current	open	run	

Reserved Words

NULL
SYSDATE
action_text
active_field
active_page
active_row
active_wl
alarm
all
append
as
ascii
bell
block
block_count
block_seq
break
browser
call
call_level
char
chr
clipboard
close
commit
confirm
connect
count
crypt
cursor_col
cursor_pos
cursor_row
cursor_wait
cuserid
datatype
datetime
else
db_command
db_id
db_msg
db_msgdump
db_mux
db_rows
db_sqlcode
debug
decode
delete
design_name
dialog_button
dir
dump_scredit_text
error
error_msg
error_trap
escape
exec_proc
exec_row
exec_sql
exec_usr
execute
export
field
field_attr
field_color
field_count
field_d
field_dynattr
field_exec
field_flag
field_helpname
field_init
field_mask
field_name
field_offset
field_page
field_rows
field_seq
field_set
field_sysattr
field_test
field_tid
field_val
field_visual
field_width
file_copy
focus
formfeed
for
gen_time
getenv
go_field
greatest
gui_id
gui_info
gui_linesize
gui_listen
gui_winattr
heapsize
if
import
in
input
input_screen
input_timer
input_visual
insert_mode
instr
int
key_exec
key_reset
key_set
key_type
kill
least
length
like
list
list_close
list_colix
list_colnam
list_cols
list_colwid
list_copy
list_curr
list_dbbind
list_dup
list_edit
list_eos
list_file
list_find
list_get
list_index
list_ixed
list_merge
list_mod
list_modcol
list_more
list_next
list_open
list_pos
list_prev
list_read
list_refcnt
list_rows
list_seek
list_sort
list_stat
list_sync
list_view
list_view2
list_view3
list_vis
lock_rowlog
ltrim
max
mdi
min
move_f2l
move_l2f
name_in
numeric
nvl
paginate
popup_menu
pset
query_count
raw_input
record_exec
return
rollback
sizeof
string
sum
table_exec
trap
trigger
while
window
window_attr
window_count
window_name
window_seq
window_table

action_text

Modifies the action button text.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int action_text(button_name,text)
ident           button_name
string          text
```

Description

Changes the text of all buttons of the current window that match the *button_name* specification.

button_name specifies the name of the button(s) to be modified

text specifies the new text for the button

active_field

Sets the active field of a window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int active_field([field])
int               field
```

Description

Always returns the new active field, which has been stored in the *window-name*.AF variable.

field (optional) specifies a new active field by its count. If field's value exceeds the window's field count, the function assumes the field with the highest value. If you don't give a value for field, the function returns the existing (current) active field.

Notes

Use active_field() when you want an active field if the current window.

Use *window-name*.AF, where *window-name* is the name of the window, when you need the active field of a specific window.

Example

Useful in the [TAB] trigger.

```
{
if (active_field() == active_field(active_field()+1))
  active_field(0);
}
```

active_row

Sets the active row of a multi-record window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int active_row([row])
int             row
```

Description

Always returns the active row number, which has been stored in the *window-name*`.AR` variable.

row (optional) specifies a new active row. You can specify the row by its count. If `row`'s value exceeds the window row repeat count, the function assumes the row with the highest count. If you don't specify `row`, the function returns the existing (current) row.

Notes

Use `active_row()` when you need the active row of the current window.

Use *`window-name`*`.AR` when you want the active row of a specific window.

Example

```
Calculates the number of rows in the window and moves the data
from the list to the window.
{
int  i,ari,pos,rows;

ari  = active_row();
pos  = list_pos(p.wl);
rows = active_row(99999);

for (i=0;i<=rows;i++) {
 active_row(i); move_l2f();
 field_exec(uat_calc,true);
 if (list_pos(p.wl) == list_next(p.wl)) break;

for (i++;i<=rows;i++) { active_row(i); field_set(NULL);

active_row(ari);
}
```

active_wl

Gets the active window list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
list active_wl()
```

Description

Always returns the active window list.

Example

You can switch to a new window list by using `WL = active_wp (tmp);`

The following example performs a DESCRIBE on the window select without destroying the current window list.

```
{
 list wl_slave, tmp;
 int pos;
 pos = list_pos(active_wl ());
 wl_save = active_wl();
 query(-1);
 tmp = active_wl();
 active_wl(wl_save);
 list_seek(active_wl(), pos);
 list_close (wl_save);
 return (list_close(tmp));
}
```

alarm

Sets an alarm.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int alarm(seconds)
integer     seconds
```

Description

The alarm, when triggered, sends (raises) Ctrl-C (SIGINT) on Unix and (SIGABRT) on Win32. The function returns the number of seconds remaining from a previously set alarm for Unix and false (0) if not previously set, or true (non-zero) if previously set for Win32.

seconds specifies the number of seconds for which the alarm should be set. Zero (0) turns the alarm off.

Example

None.

append

Opens a report output file in append mode.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
void   append([filename])
string         filename
```

Description

filename (optional) specifies the name of the file to open. If you specify an empty string, `append()` sets the output file to `STOUT`. If you do not specify a `filename`, the function sets the output file to *`run_file`*`.OUT`.

For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

Example

Sends the report output to different locations based on a department identifier, `DEPTNUM`:

```
{
.
.
if (DEPTNUM == 42) {
 close();
 append("dept42.out");

else {
 close();
 append("other.out");

.
.
}
```

ascii

Returns an integer representation of the character.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
int  ascii(character)
char       character
```

Description

Returns ascii integer value of `character`.

character specifies the char (alphanumeric or space) to process.

Example

Prints the ascii code for each character or space as given.

```
{
int  i,len;
char s[80];
char ch;
s = "Hi my name is Bentley";

for (i=1;i<=length(s);i++) {
  ch = substr(s,i,1);
  printf("The ASCII value of '" ^^ ch ^^ "' is " ^^ ascii(ch));
  }
}
```

bell

Rings the (user alert) bell.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void bell()
```

Description

Makes a noise to alert the end user.

In a window application the bell beeps even if no windows are open.

Example

Set [Enter] to ring a bell.

```
key_set(key_enter,{ bell(); );
```

block

Invokes a block.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
void  block(block-name | data)
ident       block-name
int         data
```

Description

Invokes a specified block. You can specify any block from anywhere in the report.

block-name specifies the name of the report block to invoke.

data specifies the sequence number of the report block to invoke.

Notes

DVreport creates the following triggers by default:

- Main — block (first-block-name)
- Child — block (child-block-name)
- Post-block — block (sibling-block-name)

Example

Placed in a child trigger, conditionally invokes different child report blocks based on the value in SAL.

```
{
if (SAL > 20000) block(CHILD_OVER_20K);
else             block(CHILD_BELOW_20K);
}
```

block_count

Returns the number of report blocks in the report.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
int block_count()
```

Example

Executes all the blocks in a report.

```
{
int i;

for(i=0;i<block_count();i++) block(i);
}
```

block_seq

Returns the sequence number of a report block.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
int block_seq(block-name)
ident         block-name
```

Description

Returns the sequence number of a report block.

block-name specifies the name of the block to identify by number.

Example

None.

call

Invokes a specified application.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr call(run-file[,parm,...])
string    run-file
expr      parm
```

Description

Returns the results of `run_file`, up to 400 bytes.

run-file specifies the name of the function to call.

parm specifies the function's parameter(s), which can also be accessed in the trigger's main application.

Notes

You must ensure that either the `runpath` is specified in `trim.ini` or provide the full pathname of `run-file`.

Specifying the `.run` extention is optional.

This function is similar to `overlay()`, except that the calling function `call()` receives the `run-file`'s results. `overlay()` replaces itself with the `run-file` and the results are returned to a main trigger.

Example

Invokes TRIMmenu:

```
call("trimmenu.run");
```

The following example invokes an application and passes the TRIMrun start time as a parameter.

```
call("app1.run",g.time);
```

call_level

Returns the current call level.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int call_level()
```

Description

Returns the current `call()` level. The starting level is 0.

Example

Determine if the application was called from DVrun or by a `call()`:

```
if(call_level()) printf("We were invoked via call()");
else             printf("We were invoked via DVrun");
```

chr

Returns a character string the length of the number of parameters.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string   chr(parm1[,parm2,...,parm\sl{n])
int      parm
```

Description

Returns a string that has the same number of characters as the number of `parms` specified.

parm integers that specify a character place. Each one is processed by Modula 256.

Notes

Be careful when you display any of these strings since they can include nonprintable characters.

Example

Prints the character representation of the given ascii code.

```
{
 int i;

 i=65;
 printf("The ASCII code of '" ^^ i ^^ "' is a(n) " ^^ chr(i));
}
```

chr2

Returns a unicode character string the length of the number of parameters.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string   chr2(parm1[,parm2,...,parm\sl{n])
int      parm
```

Description

Returns a string that has the same number of unicode characters as the number of `parms` specified.

parm integers that specify a character place. Each one is processed by Modula 65536.

Notes

Be careful when you display any of these strings since they can include nonprintable characters.

Example

Prints the character representation of the given unicode values.

```
{
 int i;

 i=65;
 printf("The Unicode character of '" ^^ i ^^ "' is a(n) " ^^
chr2(i));
}
```

clipboard

Manages the TRIMpl clipboard.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
retval   clipboard(command[,parm[,...]])
expr     retval
int      command
expr     parm
```

Description

Manages the TRIMpl clipboard. The clipboard is a memory area that can hold any datatype. The return value is determined by the command. Offsets are zero-based.

command specifies what clipboard operation to execute (see trim.h for cb_* values) and is one of::

cb_alloc,[size[,inc]] -- Allocate the clipboard with two optional parameters.The first parameter is the initial clipboard size and the second is the increment size. The defaults are initial size of 128,000 bytes and increment size of 32,000 bytes.

cb_free -- Frees the clipboard and its resources

cb_length[,length] -- Returns the current length of the clipboard. If the length is given, sets the current length of the clipbaord and returns the new length.

cb_append[,data] -- Appends the data in the parm and returns the new length of the clipboard. The data is not converted.

cb_insert[,data[,offset]] -- Inserts the data at the given offset. The data is not converted. The default offset is 0. Returns the new length.

cb_delete[,offset[,length]] -- Deletes the data starting at the given offset. If the optional length is not given, then the clipboard is truncated at the offset. The default offset is 0 and the length is remainder of the clipboard. Returns the new length.

cb_extract[,offset[,length]] -- Returns the data starting at the given offset. If the optional length is not given, then all the data from the offset to the end is returned. The default offset is 0 and the length is remainder of the clipboard. The data is returned as a character datatype.

cb_file,filename[,write[,append]] -- Reads or writes the file (see file specification) . If write is false, then the file is read. If append is false, then the data is not appended to the file or clipboard. Returns the length of the clipboard.

close

Closes the current report output file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
void close()
```

Example

Part of the default main trigger code:

```
{
.
.
paginate(break|footer);
close();
}
```

commit

Executes the SQL statement COMMIT WORK.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void commit([update])
keyword       update
```

Description

Starts an update transaction; default is read-only.

update (optional) specifies that the transaction is (read-write).

Notes

Some database management systems do not allow applications to specify read-only (update) transactions. Those systems ignore the `update` option.

Sybase 4.9

When the function executes `commit(update)`, all Sybase access is performed through one database process. All `list_open()` calls must return all the desired data before another Sybase access is attempted. Otherwise, Sybase returns an error.

`lock_row()` performs an implicit SELECT to Sybase. Executing a `lock_row()` with results pending causes an error.

Example

Inserts into a table and commit it.

```
{
commit(update);
exec_sql("INSERT INTO log VALUES (10,SYSDATE)");
commit();
}
```

confirm

Displays a confirmation window and returns the value for the user's choice.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
int confirm(title,msg[,default[,icon[,button]]])
string      title, msg
int         default, icon, button
```

Description

Displays a confirmation window with the `title` and `msg` and returns the specified value.

default (optional) specifies one of the following buttons:

Mmemonic	Value	Description
confirm_no (default)	0	no button
confirm_yes	1	yes button
confirm_ok	2	ok button
confirm_cancel	3	cancel button
confirm_retry	4	retry button
confirm_abort	5	abort button
confirm_ignore	6	ignore button

icon (optional) specifies one of the following icons:

Mmemonic	Value	Description
confirm_quest (default)	0	question mark
confirm_exclam	1	exclamation mark
confirm_stop	2	stop sign
confirm_info	3	information icon
confirm_none	4	no icon

button (optional) specifies one of the following buttons:

connect

Establishes a database connection.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void connect(id[,connect-string])
int          id
string       connect-string
```

Description

Establishes a connection to a database with the specified id, using the specified `connect-string`. If the function is successful, the connection becomes the active database.

id identifies the connection

connect-string (optional) specifies the connect string to use for the database connection. If no `connect-string` is specified, the previously established connection identified by `id` becomes the active connection.

Notes

DesignVision Users Guide details the connect string syntax.

Example

Displays a list from an Rdb database based on data from an Oracle database:

```
{
connect(db_oracle,"niklas/back");
connect(db_rdb,"salary_data");
.
.
connect(db_oracle);
L1 = list_open("SELECT name FROM staff",1000,"Staff");
list_view(L1,0);
connect(db_rdb);
L2 = list_open("SELECT sal FROM personnel "
               "WHERE name = &/list_curr(L1,0)/",1,"Salary");
.
.
}
```

convert

Returns a string converted from one format to another..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string convert(src,srctype,desttype)
string         src
int            srctype
int            desttype
```

Description

Returns a string that has been converted from one format to another. The currently support conversions are defined in `trim.h`..

cdty_char ASCII characters

cdty_utf8 UTF-8 characters

cdty_ucs2 UCS-2 characters

Example

Convert an ASCII string to a UTF-8 string:

```
{
char abuf[100];
char ubuf[200];

abuf = "This is a wonderful day";
ubuf = convert(abuf,cdty_char,cdty_utf8);
}
```

count

Counts the number of fetches or parameters.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int count(field | parm)
ident     field
keyword   parm
```

Description

field (only used in Reportwriter) specifies that `count()` returns the number of fetches that returned a non-NULL value for that field in the invocation of the current block.

parm specifies that `count()` returns the number of parameters used in calling the current trigger.

Notes

If `field` is not a database field, the result is undefined. `count(parm)` is always valid.

Example

Counts the number of non-NULL salaries.

```
cnt_sal = count(SAL);
```

Count the number of parameters passed to the trigger.

```
parm_cnt = count(parm);
```

crypt

DES password and data encryption.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string crypt(key,salt)
string       key
string       salt
```

Description

`crypt()` returns an encrypted string base on the salt. The first two characters are the salt that was used to do the encryption..

key the password or data to encrypt

salr the two-character salt used to randomize the encryption. The non-DES salt format is $<ID>$<SALT>$

There are several encryption methods available which are triggered by the contents of the salt string:

Method	Salt prefix	Max salt length	Max key length	Max encoded length	Returns
DES	N/A	2	8	13	<salt><pwd>
MD5	1	6	unlimited	22	$<1>$<salt>$<pwd>
Blowfish	$2a$	16	72	53	$<2a>$<salt>$<pwd>
SHA-256	5	16	unlimited	43	$<5>$<salt>$<pwd>
SHA-512	6	16	unlimited	86	$<6>$<salt>$<pwd>

Notes

Only DES is supported on Windows, VMS, and MVS.

Example

Encrypt the password using a DES salt and store it in the database:

```
pwd = prompt("Please enter password ==> ");
pwd = crypt(pwd,"n9");
```

```
exec_sql(“insert into users values(:1,:2)”,uid,pwd);
```

Verify that the password is correct:

```
uid = prompt(“Please enter userid ==> “);
pwd = prompt(“Please enter password ==> “);
pwd2 = list_curr(list_open(“select pwd from users where uid =
&uid”,1),0);
pwd = crypt(pwd,substr(pwd2,1,2));
if (pwd == pwd2) printf(“Passwords match”);
else             printf(“Passwords do not match”);
```

```
Encrypt the password using a SHA-512 salt:
pwd = prompt("Please enter password ==> ");
pwd = crypt(pwd,"$6$BF6pxKiRWVZl$");
printf("SHA-512 value is "^^pwd)
```

cursor_col

Gets/sets the cursor column position.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int cursor_col([position])
int             position
```

Description

Returns the cursor's column location.

position (optional) specifies cursor position as column in the current row.

Notes

The range is between 0 and `G.SCREENCOLS - 1`.

If `position < 0` then 0 is used.

If `position > (G.SCREENCOLS - 1)` then `G.SCREENCOLS - 1` is used.

Example

Positions cursor in the middle of the screen on the current row.

```
cursor_col(G.SCREENCOLS / 2);
```

cursor_pos

Gets/sets cursor screen position.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int cursor_pos([position])
int             position
```

Description

Returns the cursor's screen position (row * `G.SCREENCOLS` + column).

position (optional) specifies a cursor position (`position /G.SCREENCOLS, position %G.SCREENCOLS`).

The value of `position` must be between 0 and (`G.SCREENROWS * G.SCREENCOLS - 1`).

If the value specified is < 0, then `position` is 0.

If position is > (`G.SCREENROWS * G.SCREENCOLS - 1`) then `position` is (`G.SCREENROWS * G.SCREENCOLS - 1`).

Example

Places the cursor in the lower right hand side of the screen.

```
cursor_pos(G.SCREENROW * G.SCREENCOLS - 1);
```

cursor_row

Gets/sets cursor row position.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int cursor_row([position])
int             position
```

Description

Returns the cursor's current row location.

position specifies the cursor position as row in current column.

The range is between 0 and `G.SCREENROWS - 1`.
If the value of `position` < 0 then 0 is used.

If `position` > `G.SCREENROWS - 1` then `G.SCREENROWS - 1` is used.

Example

Positions cursor in the first row of the screen at the current column.

```
cursor_row(0);
```

cursor_wait

Turns the window's busy indicator (hourglass) on or off.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void cursor_wait(onoff)
int              onoff
```

Description

onoff specifies the indicator setting. If the value is true (any non-zero value), the busy indicator is turned on. A zero value turns the busy indicator off.

Notes

The busy indicator is automatically turned off by any subsequent I/O operation. Most applications do not require an explicit `cursor_wait(false)`.

Example

None.

cuserid

Returns the user login name from the operating system.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string cuserid()
```

Example

Compares the user's login to list of authorized users.

```
{
.
.
if (list_find(auth_list,0,cuserid()) == NULL) {
  printf("Login not authorized!!!");
  escape();
  }
.
.
}
```

datadump

Returns informatin about the internal representation of the variable.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
char datadump(variable)
ident         variable
```

Description

Returns a string value containing the datatype, flag, length, and hex representation of the variable.

Example

```
{
int      i;
numeric  n;
char     c;
datetime d;

i = 10;
n = 5.7;
c = 6;
d = to_date("10-JUL-90");

printf(datadump(i));
printf(datadump(n));
printf(datadump(d));
printf(datadump(c));
}
returns
dty: 0, flg: 0, len: 4, 0A000000
dty: 2, flg: 0, len: 3, C10647
dty: 12, flg: 0, len: 7, 77BE070A010101
dty: 1, flg: 0, len: 1, 36
```

datatype

Returns the value of the data type.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
char datatype(variable)
ident         variable
```

Description

Returns a value that corresponds to a data type.

variable is the name of the variable to check.

Variable's Data Type	Returned Value
char/string	C
datetime	D
int	I
list	L
numeric	N
trigger	T

Example

Prints out the data type values:

```
{
char     g[1];
datetime h;
int      i;
list     j;
numeric  k;
trigger  t;

printf("char is " ^^ datatype(g));
printf("datetime is " ^^ datatype(h));
printf("int is " ^^ datatype(i));
printf("list is " ^^ datatype(j));
printf("numeric is " ^^ datatype(k));
printf("trigger is " ^^ datatype(t));
}
```

db_command

Sends a DBMS driver command.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void db_command(command,spec)
int             command
string          spec
```

Description

Sends the specified command to the DBMS driver. The `command values are defined in the trim.h` file. The `spec` string contains the values appropriate for the command.

Cmd	Spec	DBMS	Duration	Default	Description
0	n	All	Session	Differs	Set resource timer to n seconds
1	yes|no	Sybase	Session	No	Use a single PID within a transaction
3	dbname	Sybase	Session	N/A	Use database dname
4	language	Oracle	Session	See oparse() or StmtPrepare	Set SQL syntax and datatypes.
5	yes|no	All	Session	No	Return raw datetime
7	dty	Oracle	Session	1	Map VORTEX char type to Oracle type dty.
8	N/A	All	Call	N/A	Lock VORTEXaccelerator slave on next call
9	yes|no	Sybase	Session	No	Return single blank varchar as blank instead of null
10	yes|no	Sybase	Session	No	Batch INSERT, UPDATE, and DELETE statements
11	on|off	GENESIS	Session	off	Automatically commit after every SQL statement

Cmd	Spec	DBMS	Duration	Default	Description
12	N/A	Informix	Until next use	N/A	Sets the current connection to dormant
13	N/A	Oracle	Call	N/A	Returns the address of the LDA (Oracle 7 only)
14	value	TRIMrpc	Call	N/A	Cause a raise(value) error condition
15	message	TRIMrpc	Call	N/A	Return the message as an error
16	N/A	VORTEX webd	Call	N/A	Execute a temporary release
17	value	GENESIS /SDMS	Session	N/A	Call sdms2_initx() using value
18	n	GENESIS	Session	2	Set query timeout to n seconds
19	n	GENESIS /SDMS	Session	2	Set fetch timeout to n seconds
20	yes|no	ODBC	Session	connection default	Use dynamic cursors
21	N/A	GENESIS	Session	N/A	Return GENESIS syntax level
22	yes|no	All	Session	yes	Return ROWID on insert
23	yes|no	Oracle	Session	yes	Use new Oracle Blob/ Clob
24	yes|no	Various	Session	no	Use read-only cursors
25	n	ODBC	Session	connection default	Set cursor type to n
26	n	ODBC	Session	connection default	Set transaction isolation level to n
27	scroll option	Oracle Informix DB2 Adabas D ODBC	Cursor	N/A	Set the scrolling direction or position of a scrollable cursor (see VTXOPEN). Scroll options are defined in vortex.h as TDB_SMD_SCROLL_xx.

Cmd	Spec	DBMS	Duration	Default	Description
28	N/A	All	N/A	N/A	Return the hostname of the machine where the DBMS driver is running.
29	yes|no	All	Session	no	Return fully qualified column names.
30	N/A	ODBC	Cursor	N/A	Free the locks held by the cursor.
31	yes|no	ODBC SQL Server	Session	no	Describe W[VAR]CHAR columns as CHAR.
32	yes|no	ODBC SQL Server	Session	no	Force zero length parameter strings to be bound as NULL.
33	yes|no	DB2	Session	no	Process stored procedure cursors with EXIO.

Example

Notify the DBMS to not return the inserted row's ROWID value:

```
db_command(db_cmd_return_rowid,"NO");
exec_sql("insert into staff values(:1,:2,:3,;4,:5,:6,:7)",ll);
db_command(db_cmd_return_rowid,"Yes");
```

db_id

Returns the current database connection ID.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int db_id()
```

Description

Refer to `trim.h` for the possible `db_id()` values.

Example

Executes a different SQL statement based on current database connection.

```
if (db_id() == db_oracle)
  ll = list_open("SELECT decode(ID,1,'Clerk',2,'Mgr','Unknown') "
                 "FROM staff",100);
else if (db_id() == db_rdb){
  ll = list_open("SELECT id FROM staff",100);
  while (true) {

list_modcol(ll,0,decode(list_curr(ll,0),1,"Clerk",2,"Mgr","Unknown
"));
    if(list_pos(ll) == list_next(ll))break;
    }
  }
```

db_msg

Returns last database error message.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string db_msg()
```

Description

If the last database operation was a successful insert, `db_msg()` returns the unique row identification for the new row if the database supports this response. Otherwise the function returns the last database error message.

Example

Logs database error messages to a file.

```
if (g.error_code == err_DB) log("dbms.err",db_msg());
```

db_msgdump

Displays last database error message in a window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void db_msgdump(row,col)
int             row,col
```

Description

Displays the last database error message in a dialog box and waits for the user to acknowledge the message.

row specifies the row position for the displayed window. -1 places the upper left of the box at the current position. -2 places the box in the center of the screen.

col specifies the column position for the displayed window. -1 places the upper left of the box at the current position. -2 places the box in the center of the screen.

Example

Displays database error message and wait for the user to acknowledge.

```
if (g.error_code == err_DB) db_msgdump(3,4);
```

db_mux

Sets the deferred database slave lock if the VORTEXaccelerator is active.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void db_mux()
```

Description

If the VORTEXaccelerator is active, then `db_mux()` sets the deferred database slave lock flag, which locks the slave to the client on the following database operation.

You need to execute a COMMIT/ROLLBACK to free the slave.

Example

Locks the VORTEXaccelerator to the client to prevent a potential deadlock.

```
db_mux();
ll = list_open("select * from staff for update of id,salary",100);
exec_sql("update staff set salary = salary * .60
               where id = :1",list_curr(ll,0));
```

db_rows

Returns the number of rows affected by the last INSERT/UPDATE/DELETE operation on the current database connection.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int db_rows()
```

Description

Returns the number of rows affected by the last INSERT/UPDATE/DELETE database operation. If no error occurred, this value is the same as that returned by `exec_sql()`. If an error occurs during a bulk database operation, this value indicates how many INSERT/UPDATE/DELETEs were performed prior to the error.

Example

Performs a bulk insert into a table using a list. Reports how many rows were actually inserted.

```
if (trap( { exec_sql("insert into staff values (:1)",L1); } ))
  prompt("Insert failed after " ^^ db_rows() ^^ " rows");
else prompt(db_rows() ^^ " were inserted successfully");
```

db_sqlcode

Returns last database SQLCODE error code.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int db_sqlcode()
```

Description

Always returns the last SQLCODE value.

Example

Logs database error codes to a file.

```
if (g.error_code == err_DB) log("dbms.err",db_sqlcode());
```

debugger

Invokes the runtime debugger.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
void debug()
```

Notes

To use this function you must run the `.app` or `.rep` file with TRIMgen's -g option. For more information on debugging, refer to the ***DesignVision Users Guide***.

Example

Calls the debugger when a certain employee has been processed.

```
if (NAME == "JONES") debugger();
```

decode

Decodes an expression.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr decode(code,val1,ret1[,val2,ret2,...],default)
expr        code, valn, retn, default
```

Description

Compares `code` to each `val`*n*. When it finds a match, it returns `ret`*n*. If it doesn't find a match, it returns `default`.

Note: All of the expressions, that is `val`*n* and `ret`*n*, are evaluated before code is compared against the `val`*n* expression(s).

Example

Decodes user responses; `-1` indicates an invalid response:

```
{
int  confirm;
char response[80];

response = prompt("Enter YES or NO ==> ")
confirm  = decode(response,"N",0,"NO",0,"Y",1,"YES",1,-1);
}
```

Executes different code based on variable:

```
execute(decode(case,1,trigger_code1,2,trigger_code2,default_code));
```

delete

Deletes a file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int delete(file-name)
string     file-name
```

Description

Deletes a file. Returns the error code, if any, received from the operating system.

file-name specifies the file to delete.

For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

Example

Deletes a temporary list file.

```
delete("listtemp.tmp");
```

design_name

Returns the name of the application or report.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		

Syntax

```
string design_name()
```

Description

Returns the name of the application or report.

Example

None.

dialog_button

Adds action buttons to list_view3 dialogs.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
int dialog_button([index[,code[,text]]])
int                index
int                code
string             text
```

Description

Manages the additional aaction buttons in list_view3 dialogs. You must specify opt_buttons in the list_view3 option parameter.

index Specifies which button to manager, 0-3.

code Specifies the value to be returned when the button id pressed.

text Specifies the text for the button.

If no parameter is specified, then the `code` of the last button pressed is returned. If `index` is specified, then the `code` for that button is returned. If `code` is specified, then the `code` for that button is set and returned. If `text` is specified, then the `text` for that button is set and the `code` is returned.

dir

Performs various directory functions.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr dir(operation[,name])
int             operation
string          name
```

Description

You can perform one of five operations with `dir()`. Select the operation by specifying the appropriate operation parameter:

0 Make directory. Requires the ***name*** of directory to make. Returns 0 on success.

1 Remove directory. Requires the ***name*** of directory to remove. Returns 0 on success.

2 Change directory. Requires the ***name*** of directory to go to. Returns 0 on success.

3 Get current directory. Returns working directory.

4 Get directory separator. Returns directory separator character.

Example

Change the current working directory.

```
{
char wd[128];
wd = prompt("New Directory:");
if (!dir(2,wd)) printf ("cwd set to "^^wd);
else printf("Invalid directory: "^^wd);
…
}
```

dump_scr

Dumps screen contents to a file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
	X			

Syntax

```
void dump_scr(file-name)
string     file-name
```

Description

Deletes a file. Returns the error code, if any, received from the operating system.

file-name specifies the file to in which to dump screen contents.

> For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

Screen fields are delimited by characters 15 and 16. Graphic characters are dumped as detailed in the following list. Subsequent calls with the same file name append the screen contents to the same file.

Value	Description
0	Upper left corner of box.
1	Lower left corner of box.
2	Upper right corner of box.
3	Lower right corner of box.
4	Top horizontal line.
5	Center horizontal line.
6	Bottom horizontal line.
7	Left vertical line.
8	Center vertical line.
9	Right vertical line.
10	Center line intersection.
11	Left tee.
12	Right tee.
13	Top tee.
14	Bottom tee.

Example

None

edit_text

Displays a window for edit texting.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int edit_text(text,max_len,row,col,height,width,wrap)
expr           text
int            max_len, row, col, height, width, wrap
```

Description

`edit_text()` opens an edit window using the data in the text parameter. It returns false and leaves the data unmodified if the user clicks [Cancel] . If the user clicks [OK], it returns true and returns the modified char/string/list in the original text parameter.

text either a char/string or list datatype (GUI only), specifies the text to edit.

max_len when text is a char/string, specifies the maximum edit return text length. If max_len is zero, a read-only window opens with only an [OK] button. Any text that is a one-column list causes the window width to adjust to the max list row size. This is useful for browsing text, such as help information. If the list has a title, the first line appears in the window title. If text is a list, this parameter determines whether a window is read-only or modifiable.

row specifies the row position for the edit window. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

col specifies the column for the edit window. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

height specifies the window's height.

width specifies the window's width.

wrap controls word wrapping. Set to true, wraps words.

Example

Edits the comment field in a customer order application.

```
edit_text(comment,field_width()-1,5,5,5,72,true);
```

error

Displays a user-invoked error.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void error(message)
string     message
```

Description

In DVreport displays a message and aborts the report.

In DVapp executes the code that has been specified by `error_trap()` and returns control to the previous escape level.

`message`, which specifies the contents of the variable `parm[0]`, is read by `error_trap()`.

Example

Checks for negative salaries.

```
if (SAL < 0) error("negative salary: " ^^ SAL);
```

error_msg

Returns last error message.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string error_msg()
```

Description

Returns the last error message encountered.

Example

Logs the `error_msg` after a `trap()` call.

```
if (trap({list_open("phone.lst",100);}))
  log("listopen.log",error_msg());
```

error_trap

Defines code called by `error()`.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void error_trap(error-code)
block           error-code
```

Description

Defines code that is called by `error()`. It reads the contents of `error()`'s variable in `parm[0]`.

Example

Placed in the Global window trigger by default:

```
error_trap({ g.msg = parm.0; bell(); });
```

During the development phase of a system, you may find the following `error_trap` useful:

```
error_trap({ g.msg = parm.0; log("error.log",parm.0); bell(); });
```

escape

Returns to previous escape level.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void escape([input])
keyword      input
```

Description

Returns through multiple escape levels. Currently two are available:

- ***Application*** — On entry to the application.
- ***Window*** — On entry to the window.

Typically, `escape()` is used in application key triggers to return to the window trigger immediately following the current `go_field()`} or `input()` statement.

In reports, `escape()` exits the report and returns to the operating system.

input (optional) if placed in a validation or key trigger, specifies that the execution continues after the `[raw_]input()` call. (This action resolves the problem of exiting input (quit key) from a not-null field that is empty; for example, `escape(input)` in a prevalidation trigger.) Otherwise, the execution goes to window top.

Example

Assigned to [F3] (End) in the application builder, forces an exit to the window trigger if the `go_field()` function is used:

```
escape();
```

exec_proc

Executes a database stored procedure statement.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int exec_proc(statement[[,variable-name,data,flags]...])
string        statement, variable-name
expr          data, flags
```

Description

Executes the specified stored procedure, returning `0` if completed successfully. If an error occurs, and `exec_proc()` is part of the expression, the function returns `-1`.

If `exec_proc()` isn't part of the specification, the function escapes to the window trigger error trap or, for reports or stand-alone TRIMpl, the operating system command shell.

Not all databases support stored procedures. Furthermore, only Oracle accepts array parameters. If you pass arrays to a database that does not support them, the database returns an error.

statement specifies the procedure to execute. Multiple `statements` can exist.

variable-name (optional) specifies the name of a given argument.

data (optional) must be aTRIMpl variable if the argument is an output variable.

flags (optional) indicates whether the variable is an input and/or an output variable. `flags` possible values are:

- 1 — Input
- 2 — Output
- 3 — Both input and output (bitwise OR of Input and Output)

and are defined in header files (`trim.h` or `dv.h`).

Example

Accesses a function that has been stored in an Oracle database.

```
{
int i,j,k;

for (i=4,j=4;i;i--) {
  exec_proc("begin :arg0 := sample6.add_func(:arg1, :arg2); end;",
            ":arg0",k,2,":arg1",i,1,":arg2",j,3);
  printf("i: " ^^ i ^^ ", j: " ^^ j ^^ ", k: " ^^ k);
  }}
```

exec_row

Executes a row-oriented SQL statement.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int exec_row(statement[,host-variable,...])
string       statement
expr         host-variable
```

Description

Uses the current database connection to determine the name of the unique row identification column. For example, Oracle uses ROWID and Rdb uses DBKEY. The host variable reference character varies between databases. To maintain database portability, use the Oracle ‘‘:n’’ notation. All other systems replace the ‘‘:n’’ with ‘‘?’’ notation.

Adds a database-specific unique row identification column to the WHERE clause of a statement and executes it. Returns the number of affected rows for UPDATE and DELETE.

For other operations, it returns zero (0). If an error occurs in the SQL statement and `exec_row()` is part of an expression, the function returns -1.

statement specifies the SQL statement to execute.

host-variable (optional) specifies the bind variables (either as variables or hard-coded) for the `statement`. If `host-variable` is a list, all `host-variable` references are resolved from the list.

If `exec_proc()` isn’t specified, the function escapes to the window trigger error trap DVapp operating system command shell (DVrep or stand-alone TRIMpl systems).

Notes

Uses the current database connection to dtermine the name of the unique row identification column. For example, oracle uses ROWID and Rdb uses DBKEY. The host variable reference character varies between databases. To maintain database portability, use the Oracle “:n” notation. All other systems replace the “:n” with “?” notation.

Example

Deletes a particular window list row from the database. The row identification value is in the first column of the window list.

```
{
.
exec_row("DELETE FROM staff WHERE :1 = ",list_curr(p.wl,0));
.
}
```

Deletes several rows from the database.

```
{
list dl;         /* dl has 1 column */
exec_row("DELETE FROM staff WHERE :1 = ",dl);
}
```

exec_sql

Executes a SQL statement.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int exec_sql(statement[,host-variable,...])
string       statement
expr         host-variable
```

Description

Executes a SQL statement and returns the number of affected rows for INSERT, UPDATE, and DELETE. For other operations, function returns 0. If an error occurs in the SQL statement and `exec_sql()` is part of an expression, the function returns -1.

host-variable (optional) specifies the bind variables (either as variables or hard-coded) for the `statement`. If `host-variable` is a `list`, all references are resolved from the list.

Notes

If `exec_proc()` isn't specified, the function escapes to the window trigger error trap or, for reports and stand-alone TRIMpl, operating system command shell.

The host variable reference character varies between databases. To maintain database portability, use the Oracle ''**:**n'' notation; on all other systems, the ''**:**n'' is automatically replaced by ''?'' notation.

Sybase

Sybase stored procedures can be executed by using `exec_sql()`, which returns either the number of rows processed or the raiserror value in case of error. The raiserror or print message can be retrieved with `db_msg()`.

```
count = exec_sql("insert_proc",id,name);
```

Example

Increases the salary of everyone in department 58; a prompt asks for the percent increase for each person.

```
if (DEPT == 58)
  exec_sql("UPDATE staff SET sal = sal * " ^^
           1 + prompt("Enter increase for " ^^ NAME ^^ " ==> ") /
100 ^^
           " where NAME = " ^^ NAME);
```

Delete all employees whose salary is greater than max_salary.

```
if (exec_sql("DELETE FROM staff WHERE sal > :1",max_salary) < 0)
  printf("Error occurred while deleting overpaid employees");
```

exec_usr

Invokes user exit routine, `cexuser()`.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string exec_usr (arg1,arg2,...,argn)
expr             arg1,arg2,...,argn
```

Description

The runtime executor is linked with a sample routine (refer to the `cexuser.c` in *Notes* below. The method of passing parameters is similar to that of passing parameters to a C program:

```
char *cexuser(argc,argv)
int           argc;
char          argv[];
```

The difference is that `argc` contains the actual number of parameters; for example, `argv[0]` points to the first parameter. As in C, the parameters are all null-terminated (conversion is automatic). The return value must be a null-terminated string that is returned by the user exit function.

If your application needs multiple user exits, insert a code as the first parameter to `cexuser()` which, in turn, contains a switch statement based on this code.

Notes

The following code for `cexuser()` shows a sample exit routine invoked by `exec_usr()`. The routine prints out the number of arguments, prints out each argument, and returns a message:

```
#include <stdio.h>

char *cexuser(argc,argv)
int           argc;          /* number of arguments */
char         *argv[];        /* array of arguments */
{
int i;                       /* loop variable */

printf("cexuser called with %d argument(s)\n\n",argc);
for (i=0;i<argc;i++) printf("arg%d: %s\n",i,argv[i]);
return("All is OK");
}
```

Example

Assumes that `cexuser.c` does some special formatting on a single value.

```
new_value = exec_usr(SAL * 1.10);
```

execute

Executes a code variable.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr execute(code-var[,parm,...])
trigger      code-var
expr         parm
```

Description

Executes stored code. Returns the value the code returns; if the executed code has no return value, `execute()` returns a NULL.

code-var specifies the code to execute.

parm (optional) specifies a parameter for `code-var`.

Example

Prepares an execution stack and sends it to a user function for execution:

```
{
trigger insrt;
trigger appnd;
.
.
insrt = { move_f2l(1);  field_set(NULL); move_f2l(0); };
appnd = { move_f2l(-1); field_set(NULL); move_f2l(0); };
.
.
if (direction == up) list_mod(stack,1,appnd);
else list_mod(stack,1,insrt);
.
.
exec_stack(stack);
}

exec_stack:
{
int i;
list_seek(parm.0,0);
for (i=list_rows(parm.0);i;i-) execute(list_read(parm.0,0));
}
```

field_attr

Returns the user attribute mask for the currently active field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_attr()
```

Description

User attributes are defined in the `trim.uat` file and represent bit values corresponding to their position in the file. Typically these bit values are defined in `trim.h`.

Example

Counts the number of primary key fields in the window:

```
{
for (p.af = field_count()-1;p.af>=0;p.af--)
  if (field_attr() & uat_prikey) cnt_prikey++;
}
```

field_color

Sets field colors.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void field_color([ALL,]rgb[,[name|number] | [attribute,flag]])
expr              rgb
string            name
int               number, attribute, flag
keyword           ALL
```

Description

ALL (optional) specifies that the color applies to all rows (for multi-record layouts). Otherwise only the current row is affected.

rgb is the RGB sequence for all fields in the current window.

name specifies the name of an individual field to which changes are applied.

number must evaluate to a valid field sequence number and specifies the field (by number) to which changes are applied.

attribute specifies that all fields with specified attribute have their colors set.

flag together with `attribute` specifies action to fields with the specified attribute.

Notes

The RGB sequence must be in hexadecimal format. The RGB sequence is either 8 bytes (background color only) or 16 bytes (both back and foreground colors). Each 8-byte sequence must be in a RRGGBBOO format where:

RR RED portion. Range 00-FF.

GG GREEN portion. Range 00-FF.

BB BLUE portion. Range 00-FF.

OO where option is one of the following:
- 00 do not set (RRGGBB ignored).
- 01 reset to default colors (RRGGBB ignored).
- 02 set the colors.
- 08 set to transparent (background only).

If the option portion of the RGB sequence is 00 or 01 then the RRGGBB portion can be set to any values.

Example

Sets all rows fields' background color to red.

```
field_color(all,"FF000002");
```

Sets all fields' background color to yellow and the foreground (text) color to blue for the current record (row).

```
field_color("FFFF00020000FF02");
```

Resets all fields' background color to the default values.

```
field_color("00000001");
```

Resets all rows fields' foreground (text) color to black.

```
field_color("0000000000000002");
```

field_count

Returns the number of fields in the current window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_count()
```

Example

Modifies the active field based on the number of fields in the window:

```
{
if (p.af == 0) p.af = field_count() - 1;
else p.af - -;
}
```

field_dynattr

Assigns dynamic field attributes.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void field_dynattr([data[,[name | number] | [attribute,flag]]])
expr                data
int                 attribute,number,flag
string              name
```

Description

If no parameters are specified, the active field's dynamic attribute is returned..

data specifies a dynamic attribute. If it has no `name` or `number`, all fields in the current window are assigned the same dynamic attributes.

name (optional) identifies a single field to which attributes are assigned.

number (optional) identifies field by sequence number (must evaluate to a valid field sequence number).

attribute specifies a bitmask. The following are valid, dynamic, and can be changed at runtime:

- NOECHO
- UPPER
- RESET
- AUTOSKIP
- PROTECT

flag = true specifies that the dynamic attributes are assigned to all fields that match the bitmask.

Notes

See `trim.h` for examples of dynamic attributes.

Example

None.

field_exec

Executes the field triggers in the active window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void field_exec([name | number] | [attributes,flag])
string            name
int               number,attributes,flag
```

Description

Executes all field triggers if specific ones are not specified.

name specifies particular fields by name.

number specifies particular fields by field sequence number.

attribute specifies a bitmask.

flag = true specifies that all fields with the identified attribute are executed; if `flag` is not specified, all fields without field attributes are executed.

Notes

The value of the *`window-name`*`.AF` is preserved across the `field_exec()` call. Any changes to the active field in the field trigger are not preserved.

Example

Executes all triggers for fields with the CALCULATE and PRIMARY KEY attributes:

```
field_exec(uat_calc|uat_prikey,true);
```

field_flag

Sets/gets status of a field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_flag([data[,[name | number] [attribute,flag]]])
expr            data
int             attribute,number,flag
string          name
```

Description

Sets or gets the status of a field.

If no parameter is given, the active field's flag is returned.

The integer flag is assigned to each field in an application; each flag is initially set to 0.

data optional) specifies the new flag value, which can be one of the following (SQL*Forms compatible) flags:

Value	Flag	Description
1	flg_active	Field is currently active. If reset, field assignments are not echoed to the screen. Turned on by window (w,open) Turned off by window (w, close).
2	flg_modified	Value is assigned to the field's variable
4	flg_input	Field input complete.
8	flg_output	Field output complete.

name specifies particular fields by name.

number specifies particular fields by field sequence number.

attribute specifies a bitmask.

flag = true specifies that all fields with the identified attribute are executed; if `flag` is not specified, all fields without field attributes are executed.

These last three flags are not reset by the system; the application must reset these flags. Review `trim.h` for more details.

field_helpname

Returns the help name for the current field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
string  field_helpname()
```

Description

This function returns an identifier for the current field that you use to look up help text, for example. If no identifier (help name) has been defined, it returns NULL.

Example

Returns the help name (identifier) for the current field:

```
{
list_view(list_open("select * from myhelp
           where helpname="^^field_helpname(),100),0);
}
```

field_init

Executes the field initialization triggers in the active window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void field_init([name | number] | [attributes,flag])
string           name
int              number,attributes,flag
```

Description

Executes all field initialization triggers if specific ones are not specified.

name specifies particular fields by name.

number specifies particular fields by field sequence number.

attribute specifies a bitmask.

flag = true specifies that all fields with the identified attribute are executed; if `flag` is not specified, all fields without field attributes are executed.

Notes

The value of the *`window-name`*`.AF` is preserved across the `field_init()` call. Any changes to the active field in the field trigger are not preserved.

Example

Executes all initialization triggers for fields with the CALCULATE and PRIMARY KEY attributes:

```
field_init(uat_calc|uat_prikey,true);
```

field_mask

Returns the field format mask for the current field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
string field_mask()
```

Example

Formats date data to match screen field mask:

```
field_d = to_char(SYSDATE, field_mask());
```

field_name

Returns the name of the currently active field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
string field_name()
```

Example

Creates a generic validation trigger that displays the name of the field in which the error occurs:

```
if (parm[0] < 0) error("Field " ^^ field_name() ^^ " must be >
0");
```

field_offset

Returns the column offset of a field within the report page or window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		

Syntax

```
int field_offset()
```

Notes

Valid only in field triggers.

Example

Calculates how much space is available between the field and the right edge of the page.

```
space_left = G.PAGEWIDTH - field_offset();
```

field_rows

Returns the number of rows in a multi-row window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_rows()
```

Example

Useful as part of the trigger code for the down-arrow key:

```
{
int rows;
if (list_pos(p.wl) := list_next(p.wl)) { /* did we move? */
  rows = field_rows() - 1;               /* # of rows */
  if (rows > 0) {                        /* multi row window? */
.
}
```

field_seq

Returns the field sequence number.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_seq(field-name | field-num[,list])
ident         field-name
keyword       list
expr          field-num
```

Description

field-name (optional) specifies the field by name.

field-num (optional) specifies the field by sequence number.

list (optional) specifies that only those fields with the LIST attribute are used to determine the field's sequence number.

Example

Changes the field flow based on the current data:

```
{
.
if (salary > 15000) af = field_seq(decrease_field);
else                af = field_seq(increase_field);
.
}
```

field_set

Sets a field value.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void field_set(data[,[name | number] | [attribute,flag]])
expr           data
int            attribute,number,flag
string         name
```

Description

data specifies the value to use.

name (optional) specifies a field by name.

number (optional) specifies a field by sequence number. It must evaluate to a valid field sequence number.

attribute (optional) specifies the user bitmask.

flag = true (optional) specifies that all fields that have any of the user attributes in the bitmask are set; otherwise, all fields that do not have any of the user attributes are set.

Example

Resets all fields:

```
field_set(NULL);
```

Resets all non-PRIMARY KEY fields:

```
field_set(NULL,uat_prikey,false);
```

field_sysattr

Returns the system attribute mask of the current active field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

int field_sysattr()

Notes

The values are defined in `trim.h`.

Example

Counts the number of database fields in the window:

```
{
for (p.af = field_count()-1;p.af >= 0;p.af--)
  if (field_sysattr() & sat_database) cnt_DBMS++;
}
```

field_test

Returns the number of fields in the current window whose value matches `data`.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_test(data[,[name | number] | [attribute,flag]])
expr           data
string         name
int            attribute,number,flag
```

Description

data specifies the value against which to test fields.

name (optional) specifies the name of a particular field to test.

number (optional) specifies the value of a field to test. It must evaluate to a valid field sequence number.

attribute (optional) specifies a bitmask.

flag (optional) if set to true, specifies that all fields with any of the user attributes are checked; otherwise, all fields that do not have any of the user attributes are checked.

Example

Modifies the active field based on the data in a field:

```
{
if (field_test(NULL,"ADDRESS") > 0) p.af = field_seq("PHONE");
else                                p.af = field_seq("ZIPCODE");
}
```

Tests to determine if the primary key is complete:

```
if (field_test(NULL,uat_prikey,true) := 0)
  error("Primary key is not complete.");
```

field_tid

Returns the active field's table ID.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
string field_tid()
```

Description

Returns a character identifying the current table: The first table's ID is A, the second table's ID is B, etc.

Example

None.

field_type

Returns the type of the active field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_type()
```

Description

Returns the type of the active field. Possible return values are defined in *trim.h*:

fty_comples complex field

fty_edit edit field

fty_grid grid field

fty_list list field

Example

None.

field_val

Executes the active window's field validation triggers.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_val(msgbuf,[[name | number] | [attribute,flag]])
string        msgbuf,name
int           attribute,number,flag
```

Description

Returns -1 if valid. Otherwise, the function returns the field sequence number of the first invalid field in the set of matching fields.

msgbuf contains the error message returned by `error()`, if one exists.

name (optional) specifies a field validation trigger by name.

number (optional) specifies a field validation trigger by number. It must evaluate to a valid field sequence number.

attribute (optional) specifies a bitmask.

flag = true (optional) specifies that the validation triggers are executed for all fields that have any user attributes that match the bitmask; otherwise, the validation triggers are executed for all fields that do not have any of the user attributes.

Notes

The *`window-name`*`.AF` is preserved across `field_exec()`. Any changes to the active field in the field trigger are not preserved.

Example

Validates all fields and moves cursor to the field containing an error:

```
{
int fnum;
fnum = field_val(g.msg);
if (fnum := -1) active_field(fnum);
}
```

field_visual

Sets/gets field attributes.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int field_visual([data[,[name|number] | [attribute,flag]]])
expr               data
int                attribute,number,flag
string             name
```

Description

If no parameters are given, the active field's visual attribute is returned.

data (optional) specifies the visual attribute index. Without qualifiers, all fields in the current window are assigned the same attributes.

name (optional) specifies the name of a field to which attributes are assigned.

number (optional) specifies the number of a field to which attributes are assigned. It must evaluate to a valid field sequence number.

attribute (optional) specifies a bitmask.

flag = true (optional) specifies that the attributes are assigned to all named fields; if only `data` is given, all fields in the current window are assigned the visual attributes..

Notes

See `trim.h` for examples of visual attributes.

Example

None.

field_width

Returns the field's width.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		

Syntax

```
int field_width()
```

Notes

This function is valid only in field triggers.

Example

Returns column offset following the field.

```
offset = field_offset() + field_width();
```

file_copy

Copies a file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void file_copy(source,destination[,file-type])
string         source,destination,file-type
```

Description

source specifies the file to copy.

For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

destination specifies the new filename.

file-type (optional) specifies the type

- a = ascii text
- b = binary copy (default)

Notes

When specifying file names the extended file names are available. See ***DesignVision Users Guide*** for details on specifying file names.

Example

Copies a file to the Windows machine's clipboard.

```
file_copy("/tmp/bad.log","gui:clipboard","a");
```

focus

Returns the current object focus.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
int focus(type[,id])
int       type,id
```

Description

Returns the id of the field in the current window that has focus or, if no field has focus, it returns false (zero). If no window has focus, it returns-1.

type specifies focus. Zero (0) specifies field focus; Nonzero specifies window focus.

id (optional) specifies a widget id when type = 0. if `id` is specified and `type` is false then the focus is set on widget `id`.

Example

A sample focus change event trigger could perform:

```
{ int i;
  i = p.ar;
  p.ar = focus(false) % field_rows();
  list_seek(p.wl,list_pos(p.wl)+p.ar-i);
  p.af = focus(false) / field_rows();
}
```

formfeed

Controls formfeed behavior in a report.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
void formfeed(flag)
int           flag
```

Description

flag specifies whether the formfeed value is written at the end of a report page.

gen_time

Gets the datetime when the application was generated.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
datetime gen_time()
```

Description

Returns the datetime when the application was generated with TRIMgen.

Example

Displays the application generation datetime information on the screen.

```
field = gen_time();
```

getenv

Reads an environment variable from the operating system.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string getenv(env-name)
```

Description

env-name specifies the variable to read. If none is defined, the function returns an empty string.

Notes

In VMS, environment variables correspond to logicals and symbols.

Example

Finds the name of the output device that is to be used.

```
printer = getenv("PRINTER");
```

See also

"***putenv***" on page 205

go_field

Invokes specified field and moves cursor to it.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void go_field([name | number])
ident          name
int            number
```

Description

name specifies the field by name.

number specifies the field by absolute field number. Otherwise, the last field number set in the *window-name*.AF variable is used.

Notes

When `go_field()` is specified in the window trigger, the first field is considered to be the current field. In this case, `go_field(0)` invokes the first field.

Example

Useful in the field triggers to move between fields:

```
{
if (p.mode == 0) input(p.input_var); /* normal mode */
else raw_input(p.input_var);         /* query mode */
go_field();                          /* go to field specified by
p.af */
}
```

greatest

Gets the greatest (maximum) value in a set.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr greatest(data1,[data2[,...]])
expr          data1, data2
```

Example

Find the latest of several datetime values.

```
printf("The latest date is " ^^ greatest(dt1,dt2,dt3));
```

gui_canvas

Returns or sets the GUI canvas data for the specified canvas..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
string gui_canvas(canvas_name[,data_string])
string            canvas_name
string            data_string
```

Description

The `canvas_name` parameter is the name of the canvas object. If `data_string` is not .specified, then the data values of all the modifiable named objects in teh canvas is returned in an XAML string. To set data values, set `data_string` to

```
<DATA object_name="value">
```

where `object_name` matches the object's name in the XAML string that defines the canvas.

There are two ways to load the HTML/XAML canvas definition. The first is using dvApp to insert the definition which is then stored as part of the design however the WPF client expects XAML and will fail if HTML is found in the canvas. Thus if the goal is to have a shared WPF and HTML5 browser project, then it is better to leave the dvApp canvas object definition empty and dynamically load it using `gui_canvas()` at runtime. The `gui_id()` function returns the type of the client being used.

WPF

WPF applications use XAML to define the canvas. While any valid XAML may be used, only the following object types and values can be read or written by `gui_canvas()`:

.

Control	*Write*	*Read*	*Format*
Button	Button name	N/A	Text
CheckBox	State	State	"":null, 0:unchecked, non-zero:checked
Image	URI	URI	URI
ProgressBar	Progress	Progress	nn.nn, the value between the minimum (default 0) and maximum (default 100)

Control	*Write*	*Read*	*Format*
Slider	Position	Position	nn.nn, the value between the minimum (default 0) and maximum (default 10)
TextBlock	Content	N/A	Text
TextBox	Content	Content	Text
WebBrowser	URL	URL	URL

HTML5 Browser

HTML5 browser applications use HTML to define the canvas. While any well-formed HTML may be used, only the object types defined for WPF above can be read or written by `gui_canvas()`.

Example

WPF

A canvas object named CV has the following definition in the .gap file:

```
<StackPanel>
 <TextBlock Name="TITLE">A Sample Canvas</TextBlock>
 <Button Name="B1" Tag="A909">Press Me</Button>
 <Button Name="B2" Tag="A808">And me!</Button>
 <StackPanel Orientation="Horizontal">
  <TextBox Name="FNAME"/>
  <Separator/>
  <TextBox Name="LNAME"/>
  <Separator/>
  <CheckBox Name="CB"/>
  <Separator/>
  <ProgressBar Name="PB" Width="200"/>
 </StackPanel>
<Slider Name="SL"/>
<WebBrowser Name="WB" Width="580" Height="300" Source="http://
www.trifox.com"/>
<Image Name="IM" Width="100" Height="100" Source="run.png"/>
</StackPanel
```

To set the FNAME, LNAME, and CB values, use

```
gui_canvas("CV","<DATA LNAME='Doe' FNAME='John' CB='1'/>");
```

and these values will appread in the respective canvas objects with the checkbox showing checked. To read the current values,

```
buf = gui_canvas(“CV”);
```

and buf will contain

```
<DATA LNAME=’Doe’ FNAME=’John’ CB=’1’/>
```

HTML5

A canvas object named CV has the following definition in the gap file:

```
<iframe id='WB1' src='http://www.trifox.com'></iframe>
<iframe id='WB2'></iframe>
<iframe id='WB3'></iframe>
<iframe id='WB4' src='http://www.aptean.com'></iframe>
```

To set the FNAME, LNAME, and CB values, use

```
gui_canvas("CV","<DATA WB2='http://www.expressen.se' WB3='http://
www.oracle.com'>");
```

and these values will appear in the respective canvas browser objects. To read the current values,

```
buf = gui_canvas("CV");
```

and buf will contain

```
<DATA WB1='http://www.trifox.com' WB2='http://www.expressen.se'
WB3='http://www.oracle.com' WD4='http://www.aptean.com'/>
```

As a shortcut, if you want to simply fill the canvas with a URL, you can just send

```
gui_canvas("CV","http://www.trifox.com");
```

gui_config

Sets GUI configuration options...

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_config(option,[,data])
int             option
expr            data
```

Description

The `option` parameter values are defined in `dv.h`. Valid values are::

.

gui_config_textlist Symbolic text list. `data` is required and is either a NULL list variable, e.g. `list ll; ll=NULL;`, to turn off symbolic text substitution or a two column list. The two column list has the symbolic text in the first column and its relacement text value in the second column. The list must be indexed on the first column. Symbolic texts are prefixed with "$", eg, "$LNAME". If "$LNAME" exists in the first column of the list, then it will be replaced by whatever value is in the second column, eg. "Lastname". If "$LNAME" does not exist, it is replaced by "LNAME". If the list is not set, no replacement is attempted and "$LNAME" is displayed.

Example

Switch between the original and replacement texts every time an event returns from input_r:

```
{                                   /* User window trigger      */
list LL;
int  i = true;

LL = list_open("10 10",0);

list_mod(LL,1,"AAAAA", "Not Found!!!");
list_mod(LL,1,"Email", "Electronic Mail");
list_mod(LL,1,"Tables","Tabular bells");
list_mod(LL,1,"Cancel","Fork");
```

```
list_index(LL,idx_cre_btree,0,0);

window(W2,open);
while (true) {               /* loop forever                    */
  input_r;
  if (G.key == key_quit) break;

  if (i) { i = false; gui_config(gui_config_textlist,LL);   }
  else   { i = true;  gui_config(gui_config_textlist,NULL); }

  window(W2,close);
  window(w2,open);
  }                          /* while loop forever              */
}
```

gui_grid

Send grid commands to the gui client..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_grid(cmd,parameter[,parameter...])
int           cmd
expr          parameter
```

Description

Sends commands to the current grid object. Commands are defined in dv.h:

.

Command	Description	Parameters
gui_grid_clear	Clears current grid contents	Number of null rows to initialize. 0 - original count.
gui_grid_delete	Deletes specified number of rows	Row index to start delete operation. Optional number of rows to delete (default 1).
gui_grid_freeze	Freeze column	Column number
gui_grid_header	Change column header	Column number, header
gui_grid_modify	Modify or insert a row of NULLs	Row index of affected row. < 0 Insert NULL row before Row index .= 0 Clear out Row index row. > 0 Insert NULL row after Row index
gui_grid_move	Move column	Column number, new column position
gui_grid_rgb	Set the cell color	String: row col color. If row or col are <0, then all rows or cols are affected. If row or col are greater than the row or column count, then the last row or column is affected. Refer to field_color for color definitions
gui_grid_rowoffset	Set the current grid offset	New row offset

gui_grid_save	Send grid data to clipboard and optionally to Excel or a file in CSV format	Include headers (True/False) [,Excel | Filename]
gui_grid_scroll	Scroll to the indicated position.	"Bottom" Scroll vertically to the end of the grid. Optional parameter 1 (non-zero) positions to the last row. "End" Scroll vertically to the end of the grid "Home" Scroll vertically to the beginning of the grid "Last" Scroll to the previously saved scrolling offset (client keeps the last scrolling offset automatically). "LeftEnd" Scroll horizontally to the beginning of the grid "RightEnd" Scroll horizontally to the end of the grid "Top" Scroll vertically to the beginning of the grid. Optional parameter 1 (non-zero) positions to the first row.
gui_grid_style	Style column	Column number, style name
gui_grid_unfreeze	Unfreeze column	Column number. Column is moved to the first unfrozen position. Column < 0 unfreezes all frozen columns.
gui_grid_width	Change column width	Column number, width. Width < 0 - auto sizing Width = 0 - hidden else size in pixels.

Notes

The column number is always the zero-based column number in the original grid. Frozen columns are appended from the left. The parameters may be comma

separated, e.g. `gui_grid(gui_header,1,"New Header")`, or in a string, e.g. `gui_grid(gui_header,"1 New Header")`.

Example

Freeze columns 3 and 6:

```
gui_grid(gui_grid_freeze,3);
gui_grid(gui_grid_freeze,6);
```

Columns 3 and 6 are now frozen on the left side of the grid.

Send grid data to Excel:

```
gui_grid(gui_grid_save,"Excel");
```

gui_id

Gets the type of gui client in use..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int gui_id()
```

Description

Returns an integer describing the gui client in use. The possible return values are defined in *dv.h* and are one of:

.

gui_id_activex MS Windows ActiveX

gui_id_jscript Javascript

gui_id_mswin MS Windows WIN32

gui_id_none none or not initialized yet

gui_id_silver Silverlight

gui_id_wpf WPF

Example

None

gui_info

Get information about the GUI environment..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int gui_info(option)
int          option
```

Description

Returns the GUI information requested based on the option value.:

.

option specifies the information to be returned (all sizes are in pixels) and is defined in *dv.h*:

gui_info_ansi_X Ansi X size

gui_info_ansi_Y Ansi Y size

gui_info_button_X Button X size

gui_info_button_Y Button Y size

gui_info_cell_X Cell X size

gui_info_cell_Y Cell Y size

gui_info_field_X Field font char X size

ui_info_field_Y gField font char Y size

gui_info_fixed_X Fixed X size

gui_info_fixed_Y Fixed Y size

gui_info_label_X Label X size

gui_info_label_Y Label Y size

gui_info_system_X System font char X size

gui_info_system_Y System font char Y size

gui_info_window_X Screen X size

gui_info_window_Y Screen Y size

gui_info_version Returns client version as a string

Notes

At least one GUI window must have been opened before calling *gui_info()*

gui_ipc

Communicate between spawned applications..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_ipc(name[,action[,option[,text]]])
string       name
int          action
int          option
```

Description

Communicates between spawned applications. The target application receives the event specified by `action`. Also used to set the current application's IPC name. Multiple applications can have the same name. .

name identifies the name of the target application. If `name` is the only parameter, then this sets the current application's IPC name. If `name` is NULL, then the action is sent to all other applications.

action is the action (0 - 96) to send to the target application(s). If `action` = (-1), then `pev_ipc` (86) is sent.

option sets the G.aux value for the target application(s).

text sets string for the target application(s), retrieved using `gui_info()`.

Notes

At least one GUI window must have been opened before calling gui_ipc.

gui_linesize

Set the line heights for the next window() open call..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_linesize(sizes)
string            sizes
```

Description

Sets the line heights for the next window() open. Sizes are in pixels and can be either absolute or relative values. The sizes are blank deliniated:

.

sizes specifies the line heights

Notes

Once the next window() open occurs, the line height values return to the original values.

Example

Change the line height of the first three lines to 10, +3, and -5 pixels.

```
gui_linesize("10 +3 -5");
```

This means that the first line will be 10 pixels, the second the cell height plus 3 pixels, and the third line the cell height minus 5 pixels. The rest of the lines in the window will be cell height. The values +/-0 can be used as fillers to skip lines.

gui_listen

Starts a listener process on the client machinel..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_listen(port)
int             port
```

Description

Starts a listener process on the client system. If a listerner process already exists, then this command has no effect.

.

port specifies the port number on which the listener will listen

gui_spawn

Starts a new application in an empty contained window..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_spawn(cmd)
string         cmd
```

Description

Starts a new application in an empty contained window based on the XAML in cmd, e.g. <DV><INI..../></DV>. The application is always started in

auto_start yes

mode regardless of any XAML setting to the contrary. `gui_spawn` must be called from the empty contained window's window or initialization trigger

gui_util

Send commands to the gui client..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void gui_util(cmd[,parameter])
int           cmd
expr          parameter
```

Description

Sends special commands to the DesignVision client. Commands are defined in dv.h:

Command	Description	Parameter
gui_util_call-parent	Calls the DVJScall function with the supplied parameter.	Parameter to pass to DVJScall().
gui_util_clr_keys	Clears the keystroke stack.	N/A
gui_util_clr_lastact	Clears the last stacked action.	N/A
gui_util_heartbeat	Sets the heartbeat timeout value.	Value in milliseconds
gui_util_help winname	Sets the help window name.	Window name
gui_util_kill_cookie	Removes the DVJS_COOKIE.	N/A
gui_util_killwin_prompt	Sets the kill window prompt. If set, then DVJS will open a prompt box when the user tried to close the window via Alt-F4 or the X. To reset, send the command with no parameter.	String to display in the prompt
gui_util_string	Sends a string to the evSendString handler (WPF)	String to send
gui_util_topmost	Sets or resets the window to be topmost (WPF)	True (1) or False (0)

gui_util_winicon	Sets the name of the image to display as the window icon	String with the imagename, either preloaded or a URI reference

gui_winattr

Returns or sets the GUI window attribute mask for the specified window..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
int gui_winattr(window_id[,attributes])
int             window_id
int             attributes
```

Description

The `window_id` parameter is the window sequence number which can be determined by using `window_seq(window_name)`. The `attributes` parameter is the new attribute mask for the window. These valus are defined in `dv.h`. The current attribute is always returned. The `gui_winattr()` call must be made before the target window is opened.

gui_winmod

Sets the GUI window dynamic attributes of the currently active window..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
int gui_winmod(xml)
string         xml
```

Description

The `xml` string is similar to the optional `xml` string in the `window` function however it only operates on the currently active window. The `xml` string format is

```
<WINDOW TOP='pos' LEFT='pos' WIDTH='size' HEIGHT='size'
UNITS='units' LKEYS='enum'>
```

where 'pos' is either 'center' where the window is centered on the screen, or 'n', 'size' is either 'auto' where the window adjusts to fit its contents, or 'n', and 'units' is either 'cell' or 'pixel' (default). If 'n' is prefixed with '-' or '+', then 'n' is a delta applied to the window's current setting for that value; otherwise it is an absolute value.

LKEYS sets the keys that will be treated as local keys for any lists in that window that have the Local Keys attribute set. The key names must match the WPF Key enum. These definitions are found here:

http://msdn.microsoft.com/en-us/library/system.windows.input.key.aspx

heapsize

Sets the miminum size of allocated heaps..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int heapsize(heapsize)
int          heapsize
```

Description

Sets the memory manager's minimum heap size, overriding the `heap_block_size` value in `dv.ini/trim.ini`. The new heapsize is returned. The current heapsize value is available using `sysinfo(sysinfo_heapsize)`. Every list object in TRIMpl has its own heap so an application with many lists and a large heapsize may waste memory. Conversely a small heapsize will result in many irregularly sized heaps in the free heap list.

input

Returns user input from screen field with validation.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void input([variable])
ident       variable
```

Description

Validates the data entered into the currently active field based on the field type and attributes. If the data is valid, it is placed in the field variable.

If the input field resides on a page that isn't the active page, the active page is changed.

If variable is not supplied, the application waits for an active key press.

variable (optional) contains data that has been input and validated. It must be char/string and at least as large as the field.

Notes

The alarm setting, `busy_alarm,` in `trim/dv.ini` helps you control a runaway query or looping function. For complete information, refer to `trim/dv.ini` documentation in the ***DesignVision Users Guide.***

Can only be used when in a field. An input(*variable*) call causes the following sequence of events:

1. G.MODIFIED is set to false.
2. Waits for user input; G.KEY holds the input-terminating key. If the user enters anything, G.INPUT_DATA is set to true.
3. any_key trigger is executed (if it exists).
4. raw_input data moves from temporary storage to *variable*.
5. G.INPUT_DONE is set to true.
6. If there is a pre-validation key trigger, execute key trigger (parm[0] is variable).
7. If G.INPUT_DONE is false, goes to step 1.
8. Validates NULL, TYPE, and MASK. Goes to step 1 whenever validation fails.
9. Converts and moves the data in *variable* to temporary storage (parm[0] in validate trigger).
10. If there is a foreign key trigger:
    - Executes foreign key trigger.
    - Goes to step 1 on failure (`error()` is invoked).

- If G.INPUT_DONE is false, goes to step 1.

11. If there is a validate trigger:
    - Executes validate trigger.
    - Goes to step 1 on failure (`error()` is invoked).
    - If G.INPUT_DONE is false, goes to step 1.
12. If G.MODIFIED is false, moves converted data from temporary storage to field variable. Set G.MODIFIED to true.
13. If there is a post-validation trigger, executes key trigger (parm[0] is variable).
14. If G.INPUT_DONE is false, goes to step 1.
15. Returns to caller.

Example

If get_data is true, then wait for input and do validation; otherwise simply wait for an event trigger (function key or mouse event).

```
{
if (get_data) input(input_var);
else input();
}
```

input_screen

Returns user input from all screen fields. Used for CGI applications.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void input_screen([validate[,form[,html_header[,html_leader
                  [,html_trailer]]]]])
int           validate
int           form
expr          html_header
expr          html_leader
expr          html_trailer
```

Description

Updates the HTML page with the current field values. Reads the values into the field_d variables and optionally runs the validation triggers for the fields. If the field values are valid, it moves them to the field variables.

validate (optional) If true, run the validation triggers for the fields. The default is false. Validate = true is similar to input(expr) whereas validate = false is similar to raw_input(expr).

form (optional) If true, then the HTML FORM from the file `runfile.window_id.html` is used for the screen, where `runfile` is the name of the application and `window_id` is the window ID. The HTML FORM file is generated by the **dvhtml** program. If false, then an internally generated FORM is used.

html_header (optional) This can be either a string with the name of the HTML file that contains the FORM page header or a list containing the HTML for the header. The default is NULL.

html_leader (optional) This can be either a string with the name of the HTML file that contains the FORM page leader or a list containing the HTML for the leader. The default is NULL.

html_trailer (optional) This can be either a string with the name of the HTML file that contains the FORM page trailer or a list containing the HTML for the trailer. The default is NULL.

Notes

The alarm setting, `busy_alarm,` in `dv.ini` helps you control a runaway query or looping function. For complete information, refer to `dv.ini` documentation in the ***DesignVision Users Guide*** or ***VORTEX Installation and Usage Guide.***

Can only be used when in a field. An input_scrren() call causes the following sequence of events:

1. G.MODIFIED is set to false.
2. G.INPUT_DATA is set to false.
3. HTML page is populated with actual data.
4. DVrun.cgi waits for user input (via DVnode).
5. All modified data is moved into field_d variables.
6. any_key trigger is executed if it exists.
7. If there is a pre-validation key trigger, execute key trigger (parm[0] is variable).
8. Validation triggers for all modified fields are executed if requested.
9. Post-validation key trigger is executed.
10. Returns to caller.

input_timer

Sets the `input()` timer's timeout value.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void input_timer(time)
int              time
```

Description

time specifies the timeout value in milliseconds. This value is in effect until the next `input_timer()` call.

Example

Sets the input timer's timeout value to 10 seconds.

```
input_timer(10000);
```

input_visual

Sets/gets the input field visual index.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int input_visual([vix])
int               vix
```

Description

vix (optional) specifies that the visual attribute index used to indicate the current input field is set to `vix`. If none is provided, then the function returns the field's current visual index. The vix default is 1, which typically indicates highlighting.

Specifying a vix of -1 turns off any changes for ''active'' fields.

The index's associated visual attributes are stored in `.key` files for each terminal type. (See *Key Mapping* in the ***DesignVision Users Guide*** for more information.)

Example

Changes the input visual attribute to blinking (see `trim.vis`).

```
input_visual(3);
```

insert_mode

Sets/gets insert mode.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
	X			

Syntax

```
int insert_mode([false|true])
keyword      false,true
```

Description

Sets insert mode if `true` or `false` are specified. If no parameter is given, current mode is returned.

true (optional) specifies that insert mode is turned on.

false (optional) specifies that insert mode is turned off (overwrite mode).

Example

None.

instr

Searches for a string within a string.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int instr(char1,char2[,n[,m]])
string    char1,char2
int       n,m
```

Description

char1 specifies the string in which to search.

char2 specifies the string to search for. The position of `char2` is relative to the first character of `char1`, even when `n > 1`.

n (optional) specifies the place at which to begin searching. If the value of the parameter is negative, the search begins from the back of the string.

m (optional) specifies the occurrence.

Notes

If `m` and/or `n` are not specified, 1 is assumed. If `char1` or `char2` is NULL, a NULL is returned. If `char2` does not exist in `char1`, 0 is returned.

Examples

Prints the location of two strings within a given string.

```
{
 char s[80], t[80], u[80];
 s = "This is an example for Trifox's TRIMpl";
 t = "Trifox";
 u = "No";
 printf("The location of Trifox within the string is " ^^
instr(s,t));
 printf("The location of No within the string is " ^^ instr(s,
u));
}
```

The following returns the position of the last backslash (/), in this case, 22. Note that the position is always returned from the beginning of the string and that it is 1-based.

```
instr("/usr2/rad/cheetah/lib/trim.ini", "/", -1)
```

key_exec

Executes a key trigger.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void key_exec(data)
int           data
```

Description

data specifies a key (from 0 - 63) to execute. If it is negative, the special [ANY] key is executed.

Example

Executes the key trigger associated with the [Enter].

```
key_exec(key_enter);
```

key_reset

Resets specific key's trigger code.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void key_reset(data)
int            data
```

Description

data specifies the key.

See `trim.h` for key values.

Example

Resets the **[Enter]** key to its original code (if any).

```
key_reset(key_enter);
```

key_set

Sets specific key's trigger code.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void key_set(data,trigger-code)
int          data
trigger      trigger-code
```

Description

data specifies the key.

trigger-code specifies the trigger to set.

See `trim.h` for key values.

Example

Sets **[Enter]** to ring a bell.

```
key_set(key_enter,{ bell(); });
```

key_type

Gets a key trigger's type.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int key_exec(data)
int          data
```

Description

data specifies the key (0 - 63) to examine. If the key is post-validate, the function returns true (nonzero), else it returns false (zero).

Example

None.

least

Gets the least (minimum) value in a set.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr least(data1,[data2[,...]])
expr       data1
```

Description

data specifies a items to compare. Mixed datatype data items are converted to the closest approximation. The resulting datatype depends on the datatype of the greatest item.

Example

Finds the earliest of several datetime values.

```
printf("The earliest date is " ^^ least(dt1,dt2,dt3));
```

length

Gets the current length of a variable.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int length (variable)
ident       variable
```

Example

Gets the length of the ID for the user executing the application or report.

```
{
.
.
userid = cuserid();
ulen   = length(userid);
.
.
}
```

list_close

Closes a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
list list_close(list-name)
list            list-name
```

Description

Decrements the list's reference count. If the reference count is 0 and `list_close()` is not in an expression, the list and all of its resources, such as memory, are released. If the function is part of an expression, the list is not freed even if the reference count is 0.

The reference count of a list is also decremented whenever a new value is assigned to it. Thus `list_close(list-name)` is equivalent to `list-name = NULL`.

Example

Assigns a list from one variable to another without creating a new list reference.

```
WL = list_close(LL);
WL = list_close(get_list());
```

Where `get_list()` is a user function that returns a list.

Returns a list from a user function without creating a new list reference.

```
{
list LL;
LL = list_open("SELECT name,id FROM staff",1000);
return(list_close(LL));
}
```

Closes and drops a list.

```
{
list LL;
LL = list_open("SELECT name,id FROM staff",1000);
list_view(LL,0);
list_close(LL);
}
```

list_colix

Gets the index of a lists's column.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_colix(list-name,col-name)
list           list-name
expr           col-name
```

Description

Returns column index; if the specified column is not found, returns -1.

list-name specifies the name of the list that contains the column to index. The list must have been created with a SELECT statement to have column names.

col-name specifies the column to index.

Example

Prints the index of the specified column in a given list.

```
{
 list xx, yy;

 xx = list_open("SELECT * FROM org", 1000);
 yy = list_open("20 16 10", 1000, "LMD");

 printf(list_colix(xx, "DEPTNUMB"));
 printf(list_colix(xx, "DEPTNAME"));
 printf(list_colix(yy, "DEPTNUMB"));
 list_close(xx);
 list_close(yy);
}
```

list_colnam

Gets the name of a column in a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string list_colnam(list-name,col-index[,col-name])
list               list-name
expr               col-index
string             col-name
```

Description

Returns the name of a column specified by index. If the specified list does not contain column names, `list_colnam()` returns a NULL. If `col-name` is specified, this function names the column and returns the new name.

list-name specifies the list to search.

col-index specifies the column to name. Lists created with a SELECT statement have column names associated with them.

col-name (optional) specifies a name for the column. If the column has a name, it is replaced.

Example

Prints the name of the specified column in a given list.

```
{
 list xx, yy;

 xx = list_open("SELECT * FROM org", 1000);
 yy = list_open("20 16 10", 1000, "LMD");

 printf(list_colnam(xx, 0));
 printf(list_colnam(xx, 1));
 if (list_colnam(yy, 1) == NULL)
   printf("No name exists");
 else printf(list_colnam(yy,1));
 list_close(xx);
 list_close(yy);
}
```

list_cols

Returns the column count of the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_cols(list-name)
list          list-name
```

Example

Prints the number of columns in the given list.

```
{
 list xx;

 xx = list_open("20 16 10", 1000, "RT List8");
 list_mod(xx, 1, "1", "2", "3");

 printf(list_cols(xx));
 list_close(xx);
}
```

list_colwid

Gets/sets the width of a list column.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_colwidth(list-name, col [,width])
list              list-name
int               col, width
```

DescriptionDescription

Gets or sets the width of a list column. Use the function to set widths for `list_view( )` and set the width in a list that is passed back via TRIMrpc.

list-name specifies the list to format.

col specifies the column (zero-based) to adjust.

width specifies the width in characters of specified column.

Example

Adjusts the fourth column to be 10 characters wide.

```
{
 list xx;

 xx = list_open(select * from staff", 1000, "RT List8");
 list_colwid(xx,3,10);
 list_view(xx,0);
}
```

list_copy

Copies a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
list list_copy(list-name)
list           list-name
```

Description

Returns a duplicate list.

list-name specifies the list to copy.

Example

Prints part of the original list and then the same part of the copied list.

```
{
 int i;
 list xx, yy;

 xx = list_open("20 16 10", 1000, "List A");
 list_mod(xx, 3, "1", "2", "3");
 yy = list_copy(xx);
 list_seek(xx,0);
 for (i=list_rows(xx);i;i--) {printf(list_read(xx,0));
                              printf(list_read(xx,2));
 list_seek(yy,0);
 for (i=list_rows(yy);i;i--) {printf(list_read(yy,0));
                              printf(list_read(yy,2));
 list_close(xx);
 list_close(yy);
}
```

list_copy2

Copies a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
list list_copy2(src-list,cnt | fnc[,col,...])
list            src-list
int             cnt
trigger         fnc
expr            col
```

Description

Returns a modified copy of `src-list`.

src-list specifies the list to copy.

cnt specifies the rows to be copied.
-1 all rows
0 no rows, only the list format
n n rows, beginning at the current item

fnc a function that receives `src-list` as parm.0 and returns `true` if the row is to be copied, `false` if it is not to be copied. If the function modifies the contents of the row, then the original row in `src-list` is also modified.

col a list of column numbers to be copied. Column numbers can be any expression that evaluates to either an integer or a char string containing space-delimited integers or any combination of these. Columns may be re-ordered or duplicated however the new list cannot have more columns than the original list

Notes

Any indexes on `src-list` will not be carried over to the new list.

Example

Copies columns 1 and 0 from all records where data in column 1 begins with either 'S' or 'M".

```
{
char re[80] = "^[SM]";
list LL,NL;

trigger want = { list_modcol(parm.0,0,1000*list_curr(parm.0,0));
```

```
                  return(regexp(regexp_exist,list_curr(parm.0,1)));
                };

regexp(regexp_init,re);

LL = list_open("select * from staff",99,"Regular expression: " ^^
re);

NL = list_copy2(LL,want,1,0);

list_view(NL,0);
}
```

list_curr

Reads a data item from a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr list_curr(list-name,col)
list           list-name
int            col
```

Description

Returns the value from the current item. Similar to `list_read();` however, this function does not advance the current item pointer.

list-name specifies the list.

col specifies the column.

Example

Prints the data item at the current item position.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "RTList");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");
 printf(list_curr(xx, 2));
 list_close(xx);
}
```

list_dup

Duplicates the current item in a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_dup(list-name)
list          list-name
```

Description

Duplicates an item, sets the item as current and sets its status to `item_insert`.

Example

Prints the original and duplicated current item and its status.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "Data List");
 list_mod(xx, 1, "1", "2", "3");

 printf("Current item: " ^^ list_curr(xx,0));
 printf("Current status: " ^^ list_stat(xx));
 list_dup(xx);
 printf("New item: " ^^ list_curr(xx,0));
 printf("New status: " ^^ list_stat(xx));
 list_close(xx);
}
```

list_edit

Invokes a list editor window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void list_edit(list-name,trigger-name [,menu-item1 [, ...]])
list           list-name
trigger        trigger-name
string         menu-item1
```

Description

This function invokes the windows list editor and displays the specified list from memory. It can only use data that is in memory; to display more data you need to have created a trigger that fetches it. .

list-name specifies the list to edit.

trigger-name (optional) specifies the user-written trigger to invoke for customized editing function control. `list_edit()` invokes the trigger with four parameters:

- parm[0] — A `list` that specifies the list to edit.
- parm[1] — An `int` that indicates one of the following event codes, depending on the end-user action:

When	Event Code	Callback Returns Value...
End user edits the data field.	list_edit_change	The trigger must return `char` data. If no modifications are made, the trigger retirns the contents of parm[3].
End user selects an item from the edit submenu.	list_edit_delete list_edit_insert list_edit_append list_edit_dup	-1 — Let `list_edit()` handle action. 0 — Nothing done. 1 — Trigger called and actions executed. Requests that `list_edit()` refresh the screen.
End user presses [PgDn] or [End] and `list_eos()` is false.	list_edit_more	0 — Nothing done. 1 — More data available.

When	Event Code	Callback Returns Value...
End user selects an item from User, as you have defined.	list_edit_user list_edit_user+1 …	Trigger must return `char` data for field update. If it returns `integer` data, then the following options can be used: -1 — Let list_edit() handle action. 0 — Nothing done. 1 — Trigger called and actions executed. Requests that `list_edit()` refresh the screen.

- parm[2] --- (optional) An `int` that specifies the current column.
- parm[3] --- `char` (optional) data from edit field (the text that is being modified).

menu_item1, menu_item2, … specifies the names of items you can specify in the default **User** in the menu bar. If you don't specify any items, then the menu item is a single callback event, `list_edit_user`.

If you define and list some menu_items, then the events are specified as `list_edit_user`, `list_edit_user+1`, etc. By returning data of different types (for example, `int` or `char`, you can control if a single field is to be updated or the whole grid, which in the **Edit** window is defined as 128 rows by 256 columns.

Edit menu items

If you don't specify a trigger, or if the trigger returns a -1 for any item on the \obj{Edit menu, `list_edit` uses its default behavior:

- list_edit_change —Updates field (subject to legal conversion)
- list_edit_insert — Adds blank row before current row
- list_edit_append— Adds blank row after current row
- list_edit_dup — Duplicates current row
- list_edit_delete — Deletes current row
- list_edit_more, — Nothing
- list_edit_user, list_edit_user+1, … — Nothing

Edit window behavior with lists

The **Edit window** can hold the ''maximum *grid size*,'' which is 128 rows by 256 columns. In this context a grid is a page's worth of data.

Because the scrollbar handling is local to the grid, scrollbar actions only move the cursor within the same 128x256 range. **Navigate** events, which end users can access from the menu or from function (or command) keys, also operate on grids, either within an

existing 128x256 range, or by fetching a new grid. For example, paging down actually displays 128 new rows.

Command	Action
[Enter]	Enters the edit field and invokes the callback trigger the `list_edit_change` option if the data has changed.
[Home]*	Moves the first page of the list into the grid with row 0, column 0, as the active position.
[PgUp]*	Moves a new page into the grid.
[PgDn]*	This function requires a trigger to handle the action. If you haven't specified a trigger, [PgDn] does nothing. Moves a new page into the grid. If not enough new rows exist in memory and the `list_eos()` is false then, it needs a callback trigger (that you have provided) with the `list_edit_more` option, to fetch more data into memory.
[End]*	The last page of the list is moved into the grid. If `list_eos()` is false then a trigger (that you have supplied) is invoked with the `list_edit_more` option. The last column of the last row becomes the active item.
[Tab]	Completes any edits and moves the cursor to the next column in the current row, making it active. It remains in a row and when it reaches the last column, wraps to the first column in the same row.
[Shift,Tab]	Completes any edits and moves the cursor to the previous column in the current row, making it active. It remains in a row and when it reaches the first column, wraps to the last column in the same row.
[Down]	Moves the cursor to the next row, making it active. When the end user reaches the last row of the page, it stops.
[Up]	Moves the cursor to the previous row, making it active. When the end user reaches the first row in the page, it stops.

* These commands are available from menu drop downs, as well as specified keys.

Example

This simple example prompts for a table name and then lets the user edit the list. Entering a empty table name terminate the application.

```
/*
** A simple table list editor
*/
{
list LL;
char ss[256];
trigger tt = { if (parm.1 == list_edit_change) return(parm.3);
           if (parm.1 == list_edit_delete) {
           list_mod(parm.0,0);
           return(1);
          }
          if (parm.1 == list_edit_more) {
```

```
            list_more(parm.0,1000,false);
            return(1);

          if (parm.1 == list_edit_user+0) return(upper(parm.3));
          if (parm.1 == list_edit_user+1) return(lower(parm.3));
          if (parm.1 == list_edit_user+2) {
          int i;
          i = to_int(prompt("How many rows so you want to delete?"));
          while (i > 0) { list_mod(parm.0,0); i--;
          return(1);
          }
        return(-1);
       };

while (true) {
     ss = prompt("Enter table name");
     if (ss == NULL) break;
     if (trap( { LL = list_open("select * from " ^^ ss,1000);  ))
        prompt(error_msg());
     else list_edit(LL,tt,"Uppercase","Lowercase","DeleteManyRows");
   }
}
```

list_eos

Checks list load status; closes cursor/file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_eos(list-name[,close])
list         list-name
int          close
```

Description

If the `list_open()`, `list_more()`, or `query()` call that loaded the list returns no more rows than the maximum specified in the call, then this funtion returns true (1), otherwise it returns false (0).

close (optional) can have one of two values. If it is true (nonzero), the underlying cursor or file is closed and the list remains open. Else the list is closed along with the cursor.

Example

Prints the list load status.

```
{
 list xx, yy;
 xx = list_open("20 16 10", 1000, "RT2");
 list_mod(xx,1,1,2,3);
 list_mod(xx,1,4,5,6);
 list_mod(xx,1,7,8,9);
 yy = list_open("SELECT * FROM staff", 1000);

 printf(list_eos(xx));
 printf(list_eos(yy));
 list_close(xx);
 list_close(yy);
}
```

list_file

Saves a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_file(list-name,file-name,file-type[,save-col,...])
list           list-name
string         file-name
string         file-type
int            save-col
```

Description

Saves a list to a specified location.

list-name specifies the list to save.

file-name specifies the name to save to. Whenever a file name is specified the extended file names are available.

For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

file-type specifies the type. Values can be:

- ***a*** — store as text. (Loses column structure. All columns are stored as one text line.)
- ***aa*** —append to text file.
- ***b*** — store as binary file. (Maintains list's column structure.)
- ***ba*** — append to binary file.
- ***s*** — store in shared memory.

save-col (optional) specifies the columns to display. Default is all. This option is identical to `view-col` in the `list_view()` function.

Notes

To use the shared list option, the `TRIM_SHM_FILE` environment variable must point to a valid `file-name`. Also, when using the shared list option, `save-col` is ignored; in other words, all columns are stored.

Example

Saves the parts list for later use.

```
list_file(xx,"parts.lst","b");
```

Prints part descriptions.

```
list_file(xx,"/dev/lp","a",1);
```

list_find

Searches a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr list_find(list-name,col,target[,col,target,...][,direction])
list           list-name
int            col
int            direction
expr           target
```

Description

Searches a list and sets item pointer to first position that meets criteria. Returns first `target`. Otherwise returns NULL.

list-name specifies list to search.

col specifies column(s).

target specifies criteria in specificed column.

direction (optional) specifies the direction from current position (starting with current position) in which to search.

- -1 — backwards.
- 1 — forwards.
- 0 — from first item in list. (default)

Example

Searches parts list for specific part.

```
{
if (list_find(xx,1,"crankshaft") == NULL)
 printf ("No crankshaft found in list");
}
```

list_get

Returns multiple columns from a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_get(list-name,variable[,variable,...])
list          list-name
ident         variable
```

Description

Reads the data columns from the current list position into `variable`.

Example

Loads the first three columns from the window list into local variables:

```
list_get(wl,rowid,name,salary);
```

list_index

Performs index operations on a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr list_index(list-name,option[[,index][,expr0[,...exprn]]])
list            list-name
int             options,index
expr            expr0,...exprn
```

Description

Manages indexes on lists and positions the current record based on a predicate. Both btree and hash indexes are supported. Column references are always zero-based.

list-name specifies the list to use.

option specifies what to do. It can be one of the following (defined in trim.h).

- idx_cre_btree - Create a btree index

`index` is the index number to create and `expr0[,...exprn]` are the columns to use. The number of unique key items is returned. If this number is the same as the number of rows in the list, then the index is unique.

```
list_index(LL,idx_cre_btree|idx_delete,0,1);
```

Deletes and creates a btree index on column 1.

- idx_cre_hash - Create a hash index.

`index` is the index number to create and `expr0[,...exprn]` are the columns to use. The item count for the hash bucket with the most chained items is returned. This is the worst case number of comparisons needed.

```
list_index(LL,idx_cre_hash|idx_delete,0,1,0);
```

Deletes and creates a hash index on columns 1 and 0.

- idx_delete - Delete an index.

This option may be OR'd (|) with any of the create options. No error is issued if the index does not exist.

- idx_uniquecnt - Return number of unique key items in index.

This value is always 0 for a hash index.

- idx_maxchain - Return item count for hash bucket with most chained items.

This value is always 0 for a btree index.

- idx_indexcnt - Return number of indexes in the list.

- idx_indexcols - Return the index columns.

The index columns are returnedas a string with blank separated integers.

- idx_pos_XX - Position the current item.

`index` is the index number to use and `expr0[,...exprn]` are the data values for the comparison. They must match the index columns, both in number and individual datatypes. No conversions are done during comparisons. Only the equal operator (idx_pos_eq) is allowed for hash indexes.0 is returned if the positioning failed. On success 1 is returned and the current item is set based on the operator as follows:

- idx_pos_eq - The actual item if it is unique. For non-unique indexes it is positioned on the first item for btrees. For non-unique hash indexes it is position on the first item in the bucket chain.
- idx_pos_lt - The closest item that is smaller
- idx_pos_leq - The closest item that is smaller if the item itself is not found. If the item is found then it is positioned at the last one (if non-unique).
- idx_pos_gt - The closest item that is greater.
- idx_pos_geq - The closest item that is greater if the item itself is not found. If found it is positioned as with idx_pos_eq.

Notes

When creating an index the regular sequence will always be set to the sequence of the particular index type created: sortd for btrees and hash function order for hash indexes.

When positioning within an index this order becomes the order used for all list functions such as list_seek(), etc.

The indexes are lost when the list is written into a regular file using list_copy() and list_file(). The indexes are preserved for list_copy() and list_file() into shared memory however this will use more memory when more than one index exists.

Hash indexes work best for unique integer keys.

Lists with indexes are read-only. Once the last index has been dropped the list becomes modifiable again.

Example

Create an index on two columns and search for a value:

```
      ll = list_open("select id,name,dept,job,years,salary,comm from
staff",10000);
      list_index(ll,idx_cre_btree,0,0,1);
      list_index(ll,idx_pos_eq,0,10,"SANDERS");
```

list_ixed

Reads a list value at an absolute position.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr list_ixed(list-name,row,col)
list           list-name
int            row,col
```

Description

Returns the data item from a list. Does not modify current item pointer.

list-name specifies the name of the list.

row specifies row number.

col specifies column number.

Example

Prints the data item located at the specified position.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "ListC");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");
 printf(list_ixed(xx, 1, 1));
 list_close(xx);
}
```

list_merge

Merges one list into another at a specified location.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
list list_merge(list1,list2,loc)
list            list1,list2
int             loc
```

Description

Returns the resulting (merged) list. Deletes `list2`.

list1 specifies the list to merge into.

list2 specifies the list to merge and delete.

loc specifies the point of merge relative to the current item:

- < 0 Before current item in list1.
- 0 After last item in list1.
- > 0 After current item in list1.

Example

Prints the first column of the original list, and then prints the first column of the merged list.

```
{
 int i;
 list xx, zz;
 xx = list_open("20 16 10", 1000, "List6");
 zz = list_open("20 16 10", 1000, "List7");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");
 list_mod(zz, 1, "7", "8", "9");

 list_seek(xx,0);
 for (i=list_rows;i;i--) printf(list_read(xx,0));
 xx = list_merge(xx, zz, 0);
 list_seek(xx,0);
 for (i=list_rows;i;i--) printf(list_read(xx,0));
 list_close(xx);
 list_close(zz);
}
```

list_mod

Updates, inserts, or deletes an item (including image (bitmap) for buttons or graphics lists) in a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_mod(list-name,action[,data-col,...])
list          list-name
int           action
expr          data-col
```

Description

list-name specifies list to modify

action specifies modification type:
- 0 — at current item.
- < 0 — before current item.
- > 0 — after current item.

data-col (optional) specifies the contents `action` modified. If no `data-col` is specified for 0 action deletes the current item and moves the current item pointer forward, if possible. Otherwise, 0 action updates the current item with the value(s).

If no `data-col` is specified for either < 0 or > 0 actions, the function inserts a row of NULLs for the current item.

Notes

For inserts, the current item pointer is set to the new item and its status is set to `item_insert`. For updates, the item status is set to `item_update`.

If the list is generated by a query and if the list is the window list and `list_mod()` updates the current item, the original value is stored if this has not already been done. `lock_row()` uses the stored version to compare to the current database value to determine if the row has already been modified. See the `lock_row()` function for details.

`list_mod()` is not allowed for lists in shared memory or if the list has multiple references (for example, two or more list variables point to the same list).

If the list is generated by a query, is a window list, and `list_mod()` updates the current item, it also stores the original value (creating a ''shadow row''). `lock_row()` uses the stored version to compare to the current database value to determine if the row has already been modified. See `lock_row()` for details.

`list_mod()` is not allowed for lists in shared memory or lists with multiple references (for example, two or more list variables point to the same list).

Example

Creates a list with parm[0] columns and inserts a row of NULL columns:

```
{list LL;                    /* The list  */
char def[160];               /* list_open define string */
int  i;                      /* Loop variable */
for(i=parm.0,def="";i;i--)   /* Through the columns */
  def = def ^^ "10";         /* Display width of column */
LL = list_open(def,0);       /* Create the list */
list_mod(LL,1);              /* Insert one row of NULL columns */
return(list_close(LL));     /* That's it, folks */
```

list_modcol

Modifies a single column in a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_modcol(list-name,col,data-col)
list             list-name
int              col
expr             data-col
```

Description

list-name specifies the list.

col specifies the column to modify.

data-col specifies the data to put into `col`.

Notes

If the list is a query-generated windows list and `list_mod()` updates the current item, the original value is stored. `lock_row()` uses the stored version to compare to the current database value to determine if the row has already been modified. See `lock_row()` for details.

`list_modcol()` is not allowed for lists in shared memory.

Example

Prints the data item that has been modified.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "A List");
 list_mod(xx, 1, "1", "2", "3");

 list_modcol(xx, 0, "2");
 printf(list_curr(xx,0));
 list_close(xx);
}
```

list_more

Fetches data for a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_more(list-name,limit,clear)
list           list-name
int            limit,clear
```

Description

list-name specifies the list to populate.

limit specifies the most rows that can be fetched.

clear can be true or false. If true (nonzero), the function overwrites existing data. If false (0), the function appends new data to the list's existing data and sets the current item pointer to the first row of the new set of data.

Example

Prints the first column of five rows, then fetches two more rows and appends that data to the list. Finally, it fetches two more rows and overwrites the previous data.

```
{
 list xx, yy;
 int i;
 xx = list_open("SELECT * FROM staff", 5);
 for (i=list_rows(xx);i;i--) printf(list_read(xx,0));

 list_more(xx, 2, false);
 list_seek(xx,0);
 for (i=list_rows(xx);i;i--) printf(list_read(xx,0));

 list_more(xx, 2, true);
 list_seek(xx,0);
 for (i=list_rows(xx);i;i--) printf(list_read(xx,0));
 list_close(xx);
}
```

list_next

Reposition to the next item in the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_next(list-name)
list          list-name
```

Description

Increments the current list pointer for the list, if possible. Returns the current position in the list.

Example

Useful as part of the default trigger code for the down-arrow key:

```
{
int rows,pos;
pos = list_rows(p.wl);         /* remember current position */
if (list_next(p.wl) != pos) { /* did we move? */
.
.
}
```

list_open

Opens a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
list list_open(spec,limit[,title[,...]])
expr           spec
int            limit
string         title
```

Description

Creates a list and loads it with rows.

The current item pointer is set to the first item in the list.

All items in the list are set to item_select status.

limit specifies the maximum number of rows that can be loaded into the new list. If `spec` is a SELECT, and this parameter is -1, the function executes a SQL DESCRIBE with the resulting list containing the column information.

spec can be a:

- ***SELECT statement*** — Must begin with the keyword SELECT (case insensitive.) `list_open()` executes the SELECT statement and loads the list with the data returned from the database.
- ***file name*** — Whenever a file name is specified the extended file names are available. For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***
- ***"s" followed by `list-name`*** — Considered a shared list. TRIMpl looks in the index file specified by the `TRIM_SHM_FILE` environment variable to find the memory address of the shared list identified by `list-name`. The list must have been previously filed as a shared list.
- ***Definition***— Begins with a digit. A definition consists of a list of numbers that represents the display width of the columns. No data is loaded.
- ***Control List*** — Can be edi, variable or fixed:

EDI Control List Format

Item	Content	Description	Example
0	filename [logfile]	Name of file to open; optional log file	myedi.txt"
1	edi	Indicates EDIFACT ISO ISO9735 format	edi
2	S	S - Default record separator character	‘

Variable Control List Format

Item	Content	Description	Example
0	filename [logfile]	Name of file to open; optional log file	"/tmp/test.exp mylog.txt"
1	variable	Indicates variable format	variable
2	S[E][N]	S, separator character E, optional enclosing character N, string indicating a NULL	,'NULL
3-*n*	D[F]	D, datatype (C, D, I ,N) see datatype() F, format mast	D YY/MM/DD

Fixed Control List Format

Item	Content	Description	Example
0	filename [logfile]	Name of file to open; optional log file	"/tmp/test.exp mylog.txt"
1	fixed	Indicates fixed format	fixed
2	[N]	N, string indicating a NULL	NULL
3-*n*	D O L[F]	D, datatype (C, D, I ,N) see datatype() O, column offset (0-based) L, column length F, format mast	N 0 10 99,999.99

title specifies the title that appears at the top of the list box during `list_view` operations.

In fixed format files or un-enclosed columns, leading and trailing blanks are stripped. If you specify a logfile then lines that do not conform to the specification are logged to this file. Otherwise an error is triggered.

All other cases are considered to be file names with the file being either a previously stored list or a flat text file. If it is a flat text file, then the list consists of one column.

Note that control lists can only have ***one*** column. You specify these lists like strings, enclosed by parentheses, and separate the elements with spaces.

Sybase

Sybase stored procedures can return a result list by using `list_open()`:

```
ll = list_open("select_procedure-name [&parm1 &parm2 ...]",...)
```

For example, staff2 is a stored procedure defined as:

```
create procedure staff2 @p1 integer as
SELECT * FROM staff WHERE id > @p1
```

To return the first 20 rows where id > 42:

```
int id;
id = 42;
ll = list_open("select_staff2 &id",20);
```

Examples

Executes a DESCRIBE on the table.

```
xx = list_open("SELECT * FROM " ^^ tname,-1,tname);
```

Loads a list from the database.

```
xx = list_open("SELECT * FROM emp",1000,"All employees");
```

Defines a list.

```
xx = list_open("20 16 10",1000,"Beers of the world");
```

Loads a list from a file.

```
xx = list_open("/etc/passwd",1000,"Users","(from passwd file)");
```

Loads a previously stored shared list.

```
xx = list_open("s zipcode",1000,"Zip Codes of USA");
```

The following example loads a delimited file into a list. This example is an actual working application that uses an FTP logfile, loads it into a list where it is processed for database entry.

```
int    i;
  list     CL, LL, NL;
  datetime dt;
  char     buf[80], host [160];

/* building the spec  */

  list_mod(CL, 1, ''xferlog'');
  list_mod(CL, 1, ''variable'');
```

```
  list_mod(CL, 1, `` ``);
  list_mod(CL, 1, `` ``);
  list_mod(CL, 1, ``C'');          /* 0 -- day of week */
  list_mod(CL, 1, ``C'');          /* 1 -- month       */
  list_mod(CL, 1, ``C'');          /* 2 -- day         */
  list_mod(CL, 1, ``C'');          /* 3 -- time        */
  list_mod(CL, 1, ``C'');          /* 4 -- year        */
  list_mod(CL, 1, ``N'');          /* 5 -- transfer time */
  list_mod(CL, 1, ``C'');          /* 6 -- host        */
  list_mod(CL, 1, ``N'');          /* 7 -- file size   */
  list_mod(CL, 1, ``C'');          /* 8 -- file name   */
  list_mod(CL, 1, ``C'');          /* 9 -- ascii/binary transfer
*/
  list_mod(CL, 1, ``C'');          /* 10 - special action */
  list_mod(CL, 1, ``C'');          /* 11 - direction    */
  list_mod(CL, 1, ``C'');          /* 12 - access mode  */
  list_mod(CL, 1, ``C'');          /* 13 - user name    */
  list_mod(CL, 1, ``C'');          /* 14 - service name */
  list_mod(CL, 1, ``C'');          /* 15 - auth method  */
  list_mod(CL, 1, ``C'');          /* 16 - auth ID      */


/*  opening the list using the spec */

LL = list_open(CL, 10000000);

/* now process the list */
```

list_pos

Gets current position in a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_pos(list-name)
list         list-name
```

Description

Returns the 0-origin based current position in the list. If the list is empty, returns -1.

Example

Prints the current positions of the specified items.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "List1");
 printf(list_pos(xx));

 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");
 printf(list_pos(xx));
 list_close(xx);
}
```

list_prev

Repositions pointer to the previous item in the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_prev(list-name)
list          list-name
```

Description

Decrements the current list pointer for the specified list if possible.

Example

Useful as part of the trigger code for the up-arrow key:

```
{
if (list_pos(p.w1) != list_prev(p.w1)  /* did we move? */
.
.
}
```

list_read

Reads from a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr list_read(list-name,col)
list           list-name
int            col
```

Description

Returns the column of the current position and advances the current item pointer.

Example

Prints the data items specified to be read.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "Media Report");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");

 printf(list_read(xx, 0));
 printf(list_read(xx, 1));
 printf(list_read(xx, 2));
 list_close(xx);
}
```

list_refcnt

Returns the number of references to the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr list_refcnt(list-name)
list              list-name
```

Description

Returns the number fo references to the list.

Example

Check for multiply referenced lists:

```
#trigger list_check
{
 int i;
 list_seek(parm.0,0);
 while(true) {
   for (i=0;i<list_cols(parm.0);i++) {
     if (datatype(list_curr(parm.0,i)) == "L") {
       if (list_refcnt(list_curr(parm.0,i)) > 1)
         printf("List: "^^parm.1^^", row: "^^
                list_pos(parm.0)^^" col: "^^
                i^^", the list item has refcnt > 1");
       list_check(list_curr(parm.0,i),list_pos(parm.0)^^":"^^i);
       }
    }
  }
}
```

list_rows

Gets the row count of the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_rows(list-name)
list          list-name
```

Example

Prints the number of rows in the list.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "Data Report");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");

 printf(list_rows(xx));
 list_close(xx);
}
```

list_seek

Moves the current item pointer in a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_seek(list-name,row)
list           list-name
int            row
```

Description

Positions the current item pointer to row.

Example

Seeks a position and prints from it.

```
{
 list xx;
 int i;
 xx = list_open("20 16 10", 1000, "RT1");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");

 list_seek(xx,0);
 for (i=list_rows;i;i--) printf(list_read(xx,0));
 list_close(xx);
}
```

list_sort

Sorts a list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void list_sort(list-name,col[,col,...],direction)
list           list-name
int            col,direction
```

Description

Sorts a list; leaves current list item unchanged.

list-name specifies the list to sort.

col specifies the column on which to sort. If multiple columns are specified, the sort is performed on all the columns in the order specified.

direction specifies sort order:
- 1 — Ascending order.
- 2 — Descending order.

Example

Sorts a given list by the specified column and order.

```
{
 list xx;
 int i;
 xx = list_open("20 16 10", 1000, "Improvements");
 list_mod(xx, 1, 5, 10, 15);
 list_mod(xx, 1, 15, 10, 5);

 printf("col 2, asc\n");
 list_sort(xx, 2, 1);
 list_seek(xx,0);
 for (i=list_rows(xx);i;i--) printf(list_read(xx,2));

 printf("col 0, des\n");
 list_sort(xx, 0, 0);
 list_seek(xx,0);
 for (i=list_rows(xx);i;i--) printf(list_read(xx,0));
 list_close(xx);
}
```

list_stat

Sets/gets the status of the current item in the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_stat(list-name[,status])
list          list-name
int           status
```

Description

If status is provided, updates current item's status before returning.

list-name specifies the list in which current item resides. If it is WL and the new status is `item_select`, the item's saved row is reset.

status specifies a new current item status. The possible values for status are listed in `trim.h`:

- item_delete 0
- item_select 1
- item_update 2
- item_insert 3
- item_tagged 4

Notes

See `lock_row()` for a description of the use of the saved row.

The following TRIMpl functions look at or modify the item status internally:

list_dup() New item status set to item_insert.

list_mod() Update: status set to item_update; Insert: status set to item_insert.

list_modcol() Status set to item_update.

list_view2() Chosen item status set to item_tagged.

move_f2l() Move field values to window list and set status to item_update if not already item_insert.

Example

Prints the current item's status.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "RT 20");
 list_mod(xx, 1, "1", "2", "3");
```

```
 list_mod(xx, 1, "4", "5", "6");
 printf("Current item: " ^^ list_curr(xx,0));
 printf("Status: " ^^ list_stat(xx));
 list_close(xx);
}
```

list_sync

Resynchronizes the displayed list with the latest list position.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				X

Syntax

```
void list_sync(list-name)
list           list-name
```

Description

If the list is currently displayed, updates the display position with the current list item position.

Example

Repositions highlighted list item.

```
list_seek(ll,42);
list_sync(ll);
```

list_treeview

Displays the contents of a specified list in treeview format.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
expr list_treeview(list-name,row,col,height,width)
list               list-name
int                row,col,height,width
```

Description

list_name specifies the list.

row specifies the row position of the view box. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

col specifies the column location of the view box. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

height specifies the height for the displayed window in cells.

width specifies the width for the displayed window in cells.

Notes

The `list-name` is a two-column list. The first column is the description and the second column is either the value to be returned for that leaf node or another two-column list. `list_treeview` returns NULL if either Cancel, Close Window, or OK without a selection is chosen. Otherwise it returns the integer value that is stored in that leaf node's second column.

Example

```
{
int  i,rc;
list LL,L2;

LL = list_open("20 6",0,"A brand new title","Line one","Second
line");
L2 = list_open("20 6",0);

list_mod(L2,1,"One","1");
list_mod(L2,1,"Two",2);
```

```
list_mod(L2,1,"Three","3");

list_mod(LL,1,"Level",L2);
list_mod(LL,1,"Leaf","99");

while (true) {
  rc = list_treeview(LL,-2,-2,8,20);
  if (rc == NULL) break;
  prompt(rc);
  }
}
```

Load a list with departments and the employees who work in each department.

```
{
int  i,rc;
list LL,L2;

LL = list_open("select deptname,deptnumb from org order by
1",999);

for (i=list_rows(LL);i;i--) {
  L2 = list_open("select name,id from staff "
                 "where dept = &/list_curr(LL,1)/ order by 1",
999);
  list_modcol(LL,1,L2);
  list_next(LL);
  }

prompt(list_treeview(LL,-2,-2,10,40));
}
```

list_view

Displays the contents of a specified list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
expr list_view(list-name,ret-col[,view-col,...])
list           list-name
int            ret-col,view-col
```

Description

list-name specifies the list.

ret-col is the absolute zero-based column position of the data to be returned.

view-col (optional) specifies which columns to display. Default behavior displays all columns.

Notes

The `list_view()` box has basic scrolling capabilities. End users select an item by moving the cursor to the desired item's position, using the arrow, **[PgUp/PgDn]**, and **[Home] [End]** keys, and pressing **[Enter]**. The chosen position becomes the current item pointer.

This function restricts end users to one column for which to return a value. See function descriptions for more information.

Example

Loads a list with part codes and descriptions; then, prompts the user to select a part based on the description and return its code.

```
xx   = list_open("SELECT code,description FROM
parts",1000,"Parts");
code = list_view(xx,0,1);
```

list_view2

Displays the contents of a list, allowing multiple items to be tagged.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
void list_view2(list-name,row,col,size,exit-key,[,view-col,...])
list            list-name
int             row,col,size,view-col,exit-key
```

Description

list-name specifies the list to display.

row specifies the row position for the displayed window. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

col specifies the column position for the displayed window. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

size specifies the number of rows to display. -1 indicates that the function include as many rows as fit on the display.

exit-key specifies the key that exits the view.

view-col specifies the columns to display. Default behavior is to display all columns.

Notes

In addition to the `list_view()` selection ability, this function allows you to specify the view box location and the number of rows to display, and choose an exit key for the application.

The item's status toggles between `item_tagged` and `item_select` and the last tagged item becomes the current item pointer.

`list_view3()` adds an abort option, lets you define display options, and lets you choose the column to return.

Example

Creates a delete utility that prompts for file names:

```
{
 list files;
  char fn[80];
  int  cnt;

  fn = tmpnam();
  system("ls > " ^^ fn);
```

```
  files = list_open(fn,1000,"Hit Enter to tag a file for delete");
  list_view2(files,5,5,10,key_f3);
  cnt = list_rows(files);
  list_seek(files,0);
  while (cnt) {
    if (list_stat(files) == item_tagged)
delete(list_curr(files,0));
    list_next(files);
    cnt--;

  delete(fn);
  }
```

list_view3

Displays the contents of the specified list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
expr list_view3(list-name,row,col,size,abort-key,
                          options,ret-col[,view-col,...])
list            list-name
int             row,col,size,view-col,abort-key,options,ret-col
```

Description

Returns the value in `ret-col` for the selected item.

item-name specifies the list.

row specifies the row position for the displayed window. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

col specifies the column position for the displayed window. -1 places the upper left of the box at the cursor position. -2 places the box in the center of the screen.

size specifies the number of rows to display; if it is -1, the function displays as many rows as fit.

view-col (optional) specifies the columns to display; By default, all columns are displayed.

abort-key if greater than zero, specifies the key that can be used to cancel the function's operation (returns NULL).

options define how the list is to be displayed. See ***DesignVisionUsers Guide*** for information on options.

ret-col the absolute zero-based column position of the data to be returned.

Notes

`list_view3()` includes the same scrolling capabilities as `list_view2()` and `list_view()` with the addition of an ''abort'' feature. If the end user cancels the operation, this function restores the current item.

In addition, you can define display options for your end user and specify the column to return.

Example

Creates a file-editing prompting utility:

```
list    files;
  string  fn[80];
  string  fn2[80];
  int     cnt;

  fn = tmpnam();
  system("ls > " ^^ fn);
  files = list_open(fn,1000,"Hit Enter to edit a file, F3 to
quit");
  while (true) {
    fn2 = list_view3(files,5,5,10,key_f3,opt_highlight,0);
    if (fn2 == NULL) break;
    system("vi " ^^ fn2);

  delete(fn); }
```

list_vis

Gets the number of visible rows in the list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int list_vis(list-name)
list         list-name
```

Description

Returns the number of rows whose status is not set to item_delete and remain ''visible.''

Example

Prints number of visible rows in a given list.

```
{
 list xx;
 xx = list_open("20 16 10", 1000, "New Data");
 list_mod(xx, 1, "1", "2", "3");
 list_mod(xx, 1, "4", "5", "6");
 list_mod(xx, 1, "7", "8", "9");
 printf(list_vis(xx));
 list_close(xx);
}
```

lock_row

Locks the current window list row for update in the database.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int lock_row(table-name,row-id,update[,original])
ident        table-name
expr         row-id
int          update
list         original
```

Description

Locks the specified row. Returns status as:

- 0 — Lock succeeded.
- 1 — Could not acquire lock.
- 2 — Database row has been modified since it was read into the window list.

table-name specifies the database table.

row-id specifies the row in the database table.

update If update is true, then the item's original value (previously saved) is compared to the current value in the database. If these values do not match, the following status is returned:

```
2
```

original (optional) must be a list in which the current database values for columns without the NOUPDATE attribute are placed.

The `lock_row()` function uses these values internally to compare to the shadow values.

Notes

TRIMpl uses optimistic locking, only locking a row when explicitly required and not when the corresponding item is updated.

Example

Deletes all rows in the delete list:

```
{
.
for (i=list_rows(DL);i;i-,list_next(DL)) {
  lock = lock_row(EMP,list_curr(DL,0),false);
  if (lock := 0) g.msg = "Could not acquire lock.";
  else
    exec_sql("DELETE FROM emp WHERE rowid = " ^^ list_curr(DL,0));
  }.}
```

log

Writes text to a file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string log(file-name,text)
string     file-name,text
```

Description

Writes text to logfile. Flushes file after each write so no data is lost by a system crash or similar disaster.

file-name specifies the name of the logfile. Whenever a file name is specified the extended file names are available.

For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

text specifies the text to append to the file.

Example

Logs two messages to a specified file and print the messages.

```
{
 list xx;
 log ("ttt", "whoa");
 log ("ttt", "nelly");
 xx = list_open("ttt", 1000);
 printf(list_read(xx,0));
 list_seek(xx,1);
 printf(list_read(xx,0));
}
```

ltrim

Trims blanks to the left of a char string.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string ltrim (string[,charset])
string        string, charset
```

Description

This function strips the chars specified in `charset` from `string` until it reaches a character not specified in `charset`. If no char is specified, the function assumes blanks.

Example

The following strips blanks from the left of the string located in buffer parm[3].

```
buf = ltrim(parm[3])
```

mask_chk

Validates format mask.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int mask_chk(datatype,mask)
char         datatype,mask
```

Description

Checks if the format mask is valid for the given datatype.

Returns 1 if mask is valid, else 0.

datatype can be:

- C — Character
- D — Datetime
- I — Integer
- N — Numeric

mask specifies the mask definition (see ***DesignVision Users Guide***.)

Notes

The format mask for char is case-sensitive.

Example

Prompts the user for the format mask and validate it before using:

```
{
char mask[40];
mask = prompt("Please enter the format mask ==> ");
while (mask_chk(datatype(parm.0),mask) == 0)
  printf("Mask is invalid for " ^^ datatype(parm.0) ^^ "
datatype.");
.
.
}
```

max

Gets the maximum (larger) of two expressions.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr max(data1,data2)
expr     data1, data2
```

Example

Returns the maximum commission.

```
{
numeric max_comm = 0;
.
.
max_comm = max(max_comm,COMM);
.
.
}
```

message

Writes a message to a window message box.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
			X	X

Syntax

```
string message(msg)
string         msg
```

Description

Writes specified text to the window message box.

msg specifies the text to write.

If no message box is open, msg is displayed in a dialog box and the user must acknowledge the message to continue.

Example

Indicates the number of database rows modified.

```
i = exec_sql("update staff set salary = salary *
             .75 where salary > :1",limit);
message(i^^ "rows updated");
```

min

Gets the minimum (smaller) of two expressions.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr min(data1,data2)
expr     data1,data2
```

Example

Returns the minimum salary.

```
{
numeric sal_min = 0;
.
.
sal_min = min(sal_min,SAL);
.
.
}
```

move_f2l

Moves data from the window fields to the window list.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X			X	

Syntax

```
void move_f2l(action)
int           action
```

Description

Copies the data from all the fields with list attribute set into the current window list based on the value of action.

action specifies the action to take and can be:

- 0 — Update current item with the field data values. If the list is empty, a list row is created with the field data values.
- < 0 — Insert field data values before the current item.
- > 0 — Insert field data values after the current item.

Notes

For INSERT, the window list current item pointer is set to the new item and its status is set to item-insert. For UPDATE, the item status is set to `item_update`.

Example

Updates the current record in the window list and moves to the next record in the list:

```
{
move_f2l(0);
list_next(p.wl);
}
```

move_l2f

Moves data from the window list to the window fields.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void move_l2f()
```

Description

Copies the data from the current window list to the window fields.

Example

Moves to the next item in the window list and updates the window fields:

```
{
list_next(p.wl);
move_l2f();
}
```

name_in

Gets the contents of a variable.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr name_in(varname)
string       varname
```

Description

varname specifies the variable; it must evaluate to a fully-qualified variable name.

Notes

Specify -s with TRIMgen to include the symbol table in the runtree.

Example

Prints the content of the specified variable.

```
{
 char varname[4];
 datetime dt;
 dt = "15-JUN-75";
 varname = "g.dt";
 printf("Variable content is " ^^ name_in(varname));
}
```

nvl

Returns null value.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr nvl(data,null-value)
expr     data,null-value
```

Description

data specifies the expression to evaluate.

null-value specifies what to return if `data` evaluates to NULL.

Example

Prints ‘‘N/A” for null values.

```
printf("Salary = " ^^ nvl(SALARY,"N/A"));
```

open

Opens a specified report output file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
void open([file-name])
string     file-name
```

Description

file-name (optional) specifies a filename for the report output. If it is not supplied, the output file is set to **`run-file`**`.OUT`. If file-name is an empty string, the output file is set to STDOUT.

For more information about specifying files, see *Filename Specifications* in the ***DesignVision Users Guide.***

Example

Part of the default MAIN trigger:

```
{
open();
paginate(header);
.
.
}
```

os_id

Returns the operating system ID.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int os_id()
```

Description

Returns the OS (Operating System) ID as follows (from `trim.h`):

```
#define  os_unix       0     /* All Unix derivatives          */
#define  os_windows    1     /* Windows NT/9x                 */
#define  os_mvs        2     /* MVS (OS/390)                  */
#define  os_vms        3     /* OpenVMS                       */
```

Example

The following prints out the computer's operating system.

```
{
prompt ( decode(os_id(), os_unix, "Unix",
                         os_windows, "Windows",
                         os_mvs, "MVS",
                         os_vms, "VMS",
                         "undefined"));
}
```

overlay

Overlays the current application with the specified application.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void overlay(run-file[,parm,...])
string       run-file
expr         parm
```

Description

Replaces the calling application with a specified one. Returns nothing because the calling application is replaced with the new one.

run-file specifies the application to overlay. Unless runpath is specified in `trim.ini`, you must give the full path name. Specifying the `.run` extention is optional.

parm (optional) specify parameters of the overlaying function. These parameters can be accessed in the main trigger of the application.

Notes

`overlay()` is similar to `call()` and is useful in applications where the return path does not have to be preserved, such as a multi-level menu system. `overlay()` can also help in memory-constrained environments.

Example

Invokes an application and passes the current time as a parameter.

```
overlay("app1.run",g.time);
```

paginate

Creates a page break.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
	X			X

Syntax

```
void paginate( [header] [| break] [| footer]  )
keyword     header,break,footer
```

Description

Creates the following page break for each keyword:

footer Appends the current block's page footer (if any).

break Writes out the report page, clears report page, resets line count (G.LINENUMBER = G.PAGEHEADERLINES) and increases page number (G.PAGENUMBER = G.PAGENUMBER + 1).

header Inserts the current block's page header (if any).

Notes

If no block is currently active (for example, the main trigger), then the first block's page header is printed. If a different header is required, remove paginate(header) from the main trigger and put it in the PRE-BLOCK trigger of the first block.

Example

By default the page trigger contains the single statement:

```
paginate(footer|break|header);
```

For special effects, the page trigger can be changed to perform actions such as resetting offsets.

popup_menu

Manages floating popup menus.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void popup_menu(command[,data,...])
int             command
expr            data
```

Description

command specifies the action. Its value can be:

- 0 — Loads the popup menu from a list. In this case, data must be a list. Only the first column of the list is used and it must char with one of the two following formats:

  < menu item label > is a simple popup menu is loaded where the list's sequence becomes the menu item ID.

  < ID > <level > <menu item label > is used for more complex popup menus where the main menu is level 0.

  The popup menu is displayed by either hitting the right mouse button in a non-child area of the window or by calling popup_menu(1).

  When an item is picked from the menu the pre-defined event trigger is invoked and the menu ID is placed in `G.aux`.

- 1— Display the popup menu.
- 2 — Attach the popup menu to the windows menu bar. `data` must be a numeric expression indicating where to attach the popup menu. For example if this parameter is *n*, then the popup menu is attached to the *n*th item on the window menu bar. *n* is zero based. When attached the popup menu is still available via the right mouse button or via popup_menu(1).
- 3 — Set popup menu item attribute. The parameters are pairs of ID and flags where flags (found in dv.h) are the same values as used when setting other fixed actions flags.
- 4 — Delete the popup menu.

Example

```
{
list LL;
list_mod(LL,1,"Item One");        /* Load the list                 */
list_mod(LL,1,"Item Two");
list_mod(LL,1,"Item Three");
popup_menu(popup_load,LL);        /* load the popup menu from list  */
popup_menu(popup_attach,3);       /* attach popup menu the 4th item */
                                  /* on the menu bar                */
popup_menu(popup_flag,0,
                flag_tagged,      /* place check mark on "Item One" */
                2,flag_disable); /* disable(gray) "Item Three"     */
}
```

postmessage

Posts a message to window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				X

Syntax

```
void postmessage(hwnd,msg,wparam,lparam)
int              hwnd,msg,wparam,lparam
```

Description

Passes the parameters to the MS Windows PostMessage() function. Use this function at your own risk.

Example

Terminate all MS Windows: (not recommended)

```
postmessage(-1,18,0,0);
```

power

Returns *m* to the power of *n*.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				X

Syntax

```
power (m,n)
numeric    m
int        n
```

Description

Returns *m* to the power of *n*. The base and exponent *n* can be any number but *n* is always truncated to an integer before calculation.

Example

None.

printf

Prints to display.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
string printf (message)
string         message
```

Description

Writes message to STDOUT and returns message.

Example

Print names and salaries of employees whose salary > 50000.

```
if (SAL > 50000) printf(NAME ^^ " makes " ^^ sal);
```

prompt

Prompts for user input.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
string prompt(message)
string        message
```

Description

After message has been displayed on STDOUT, the program waits until the user responds by entering data and pressing **[Enter]**. The user input is returned.

Notes

If prompt is used within a TRIMapp application, the message and prompt area are placed within a box.

Example

Prompts the user for a response.

```
field = prompt("Please enter your name ==> ");
```

prompt2

Creates a box for prompt text.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X		X

Syntax

```
string prompt2(row,col,text[,text...],field-text,len,visual,box)
int            row,col,len,visual,box
string         text,field-text
```

Description

Returns user input.

row specifies the row position for the displayed window. -1 places the top of the box at the cursor position. -2 places the box in the center of the screen.

col specifies column location of the view box; -1 places the left corner of the box at the cursor position. -2 places the box in the center of the screen.

text (optional) specifies one or more text lines.

field-text specifies text to be placed in field.

len specifies the length of text field.

visual specifies visual attribute of field.

box allows you to draw a box around prompt. If this value is true (nonzero), the function draws a box.

Notes

The number of text lines and the last `text` plus `len` determine the box size. The field is placed after the last text line. Keys 0 and 3 return except if `len` is 1, in which case all keys return.

The prompt window is always closed on exit. Use `refresh(false)` to give the appearance of remaining open if needed for error handling.

pset

Sets printer code.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
void pset(row,column,code-name | code)
int       row,column
ident     code-name
string    code
```

Description

Sets a printer escape sequence at the given row and column coordinate within the report page.

row specifies row coordinate. If negative, uses G.LINENUMBER.

column specifies column coordinate. If negative, uses G.PAGEOFFSET.

code-name specifies the escape sequence by symbolic name. A hex string must be prefixed with 0x. The symbolic name must be in the printer control code file (default: `trim.cc`). The following example shows the format of the file:

```
BOLD_ON  "0x1b47"
BOLD_OFF "0x1b48"
```

code specifies the escape sequence by code. The escape sequence can be a C string. A hex string must be prefixed with 0x.

Example

Turns on bold after page header and turn off bold before page footer in the paginate trigger.

```
{
.
.
pset(G.PAGEHEADERLINES,0,"0x1b47");
pset(G.PAGELENGTH-G.PAGEFOOTERLINES,0,BOLD_OFF);
}
```

putenv

Sets an operating system environment variable.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void putenv(env-pair)
string      env-pair
```

Description

Sets or gets one or more operating system environment variable(s). The env-pair variable is a double-quoted string of an environment name and its value. The environment variable change is not persistent. That is, it is only set for the session

env-pair Specifies the value(s) to set and its variable(s).

Example

Sets the environment variable PRINTER to lpt300.

```
putenv("PRINTER=lpt300");
```

Sets multiple environment variables: the printer as well as paper size.

```
putenv("PRINTER=lpt3001,PAPER=letter");
```

See also

"*getenv*" on page 97

query

Loads the window list (WL) using the window SELECT statement.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void query(data[,add-where[,add-end]])
int        data
string     add-where,add-end
```

Description

Fetches a maximum of data tuples. To fetch more tuples, use the `list_more()` function.

data specifies the maximum number of tuples to fetch. If data = -1, the function performs a describe on the window SELECT statement and puts the results in the window list.

If the window has the default triggers attribute, then the appropriate row identifier column becomes the first column of the window SELECT list. For example, in Oracle, A.row-id becomes the first column of the window list.

add-where (optional) is appended to the `#where_begin #where_end` sequence; WHERE and AND are inserted where needed. If the query does not have the #where_begin/#where_end tags, this parameter is ignored.

add-end (optional) is appended to the SELECT statement; no blank is inserted before add-end.

Example

Retrieves up to 1,000 tuples as specified in the

SELECT statement:

```
query(1000,"salary BETWEEN 10000 AND 20000","ORDER BY salary
DESC");
```

See the default window trigger in ***DesignVision Users Guide***.

query_count

Gets the number of qualifying tuples in window SELECT statement.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
numeric query_count([add-where[,add-end]])
string           add-where,add-end
```

Description

Returns the number of qualifying tuples in window SELECT.

add-where (optional) is appended to the `# where_begin # where_end` sequence; WHERE and AND are inserted where needed.

add-end (optional) is appended to the SELECT statement; no blank is inserted before add-end.

Example

Prompts user for confirmation before fetching data:

```
{
numeric cnt;
char    answer[1];
cnt    = query_count();
answer = prompt("Query returns " ^^ cnt ^^ " tuples.  OK(Y/N)?");
if (answer in ("Y","y")) query(cnt);
 }
```

raw_input

Gets user input from screen field without validating the data.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void raw_input([variable])
ident           variable
```

Description

Accepts data entered into the currently active field by the user. It does not validate the data and the data is not moved to the field. Can be used anywhere on the screen.

variable (optional) specifies the variable in which to put the user data. The variable must be a char/string and at least as large as the field. If no variable is specified, the application waits for an active key press.

If the input field resides on a page that isn't the active page, the active page is changed.

Notes

The alarm setting, `busy_alarm,` in `trim/dv.ini` helps you control a runaway query or looping function. For complete information, refer to `trim/dv.ini` documentation in the ***DesignVision Users Guide*** or ***Trifox Resource Manual.***

Example

Wait for user to fill in query value in field.

```
{
raw_input(input_var);
}
```

record_exec

Executes the window's record trigger.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
ret_value record_exec([parm[0][,parm[1][,… ]]])
```

Description

A window can have a record trigger where record-oriented activities can be centralized. For example, a navigation key trigger could call the record trigger to make sure that the record is complete before allowing the user to move off the current record.

The function can return any data type, or it can return void, depending on how the record trigger has been coded.

parm[*n*] (optional) parameters.

Example

Check the record trigger before allowing the user to move to another record.

```
{
   if (G.modified) if (record_exec(PEV_UPDATE)) break;
   edit_win(p.wl,edt_forward,false);/* Down arrow (forward one record)*/
   if (child) window(child,query);  /* query children (if any)        */
}
```

redraw

Redraws the screen.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void redraw()
```

Description

Performs a screen draw (as does **[Ctrl-R]**).

Useful for repainting the screen after a system call or user exit.

refresh

Controls screen updating.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void refresh([onoff])
expr          onoff
```

Description

onoff (optional) specifies action to take with pending updates.

- ***True*** — (not zero) All pending updates are performed. In addition a flag is set for the current window indicating that all future updates to that window should be performed immediately.
- ***False*** — (zero) no screen updates are performed until either refresh() or refresh(true) is called, or an input call is made.
- ***Not specified***— All pending updates are performed.

Example

Opens several windows with only one screen update:

```
{
.
.
refresh(false);
window(W1,open);
window(W2,open);
window(W3,open);
refresh(true);
.
.
 }
```

regexp

Regular expression processor..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int regexp(option [| flags][,expr1[,expr2]])
int        option
string     expr1
string     expr2
```

Description

Applies a regular expression against a string and returns true if there is a match or false if there is no match. The options and flags are defined in trim.h and are:

regexp_end (option) returns the ending offset of the match after regexp_exists returns true. Requires the regexp_offsets flag.

regexp_exist (option) executes the cached regular expression tree against `expr1` and returns true if there is a match, otherwise false.

If `expr2` is supplied, then the regexp processing uses it as the regular expression instead of the cached one. The cached regular expression is not affected.

regexp_free (option) frees the cached regular expresssion.

regexp_init (option) initializes and caches the regular expresssion `expr1`. Any existing regular expression is automatically replaced.

regexp_start (option) returns the starting offset of the match after regexp_exists returns true. Requires the regexp_offsets flag.

regexp_icase (flag) ignore case when checking for matches.

regexp_newline (flag) "match any" in expr1 does not match newline.

regexp_offsets (flag) calculate start/end offsets.

Notes

If the same regular expression is to be used multiple times, then it is much more efficient to use regexp_init, regexp_exist, regexp_free than regexp_exist with the regular expression.

Regexp_start and regexp_end are only valid when regexp_exists is used without the regular expression argument.

Example

The following demonstrates the performance difference between cached and uncached regular expressions.

```
#define iters 100000

{
int  i;
char exp[20] = "N*s";
char str[20] = "Niklas Back";

timestamp();

regexp(regexp_init,exp);

for (i=iters;i;i--) regexp(regexp_exist,str);

printf(regexp(regexp_exist,str) ^^ ", Seconds: " ^^ timestamp());

for (i=iters;i;i--) regexp(regexp_exist,str,exp);

printf(regexp(regexp_exist,str,exp) ^^ ", Seconds: " ^^
timestamp());
}
```

The following finds the beginning and ending offsets in a string.

```
{
char exp[20] = "N.*?s";
char str[20] = "Back, Niklas";

regexp(regexp_init|regexp_offsets,exp);
regexp(regexp_exist,str);
printf(regexp(regexp_start));
printf(regexp(regexp_end));
}
```

returns

6

12

replace

Searches for a string within a string and replaces all found occurrences.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int replace(str1,str2[,str3])
string      str1,str2,str3
```

Description

str1 specifies the string in which to search.

str2 specifies the search string.

str3 (optional)specifies the string with which to replace str2 if found in str1. If str3 is not specified, then all occurrences of str2 will be deleted from str1..

Example

Find and replace all occurrences of "San Jose" with "Los Gatos".

```
{
 char s[80], f[80], r[80];
 s = "San Jose is a lovely town. Do you know the way to San
Jose?";
 f = "San Jose";
 r = "Los Gatos";

 printf(s);
 printf("Replace '"^^f^^"' with '"^^r^^"'");
 s = replace(s,f,r);
 printf(s);
 f = "the way to ";
 printf("Remove '"^^f^^"' from '"^^s^^"'");
 s = replace(s,f);
 printf(s);
}
```

returns

```
San Jose is a lovely town. Do you know the way to San Jose?
Replace 'San Jose' with 'Los Gatos'
Los Gatos is a lovely town. Do you know the way to Los Gatos?
Remove 'the way to ' from 'Los Gatos is a lovely town. Do you know
the way to Los Gatos?'
Los Gatos is a lovely town. Do you know Los Gatos?
```

ret_freeheap

Releases all heap blocks in the free heap list to the operating system..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int ret_freeheap()
```

Description

DesignVision allocates memory in blocks at least as big as specified in `dv.ini/trim.ini` with the `heap_block_size` keyword (default: 4000). When a memory block is no longer needed, for example when a list is freed, the memory blocks are placed in a free heap list. New allocations search this list first. This eliminates any potential memory fragmentation problems. If an application uses a large amount of memory, for example by loading many very large lists and then freeing them, on a system with many users, then it might be beneficial to return this unused memory to the operating system by calling `ret_freeheap()`. `ret_freeheap()` also returns the number of freed heap blocks.

rollback

Execute the SQL ROLLBACK WORK statement.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void rollback([update])
keyword    update
```

Description

update (optional) specifies that the transaction is for update. The default (unspecified) is read-only.

Notes

Some database systems do not provide the capability of specifying read-only or update transactions. On these systems, option update is ignored.

See also `commit()`.

Example

Deletes rows in a table and confirm it with the user before committing the work.

```
{
int n_rows;
n_rows = exec_sql("DELETE FROM staff WHERE id = 20");
if (n_rows > 0) {
  printf("You are about to delete" ^^ to_char(n_rows) ^^ "rows");
  if (prompt("Proceed (Y/N) ==> ") == "Y") commit();
  else rollback();
   }
}
```

round

Gets the rounded value of the operand.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr round(data,[position])
expr       data,position
```

Description

Returns the rounded value of data.

data specifies the value to round.

position (optional) specifies the rounding position, positive indicates positions to the right of the decimal point, negative, the left of the decimal point. The default value of position is 0.

If data is a datetime value, and position is specified, its possible values are:

Mnemonic	Description
SCC,CC	Century
SYYY,YYYY,YEAR,SYEAR,YYY,YY,Y	Year
Q	Quarter
MONTH,MON,MM	Month
WW,W	Start of Week
DDD,DD,J (Default)	Day
DAY, DY,D	Nearest Sunday
HH,HH12,HH24	Hour
MI	Minute

Example

Displays salaries to the nearest thousand.

```
while(list_pos(ll) := list_next(ll))
  list_modcol(ll,6,round(list_curr(ll,6),3));
list_view(ll,0);
```

rtrim

Trims blanks to the right of a char string.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
rtrim (string[,charset])
string        string
string        charset
```

Description

This function strips the chars specified in `charset` from `string` until it reaches a character not specified in `charset`. If no char is specified, the function assumes blanks.

Example

The following strips blanks from the right of the string located in buffer parm[3].

```
buf=rtrim(parm[3])
```

scribble

Writes the text to the screen at the absolute row and column location.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
	X			

Syntax

```
string scribble(row,col,text[,visual-attr])
int             row,col,visual-attr
string          text
```

Description

Returns text, writes it to the screen at specified location.

row specifies the row position for the displayed window. -1 places the top of the box at the cursor position. -2 places the box in the center of the screen.

col specifies column position of the window; -1 places the left corner of the box at the cursor position. -2 places the box in the center of the screen.

text specifies the text to write.

visual-attr (optional) specifies attirbutes. See `field_visual()` for details.

Notes

Use this function, which writes directly to a screen, to debug or continuously update a particular position on the screen, such as a counter.

Example

None.

set_option

Sets various runtime options..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int set_option(option[,data])
int            option
expr           data
```

Description

The `option` parameter values are defined in `trim.h`. The current value is always returned. Valid values are

soc_heapsize Sets the minimum heapsize in bytes. Larger values may improve performance as fewer heap blocks will be required, e.g. when loading a list, however there may be wasted memory space. A good start is 16380.

soc_ucs2conv Sets the default conversion method used to convert between one and two byte character strings. Option values are:

0 Characters are moved to/from 8-bit and 16-bit values (default):
1 8-bit characters > 127 are converted based upon the definitions in `$TRIM_HOME/lib/dv.a2u`.
81 8-bit characters are assumed to be utf-8.

signal

Installs/removes a signal handler..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
void signal(signal[,trigger])
int         signal
code        trigger
```

Description

Installs or removes a signal handler. If no `trigger` parameter is given, then any existing signal handler for that `signal` is removed. If `trigger` is given, then it will be excuted whenever the process receives a `signal` interrupt. This function is similar to its POSIX counterpart.

signal specifies the signal number.

trigger (optional) specifies the code to execute when the `signal` is received.

Notes

By using `escape()` in the trigger code, you can exit any wait state that you may be in.

Example

```
{
int     i,j;
trigger t1 = { printf("Signal caught/escaping"); escape(); };
trigger t2 = { printf("Signal caught/returning"); };

i = to_int(prompt("Enter signum: "));

/*************************************************************/
/* Set the handler                                           */
/*************************************************************/
for (j=0;j<3;j++) {
  if (trap({
     signal(i,t1);
     prompt("Let's wait (" ^^ j ^^ ") escape ... ");
     }));
  }

/*************************************************************/
```

```
/* If you get the signal now, the process will exit.            */
/*****************************************************************/
signal(i);
prompt("Let's wait ... no signal handler");

/*****************************************************************/
/* Set the handler.                                              */
/*****************************************************************/
signal(i,t2);

for (j=0;j<3;j++) {
  prompt("Let's wait (" ^^ j ^^ ") return ... ");
  }

printf("All done");
}
```

signal_client

Sends a signal to the DVSlave process..

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void signal_client(port)
int                port
```

Description

Sends a signal to the DVslave process. If DVslave is started with the `-L[port]` option, then sending a signal to that port returns a `WM_USER+998` event. If `[port]` is not given, then the listening port is DVslave's port + 1.

Example

Using the main trigger of an application,

```
{
int          update_error;    /* update trigger error flag      */
int          changed;         /* global change flag             */
char         input_var[256];  /* input variable                 */
char         query_buf[128];  /* for expanded queries           */

trigger      test = { signal_client(1959); };

error_trap({ status(-1,parm.0); bell(); }); /* set error trap   */

key_set(key_any,{ status(-1,"Current version: " ^^ g.version); });

signal(10,test);

window(0,open|run|close);     /* Start at first window          */
}
```

Any time a signal 10 is received, the application will send a signal to the DVslave process on port 1959. The application will then receive an event `WM_USER+998`.

sizeof

Returns the allocated size of a variable.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int sizeof (variable)
ident       variable
```

Description

variable calculates during generation while the `length()` function calculates at runtime.

Example

Returns the allocated length of the user ID variable.

```
usize = sizeof(G.USER);
```

sql_xlate

Converts `statement` using the function mapping specified in the `sql_xlate_file`, which is defined in `trim/dv.ini`.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string sql_xlate(statement[,did])
string       statement
int          did
```

Description

Returns the translated string.

statement Specifies the SQL statement to translate.

did (optional) specifies the target database (as defined in `trim/dv.h`) so the function can take into account specific databases' characteristics, such as case-sensitivity.

Notes

You must put the function mapping strings you intend to use in the file specified by `sql_xlate_file`. For more information about SQL syntax mapping, refer to ***DesignVision Users Guide***.

Example

The following example converts from Oracle to DB2 syntax. `sql_xlate_file` points to a valid function mapping file.

```
{
…
db2sql = sql_xlate("select decode (id,42,16,23) from staff");

…
}
```

sqrt

Returns the square root of an expression.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
numeric sqrt(expr)
numeric      expr
```

Example

Calculates the square root of the sum of salaries.

```
sqrt_sum_sal = sqrt(sum(SAL));
```

status

Writes a message to the window status line.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
string status([wid,]msg)
string        msg
int           wid
```

Description

wid (optional) specifies the window number to use. Default is current window. If -1 is specified, msg is sent to all open window status lines.

msg specifies the message to send to window status lines.

Example

Indicates the number of database rows modified.

```
i = exec_sql("update staff set salary =
              salary * .75 where salary > :1",limit);
status(i^^ "rows updated");
```

substr

Gets a substring.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string substr(data,start[,length])
string        data
int           start,length
```

Description

Returns a substring.

data specifies the expression that may contain the substring.

start (1-based) specifies the start location. If start = 0, the whole string is returned. If it is a negative number, the number of characters specified is returned starting from the end of the string.

length (optional) specifies the minimum string length to return. Default is to return the string from start to end.

Example

Extracts the second half of a string.

```
new = substr(old,length(old)/2,length(old)/2);
```

sum

Gets the sum of a field.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
		X		

Syntax

```
numeric sum(field)
ident       field
```

Description

Returns sum of specified field. Valid only on numeric database fields.

field specifies the field to sum

Example

Gets the current sum of salaries.

```
sum_sal = sum(SAL);
```

sysinfo

Returns various system and application information values.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr sysinfo(option)
int          option
```

Description

Sysinfo returns an expression for various system and application information. The valid option values are defined in `trim.h`:

Option	Returns	Value
sysinfo_bits	Integer	The CPU word's bitsize.
sysinfo_dyncurs	List	Returns a three column list of the current cached dynamic cursors: Logical cursor #, Select statement length, Select statement text. The maximum number of cached dynamic cursors is controlled by the `dynamic_cursors` setting in `dv.ini/trim.ini`.
sysinfo_heapsize	Integer	The minimum heap size for the application. This is the value of `heap_block_size` in `dv.ini/trim.ini`.
sysinfo_memory	integer	The total amount of memory allocated by the application.

syslog

Writes a message to a system log file.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string syslog(level,message)
integer       level
string        message
```

Description

Returns the value of `message`. On Unix, this function writes to the actual syslog; on NT it writes to the operating system's event log.

level specifies the level of logging:
- 0 — Informational message
- 1 — Warning message
- 2 — Error message

message The text written to the system log file.

Example

Logs two messages to a system logfile.

```
{
 log ("0", "whoa");
 log ("0", "nelly");
}
```

system

Invokes a system call.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int system(command[,wait])
string     command
expr       wait
```

Description

Invokes an operating system command using the C-library call `system()`. The return value is operating-system dependent. See the reference manual for your operating system.

command specifies the operating system command to invoke.

wait (optional) specifies a pause. If true, the function displays a prompt before returning to the application.

Example

In a Unix environment, searches all files in the temporary directory for the user ID of the person running the report.

```
system("fgrep " ^^ cuserid() ^^ " /tmp/*",wait);
```

table_exec

Executes the window's table trigger.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
ret_value table_exec([parm[0][,parm[1][,… ]]])
```

Description

A window can have a table trigger where table-oriented activities can be centralized. For example, deleting a record may require foreign key actions that are best handled in one central trigger for this table.

The function can return any data type, or it can return void, depending on how the record trigger has been coded.

parm[*n*] (optional) parameters.

Example

If the record trigger indicates that there are no dependents on the record, then call the table trigger.

```
if (!record_exec(PEV_DELETE)) {   /* check for dependants          */
table_exec(PEV_DELETE);
edit_win(p.wl,edt_delete,p.dl);    /* delete a record               */
}
```

timestamp

Sets the timestamp or returns the delta timestamp.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
numeric timestamp()
```

Description

If the `timestamp()` call is not part of an expression, then it sets the current timestamp. Otherwise it returns the delta timestamp in seconds and sets a new one. The granularity of the result is in microseconds.

Example

Displays the application run time for 10000 operations:

```
timestamp();
for (i=10000;i;i--) j = i * i * i * i;
printf("10000 operations took "^^timestamp()^^" seconds");
```

tmpnam

Gets a unique file name.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string tmpnam()
```

Example

Creates a temporary file in which to store a list.

```
{
.
.
fname = tmpnam();
list_file(wl,fname,"b");
}
```

to_char

Converts an expression to a character string.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string to_char(data[,format])
expr           data
string         format
```

Description

data specifies the expression to convert. If data is already a char, no conversion takes place.

format (optional) specifies a mask. If none is provided, a default conversion is performed. Refer to ***DesignVision Users Guide*** for the valid formats.

Example

Format salaries with two decimal places and a currency symbol.

```
field = to_char(SAL,"$99,999.99");
```

to_date

Converts an expression to date and time based on the format specified.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
datetime to_date(data[,format])
expr             data
string           format
```

Description

If data cannot be converted to format, the conversion fails and an error is returned. If data is already a date, no conversion occurs and the original value is returned.

data specifies the expression to convert.

format (optional) specifies a mask. If none is provided, a default conversion is performed. Refer to ***DesignVision Users Guide*** for the valid formats.

Example

Formats date in European format.

```
field = to_date(G.TIME,"DD/MM/YY");
```

Rdb

Formats date to include hours, minutes, and seconds.

```
{
list     t1;
char     td1[25];
char     td2[25];
datetime dd1;
datetime dd2;
/*************************************************************/
/* Pass datetime variable into SELECT statement for Rdb      */
/*************************************************************/
dd1 = to_date("01-FEB-1992 00:00:00","DD-MON-YYYY HH:MI:SS");
dd2 = to_date("15-FEB-1992 00:00:00","DD-MON-YYYY HH:MI:SS");

/* -----   ship_dt is DATE type of Rdb   ----- */
t1 = list_open("SELECT custid,ship_dt FROM tk_ship WHERE ship_dt BETWEEN "
               " '" ^^ to_char(dd1,"DD-MON-YYYY HH:MI:SS") ^^ "' and "
               " '" ^^ to_char(dd2,"DD-MON-YYYY HH:MI:SS") ^^ "' ",1000);
list_view(t1,0);
/*************************************************************/
/* Hard code data into SELECT statement for Rdb              */
/*************************************************************/
t1 = list_open("SELECT custid,ship_dt FROM tk_ship WHERE ship_dt BETWEEN "
               " '" ^^ "01-FEB-1992 00:00:00.00" ^^ "'  and "
               " '" ^^ "15-FEB-1992 00:00:00.00" ^^ "'   ",1000);
```

```
list_view(t1,0);
}
```

to_int

Converts an expression to an integer.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int to_int(data)
expr       data
```

Notes

Truncates the expression. If data is already an int, no conversion takes place and the original value is returned.

Example

Converts salary field datatype to eliminate decimals.

```
field = to_int(SAL);
```

to_number

Converts an expression to numeric data type.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
numeric to_number(data[,format])
expr              data
string            format
```

Description

Returns an error if data does not match the specified format.

format (optional) specifies a mask. If none is provided, a default conversion is performed. Refer to ***DesignVision Users Guide*** for the valid formats.

Notes

Conversions are performed automatically within TRIMpl.

`to_number()` provides an explicit type conversion and allows format restrictions to be specified.

If the datatype of data is datetime, the number of days since January 01, 0000 is returned. To convert to Oracle Julian date, add 1,721,061.

Example

Converts a character string to a numeric. `to_number()` returns an error if price is greater than 99.99 or has more than two decimal places.

```
field = to_number(price,"99.99");
```

tokenize

Tokenize a character string.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
list tokenize(data[,separator[,quote[,nulls]]])
string        data,separator,quote
int           nulls
```

Description

Breaks up the tokens in `data`. The default token separator is a blank character. The `separator` string overrides this by providing a string of characters to be used as token delimiters. There is no default quote character. If `quote` is provided, then all characters within a pair of `quote` characters are returned as one token. The `nulls` flag controls how `tokenize` treats leading, trailing and consecutive separators. The default is false which means that leading, trailing and consecutive separators will just be ignored. Setting `nulls` to true means that an empty list row will be inserted wherever there are leading or consecutive separators and appended if there is a trailing separator. The tokens are returned in a single column list.

data specifies the string to tokenize

separator (optional) specifies the character(s) to use as token delimiter(s)

quote (optional) specifies the character(s) to use as quote(s)

nulls (optional) specifies how consecutive separators are treated. The default is false which does not insert an empty list row for consecutive separators.

Example

Simple blank delimited string with no quotes:

```
LL = tokenize("Trifox is located in California");

         LL will contain: Trifox
                          is
                          located
                          in
                          California
```

Use "/" and "." as the token delimiters:

```
LL = tokenize("/etc/mail.log","/.");

         LL will contain: etc
                          mail
                          log
```

Use " " and "," as token delimiters and "'" as the quote character:

```
LL = tokenize("what, might 'this be'"," ,","'");

         LL will contain: what
                          might
                          this be
```

Use "," as the token delimiters:

```
LL = tokenize("what,might,,be",",","'",true);

         LL will contain: what
                          might

                          be
```

translate

Translates a character string.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
string translate(data,from,to)
string           data,from,to
```

Description

Replaces all characters in data as specified.

from specifies the characters to replace.

to specifies the new characters to use.

If the number of characters in *to* is a multiple of the number of characters in *from*, for each character in from, that multiple of characters from to replaces that single character.

If to is empty, all characters found in from are deleted.

Example

Replaces all digits with asterisk (*).

```
field = translate(SAL,"0123456789","**********");
```

Replaces all single quotes with two single quotes (useful for SQL INSERT commands).

```
exec_sql(translate(sqlstmt,"'","''"));
```

Deletes all commas from character representation of numerics.

```
buf = translate(line,",","");
```

Validates that a string contains only uppercase letters.

```
if (translate(line,"ABCDEFGHIJKLMNOPQRSTUVWXYZ","") := "")
  error("Line may only contain letters");
```

trap

Traps any errors that occur during the execution of code.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int trap(code)
block    code
```

Description

Returns true on error, otherwise false.

Example

Catches database error if table does not exist.

```
{
list LL;

if (trap({ LL = list_open("SELECT * FROM " ^^ parm.0,100); ))
  error("Table " ^^ parm.0 ^^ " does not exist");
.
.
}
```

unicode

Returns an integer representation of the unicode character.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
int unicode(character)
char2       character
```

Description

Returns unicode integer value of `character`.

Character specifies the char (alphanumeric or space) to process.

Example

Prints the unicode code for each character or space as given.

```
{
int    i,len;
char2  s[80];
char2  ch;
s = "Hi my name is Bentley";

for (i=1;i<=length(s);i++) {
  ch = substr(s,i,1);
  printf("The UNICODE value of '" ^^ ch ^^ "' is " ^^
unicode(ch));
  }
}
```

update

Invokes the update trigger for the current window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void update()
```

Notes

The update trigger of another window can be executed at any time by calling `window(window-name |expr,update)`.

Example

See the default window trigger in ***DesignVision Users Guide***.

window

Calls a window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void window(window-name | value,
             [open |,run |,update |, close |, initialize |, query
])
ident       window-name
int         value
keyword     open,run,update,close,initialize,query
```

Description

Calls a specified window. You can call any window from anywhere in the application.

window-name specifies the window to call by name.

value specifies the window by sequence number.

options specify the action(s) to perform on the window. This value can be:

- ***open*** — If the window does not already exist, create it. In either case, put it in front of any other windows on the screen.
- ***run*** — If the window is open, put it in front of any other windows on the screen and pass control to its window trigger. If the window is not open, a user input call returns an error.
- ***initialize*** — Execute the window's initialize trigger.
- ***query*** — Execute the window's query trigger.
- ***update*** — Execute the window's update trigger.
- ***close*** — Close and remove the window from the screen.

Character-based applications have two default triggers, one for the main window (which is global) and another for the first window. Window applications have a single default trigger for the main (or first) window.

If you specify multiple options for `window()`, they execute in the following order:

1. open
2. initialize
3. query
4. update
5. close

Example

Placed in the window trigger and conditionally invokes different windows depending on the value in SAL.

```
{
if (SAL > 20000) window(CUT_SALARY,open|run|close);
else             window(RAISE_SALARY,open|run|close);
}
```

window_attr

Returns the window attributes of the current window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int window_attr()
```

Description

The window attribute is an integer value where each attribute is represented by a bit. See `watt_xxx` in `trim.h` for values.

window_count

Gets the number of windows in the application.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int window_count()
```

window_info

Returns basic information about a window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
group window_info(window_name | value)
ident              window-name
int                value
```

Description

Window_info takes a single parameter, either the window name or id, and returns information about that window. The values are returned using the TRIMpl group format. The window does not have to be opened before calling window_info.

window-name specifies the window by name.

value specifies the window by sequence number.

Currently there are four return values:

Rows per row

Row count

Number of fields

Window attributes

Example

```
{
  int rpr,rct,nfl,wat;

  [rpr,rct,nfl,wat] = window_info(W1);

  [rpr] = window_info(W2);

}
```

window_name

Gets the name of the currently active window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
string window_name([title])
string              title
```

Description

title (optional) specifies the current window's new title. It is ignored in the character based version.

Example

Logs user errors that occur including the window name:

```
error_trap({ g.msg = parm.0);
             log("logfile",window_name() ^^ ":" ^^ g.msg;
             bell();
           );
```

window_seq

Returns the zero-based sequence number of the window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
int window_seq(window-name)
ident          window-name
```

Description

window-name specifies the window by name.

window_table

Returns a table's name.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
string window_table(table-index)
int          table-index
```

Description

Returns a table's name.

table-index specifies the zero-based table index.

Example

The following displays the table name of the current field.

```
prompt(window_table(ascii(field_tid)-ascii("A")));
```

winexec

Executes a Windows program.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void winexec(command[,wait])
string       command
int          wait
```

Description

Executes a command in the Windows environment.

command specifies the command to execute.

wait (optional) specifies timing. If it is true (non-zero), the executed command must terminate before the application can continue (synchronous). Default is asynchronous.

Example

```
winexec("c:\bin\aprg -a");
```

winhelp

Invokes the Windows help system.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X				

Syntax

```
void winhelp(helpfile)
string       helpfile
```

Description

Invokes the Windows help system using `helpfile`.

Example

```
winhelp("c:\help\trimpl.hlp");
```

winprop

Display the properties window.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X			

Syntax

```
void winprop([deffile])
string        deffile
```

Description

`winprop()` activates the properties window and allows the user to alter font and other window characteristics. If `deffile` is specified then the file is used to set the windows properties.

Example

A menu item under the File menu could be called Properties and have the following trigger.

```
winprop();
```

xml

Executes an XML function.

Apps(window)	*Apps (char)*	*Reportwriter*	*RPC*	*Standalone PL*
X	X	X	X	X

Syntax

```
expr xml(opt[,parm])
int      opt
expr     parm
```

Description

Performs various XML operations based on the value of `opt`.

opt	Parm	function
xml_attr	attribute name	Returns the value of the specified attribute. If the attribute is not found, then it returns NULL
xml_data	N/A	Returns the current XML data value as a character string. If an error has occurred in any other XML function, then the error message can be retrieved using `xml(xml_data)`.
xml_endtag	N/A	Returns true if the XML stream is at the end of a tag.
xml_esc	string	Returns a string with `string's` special HTML characters converted to HTML entities *(see xml_unesc)*.
xml_free	N/A	Frees all resources associated with the XML system.
xml_init	Any valid file source	Initializes the XML system using the specified XML stream.
xml_name	N/A	Returns the current XML tag as a character string.
xml_next	N/A	Moves to the next XML tag. The entire attribute string is available using `xml(xml_data)`. Returns either the tag code value if `xml(xml_tags,list)` was called, (-1) if no tags were sent, or NULL if an error occurred. If the return code is NULL, use `xml(xml_data)` to retrieve the error message.

xml_tags	list of tag/value pairs	Sends a list of valid XML tags to the XML system in the form of a two column list. The first column is the tag and the second column is its code value.
xml_unesc	string	Returns a string with `string's` HTML entities converted to special HTML characters *(see xml_esc)*.
xml_value	N/A	Returns the current uninterpreted data from the XML stream. If it returns NULL, then if `xml(xml_endtag)` returns true, the XML stream is empty; otherwise the XML stream is positioned at a new XML tag.

Example

Convert a generic XML file to internal list format:

```
/*************************************************************
** Convert generic XML file to internal list format
**
** parm.0 - XML file
** parm.1 - Internal list file
**
** Format of list: col0: tag name
**                 col1: attributes (NULL if none)
**                 col2: NULL no data (<..../>)
**                       Char datatype then string data
**                       List datatype then sub-tree
**
*************************************************************/

#trigger next_element
/
*************************************************************
** Get next tag element
*************************************************************/
{
int rc,end;
rc = xml(xml_next);
if (!rc) error(xml(xml_data));
if (rc < (-1)) end = true;
else end = xml(xml_endtag);
return([xml(xml_name),end,xml(xml_data)]);
}                              /* next_element                */

#trigger dive
/
*************************************************************
** Process XML
*************************************************************/
{
int        i;
```

```
char       val[500],attr[1000];
local int  end;
local int  level  = parm.0;
local char s[200] = "";
local char tag[100],exit_tag[100] = parm.1;

for (i=level;i;i--) s = s ^^ " ";

while (true) {
  [tag,end,attr] = next_element();

  if (end) {
    if (tag == exit_tag) return;
    list_mod(parm.2,1,tag,attr,NULL);
    continue;
    }

  val = xml(xml_value);
  val = ltrim(rtrim(val,G.sc),G.sc);

  if (xml(xml_endtag) && (tag == xml(xml_name))) {
    list_mod(parm.2,1,tag,attr,val);
    xml(xml_next);                  /* eat                        */
    if (tag == exit_tag) return;
    }
  else {
    list_mod(parm.2,1,tag,attr,list_open("20 30 20",0));
    dive(level+1,tag,list_curr(parm.2,2));
    if (!exit_tag) return;
    }
  }
}                               /* dive                           */

#trigger
/*****************************************************************
** Main
*****************************************************************/
{
char sc[3] = chr(10,13,32);   /* trim chars                       */
list LL;

xml(xml_init,parm.0);

LL = list_open("20 30 20",0);

dive(0,NULL,LL);

list_file(LL,parm.1,"x");

xml(xml_free);
}
```

Convert an internal XML list to a generic XML file.

```
/*****************************************************************
** Convert internal XML list to generic XML file
**
** parm.0 - file with list
**
** Format of list: col0: tag name
**                 col1: attributes (NULL if none)
**                 col2: NULL no data (<..../>)
**                       Char datatype then string data
**                       List datatype then sub-tree
**
*****************************************************************/

#trigger dive
/*****************************************************************
** Process a list
*****************************************************************/
{
local int  i;
local char tag[100];
char       s[1000],attr[500];

for (i=list_rows(parm.0);i;i--) {
  tag  = list_curr(parm.0,0);
  attr = list_curr(parm.0,1);
  s    = "<" ^^ tag;
  if (attr) s = s ^^ " " ^^ attr;
  if (list_curr(parm.0,2) == NULL) printf(s ^^ "/>");
  else if (datatype(list_curr(parm.0,2)) == "C")
    printf(s ^^ ">" ^^
translate(translate(translate(translate(translate(
                                list_curr(parm.0,2),
                                "&","&amp;"),
                                "<","&lt;"),
                                ">","&gt;"),
                                "'","&apos;"),
                                """","&quot;") ^^ "</" ^^ tag ^^
">");
  else {                        /* a sub-list                    */
    printf(s ^^ ">");
    dive(list_curr(parm.0,2));
    printf("</" ^^ tag ^^ ">");
    }                           /* a sub-list                    */
  list_next(parm.0);
  }
}                               /* dive                          */
```

```
#trigger
/*****************************************************************
** Main
*****************************************************************/
{
list LL;

LL = list_open(parm.0,999999);

dive(LL);
}
```

Convert special HTML characters to HTML entities and back again:

```
{
char buf[100];
buf = xml(xml_esc,"<data wb='www.trifox.com?cow=42&moo=53'>");
printf(buf);
buf = xml(xml_unesc,buf);
printf(buf);
}
```

returns

```
&lt;data wb=&apos;www.trifox.com?cow=42&amp;moo=53&apos;&gt;
<data wb='www.trifox.com?cow=42&moo=53'>
```

E

F

G

W