CHAPTER 1 Fundamentals

Concepts

This section describes some of the internal workings of DataCAD.

Angles

All angles in DCAL (and therefore internal to DataCAD), are expressed in radians starting at 0 to the right (east). Angles increase counterclockwise.

NOTE: The internal representation has no effect on how DataCAD displays angles to the user.

Coordinate System

Internally DataCAD stores all distances and coordinates as 1 unit = 1/32 inch. This is also true for metric units.

When converting distances to strings or strings to distances, the current scaling type (feet- inches-fractions, meters, decimal feet, etc.) determines the format of distance strings. This does not, however, affect the way that DataCAD maintains distances internally.

Entities Every primitive graphic shape that DataCAD understands is called an entity. Currently supported entities include:

3D arc	ellipse	3D line
line	arc	point
associative	polygon	dimension
bezier	skew box	bezier
slab	surface	bspline
symbol	circle	text string
cone	torus	cylinder
truncated cone	dome	

Each entity has certain information associated with it such as which layer it is on, color, line type, line spacing, line weight, overshoot factor, and an attribute (integer). Each entity also has an address associated with it which is a handle to the entity in the drawing file. The address of an entity does not change, either during an editing session or between editing sessions.

Keyboard

A DataCAD macro perceives the keyboard differently from other programs. To simplify writing macros, all keys are returned from the keyboard as integers, not as

characters. This allows DataCAD to represent the entire 256 extended ASCII character set, as well as any 'special' key. 'Special' keys return escape codes.

A special key is any key that has a macro- interpretable function. For instance, the function keys are special keys, as are the arrow keys, (Ins) and (Del). Some keys are special keys only some of the time. For instance, (q) returns an escape code under some circumstances which causes DataCAD to increment the current line type, and at other times (while reading a string) returns a normal key code.

Several of the DCAL routines return a result that tells you how they were exited. These result codes are res_normal for a normal finish and res_escape for an interruption because an escape code was read. A result of res_escape typically means that the user pressed a function key while the macro was requesting some type of input. The escape code that was pressed returns from these routines as an integer.

Layers

Each entity is on a layer. Each layer is scanned during any search of DataCAD's database. When a layer is not searched, every entity on that layer is ignored. Ignoring certain layers (for instance layers that are off) or confining a search to a specific layer or set of layers improves search speed.

Selection Sets

DataCAD supports 16 selection sets. A selection set is a collection of entities. Every entity can be a member of some, none, or all of the selection sets. Selection sets are convenient ways of grouping entities so you can act upon them together. Selection sets also provide an alternative to using layers to group entities. The first eight selection sets are available to you and DCAL, while the other eight are reserved for DataCAD's internal use.

Selection sets exist across uses of a drawing file. That is, you can set up and manipulate a selection set. If the drawing is exited and then reentered, the selection set still exists. In this respect, selection sets are similar to layers as a way to group entities.

Local Variables

Routines can declare local variables. These variables can only be used inside this routine or any routine local to it. They are pushed on the stack and exist only during the lifetime of the routine.

Symbols

A symbol is similar to any other DataCAD entity. However, a symbol is a reference to a group of other entities that are drawn where the symbol is inserted. A symbol, as created at the DataCAD template menu, is referenced by its file name (including the

path). With a symbol defined by a macro, however, all you need is a unique symbol name.

Symbols are instanced, that is, their geometry exists only once in the database. They can be globally updated, but no longer individually edited.

Basics

DCAL is a block-structured language similar to Pascal and Modula-2. DCAL supports recursive procedures and functions, as well as nested routines, local variables and constants, reference and value parameters, and conformant arrays.

Case Sensitivity

The functions and procedures in DCAL can be entered in either upper- or lower-case characters. DCAL is not case sensitive, therefore, ABC, abC, and AbC are all the same to DCAL.

Comments

A comment is text that prints within the code of a macro, but is ignored by the macro. DCAL allows two different types of comments:

Comments that start with a left brace ({) and end with a right brace (}).

The other type of comment is the comment until end of line. This starts with an exclamation mark (!) and continues until the end of the current line.

Comments cannot be nested, and cannot appear within a string.

Identifiers

DCAL macros are composed of identifiers, both pre-defined and user-defined. Examples are: variable names, procedure names, and procedure parameters.

Valid DCAL identifiers begin with a letter, and are followed by letters, numbers, and underscores. Identifiers may be up to 20 characters long. Identifiers longer than 20 characters result in compile time errors.

Therefore, these are valid identifiers:

test
proc_1
alphabet

These, however, are not valid identifiers:

123test	*Starts with a number.*
_hello	*Starts with an underscore.*

abcdefghijklm nopqrstuvwxyz *More than 20 letters.*
ab$de *$ is an invalid character.*

DCAL identifiers are not case sensitive, therefore, ABC, abC, and AbC are all the same identifier to DCAL.

Keywords

A keyword is a word that is part of the DCAL language. Keywords are reserved and cannot be used for any other purpose. A complete list of keywords follows:

```
and         false       of          return
array       for         off         string
begin       function    on          then
by          if          or          to
const       in          out         true
do          include     procedure   type
else        message     program     until
elsif       mode        public      var
end         module      record      while
external    not         repeat
```

White Space

White space is ignored by the compiler. White space consists of spaces, newlines, and tabs that are not in a string or character literal.

Program Layout

Macro Header

There are two types of macro headers, depending on whether the .dcs file contains a program or a module:

Programs
As the sample macro spiral shows, DCAL program macros begin with the keyword program followed by the name of the macro, followed by a semicolon.

Modules
DCAL modules begin with the keyword module, followed by the name of the module, followed by a semicolon. Modules have no body, and thus no corresponding begin keyword. However, they do have an end followed by the module name and a period. The sample module wrtutl.dcs is an example of a module.

Constant

The Constant section, if present, starts with the keyword const. Following it are the constant identifier, an equal sign (=), followed by the value of the constant. Finally, a semicolon terminates each constant.

DCAL recognizes two types of constants, integers, and reals. The following is a valid constant section:

```
CONST
   delta = 23.43;  {real constant}
   max_count = 45; {integer constant}
```

Type

The Type section, when present, comes after the Constant section. It begins with the keyword type, followed by the type name, followed by the type definition.

Scalar Types
Scalar types consist of the DCAL built-in simple types which are described in the "Data Types" chapter. These simple types are indivisible: a DCAL macro cannot directly examine these variables to any greater degree.

Array Types
Array types are used by DCAL to group data together. To declare an array, use the keyword array, followed by a left bracket ([), followed by the lower bound, followed by two periods (..), followed by the upper bound, followed by a right bracket (]), followed by the keyword of, followed by the type of the element that the array is comprised of.

Notice that arrays can be made up of arrays. For instance:

```
TYPE   x = ARRAY [0..9] OF ARRAY [3..5] OF integer;
```

A shorthand notation for this is:

```
TYPE   x = ARRAY [0..9, 3..5] OF integer;
```

Records
Records are the other way DCAL groups data together. DCAL records are similar to Pascal records, including the use of the case variant record. The only distinctions are that in a DCAL case variant record, the case selector (tag) must be an integer, and no explicit tag variable is allowed. For example, the following are valid record descriptors:

```
TYPE
   rec1  =     RECORD
            a : integer;
            b : real;
          END;
   rec2  =     RECORD
```

```
    a : integer;
    CASE integer OF
    0 : (r : real);
    1 : (b : boolean);
  END;
```

Variables

Next comes the optional variables section. It begins with the keyword var, followed by a list of identifiers, each followed by a colon, then the type of variable, followed by a semicolon. For example:

```
VAR
   i            : integer;
   x, y         : real;
   dis_Array    : dis_Type;       { a user defined type }
```

Notice that, unlike constants, more than one variable can be given the same type, as long as they are separated by commas.

Subprograms

A subprogram, or routine, is a procedure or function. A function is identical to a procedure except that it returns a value and can be used in an expression, if its type is appropriate for the expression. In this manual, the terms subprogram and routine mean either procedure or function.

Procedures

Procedures are declared by the keyword procedure, followed by the procedure name, followed by an optional parameter list. After that, a procedure may declare local constants, types, variables, and other routines. A procedure ends with control falling through to the final end statement, or by using the return statement.

Functions

Functions are similar to procedures, except they return a value of a certain type. Functions are declared with the keyword function, followed by the function name, an optional parameter list, a colon, and the type of variable the function returns. A function may declare local constants, types, variables, and other routines. In the body of the function, the return value is returned in the return statement. The returned value follows the keyword return. If control falls through to the final end in a function without encountering a return statement, a fatal runtime error occurs. Functions may only return scalar types.

NOTE: In this manual, the function or procedure is listed with a description, followed by the parameters, with descriptions of the parameters concluding the section.

Passing Parameters

Parameters are passed to routines in one of three ways: IN mode, OUT mode, and IN OUT mode. You can specify the usage of the parameter; and the compiler determines whether to pass the parameter by reference or value.

Valid routine declarations with parameters are:

```
PROCEDURE calc (x, y : real; x1, y1 : OUT real);

FUNCTION dist (x1, y1, x2, y2 : real) : real;

PROCEDURE readint (i : IN OUT integer);
```

IN Mode
An in mode parameter may be read from, but not written to. When no mode is assigned to a parameter, in is the default. You can substitute an expression for an in parameter; therefore, both (3) and (3 * i) are valid in parameters.

OUT Mode
An out mode parameter can be written to, but cannot be read. An example is a procedure that returns several results in out parameters.

```
PROCEDURE average (data : ARRAY OF integer;
                   mean : OUT integer;
                   sumation : OUT integer;
                   median : OUT integer);
```

IN OUT Mode
in out parameters can be both read from and written to. These parameters typically have an initial value with a final value returned from the routine.

Open Array Parameters

DCAL supports open array parameters to procedures. An open array (also called a conformant array) is an array that does not have specified bounds, for example, a procedure that sorts an array of integers. The procedure can sort an array of any size. The declaration is:

```
PROCEDURE sort (i : ARRAY OF integer);
```

The functions low and high (described later) are used by the procedure to determine the actual bounds of the array parameter.

Expressions and Logical Operators

This section discusses type compatibility issues. It also discusses mathematical operators that are valid in DCAL.

Type Compatibility

DCAL features strong type checking. Any operation may be performed on a variable as long as the operation is appropriate to the base type of that variable. However, variables and expressions must be the same type to operate or appear together in assignment statements.

For instance, if you declare:

```
TYPE   count = integer;

VAR   a, b, c : count;   x, y, z : integer;
```

Then you can legally write:

```
a := b + c;
```

because a, b, and c are declared as identical types and the + operator is appropriate to the base type of count, which is integer. However, you could not write:

```
a := b + x;
```

because b and x are not of the same type.

You can escape the type checking through type casting. For example:

```
a := b + count (x);
```

Here, the type name is used to relax the type checking. For this to work, the base type of x must be the same as the base type of the type count, as it is in this example (both are integers). You can use type casting to convert any type to any other type, as long as they are the same size in computer memory.

Operators

The mathmatical operators *, /, +, and - are used for multiplication, division, addition, and subtraction, respectively. Use these operators with both real numbers and integer numbers. Both operands must be of the same type; with results of the same type.

The routines *float*, *round*, and *trunc* may be used within an expression in which both integer and real variables are mixed.

The operator *mod* is used only with integer operands and returns the integer remainder of division of the two numbers or variables. Here are some examples of these operators:

```
VAR

  rl1, rl2,           : real;
  int1, int2,         : integer;
```

```
  rlResult            : real;
  intResult           : integer;

BEGIN
  rlResult := (rl1 + rl2) / 2.0;
  intResult := (int1 - int2) * 2;
  rlResult := float (int1) * 34.75;
  intResult := 7 mod 3;
  { this gives intResult = 1 }
  intResult := 7 / 3;
  { this gives intResult = 2 }
  intResult := round (rl1) / 4;
  intResult := trunc (rl2) * 6;
END XXX.
```

Statements

This section describes the statements that make up DCAL.

Assignment Statement

The assignment statement is the statement DCAL uses to assign values to variables. Its use is identical to the Pascal assignment statement:

```
variable := expression;
```

The variable gets the value returned by the expression. The expression must evaluate to the same type as the variable (integers must be assigned to integer variables, reals to real variables, etc). The expression itself may be a variable.

IF Statement

The IF statement is used for testing Boolean conditions and branching based on the result. Several valid IF statements are:

```
IF f AND g THEN
  dofirst;
ELSE
  dosecond (f);
END;

IF apple + orange < pineapple THEN
  error1;
ELSIF i >= j THEN
  deleteFile;
ELSE
  formatDisk;
END;

IF today = monday THEN
  getOutOfBed (late);
  goToWork (late);
END;
```

Conditional Expressions

Follow the keyword *IF* with an expression which returns a Boolean result. When the result is true, the code following the *THEN* keyword is executed. When the result is false, the code following the *ELSE* keyword is executed.

THEN Statements
The statements between the *THEN* and the following *END*, *ELSE*, or *ELSIF* are executed when the Boolean expression evaluates to true.

ELSE Statements
The statements between the *ELSE* and the corresponding *END* or *ELSIF* are executed when the Boolean test expression evaluates to false. The *ELSE* section is optional. When the *ELSE* section is not present and the test expression evaluates to false, no action is taken and control falls through to the next statement following the corresponding *END* of the *IF* statement.

ELSIF
Use the keyword *ELSIF* to shorten the number of lines needed for multiple *IF* statements.

The following code:

```
IF key = 1 THEN
  code_for_1;
ELSE
  IF key = 2 THEN
    code_for_2;
  ELSE
    IF key = 3 THEN
      code_for_3;
    ELSE
      other_code;
    END;
  END;
END;
```

using ELSIF could be written as:

```
IF key = 1 THEN
  code_for_1;
ELSIF key = 2 THEN
  code_for_2;
ELSIF key = 3 THEN
  code_for_3;
ELSE
  other_code;
END;
```

Using *ELSIF* in this way can be a substitute for the Pascal CASE statement, which DCAL lacks.

Procedure Call Statement

With the procedure call statement you can execute procedures you have written or procedures that are built into DCAL and DataCAD. To call a procedure, simply use the procedure's name, followed by its parameters, if any. Example procedure calls are:

```
wrterr ('Unable to open file.');

write_prompt;{ call proc. w/o parameters }

lblson; add (x, y);

testit (3 + 4, a);
```

RETURN Statement

The *RETURN* statement occurs in two slightly different forms depending on whether it is used in a procedure or in a function. Both procedures and functions may have any number of *RETURN* statements.

RETURN Used in Procedures
When used in a procedure, the keyword *RETURN* appears by itself. It causes the execution of the procedure to terminate and control to return to the calling routine. There is an implicit *RETURN* statement at the end of every procedure.

RETURN Used in Functions
When used in a function, the keyword *RETURN* is followed by the expression that is returned as the value of the function. If control falls through to the final *END* of a function, a run- time error occurs. The value that follows a *RETURN* statement is returned to the calling routine.

WHILE Statement

The *WHILE* statement implements a loop. The keyword *WHILE* is followed by a Boolean expression, followed by the keyword *DO*. Following this are one or more statements, followed by *END*. The expression is evaluated at the top of the loop. When the statement is true, the statements in the loop are executed. The expression is then re-evaluated and the loop is rerun until the Boolean expression evaluates to false.

Notice that a loop can be executed zero times when the expression evaluates to false the first time it is tested.

A valid WHILE statement is:

```
i := 1;     { initialize counter }
done := false;
WHILE NOT done DO
  IF test (i) THEN;
    i := i + 1; { increment
                  counter }
```

```
  ELSE
    done := true;
END;
```

REPEAT Statement

The *REPEAT* statement is another looping structure in DCAL. Unlike the *WHILE* loop, the body of the *REPEAT* statement is executed at least one time. The *REPEAT* loop consists of the keyword *REPEAT*, followed by the statements that make up the body of the loop, followed by the keyword *UNTIL*, followed by a Boolean expression.

Every time control passes to the *UNTIL* statement, the Boolean expression is evaluated. When the expression evaluates to true, control passes through to the statement following *UNTIL*. When the expression is false, control goes back to the statement following the *REPEAT* statement.

A valid REPEAT statement is:

```
i := 10;
REPEAT
  write_file (i);
  i := i - 1;
UNTIL i = 0;
```

FOR Statement

The *FOR* statement is used to loop a specified number of times. The syntax of a *FOR* statement is:

FOR <ivar> := <startval> *TO* <endval> | *BY* <iconst> | *DO* <stmts> *END*;

The index variable (<ivar>) must be an integer variable that can be assigned. The optional *BY* is followed by an integer constant. This is the step size used in the *FOR* statement which may be positive or negative. When the *BY* section is not included, the step size is one (1). The <stmts> are executed until the value of <ivar> is greater than <endval>. <endval> is only evaluated once, when the loop is entered. When <startval> is greater than <endval> (and the step size is positive), the <stmts> are never executed.

CHAPTER 2 Constants

This chapter describes the constants that are built into DCAL. They are pre-declared and can be used by your programs.

NOTE: The values of these constants may change in future releases of DCAL and DataCAD; therefore, your programs should never depend on the actual value that these constants represent.

I/O Constants

File Operation
These operation constants are used and returned by file handling procedures:

fl_access_denied This error is returned when an attempt is made to access a file which cannot be accessed, such as attempting to write to a read only file.

fl_invalid_access_code If this error occurs, please notify Technical Support.

fl_invalid_function If this error occurs, please notify Technical Support.

fl_invalid_handle An attempt was made to access a file using an improper file variable. Usually, this indicates an uninitialized file variable.

fl_no_handles_left An attempt was made to open too many files at once. Some operating systems restrict the number of files that can be opened at one time. For MS-DOS, you can set this number in the config.sys file with the files= statement.

fl_not_found An attempt was made to open a file that could not be found.

fl_ok The file routines return this constant when they are successful.

fl_path_not_found A pathname was specified which does not exist.

fmode_read This constant specifies that a file is open for reading only.

fmode_read_write This constant specifies a file that is open for both reading and writing.

fmode_write This constant specifies a file that is open for writing only.

Pathnames
DataCAD defines a number of built-in pathnames, which you can access with the routines getpath and setpath. These routines take an integer constant which has one of

the following values:

pathchr	The path where DataCAD looks when loading a character set. Changing this has no affect on any previously-loaded character sets.
pathdef	This is the path from which DataCAD loads default files, such as the default angles, scales, and distances.
pathdrv	This path indicates where driver files are loaded from. This path is not used once DataCAD is running.
pathdwg	The path where DataCAD looks for drawing files.
pathdxf	This is the index of the path where DataCAD reads and writes dxf files.
pathfrm reports	This is the path where DataCAD looks for form files when executing the reports command.
pathlyr	pathlyr is the index of the path where DataCAD loads and stores layer files.
pathmcr	pathmcr is the index of the path where DataCAD looks when loading a new macro for execution. Changing this has no affect on an executing macro.
pathout	This determines where DataCAD sends text output files, such as output from the report command.
pathsup	pathsup is the index of the path where DataCAD looks for support files, such as message and label files. Do not change this pathname while DataCAD is running.
pathswp	pathswp is the path where DataCAD keeps virtual memory swap files. This may be changed while DataCAD is running without disturbing existing virtual address spaces.
pathsym	pathsym is the index of the pathname where DataCAD adds new symbol files. This is used when new symbol files are created. Once a symbol file exists in a template, the filename is stored in the template file.
pathtmp	This is the index of the path where temporary files are kept. It should be used for any temporary file that the macro needs.
pathtpl	The path where DataCAD looks for template files. This

pathname may be changed while a template file is displayed on the screen. This does not affect any already-opened files.

Data Constants

Attributes

The following constants are used to create, examine, and modify attributes.

atr_addr	The value of this attribute is a logical address (lgl_addr). These are the addresses of layers, entities, attributes, symbols, etc.
atr_ang	The value of the attribute is a real number interpreted as an angle.
atr_dis	The value of the attribute is a real number interpreted as a distance.
atr_int	An attribute that is an integer.
atr_name_len	The maximum length of the name of an attribute, 12 bytes.
atr_pnt	The value of an attribute of this type is a point.
atr_rl	The value of the attribute is a real number.
atr_str	An attribute whose value is a string of up to 80 characters.

Colors

These constants refer to the standard colors used by DataCAD.

NOTE: Because the value of these constants may change between versions of DataCAD, use the constant names.

Name	**Colour**	**Name**	**Colour**
ClrBlue	Blue	ClrBrown	Brown
ClrCyan	Cyan	ClrDkGray	Dark Grey
ClrGrn	Green	ClrLtBlue	Light Blue
ClrLtCyan	Light Cyan	ClrLtGray	Light Grey
ClrLtGrn	Light Green	ClrLtMgta	Light Magenta
ClrLtRed	Light Red	ClrMgta	Magenta
ClrRed	Red	ClrWhite	White
ClrYellow	Yellow		

Line Types

The constants in this section refer to DataCAD's built-in line types.

ltype_solid	ltype_solid refers to the line type solid.
ltype_dotted	ltype_dotted is the dotted line type.
ltype_dashed	ltype_dashed is the dashed line type.
ltype_dotdash	ltype_dotdash is the dot-dash line type.

The user-defined line types are identified by integer values starting at 4.

Entities

These constants are used in some of the database routines to refer to the different types of entities that DataCAD supports:

entar3	entar3 is used for a 3D arc (an arc not in the x-y plane).
entarc	entarc indicates that the entity is a two-dimensional arc.
entbez	entbez refers to a Bezier curve entity.
entblk	entblk represents a block. A block is similar to a slab, but has only four sides.
entbsp	entbsp indicates an entity type of a b-spline.
entcnt	entcnt represents a three-dimensional contour curve.
entcon	entcon refers to a cone. See also enttrn.
entcrc	entcrc indicates a circle entity.
entcyl	entcyl refers to a cylinder.
entdim	entdim refers to the associative dimension.
entdom	entdom indicates a dome. A dome is any section of a sphere.
entell	entell indicates an ellipse.
entlin	entlin refers to a line, the most common entity.
entln3	entln3 refers to a three-dimensional line. This entity is a free vector in 3D space.

entmrk	entmrk is the point (marker) entity type.
entpln	entpln is the entity constant for a polyline.
entply	entply represents a polygon. A polygon is a flat, closed, three-dimensional entity.
entrev	entrev indicates a surface of revolution. Like polylines, this involves the copious data of polyverts.
entslb	entslb represents a slab. A slab is similar to a polygon, but has some specified thickness.
entsrf	entsrf refers to the three-dimensional mesh surface.
entsym	entsym refers to the insertion (an instance) of a symbol. Note that this is an instance of a symbol and does not describe its geometry.
enttor	enttor indicates the entity type torus, a doughnut shape.
enttrn	enttrn represents a truncated cone.
enttxt	enttxt refers to text strings.

Layer Mode

The constants in this section describe the mode that DataCAD uses to search for entities across layers. They are used with a mode_type variable.

lyr_all	Using this mode, all layers in the drawing are searched whether they are on or off.
lyr_curr	Using lyr_curr, only the current layer is searched.
lyr_on	Using lyr_on, all layers in the drawing that are turned on are searched.

Processing Constants

Entity Drawing

These constants are used when drawing entities on the screen. They are used by some of the routines which manipulate the screen.

drmode_black	When an entity is drawn with mode drmode_black, it is erased from the screen. This makes an entity vanish from the screen, but not from the database.

drmode_flip — With drmode_flip, the entity being drawn is XORed with what is currently on the screen. Notice that two consecutive XORs result in the restoration of the original image. By using drmode_flip twice in a row, you can make an entity blink.

drmode_white — Using drmode_white causes an entity to be drawn on the screen in whatever its color is. This is the opposite operation from drmode_black.

Function Key Return

The constants in this section are returned by the routines that detect function key presses, such as getchar, getpoint, and dgetint. Also, see the function FnKeyConv.

f1, f2, f3, f4, f5, f6, f7, f8, f9, f0:
These constants are returned when any of the unshifted function keys are pressed. When function key (F10) is pressed, f0 is returned.

s1, s2, s3, s4, s5, s6, s7, s8, s9, s0:
These constants are returned when the shifted function keys are pressed. When (S10) is pressed, s0 is the constant returned.

Hatching

The constants in this section are used for hatching. See the "Processing Routines" chapter for more information.

htype_ignore — Indicates to procedure hatch_mode that all entities interior to the hatching boundary are ignored. In this case, each scan line creates one section of segments, at most.

htype_normal — Indicates that hatching should start and stop the hatching scan line as it crosses the edge of each entity in its path.

htype_outer — Indicates that no matter how many sections the scan line may be broken into, only the first and last section lying within the outer boundary will be drawn.

maxdash — Represents the maximum number of portions of dashes in a given scan line pattern. maxdash applies to portions of a broken scan line where the line is drawn and where the line is not drawn (the spaces). maxDash is equal to 6.

Object Snapping

Snap constants are used by the built-in variable osnap_mode, which controls the object snap mode. These constants can be added together.

For example, to set center, end point, and quick mode, use:

```
osnap_mode := osnap_center + osnap_endpoint + osnap_quick;
```

osnap_center	This constant sets object snap to recognize the center of arcs and circles.
osnap_endpoint	osnap_endpoint sets object snap to snap to the end points of the nearest line, point, Bezier curve, b-spline, arc, or associative dimension.
osnap_intsect	Object snapping recognizes the intersections of lines, circles, and arcs. Although powerful, this may significantly slow down snapping operations.
osnap_midpoint	When osnap_midpoint is on, snapping recognizes the mid point of the nearest line, arc, or associative dimension.
osnap_nearest	When osnap_nearest is set, snapping recognizes the nearest point on the nearest line, point, Bezier curve, b-spline, circle, or arc. Bezier curves and b-splines use their control points.
osnap_npoint	osnap_npoint divides the nearest line or arc into osnap_num divisions, and snaps to one of these points.
osnap_perp	When object snap uses osnap_perp, points which are perpendicular to the nearest line, arc, or circle, and which pass through the last point entered are recognized.
osnap_quad	Object snapping snaps to the quadrant points of arcs and circles. Quadrant points are the points at 0, 90, 180, and 270 degrees. When an arc does not pass through one or more of these points, the points are not used.
osnap_quick	When osnap_quick is used and osnap_intsect is not, the search for the nearest entity is abandoned after an entity within the current miss distance is found.
osnap_tan	When object snap uses osnap_tan, if the point entered is within the miss distance of an arc or circle, a possible snapping point is considered at the point of tangency of a line passing through the last point entered and tangent to the arc or circle.

Polylines

The following constants are used in the polyline routines:

pv_bulge	pv_bulge represents a polyvert which is an arc segment

between a polyvert and the next polyvert in the chain.

pv_vert	pv_vert represents a polyvert which is a straight line between a polyvert and the next polyvert in a chain.

Viewing

The following constants are used with the view management and calculation routines as well as with the function *getpointp*:

oblqelev	Represents an oblique projection calculated using the elevation oblique technique. See view_calcoblq for more information.
oblqplan	Represents an oblique projection calculated using the plan oblique technique. See view_calcoblq for more information.
vmode_all	Instructs DataCAD that any viewing projection is valid for a given function. See view_checkmode, getpointp for more information.
vmode_edit	Represents a viewing projection that the user can edit, including orthographic and parallel views. In general, data may be entered into a drawing in either an orthographic or a parallel projection where object snapping and all selection functions work properly. See getpointp for more information.
vmode_oblq	Represents oblique viewing projections.
vmode_orth	Represents orthographic viewing projections.
vmode_para	Represents parallel viewing projections.
vmode_pers	Represents perspective viewing projections.

Walls

These constants are used by some of the built- in variables to determine certain wall conditions:

wall_cap	The variable wallend is set to the constant wall_cap when the end condition of walls is to cap them.
wall_clip	wallend is set to wall_clip when the end condition for walls is to clip an intersecting wall, forming a T-intersection.
wall_none	wallend is set to wall_none when the end condition for a wall

	is to do nothing.
wall_off	The built-in variable wallson is set to wall_off when walls are not being drawn, in which case DataCAD draws simple lines.
wall_on	wallson is set to wall_on if DataCAD is drawing walls.

Miscellaneous Constants

ABSZero	This constant is equal to 1.0E-12. Useful for near zero tests.
halfpi	halfpi is defined as pi / 2.0.
maxpnts	maxpnts is the maximum number of points in a polygon, slab, or contour. 36 is the maximum number of points.
pi	pi is defined as 3.141592654. This value is useful in many mathematical computations, since angles are stored and manipulated in DataCAD in radians.
res_escape	res_escape is the counterpart to res_normal. It indicates that an entry was not completed correctly, but that a key (such as a function key) was pressed during input. See "getpoint" for more information.
res_normal	res_normal is returned by several of the input routines to indicate a valid entry.
SQRZero	This is equal to ABSZero * ABSZero.
twopi	twopi is defined as 2.0 * pi.
x	The constant x is used whenever the x coordinate of a point is needed, or the x axis is specified, such as in a rotation.
y	The constant y is the counterpart of x, and indicates the y coordinate or y axis.
z	The constant z is similar to x and y, and indicates the z coordinate or z axis.

CHAPTER 3 Data Types

The types explained in this chapter are built into the DCAL compiler. There are standard types (like integers, booleans, and reals) and more complex types (like entity, mode_type, and file) that are used to communicate with the DataCAD database and to read and write to files.

Standard Types

atrname	atrname is a twelve-character string used for the name of an attribute.
atraddr	atraddr is the logical address of an attrib. atraddr has a base type of lgl_addr. This is used mainly in reading attributes. For more information, see the "Data Routines" chapter.
bezarr	bezarr is defined as: `bezarr = ARRAY [1..20] OF point;` bezarr is used in b-spline and Bezier curve entities to store the array of defining points, which is restricted to twenty points in these entities.
boolarr	boolarr is defined as: `boolarr = ARRAY [1..maxpnts] OF boolean;`
boolarr	is used in polygon and slab entities to determine which line segments are drawn and which are not.
boolean	The type boolean can have only true and false values. Booleans are used for logical operations.
char	The type char is an eight-bit (0-255) ASCII value that uses the IBM extended ASCII character set. Character literals are enclosed in double quotes (") and use the same backslash sequences as string literals (see string).
integer	DCAL supports the primitive type integer, with values ranging from -32768 to +32767.
lgl_addr	lgl_addr is the base type for several types that involve drawing database memory addresses, including entaddr, symaddr, atraddr, pvtaddr, and viewaddr. These address types are of the same size, and essentially interchangeable. It is in their semantic usage that the source or type of the address is crucial. DCAL allows type conversion of equal size variables using

type casting. The procedure setnil and the function isnil take parameters of type lgl_addr.
See "pvtaddr" or "viewaddr" for an example of type casting using lgl_addr.

longint

The type longint is a four-byte long integer. The operations currently allowed on a longint are to convert a real to a longint (round4 and trunc4) and to convert a string to and from a longint (cvlntst and cvstlnt).

Do not perform arithmetic operations on long integers.

modmat

modmat is the built-in type for a modeling matrix. This type is used for general three- dimensional manipulations. It is defined as:

```
modmat = ARRAY [1..4, 1..4] OF real;
```

For more information, see "Modeling Matrix Routines" in the "Data Routines" chapter.

plot_type

plot_type is the type used by the plotting system for storing information pertaining to a single call. This type may not be examined.

pnt4arr

This type is used to define an entity type block. pnt4arr is defined as:

```
pnt4arr = ARRAY [1..4] OF point;
```

pntarr

pntarr is used in polygons, slabs, and contours to store the array of defining vertices. This version restricts this to 36 points in any of these entities. pntarr is defined as:

```
pntarr = ARRAY [1..maxpnts] OF point;
```

pntmat

This type is used to define a mesh surface entity. This built-in type is defined as:

```
pntmat = ARRAY [1..4, 1..4] OF point;
```

point

A point is a built-in record of reals. It is defined as:

```
point = RECORD
          x : real;
          y : real;
          z : real;
        END;
```

The individual coordinates that make up a point are accessed by:

```
VAR
   x1, y1, z1 : real;
   pt : point;
BEGIN
```

```
   x1 := pt.x;
   y1 := pt.y;
   z1 := pt.z;
```

pvtaddr

pvtaddr is the type of address used to access polyverts.

```
VAR
   vaddr   : pvtaddr;
BEGIN
   setnil (lgl_addr (vaddr));
   IF isnil (lgl_addr (vaddr)) THEN
     { vaddr is equal to nil }
   END;
```

The type pvtaddr has a base type of lgl_addr. DCAL allows for type conversion of variables of equal size with type casting.
The procedure setnil and the function isnil take parameters of type lgl_addr. They may be used with variables of type pvtaddr via type casting. In the above example, setnil and isnil are used on variables of type pvtaddr by the use of type casting to type lgl_addr.

real

The real type is an IEEE single precision, floating point number, having an approximate range of:
+0.1E-63 to +0.999999E63
-0.1E-63 to -0.999999E63

Real number literals cannot begin with a period, they must begin with a number. Thus, you must write ‘0.4’ instead of ‘.4’.

string

A string is a dynamic-length array of characters. Every string has two lengths associated with it, the maximum length (which is the length specified when the string is defined) and a current dynamic length. The built-in type string is not actually a type, but an operator that returns a string of a specified maximum length.

String literals are enclosed in single quotes ('). Inside a string literal the backslash (\) character is used to allow entry of control characters. The recognized sequences are:

\\	Backslash character (\)
\e	Escape character
\t	Tab character
\r	Carriage return
\n	Newline
\v	Vertical tab
\b	Backspace

\f Form feed
\’ Single quote
\" Double quote
\ddd ASCII character specified by octal ddd
\xdd ASCII character specified by hex dd

For example, declaring a variable
```
str : string (30);
```
creates a variable named str that has a maximum length of 30 characters. If we were to assign to that string
```
str := ’Good morning’;
```
then the current dynamic length of the string (as returned by the function *strlen*) would be 12.

Because a string is an array of characters, the individual characters of a string may be accessed through indices. If chr were of type char, then with chr := str [4]; the fourth character in str (d) would be assigned to chr.
str [0] contains the dynamic length of the string and should not be changed. Accessing str [13] in the example returns either garbage or a run-time error, because the current dynamic length of the string is 12.

str8 — str8 is a predefined string eight characters long. That is, str8 is the same as string (8).

str80 — str80 is a predefined string 80 characters long.

str255 — str255 is a predefined string 255 characters long.

symstr — symstr is a 63-character long string used for the internal name of a symbol.

symaddr — symaddr is the logical address of the definition of a symbol, similar to entaddr. symaddr has a base type of lgl_addr.

Complex Types

attrib

attrib is used for an attribute. An attribute can be attached to the drawing as a whole (system attribute), a layer, a symbol (the symbol definition), or to any entity (including an instance of a symbol). An attribute has a name and a value.

attrib is defined as:

```
attrib  = RECORD
          atrtype   : integer;
```

```
        name      : atrname;
        addr      : atraddr;

        visible   : boolean;
        positn    : point;
        txtcolor  : integer;
        txtsize   : real;
        txtang    : real;
        txtslant  : real;
        txtaspect : real;

        CASE integer OF
             atr_str   : (str : str80);
             atr_int   : (int : integer);
             atr_rl    : (rl  : real);
             atr_dis   : (dis : real);
             atr_ang   : (ang : real);
             atr_pn    : (pnt : point);
             atr_addr  : (lgladdr  : lgl_addr);
    END
```

Like symbol and entity, there are other fields that are not accessible. The name of the attribute must be unique for any given entity, symbol, or layer.

- addr is the address of the attribute.

- When the attribute belongs to an entity and is to be drawn, the field visible is set to true.

- The value of the attribute is converted to a string (using atr_2str) and displayed at positn.

- The text color, size, etc., are given by the txtcolor, txtsize, txtang, txtslant, and txtaspect fields. When the field name of the attribute is NOT visible, all the fields between positn and txtaspect, inclusive, are not saved in the database. You can change the type and visibility of an attribute with *atr_update*.

entity

The type entity is used to read and write from DataCAD's database. The DataCAD drawing database is a collection of entities on layers. Entities can be any of the DataCAD primitives: lines, arcs, circles, text, etc. entity is a complex case-variant record.

The description of entity follows. Notice that there is a variant for every type of entity. The description given here is NOT exact: other fields exist that are not given here; and fields may not overlap.

```
entity = RECORD
        enttype : integer;
        ltype : integer;
        width : integer;
```

```
spacing : real;
ovrshut : real;
color : integer;
attr : integer;
addr : entaddr;
lyr : layer;
sttyp : integer;
index : integer;
dbase : integer;
block : integer;
frstatr : atraddr;
lastatr : atraddr;

CASE integer OF

   entLin :  (  linPt1   : point;
             linPt2     : point);

   entLn3 :  (  ln3Pt1   : point;
             ln3Pt2     : point);

   entMrk :  (  mrkPnt : point;
             mrkVec : point;
             mrkTyp : integer;
             mrkSiz : integer);

   entArc :  (arcCent : point;
             arcRad : real;|
             arcBang : real;
             arcEang : real;
             arcBase : real;
             arcHite : real);

   entCrc :  (crcCent : point;
             crcRad : real;
             crcBase : real;
             crcHite : real);

   entEll :  (ellCent : point;
             ellRadx : real;
             ellRady : real;
             ellBang : real;
             ellEang : real;
             ellAng : real;
             ellBase : real;
             ellHite : real);

   entTxt :  (txtPnt : point;
             txtSize : real;
             txtAng : real;
             txtSlant : real;
             txtAspect : real;
             txtStr : str255;
             txtBase : real
             txtHite : real;
             txtFont : str8);

   entBez :  (bezNpnt : integer;
             bezPnt : bezarr;
             bezBase : real);
```

```
entBsp :  (bspNpnt : integer;
          bspPnt : bezarr;
          bspBase : real);

entBlk : (  blkPnt : pnt4arr);

entCnt : (  cntPnt : pntarr;
          cntTanpnt1 : point;
          cntTanpnt2 : point;
          cntNpnt : integer;
          cntType : integer;
          cntDivs : integer;
          cntStiff : real);

entDim : (  dimPt1 : point;
          dimPt2 : point;
          dimPt3 : point;
          dimTxtpt : point;
          dimExo1 : real;
          dimExo2 : real;
          dimExe : real;
          dimDli : real;
          dimAng : real;
          dimOvr : real;
          dimTxtsize : real;
          dimArrsize : real;
          dimArratio : real;
          dimTxtaspect : real;
          dimTxtang : real;
          dimBase : real;
          dimTxtslant : real;
          dimDis : real;
          dimInc : integer;
          dimNlpts : integer;
          dimTxtweight : integer;
          dimTictype : integer;
          dimType : integer;
          dimLeader : boolean;
          dimTih : boolean;
          dimToh : boolean;
          dimSe1 : boolean;
          dimSe2 : boolean;
          dimMan : boolean;
          dimTad : boolean;
          dimNolftmrk : boolean
          dimNorhtmrk : boolean;
          dimTicweight : integer;
          dimDINstd : boolean;
          dimTxtofs : real;
          dimFont : str8;
          dimLdrpnts : ARRAY [1..9] OF point;

entPln :  (plnFrst : pvtaddr;
          plnLast : pvtaddr;
          plnClose : boolean;
          plnBase : real;
          plnHite : real);

entSym : (  symName : symstr;
```

```
          symAddr : symaddr;
          symMod : modmat);

entPly : (  plyNpnt : integer;
          plyPnt : pntarr;
          plyIsln : boolarr;
          plyFrstvoid : entaddr;
          plyLastvoid : entaddr);

entSlb : (  slbNpnt : integer;
          slbPnt : pntarr;
          slbThick : point:
          slbIsln : boolarr;
          slbFrstvoid : entaddr;
          slbLastvoid : entaddr);

entAr3 : (  ar3Mod : modmat;
          ar3Div : integer;
          ar3Rad : real;
          ar3Bang : real;
          ar3Eang : real;
          ar3Close : boolean);

entCon : (conMod : modmat;
          conDiv : integer;
          conRad : real;
          conTip : point;
          conBang : real;
          conEang : real;
          conClose : boolean
          conTip : real);
          ( note: not sure if conTip is shown at correct position as
          it was not mentioned at all in the published manual)

entCyl : (  cylMod : modmat;
          cylDiv : integer;
          cylRad : real;
          cylLen : real;
          cylBang : real;
          cylEang : real;
          cylClose : boolean);

entTrn : (  trnMod : modmat;
          trnDiv : integer;
          trnCent : point;
          trnRad1 : real;
          trnRad2 : real;
          trnBang : real;
          trnEang : real;
          trnClose : boolean);

entDom : (  domMod : modmat;
          domDiv1 : integer;
          domDiv2 : integer;
          domRad : real;
          domBang1 : real;
          domBang2 : real;
          domEang1 : real;
          domEang2 : real;
          domClose : boolean);
```

```
entTor : (  torMod : modmat;
         torDiv1 : integer;
         torDiv2 : integer;
         torRad1 : real;
         torRad2 : real;
         torBang1 : real;
         torBang2 : real;
         torEang1 : real;
         torEang2 : real;
         torClose : boolean);

entSrf : (  srfPnt : pntmat;
         srfSdiv : integer;
         srfTdiv : integer);

entRev : (  revBang : real;
         revEang : real;
         revMod : modmat;
         revDiv1 : integer;
         revDiv2 : integer;
         revFrst : lgl_addr;
         revLast : lgl_addr;
         revType : integer);
END;
```

You can change all the fields except addr, which is the address of the entity in the database. Changing lyr and replacing the entity in the database moves the entity from one layer to another. See the "Data Routines" chapter for routines to manipulate the database. Each entity type is explained below:

- enttype is the type of the entity, such as entlin, entcrc, etc. This determines which of the case-variant sections apply.
- ltype is the line type.
- width is the line weight, with the default single line being a weight of one
- spacing is the distance over which the line type repeats itself. This is measured in world coordinates.
- ovrshut is the line overshoot distance, measured in world coordinates.
- color is the color of the entity, using the constants in the "Constants" chapter.
- attr is an integer used by DataCAD to determine what operation created the entity.
- lyr is the address of the layer that the entity is on.
- sttype, index, dbase, and block are reserved for the use of the macro writer. DataCAD does not change or examine these.

- frstatr and lastatr are pointers to the list of attributes for this entity.

- For an entity of type entLin, linPt1 and linPt2 are the two end points of the line. The z values of these points are used as the base and height of the line.

- For entLn3, the points ln3Pt1 and ln3Pt2 are the end points of the line. The line is drawn from ln3Pt1 to ln3Pt2.

- An entity of type entMrk is the basic marker or point entity. The marker is drawn at mrkPnt. mrkVec is currently unused, and may be used by the macro. The type of the marker is given by mrkTyp, and is given in the following table:

  1 square 2 "x" 3 diamond 4 dot

  When mrkTyp is any other value, nothing is drawn. This is a valid condition. mrkSiz is the size of the marker in pixels.

- An entity of type entArc has its center at arcCent. The z value of arcCent is unused. The radius of the arc is arcRad. arcBang is the beginning angle of the arc. arcEang is the ending angle of the arc. Angles are measured in radians, with 0 to the right, increasing in a counterclockwise direction. The base of the arc is at arcBase and the height is at arcHite.

- All fields of the entCrc entity are similar to the fields for entArc, but starting and ending angles are not required.

- entEll is the entity for an ellipse. The center is at ellCent. The z value is ignored. The distance from the center to the edge of the ellipse in the x direction is given by ellRadx, and in the y direction by ellRady. The starting angle of the ellipse is ellBang and the ending angle ellEang. The entire ellipse is rotated at an angle of ellAng. The bottom of the ellipse is at elevation ellBase, and the top is at ellHite.

- The data for a text string is given in the entTxt case. The text is drawn with the lower left corner of the string at txtPnt. The height of the characters is given by txtSize, measured in drawing units. The text is rotated at an angle of txtAng. The angle of slant is txtSlant, with 0 being no slant. The aspect ratio is txtAspect: 1 is normal, 2 results in narrow characters (still txtSize tall). The characters themselves are in the string txtStr. The bottom of the text is at elevation txtBase and the top is at txtHite. The font name for the string is txtFont, note that txtFont does not contain a pathname or extension.

- entBez is a Bezier curve entity. Bezier curves are defined by an array of points. The number of points is given by bezNpnt. The array of points is bezPnt. The curve is drawn at a z height of bezBase.

- entBsp is a b-spline entity. B-splines are defined the same as Bezier curves.

- The block entity, entBlk, is similar to a cube, but can be sheared in all three directions. All four points in the array blkPnt must be defined. The first point (blkPnt [1]) is the corner vertex for the block. The other three points define the three adjacent vertices on the block in any order. Since opposite faces of the block are parallel, each face is a parallelogram.

- entCnt is the definition of a three- dimensional contour curve. The node points are stored in the array cntPnt. The number of points on the curve is cntNpnt. cntType specifies the type of curve: 0 is a natural curve, 1 is a cyclic curve, 2 is a tangent curve. When cntType is 2 cntTanpnt1 is the tangent point at the start of the curve and cntTanpnt2 is the tangent point at the end of the curve. cntDivs is the number of line segments to draw between successive points on the curve. This is slightly different from the divisions as defined in other entities. cntStiff determines how stiff the contour is at the node points; it should be between 0.5 and 2.0, with 1.0 being neutral.

- An associative dimension is of entity type entDim; the most complicated entity type.

    - dimPt1 is the first point dimensioned to; dimPt2 is the second point dimensioned to; dimPt3 is the point that defines where the dimension line itself is drawn. dimPt3 is the third point entered at the DataCAD standard dimensioning menu.

    - dimTxtpt is the point where the text is drawn; it is valid only when the text is manually placed, dimMan is true.

    - dimExo1 is the offset from the first point, dimPt1, to the start of the first extension line; similarly, dimExo2 is the offset from the second point, dimPt2, to the start of the second extension line. dimExe is the extension line extension, the distance the extension line is drawn past the dimension line.

    - dimDli is the dimension line increment and is the distance each dimension line is incremented in the baseline dimensioning case; it is used when FixdDis is turned on in the linear dimensioning DimStyle menu.

    - dimAng is the angle of the dimension line and is only valid when the dimension style is rotated.

    - dimOvr is the dimension line overrun distance; the distance the dimension line extends past the extension lines.

    - dimTxtsize is the size of the text to draw.

    - dimArrsize is the arrow size, relative to the dimTxtsize; when dimArrsize is 1.0, the arrow size is the same as the text size. dimArratio is the aspect ratio of the arrows.

- dimTxtaspect is the text aspect ratio. dimTxtang is the text angle, relative to the x axis, not relative to the dimension line.
- dimBase is the z coordinate at which the associative dimension is drawn.
- dimTxtslant is the text slant angle. dimDis is the distance to use when the dimension is drawn. It is calculated when the entity is read from the database by ent_get. dimInc is used when baseline dimensions are changed at the dimensioning change menu.
- dimInc is the number given to the dimension when it was entered; the first dimension has dimInc = 1, the second has dimInc = 2, etc. When dimDli is changed, the dimension line position is recalculated using dimInc. dimNlpts is the number of leader points defined in the field dimLdrpnts; the maximum number of leader points is 9.
- dimTxtweight is the weight of the text when it is drawn.
- dimTictype is the tic mark type: for 0 arrows are drawn, for 1 tic marks are drawn, and for 2 circles are drawn.
- dimType is the dimension orientation type:
- for 0 the dimension is horizontal,
- for 1 the dimension is vertical,
- for 2 the text is aligned with dimPt1 and dimPt2,
- for 3, the dimension is rotated at angle dimAng.
- dimLeader is true when a leader line is drawn (see dimNlpts and dimLdrpnts).
- dimTih is the text-inside- horizontal flag; when text is automatically placed and fits inside the extension lines, and dimTih is true, text is horizontal; otherwise text aligns with the dimension line. dimToh is the text-outside-horizontal flag, and is similar to dimTih but applies when a leader is drawn (dimLeader is true).
- dimSe1 (suppress extension line 1) is true when the first extension line (the one that comes from dimPt1) is NOT drawn. dimSe2 is the same for the second extension line.
- dimMan is true when text is placed manually. When this is the case, text is placed at dimTxtpt.
- dimTad is the text- above-dimension-line flag; it controls placing the dimension text inside the dimension line. When dimTad is true, text is placed above the dimension line; when it is false, the dimension line is broken into two lines and text is placed inside the dimension line.

    - dimNolftmrk is a flag that controls the drawing of the left tic mark; when it is true, no left tic mark is drawn. dimNorhtmrk is a flag that controls the drawing of the right tic mark; when it it true, no right tic mark is drawn. dimTicweight is the weight used to draw tic marks.
    - dimDINstd is true when text is entered with a European version of DataCAD.
    - dimTxtofs is only used when dimMan is false and dimTad is true. When this is the case, dimTxtofs is the offset from the dimension line to the text string.
    - dimFont is the font used to draw the text.
- entPln is a polyline consisting of a collection of polyverts. The address of the first polyvert is plnFrst. The polyverts are in a linked list structure (see the discussion in this chapter on polyvert). The last polyvert is pointed to by plnLast. When the polyline is closed (that is, there is a line between the last polyvert and the first polyvert), plnClose is true. The polyline is drawn at a z base of plnBase and a z height of plnHite.

- An instance of a symbol is an entity of type entSym. The symbol name is symName; in the DataCAD template system, this is the same as the file that the symbol came from, without the .sym extension. This is not necessarily the case in a macro, however. The modeling matrix governing the placement of this particular symbol is symMod. The field symAddr is the address of the symbol of which this is an instance; this field is valid only after reading an entity from the database. This field should NEVER be altered.

- A three-dimensional polygon is of type entPly. The number of points in a polygon is plyNpnt; this number must be between 3 and 36. The points themselves are contained in the array plyPnt, indexed from 1 to plyNpnt. The corresponding array plyIsln is used to determine if a line is actually drawn on every edge of the polygon. When plyIsln [1] is true, a line is drawn from plyPnt [1] to plyPnt [2], and so on for every vertex. This method can be used to simulate much larger polygons by joining polygons but not drawing their connecting edges. plyFrstvoid is the first void in the chain entities which are voids in the polygon. plyLastvoid is the last void entity in the chain. For more information, see the "Data Routines" chapter on manipulating voids.

- entSlb is similar to entPly. All of the corresponding fields have the same meaning. The additional field slbThick determines the thickness and shear of the slab. The distance and angle from slbPnt [1] to slbThick determine the overall thickness and shear of the entire slab. While slbThick is interpreted relative to the first point of the slab, its value is in absolute, not relative, coordinates.

- The general three-dimensional arc is of type entAr3. The modeling matrix for

the arc is ar3Mod, which specifies all translations, rotations, and scalings for the arc. This allows you to have elliptical arcs. The number of divisions used when drawing the arc is ar3Div, with a maximum of 36. This number is based on a complete circle, so when 36 is specified and only one quarter of a circle is drawn, nine divisions are used. The radius of the arc is ar3Rad. The beginning and ending angles are ar3Bang and ar3Eang, respectively. When ar3Close is true, a line is drawn from the starting point to the ending point of the arc.

- entCon is the entity type for a cone. The modeling matrix attached to the cone is conMod. The number of divisions, conDiv, is similar to ar3Div. The radius of the base of the cone is conRad. The cone's center is at the origin of a local coordinate system. conTip is relative to the origin. conMat transforms the local coordinate system into world (drawing) coordinates. Note that this form of definition allows a skewed cone. The beginning and ending sweep angles are conBang and conEang, respectively. When conClose is true, the cone has a triangular polygon connecting the starting and ending sides. See also the entTrn description.

- The entity for a cylinder is described in the entCyl section. The modeling matrix is cylMod. The number of divisions used to draw the cylinder is cylDiv. The radius of the cylinder is cylRad. The length of the cylinder is cylLen. The center is at the origin of a local coordinate system. The beginning and ending sweep angle are cylBang and cylEang, respectively. When cylinder is closed, cylClose is true.

- The data for a truncated cone is listed in the entTrn section. The modeling matrix is trnMod. The number of divisions used to draw the truncated cone is trnDiv. This is similar to ar3Div. The center of the base of the cone is trnCent. The radius at the bottom is trnRad1, and at the top trnRad2. The beginning and ending sweep angles are trnBang and trnEang. When the truncated cone is closed, trnClose is true.

- entDom is the entity type used for a dome or sphere. domMod is the modeling matrix used to translate, rotate, and scale the dome. The suffix 1 on fields represents the sweep direction, and the suffix 2 represents the rise direction; domDiv1 is the number of divisions to use in the sweep direction, and domDiv2 is the number of divisions to use in the rise direction. The radius of the dome is domRad. The beginning and ending sweep angles are domBang1 and domEang1. The beginning and ending rise angles are domBang2 and domEang2, respectively. The range of values valid for domBang2 and domEang2 is from -halfpi to halfpi. To draw the top half of a dome, domBang2 would be 0, and domEang2 would be halfpi. When the dome is a closed figure domClose is true.

- The torus entity is entTor. The modeling matrix for this entity is torMod. Like the dome (entDom), the suffix 1 represents values for the sweep direction and

the suffix 2 is used for values in the roll direction. torRad1 is the sweep radius, and torRad2 is the roll radius. When the torus is closed torClose is true.

- The mesh surface entity is described in the entSrf section. The matrix of points defining the mesh is contained in srfPnt. The number of divisions to divide the surface into in each direction is srfSdiv and srfTdiv.

- entRev is the definition for a surface of revolution entity. A surface of revolution consists of a profile made up of a collection of polyverts, and a collection of data defining the orientation, type, and angles of the surface. revFrst and revLast are the addresses of the first and last polyverts (copious data) which define the surface's profile. Only the x and y coordinates of the polyverts are used in the definition of the profile, the z coordinates are ignored. revMod is a modeling matrix which describes the local frame of reference of the surface. The x and y coordinates of the polyverts describe the surface profile in the local xz plane; defining the profile in an xy local coordinate system. The axis of sweep for the surface, however, remains the local z axis, and is thus consistent with other circular entities such as cylinders and tori. revBang and revEang are the beginning and ending angles of sweep, respectively. revDiv1 is the number of segment divisions in the sweep direction, and revDiv2 is the number of segment divisions in the roll direction for any profile vertices which are defined as bulges (similar to a torus). revType may take on one of the follwoing values

  0 indicates an open surface,
  1 indicates an entity is closed at the sides, describing a solid, doughnut-like object.
  2 indicates closed at the ends, describing a solid object, and

entaddr.
Every entity has an address in the database associated with it. The address of the entity is of type entaddr. This address is used to read the entity from the database and to put the entity back into the database. entaddr has a base type of lgl_addr. An address which points to nothing is equal to nil. See the procedure isnil.

file
A file in DCAL represents a disk file, either a text or blocked data file. The file variable references a disk file. A filename can contain a drive and pathname. Use forward or backward slashes in the pathname. When backslashes are used, they must appear in the string as two backslashes, for example:

\\dcad\\dcad.exe or /dcad/dcad.exe See "string" in this section for more information.

Note that any file or device can be opened as a file. This is how you can write to the console. Just put the screen into text mode, open a text file called CON:, and write to it. You can also write to the printer in this way.

polyvert

polyvert is the type used for the copious data associated with polylines and surfaces of revolution. Polyverts belong only to entities of type polyline or surface of revolution. In either case, the polyverts are arranged in a double-linked chain and the addresses of the first and last polyvert in the chain are stored with the entity. For an entity of type entpln, the fields containing these addresses are ent_plnfrst and ent_plnlast. For an entity of type entrev, the fields containing these addresses are ent_revfrst and ent_revlast.

For entities of type polyline (entpln) and surface of revolution (entrev), any number of polyverts may be added to the entity (up to the limit of the size of the drawing file). Unlike polygons, slabs, and contour curves where there is currently a limit of 36 vertices or knots, polylines, and surfaces of revolution may have any number of polyverts associated with them.

```
polyvert  = RECORD
addr    : pvtaddr;
next    : pvtaddr;
prev    : pvtaddr
shape   : integer;
pnt   : point;
bulge   : real;
nextpnt : point;
last    : boolean;
END;
```

Each entity type is explained below:

- addr is the address of this polyvert in the database. Of course, if the polyvert was not read out of the database, the value of this address is invalid.

- next is the address of the next polyvert in the chain and prev is the address of the previous polyvert in the chain. When next and/or prev are nil, they are the last or first polyverts in the chain respectively.

- shape may have one of two constant values: pv_vert and pv_bulge. The constant value pv_vert coincides with a polyvert which consists of a point and a straight line connected to the next polyvert in the list. The constant value pv_bulge coincides with a polyvert which consists of a point and an arc connected to the next polyvert in the list.

- nextPnt contains the x, y, and z coordinates of the next polyvert in the list. In this way a typical call to polyvert_get reads not only the contents of the polyvert indicated by the address passed in polyvert_get, but also the coordinates of the next polyvert in the list.

- The field last is true when the polyvert is the last polyvert in the list (the next field is equal to nil).

- When shape = pv_vert, the bulge field is ignored. When a polyvert has a shape of pv_bulge, bulge is a real number which may take on a value between minus infinity and plus infinity. When bulge is zero, the polyvert indicates a straight line between it and the next polyvert in the list (same as pv_vert). When bulge is equal to 1.0 the polyvert defines a semi-circle to the left side of a line between the polyvert and the next polyvert in the list. When bulge is equal to -1.0 the polyvert defines a semi- circle to the right side of a line between the polyvert and the next polyvert in the list.

As you can see, a single polyvert by itself is insufficient to describe a given line or arc segment, but requires information from the following polyvert to determine the complete geometry between any two vertices. The record structure of a polyvert and the routines for managing polyverts are designed to take this into account. A variable of type polyvert contains both the coordinates of the vertex belonging to the polyvert itself as well as the coordinates of the polyvert which follows it in the chain. For the last polyvert in the list, DataCAD's polyvert management routines circle back to the top of the list to the "following" polyvert.

mode_type

Use mode_type to read through DataCAD's drawing database. This variable is used by DataCAD to keep track of what type of entities you want to read from the database. You can read through the database in many ways by using the appropriate mode_type.

By setting a variable of mode_type while searching the drawing database, you effectively restrict the search to a subset of entities. You could restrict the search to a group, selection set, or symbol; to a subset of layers; to a subset of types of entities; or to a certain region in the drawing. These mode search restrictions can be built upon each other to further limit the reading of the database to an intersection of the above subsets of entities. This technique of reading the database is one of the most important and powerful concepts in DCAL.

scanLineType

The type scanLineType defines the pattern for the scan lines that generate a hatch pattern.

```
scanLineType = RECORD
                    ang      : real;
                    origin   : point;
                    delta    : point;
                    numdash  : integer;
                    dash     : ARRAY [1..maxDash] OF real;
                    dashDraw : ARRAY [1..maxDash] OF
                                                        boolean;
{ following fields do NOT need to be set }
                    dashPart : ARRAY [1..maxDash] OF point;
                    dashTotal: real;
                  END;
```

Each entity type is explained below:

- The angle of the hatch pattern is ang.
- The hatch pattern is calculated starting at origin.
- delta contains the x and y values of the hatch pattern delta values.
- The number of dashes in the hatch pattern is numdash.
- The dash lengths are stored in the array dash.
- dashDraw is a corresponding array that determines whether each dash pattern is drawn or if it is a 'skip' in the dash pattern.
- The last two fields, dashpart and dashtotal, are used internally and need not be set.

symbol

The definition of a symbol is represented by a variable of type symbol. This is the definition of the geometry of a symbol, not an instance of one.

symbol is defined as:

```
symbol  = RECORD
             name      : symstr;
             frstent   : entaddr;
             lastent   : entaddr;
             frstatr   : atraddr;
             lastatr   : atraddr;
             min       : point;
             max       : point;
             addr      : symaddr;
             refFlag   : boolean;
             num       : integer;
          END;
```

All of the fields, with the exception of refFlag, may be examined but not modified. refFlag may be modified. Each entity type is explained below:

- name is the unique symbol name.
- frstent and lastent are pointers to the first and last entities that define the geometry of the symbol.
- frstatr and lastatr are pointers to the first and last attributes assigned to the definition of the symbol.
- min and max are the minimum and maximum extents, respectively, of the untransformed symbol

- addr is the address of the symbol.
- For a description of refFlag, see "sym_ref" in the "Data Routines" chapter.
- num is the number of instances of the symbol. This value is not constantly updated. See "sym_count". num may be read but never written to.

viewaddr

viewaddr is the type of address used to access saved views. The type viewaddr has a base type of lgl_addr. DCAL allows for type conversion of variables of equal size via type casting.

```
VAR
   vaddr   : viewaddr;
...
  setnil (lgl_addr (vaddr));
  IF isnil (lgl_addr (vaddr) THEN
     { vaddr is equal to nil }
END;
```

The procedure setnil and the function isnil take parameters of type lgl_addr. They may be used with variables of type viewaddr with type casting. In the above example, setnil and isnil are used on variables of type viewaddr using type casting to type lgl_addr.

view_type

Use variables of type view_type for two different purposes when working with the viewer and saved views. A view_type can be used to calculate, interrogate, and update the current view and projection throughout DataCAD. Secondly, a view_type represents the object which is stored and retrieved from a drawing database when using saved views. These two functions can be used with each other.

The information contained in a variable of view_type is not necessarily in a form which makes the physical interpretation of a view obvious. A view_type is designed for efficiency in performance and space requirements. Therefore, some of the fields in a view_type variable are used for more than one purpose depending upon the type of projection and the value of other fields in the view variable. Some of the fields of a view_type are derived so that changing them does not necessarily change the view directly. However, the parameters for manipulating views are flexible enough that you can create a wide variety of views including animated sequences, slide shows, and multiple viewport output displays.

```
    view_type = RECORD
                    addr      : viewaddr;
                    next      : viewaddr;
                    prev      : viewaddr;
                    projtype  : integer;
                    viewmat   : modmat;
                    clipmat   : modmat;
                    editmat   : modmat;
```

```
  inptmat   : modmat;
  clipon    : boolean;
  name      : str8;
  clipmin   : point;
  clipmax   : point;
  windxmin  : real;
  windymin  : real;
  scale     : real;
  scalenum  : integer;
  viewcent  : point;
  perseye   : point;
  persdis   : real;
  togglelyr : boolean;
  frstlyr   : lgl_addr;
  lastlyr   : lgl_addr;
  currlyr   : lgl_addr;
  flag1     : boolean;
  flag2     : byte;
END;
```

The meaning of the fields of a view_type are as follows:

- addr is the address of the view when the view is read from the chain of saved views in the database. addr is valid only when the view is read from the database using view_get, or added to the database using view_add. When the current viewing parameters were read into the view using view_getcurr, addr is undefined.

- next is the address of the next view in the chain of saved views. When the view variable is the last view in the chain of saved views, next is equal to nil. next is valid only when the view is read from the database using view_get, or added to the database using view_add. When the current viewing parameters were read into the view using view_getcurr, next is undefined.

- prev is the address of the previous view in the chain of saved views. When the view variable is the first view in the chain of saved views, prev is equal to nil. prev is valid only when the view is read from the database using view_get, or added to the database using view_add. When the current viewing parameters are read into the view using view_getcurr, prev is undefined.

- projtype is the type of projection which the view represents. projtype may take on one of the four constants: vmode_orth, vmode_para, vmode_pers, and vmode_oblq which represent orthographic, parallel, perspective, or oblique projections, respectively. The remaining fields in a view_type vary in their interpretation depending upon the value of projtype. Setting projtype to a value other than one these four constants causes an error. Never assign vmode_edit or vmode_all to projtype. Use these constants only with the function getpointp and the procedure view_checkmode.

- viewmat is the primary viewing matrix. The value and interpretation of viewmat depends upon the projection type and whether or not clipping cubes are turned on

for the view. For orthographic, parallel, or oblique projections with clipping cubes off, viewmat contains the complete and general viewing transformation. For orthographic, parallel, or oblique projections with clipping cubes on, viewmat contains the portion of the viewing transformation prior to clipping to the three-dimensional cube. When clipping cubes are turned on for the view, viewmat is normally the identity matrix and clipmat contains the complete and general postclip viewing transformation. For perspective projections, viewmat contains the portion of the perspective transformation prior to frustum or hither clipping. Clipping cubes are always ignored for perspective projections.

- clipmat is the secondary viewing matrix. The value and interpretation of clipmat depends upon the projection type and whether or not clipping cubes are turned on for the view. For orthographic, parallel, or oblique projections with clipping cubes off, clipmat is ignored but typically set to the identity matrix. For orthographic, parallel, or oblique projections with clipping cubes on, clipmat contains the complete and general post-clip viewing transformation that would otherwise be stored in viewmat as described above. For perspective projections, clipmat contains the portion of the viewing transformation after frustum or hither clipping.

- editmat is the complete and general viewing transformation regardless of whether or not clipping cubes are in effect. When the effect of clipping cubes is not considered, editmat is the complete concatenated viewing transformation. editmat is usually used during input operations where the effects of clipping cubes need not be considered until an input operation is complete. For perspective projections, editmat is the concatenation of viewmat and clipmat.

- inptmat is the mathematical inverse of editmat. inptmat is used for reverse transformations, usually during input operations in parallel projections when the editing plane is not parallel with the xy-plane in the world coordinate system. inptmat is constantly updated by DataCAD so that upon any call to view_getcurr, you may be assured that inptmat is the correct and current inverse viewing transformation.

- clipon is a Boolean flag which is true when clipping cubes are on for the view, and false when clipping cubes are off. clipon is ignored for perspective projections. Frustum or hither clipping is always in effect for perspective projections. clipmin and clipmax contain the minimum and maximum extents of the clipping cube. You must ensure that the values of clipmin are less than the respective values of clipmax. DataCAD does not automatically ensure their order. NOTE: Simply toggling clipon on or off does not set the viewing matrices for clipping cubes in effect. viewmat and clipmat must also be adjusted accordingly.

- name is the name of the view which appears on the function keys in the GotoView menus. name may be up to eight characters long, may contain any number of spaces, or may be a null string.

- windlft, windbot, scalenum, and scale collectively define the current two-dimensional window to viewport transformation. DataCAD maintains a separate two-dimensional, world coordinate, window-to-viewport transformation distinct from the general three-dimensional transformation. This is necessary for handling features such as user-defined line types and weighted lines. These display attributes are always applied after the general three-dimensional viewing transformation.

- windlft is the position of the left edge of the viewing window in two-dimensional world coordinates.

- windbot is the position of the bottom edge of the viewing window in two-dimensional world coordinates.

- scalenum is an integer number between 1 and 18 inclusive, and indicates which of the 18 currently-loaded scales is the current window to viewport scaling of the drawing. Since the user may edit these scales, or may load different sets of scales into a drawing, scalenum is not necessarily unique in any given drawing database or across drawings. scale is the actual window-to- viewport ratio which scalenum represents. When the user changes the scale corresponding to a particular value of scalenum, scale is adjusted accordingly upon the next reading or updating of the view. scalenum always supercedes and controls the value of scale. scale is derived through a look-up table.

- The value of viewcent changes depending upon the projection type of the view. When the view is a perspective projection, viewcent is the point in the view at which the viewer is looking; in this case viewcent defines the location of the picture plane relative to the eye point. The picture plane is perpendicular to a line which passes from perseye to viewcent and through viewcent. When the view is a parallel projection, viewcent is the point about which the view is rotated during Control menu operations or by using the globe. For parallel projections, viewcent becomes the origin of the grid for that view. Therefore, the location of this point can be particularly important during editing operations in parallel projections. When the view is an oblique projection, viewcent is the center of the view, and determines the center of shear for the view. viewcent is ignored for orthographic projections.

- perseye is used only for perspective projections, and ignored for all other projection types. perseye is the location of the eye point of the viewer.

- persdis is the distance between the points viewcent and perseye, and is always derived. persdis is used only for perspective projections, and otherwise ignored. Setting the field persdis has no effect upon the view.

- togglelyr, frstlyr, lastlyr, and currlyr manage layers depending upon the state of the LayrSet toggle on the 3D GotoView menu. Currently, you can't manipulate this aspect of saved views from within DCAL. Do not alter these fields of a view_type.

However, each view which is added to the list of saved views using view_add remembers which layers are on or off so you can adjust which layers are on prior to adding a view to the database. Furthermore, a call to view_update resets the list of layers for the views which are on.

- flag1 and flag2 are general purpose Boolean flags which are stored, updated, and retrieved with saved views. You may set or clear these flags. flag1 and flag2 are not protected from use by other DCAL applications, so their values may not remain the same until you change them. DataCAD's standard user interface does not currently use these fields as they are primarily intended for DCAL applications.

CHAPTER 4 Variables

The variables listed in this chapter access DataCAD's internal variables. These variables can only be passed as mode IN parameters; they cannot be used as mode OUT or mode IN OUT parameters. They can be read from and written to like regular variables. All these variables can be accessed from DataCAD.

When your macro performs a function similar to a DataCAD function (adding special door types, for instance), use the DataCAD variable for that function (door height, thickness, etc.).

I/O Variables

Plotter Variables

The variables in this section contain various user settings. These variables do not effect plotting as done through DCAL, but instead effect plotting done through DataCAD. They are available in DCAL so that a macro can be consistent with the DataCAD Plotting menu.

pltcentx : real;

The x value of the center of the current plot, as defined in world coordinates.

pltcenty : real;

The y value of the center of the current plot, as defined in world coordinates.

pltcolor : boolean

The current user setting for producing a color plot.

pltpcustx : real;

The custom x paper size in inches. Notice that the units are inches, not 1/32s of an inch which is the base unit in world coordinates.

pltpcusty : real;

The custom y paper size in inches. The following code determines the world coordinate sizes of the currently selected paper size:

```
IF pltpsize = 1 THEN
   x := 10.5;
   y := 8.0;
ELSIF pltpsize = 2 THEN
   x := 16.0;
   y := 10.0;
ELSIF pltpsize = 3 THEN
```

```
   x := 21.0;
   y := 16.0;
ELSIF pltpsize = 4 THEN
   x := 33.0;
   y := 21.0;
ELSIF pltpsize = 5 THEN
   x := 43.0;
   y := 33.0;
ELSE
   x := pltpcustx;
   y := pltpcusty;
END;
 { convert from inches to world   coordinates }
x := x * 32.0;
y := y * 32.0;
```

pltpensort : boolean;

The current user setting for pen sorting.

pltpenspeed : integer;

The setting of the plotter pen speed. Not all plotters support pen speeds set within the program.

pltpenwidth : integer;

The setting of the plotter pen width.

pltpsize : integer;

The paper size used by the plotter. The following values are valid:

1	8 x 10.5	2	10 x 16
3	16 x 21	4	12 x 32
5	33 x 43	6	Custom (see pltpcustx & pltpcusty)

pltrot : boolean;

true when DataCAD rotates the plots.

pltrotang : real;

When pltrot is true, pltrotang is the angle of rotation for the plot.

pltrotcentx : real;

The x value of the center of rotation when DataCAD is doing a rotated plot.

pltrotcenty : real;

The y value of the center of rotation when DataCAD is doing a rotated plot.

pltscalenum : integer;

The current plotter scale index setting. See the procedure scale_get for more information on scale indices.

Data Variables

Doors, Walls , and Windows

These variables access information about doors, walls, and windows. Most of these variables are accessible from the Architect menu.

centwall : boolean;

true if you are currently specifying walls by their center lines. When centwall is false, walls are drawn by one side and DataCAD asks you to point to the other side.

cut_lyr : layer;

The layer that DataCAD searches for wall lines when cut_srch is true.

cut_srch : boolean;

Controls whether the DataCAD cutout function searches for lines on the current layer (cut_srch = false) or on the layer determined by cut_lyr (cut_srch = true).

docut : boolean;

true when the procedure that places a door or window in a wall cuts the wall opening, otherwise the wall is not cut. This is the same variable as is toggled on the Architect menu.

dojamb : boolean;

true when jambs are drawn when an opening is cut. This is the same variable as is toggled on the Architect menu.

doorang : real;

The angle that determines the opening of a door. This is the same variable that is accessed from the Doors menu under Architect.

doorhgt : real;

The height of a door.

doorthk : real;

The door thickness.

doortype : integer;

The type of door being drawn at the DataCAD Door menu. Its values are:

0	Single	1	Double	2	Bi-fold
3	Sliding				

glassthk : real;

The thickness of the pane of glass that is drawn in the window routines. A thickness of zero means only one line is drawn, rather than two.

headhgt : real;

The header height for windows.

jambwth : real;

The jamb width for doors and windows.

sidedoor : boolean;

true when doors and windows are drawn by their sides (jambs). false when they are drawn by their center and one side. This variable is used for both doors and windows.

sillhgt : real;

The window sill height.

sillin : real;

The window sill width on the inside of the wall.

sillout : real;

The window sill width on the outside of the wall.

wallend : integer;

The wall end condition; valid values are:

wall_none	Nothing is done at the ends.
wall_clip	T-intersect with nearest wall.
wall_cap	Wall is capped.

wallhilite : boolean;

Controls wall highlighting when getesc is called.

wallhiliteout : boolean;

Controls whether wall highlighting highlights the inside or outside wall. true means the outside wall is highlighted when wallhilite is also true.

wallhilitecolor : integer;

wallhilitecolor is the color used for highlighting wall lines.

wallhilitewidth : integer;

The line width (weight) used for highlighting wall lines.

wallson : integer;

Determines when walls are drawn. The valid values for wallson are:

wall_on	Draw walls.
wall_off	Don't draw walls.

wall_on and wall_off are described in the "Constants" chapter.

wallwidth : real;

The current wall width being drawn.

Entity Property Variables

The variables in this section are associated with entities that are added to the drawing database.

lineattr : integer;

Contains the attribute that is assigned to added entities. This value is put into the attr field of a variable of type entity. This is used for such purposes as distinguishing hatching lines from wall boundary lines.

linecolor : integer;

The color assigned to the entities that are added to the database. This variable is assigned by layer, and changing its value changes the value for the current layer only. Changing this variable does not change the color of the layer name on the screen. You must call wrtlyr to force the layer name to change.

lineosht : real;

The current line overshoot factor. Each line is drawn this distance past its actual end point.

linespcg : real;

The current line spacing distance. It is the distance of each line pattern repeat (dot to dot, dash to dash, etc).

linetype : integer;

The line type used for entities added to the database. It may take any of the values: ltype_solid, ltype_dotted, ltype_dashed, or ltype_dotdash. In addition, any of the user-defined line types may be specified. These line types are assigned numbers begining at 4; the first user line type is given the number 4, the next 5, etc.

linewidth : integer;

linewidth is the line width (also called line weight) given to entities added to the database. This value is assigned to the field width of a variable of type entity.

Polygon Variables

The variables in this section are used in DataCAD's 2D Polygon menu.

plyrect : boolean;

Controls whether the 2D Polygon menu enters regular polygons or rectangles. When it is true the menu starts off in the polygon entry state.

polycntr : boolean;

Controls whether DataCAD adds a center point to the regular polygons it enters on the Polygons menu. The center point is added when polycntr is true.

polydiam : boolean;

true creates polygons by the diameter of the inscribing or circumscribing circle. When false, polygons are created by the center and one point on a radius.

polyinsd : boolean;

true inscribes polygons inside their defining circle. When false, polygons are circumscribed about the defining circle.

polysides : integer;

The number of sides used in the regular polygons.

polyvet : boolean;

true when the second point entered to define a polygon is at a vertex. When it is false, the second point defines a point on a face of the polygon.

Symbol Variables

The variables in this section are used when entering symbols from the template window.

symang : real;

symang is the current symbol rotation around the z axis.

symenlx : real;

symenlx is the current symbol enlargement factor in the x direction.

symenly : real;

symenly is the current symbol enlargement factor in the y direction.

symenlz : real;

symenlz is the current symbol enlargement factor in the z direction.

symexplode : boolean;

true when instances of symbols entered at DataCAD's Template menu are exploded into their constituent entities.

symzoffset : real;

The z offset of instances of symbols entered in DataCAD.

Processing Variables

Grid Variables

These variables control the grid on the current layer. Unless noted, every layer has its own copy of these variables; therefore, changing layers may cause the value of these variables to change. Changing these variables does not cause the grid to redraw.

grid1sz : integer;

The size of the secondary display grid (the grid of crosses) measured in pixels. This is

a global variable, that is, there is not a unique copy of this variable for every layer.

gridang : real;

The angle of the grid and cursor on the current layer.

gridclr : integer;

The color of the primary display grid (the grid of dots). This is a global variable, that is, there is not a unique copy of this variable for every layer.

gridclr1 : integer;

The color of the secondary display grid (the grid of crosses). This is a global variable, that is, there is not a unique copy of this variable for every layer.

gridorgx : real;

The x coordinate of the grid origin on the current layer. Changing this variable does not cause the grid to redraw. Each layer has its own grid origin.

gridorgy : real;

The y coordinate of the grid origin on the current layer. Changing this variable does not cause the grid to redraw. Each layer has its own grid origin.

gridshow : boolean;

Determines when the main grid on the current layer (the grid of dots) is drawn.

gridshowx : real;

The x spacing of the grid of dots on the current layer.

gridshowy : real;

The y spacing of the grid of dots on the current layer.

gridshw1 : boolean;

true when the secondary grid (the grid of crosses) is drawn.

gridshw1x : real;

The x spacing for the secondary displayed grid.

gridshw1y : real;

The y spacing for the secondary displayed grid.

gridsnap : boolean;

true when the cursor snaps to the snap grid.

gridsnapx : real;

The x distance of the snap grid.

gridsnapy : real;

The y distance of the snap grid.

numdivs : integer;

The number of divisions that orthographic mode is using. When numdivs = 8, orthmode snaps every 45 degrees.

orthmode : boolean;

true when orthographic snapping mode is on. When orthmode is false, orthographic snapping is off.

Object Snap Variables

The variables in this section control the current object snap modes.

osnap_mode : integer;

For a description of the values that this variable can have, see the "Constants" chapter.

osnap_num : integer;

This variable is used when one of the object snap modes currently in use is osnap_npoint. The line or arc is divided into osnap_num divisions before snapping occurs.

Text Variables

The variables in this section control text entry and display.

txtaline : integer;

txtaline determines the type of text alignment currently used when text is entered. The allowed values are:

0 Left justified 1 Center justified 2 Right justified

txtang : real;

The angle at which text is entered.

txtaspect : real;

The aspect ratio of text that is entered.

txtboxclr : integer;

The color of text boxes when text is too small to draw or when txton is false. If txtboxclr is 0, then text boxes are drawn as the color of the text entity currently being drawn.

txtboxmin : integer;

When text entities are less than txtboxmin pixels tall, text boxes are drawn instead of the text itself.

txton : boolean;

When txton is true, text is drawn when text entities are drawn, subject to txtboxmin. When txton is false, text is drawn as boxes when text entities are drawn.

txtsize : real;

The size (in world coordinates) of entered text.

txtslant : real;

The slant (in radians) of entered text.

txtweight : integer;

The line weight of entered text.

Hidden Line Removal

The variables in this section control the operation of the DataCAD hidden line removal system.

hidecolor : integer;

Controls the line color used when DataCAD adds hidden lines.

hideltype : integer;

Controls the line type used when DataCAD adds hidden lines.

hidepierce : boolean;

Controls the state of the pierce algorithm used in the hidden line removal system. When hidepierce is true, pierce calculations are performed.

hidespcg : real;

Controls the hidden line removal line spacing factor.

hidewidth : integer;

Controls the line width used for hidden lines.

Miscellaneous Variables

The variables in this section control several DataCAD functions. Some of them control how DataCAD looks, while others control what DataCAD does.

anglestyle : Integer;

Controls DataCAD's current angle style:

0	Degrees, Minutes, Seconds
1	Bearings
2	Not used
3	Decimal Degrees
4	Radians
5	Gradians

aperture : boolean;

Controls the display of the "miss distance" box around the DataCAD cursor.

arrowratio : Real;

Controls arrow aspect ratio: Length/Width.

arrowsize : Real;

Controls the size of arrows drawn in the text menu. Size is relative to current text size.

arrowstyl : Integer;

Controls the current style of arrow being drawn:

0	Open	1	Closed	2	Bridge
3	Dot	4	Tick		

atrdraw : boolean;

When atrdraw is true visible attributes that belong to entities are drawn along with the entity. You can't control this variable from the standard DataCAD interface.

autopath : Boolean;

Automatic creation and setting of path for symbol files.

bigcurs : boolean;

Controls the drawing of the full screen cursor. When bigcurs is true, the full screen cursor is drawn. When it is false, the small cursor is drawn. This variable is toggled by the plus (+) key.

boxsym : Integer;

Minimum size, in pixels, of symbols before boxes are drawn.

chamfera : real;

The length of the chamfer for the first line entered in the Chamfer menu.

chamferb : real;

The length of the chamfer for the second line entered in the Chamfer menu.

circlefact : Real;

Factor for 2D curve drawing precision.

clockwise : Boolean;

When true, angles increase in a clockwise direction.

constref : boolean;

true when the distance read out at the bottom of the screen is the distance from the constant reference point (usually the origin). When constref is false, the distance is measured from the last point entered (see refpnt).

copyflag : Boolean;

The AndCopy toggle in Move, Rotate, and Enlarge.

crcdiv1 : integer;

The default primary circle divisions used with three-dimensional circular entities such

as arcs and domes.

crcdiv2 : integer;

The default secondary circle divisions used with three-dimensional circular entities such as domes and tori.

curssz : integer;

The size, in pixels, of each arm of the small cursor.

curvecenters : Boolean;

Controls center point display for 2D curves.

dimcontrolpts : boolean;

Controls the display of associative dimension control points.

dimlimits : boolean;

When true, limits are added to dimensions.

DimMinusTol : Real;

Minus tolerance amount used with DimTolerance and DimLimits.

DimMinusTolAng : Real;

Minus tolerance amount used with DimTolerance and DimLimits for angular dimensions.

dimmon Boolean;

Controls the display of associative dimensions.

dimorient : integer;

The current default dimensioning orientation.

0 Horizontal 1 Vertical 2 Aligned 3 Rotated

dimplustol : real;

Plus tolerance amount used with dimtolerance and dimlimits.

dimplustolang : Real;

Plus tolerance amount used with DimTolerance and DimLimits for angular

dimensions.

dimticcolor : integer;

The color of arrows to add to all dimensions.

dimtolerance : boolean;

When true, tolerances are added to dimensions.

dimtxtcolor : integer

The color of text to add to all dimensions.

distancesync : Boolean;

The dissync toggle in the Settings menu.

distdelay : Integer;

Delay time for distance readout.

drawlines : boolean;

When drawlines are true, entities are drawn with their appropriate line type. When drawlines are false, entities, even those with a user-defined line type, are drawn with solid lines. This can speed up redrawing the display. This variable is toggled in the Display menu under Userline.

drawmarks : boolean;

Controls the drawing of markers every time a point is entered. These marks are drawn to the screen only and are not remembered. They disappear during the next screen refresh.

dynamic : Boolean;

The Dynamic toggle in 2D Curve Creation menus.

dynamictxt : boolean;

Controls the Dynamic toggle on the Text menu.

enlcopy : boolean;

Controls the state of the AndCopy flag in the Enlarge menu.

filcut : boolean;

When true, the lines that are pointed to during a fillet are cut back to join the resulting arc. When filcut is false, the lines are not modified.

filrad : real;

The radius of the arc used for a fillet.

findhatch : Boolean;

When false, hatch lines are ignored for entity searches in object snapping.

hatchon : Boolean;

Controls hatch line displays.

hither : real;

The distance from the viewer to the hither clipping plane when DataCAD is in a perspective projection.

inpstyl : boolean;

Note: inpstyl appears to be incorectly declared as a boolean in the procedural interface. You need to convert it to an integer to retrieve the following values (otherwise it is false for Relative Polar, and true for everything else)

The current style of entering points when you press (Spacebar). inpstyl can have the following values:

0	Relative polar coordinates
1	Absolute polar coordinates
2	Relative Cartesian coordinates
3	Absolute Cartesian coordinates

lastdist : Point;

Last distance used in Copy, Move, or Stretch.

lastenlpt : Point;

Last point used for center of 2D enlargement.

lastrotpt : Point;

Last point used for center of 2D rotation.

layerswitch : Boolean;

layerswitch is the LyrSet toggle in the Views menus.

layoutextents ; Boolean;

When true, only the extents of the drawing display in Plot Layout.

lyrsearch : boolean;

lyrsearch controls whether DataCAD searches for entities on the current layer or on every layer that is turned on. When this variable is true, the search occurs on all of the layers that are on. When it is false, the search occurs on the current layer only. This variable is toggled at the Line, Shape, Area menus.

See also the discussion of mode_init.

maxdrag : Integer;

maxdrag is the maximum number of entities to consider for Move/Drag and Rotate/Drag before using the extents box.

mrkdraw : boolean;

When mrkdraw is true, markers are drawn every time a point is entered.

mircopy : boolean;

mircopy controls the state of the AndCopy flag at the Mirror menu.

mirfixtext : boolean;

mirfixtext controls the state of the FixText flag at the Mirror menu.

missdis : integer;

The size, in pixels, of the miss distance applied to searches of the database. It is also the size of the aperture around the cursor when aperture is true.

See also ent_near.

movcopy : boolean;

movcopy controls the state of the AndCopy flag at the Move menu.

multipen : Integer;

multipen controls sending pen change commands to plotter.

1 Send pen change commands.

2 Do NOT send pen change commands.

nofloat : Boolean;

When true, DataCAD's distance output uses a fixed number of digits to the right of the decimal point as specified by the variable sigdigits.

noisy : boolean;

When noisy is set to true, DataCAD can beep. When it is false, DataCAD does not make noises.

nounits : Boolean;

When nounits is true, DataCAD's distance output does not append the currently-set scale units.

ovrdraw : boolean;

When ovrdraw is false, line overshoots are not drawn.

ratiobox : boolean;

When ratiobox is true, a rubberband box is drawn from the last entered point. The rubberband box is forced to be of a specific aspect ratio given by ratioRatio. This rubberbanding occurs during the next call to getpoint. Turn ratiobox on before calls to getpoint:

ratioratio : real;

ratioratio is the ratio of the rubberband box drawn when ratiobox is true.

regenorder : Integer;

regenorder controls the order in which the database is re-drawn:

0 Active layer first. 1 Active layer last. 2 In order.

rotcopy : boolean;

rotcopy controls the state of the AndCopy flag at the Rotate menu.

rubbx : boolean;

When rubbx is true, a rubberband rectangle is drawn from the last entered point. This rubberbanding occurs during the next call to getpoint. Turn rubbx on before calls to getpoint:

```
rubbx := true;
result := getpoint (pt, key);
IF result = res_escape THEN   ...
```

rubbx does not have to be turned off; the call to getpoint automatically turns it off.

rubln : boolean;

rubln controls the rubberband line being drawn. Use it in the same way as rubbx.

savedelay : Integer;

savedelay is the delay time for automatic file save.

scaletype : Integer;

Controls DataCAD's current scale type:

0	Architectural	1	Engineering
2	Decimal	3	Meters
4	Inches Fraction	5	Inches Decimal
6	Centimeters	7	Millimeters
8	Metric		

scrolldis : real;

scrolldis represents the fraction of the screen that moves during screen scrolling. scrolldis should be between 0.0 and 1.0, but DataCAD does not check this range.

selecttype : integer;

selecttype controls the current Entity, Group, Area, Fence state when getmode is called.

0	Group	1	Entity	2	Area	3	Fence

showinspt : Boolean;

Controls symbol insertion point displays.

showneg : Boolean;

When showneg is true, negative numbers show in the distance display.

showwgt : Boolean;

Controls line weight displays.

showz : Boolean;

Shows the current z-base and z-height values on the status line.

sigdigits ; Integer;

See nofloat. Must be in the range 0-3.

smallgrid : integer;

The size, in pixels, of the smallest grid that can be drawn.

snaplyrsearch : boolean;

Controls whether DataCAD uses layer searching when snapping to entities with object snapping.

snapquick : boolean;

Controls the quick snapping state in DataCAD. This controls the snapping to selected 3D entities to their control points.

snapsymfast : boolean;

Controls whether DataCAD searches through the description of symbols when snapping. When true, DataCAD considers the insertion points of symbols when calculating snap points.

srchquick : boolean;

Controls the quick searching state in DataCAD. This controls the searching of selected 3D entities by their control points.

srfgrid : boolean;

When srfgrid is true the grid associated with Bezier surface entities (entSrf) is drawn when the entities are drawn.

srfpnts : boolean;

When srfpnts is true, the control points associated with Bezier surface entities (entSrf) are drawn when the entities are drawn.

txtcurs : boolean;

When txtcurs is set to true before calling getpoint the text cursor is drawn instead of the normal cross hair cursor. This is the cursor that is used when DataCAD asks you to position some text on the drawing. The shape and orientation of the text cursor is subject to txtang, txtsize, txtslant, and txtaspect

txtuseplt : boolean;

Controls text scaling. When true, text is scaled relative to the current plot scale.

zbase : real;

The current value of Z-base as set when (z) is pressed.

zeroangle : Real;

The angle specifying zero for all of DataCAD's angle conventions.

zhite : real;

The current value of Z-height as set when (z) is pressed.

zuser1 : real;

The Z-height that appears on the DataCAD menus as Z-User1.

zuser2 : real;

The Z-height that appears on the DataCAD menus as Z-User2.

CHAPTER 5 Input / Output Routines

Input Routines

Function Key Routines

With the routines in this section you control the clearing, setting, and display of function key labels, which DataCAD saves in an internal record.

The function keys are numbered from 1 to 20. (F1) through (F10) are numbered 1 through 10. The Shifted (or Alted) function keys (S1) through (S10) are numbered 11 through 20.

fnkeyconv

```
FUNCTION fnkeyconv (key : integer) : integer;
```

fnkeyconv takes a key returned from an input routine and converts it to an ordinal function key number in the range 1 to 20, or returns -1 when the key is not a function key.

For instance, when a user presses (F3), getchar returns the constant f3. fnkeyconv returns 3 when passed f3. The constants returned representing the function keys are not in numerical order, whereas the numbers returned by fnkeyconv are.

```
fnkeyconv (S7)    returns 17
fnkeyconv (Esc)   returns -1
fnkeyconv (F10)   returns 10
```

To read any function key between (F1) and (S3), you can use the following code:

```
getchar (key);
key := fnkeyconv (key);
IF (1 <= key) AND (key <= 13) THEN   ...
```

lblmsg

```
PROCEDURE lblmsg (key : integer; str : string);
```

lblmsg sets the message that appears when a function key is pressed. The parameters are similar to lblset, but the string may be up to 60 characters in length.

lblset

```
PROCEDURE lblset (key : integer; str : string);
```

lblset sets the string str to the internal label associated with the label number key. This procedure does not cause anything on the display to change. Although the string passed to lblset can be any length, only the first eight characters appear. When the

string is shorter than eight characters, it is padded on the right with spaces.

lblsett

```
PROCEDURE lblsett (      key : integer;
                         str : string;
                         toggle : boolean);
```

lblsett is similar to lblset, but takes the additional parameter, toggle. When toggle is true, an asterisk is inserted in front of str before str is assigned to the function key. When toggle is false, a space is inserted before str.

lblsinit

```
PROCEDURE lblsinit;
```

This procedure clears DataCAD's internal record of the function key labels. Use it first prior to setting up a menu. Note that this procedure does NOT clear the function keys that are displayed. To do that, use lblsinit followed by lblson.

lblson

```
PROCEDURE lblson;
```

lblson takes the current internal function key labels and displays them on the monitor. The current labels are overwritten.

Mouse / Keyboard Routines

The following routines access the mouse and keyboard. They allow you to use the default DataCAD menus for certain functions.

answer

```
FUNCTION answer (i :integer) : boolean;
```

Use answer to ask the user a yes/no type of question.

Depending on the value of i, different responses are expected:
When i=0, the choices are Yes/No.
When i=1, the options are Continue/Stop.
The last recognized value is i=2, which is On/Off.

The function return value is true when the user picks the first option, false when the user picks the second option. You can use answer as in the example below:

```
wrtmsg ('Are you sure?');
IF answer (0) THEN   doit;
END;
```

dgetang

```
FUNCTION dgetang (ang: IN OUT real;
                      key OUT : integer) : integer;
```

dgetang is used to enter an angle while the macro handles the use of function keys. dgetang is similar to both getang and getpoint.

The return value is either res_normal or res_escape. The return value and use of the parameter key are the same as in getpoint. When you call dgetang, the default angle menu does not appear; however, the current angle style still applies.

```
done := false;
  and := pi;
  REPEAT
    lblsinit;
    lblset ( 1, '25 Degr');
    lblset (20, 'Exit');
    lblson;
    wrtlvl ('test');
    wrtmsg ('Enter angle for widgets: ');

  result := dgetang (ang, key);
  IF result = res_escape THEN
    IF key = f1 THEN
      and := radians (25.0);
      result := res_normal;
      { get out of loop }
    ELSIF key = s0 THEN
      done := true;
    END;
UNTIL done OR (result = res_normal);
IF NOT done THEN
  { do processing here }
```

One useful feature of all of the dget functions is the ability to use (S10) as an Exit key.

dgetdis

```
FUNCTION dgetdis (dis : IN OUT real;
                    key : OUT integer) : integer;
```

Use dgetdis to enter a distance while the macro handles the use of the function keys. dgetdis is similar to getdis and dgetang.

The return value is either res_normal or res_escape. The return value and use of the parameter key are the same as in getpoint.

dgetint

```
FUNCTION dgetint (int : IN OUT integer;
                    key : OUT integer) : integer;
```

dgetint is similar to dgetang and dgetdis, but the user enters an integer.

The return value is either res_normal or res_escape. The return value and use of the parameter key are the same as in getpoint.

dgetrl

```
FUNCTION dgetrl (rl : IN OUT real;
                 key : OUT integer) : integer;
```

dgetrl is used to enter real numbers (decimal numbers) while the macro handles the use of the function keys.

The return value is either res_normal or res_escape. The return value and use of the parameter key are the same as in getpoint.

dgetstr

```
FUNCTION dgetstr (str : IN OUT string;
                  len : integer;
                  key : OUT integer) : integer;
```

dgetstr reads a string from the keyboard, while the macro handles the function keys.

The parameter str must be initialized.

The parameter len is the maximum allowable length of the string. len can be less than or equal to (but not greater than) the maximum length of str.

The return value is either res_normal or res_escape. The return value and use of the parameter key are the same as in getpoint.

fgetlyr

```
FUNCTION  fgetlyr (offset : IN OUT integer;
                   key : IN OUT integer;
                   lyr : IN OUT layer) : integer;
```

fgetlayr works like getlyr except that a valid selection need not be made to exit this function.

Initialize offset to zero. This returns the number of times scrolled forward into layer selections. When the function return value is res_escape, key returns which key was pressed. lyr is the layer selected when the function return value is res_normal. lyr should be initialized to a valid value before calling the routine. The return values are the same as in the routine getpointp.

getang

```
PROCEDURE getang (ang : IN OUT real);
```

getang reads an angle from the keyboard.

It displays the default angle menu, and allows the user to use the function keys or type the angle from the keyboard. The angle is expressed in radians, using the DataCAD angle convention, regardless of the currently- selected type of angle display. Notice that the parameter ang is an IN OUT parameter; and must be initialized before calling getang. The current value of ang displays as a default when getang is called.

getchar

```
PROCEDURE getchar (key : OUT integer);
```

getchar reads characters from the keyboard. As discussed in the "Introduction" chapter, the DataCAD keyboard returns an escape code when you press certain keys. Of particular interest are the codes that function keys return. The rest of the escape codes (such as the arrow keys, (Home), etc.) are handled within DataCAD.

The escape codes returned by the function keys are represented by the built-in constants f0 through f9 for the function keys, and the constants s0 through s9 for shifted function keys.

NOTE: Function keys always return their respective keycodes even when nothing appears on them.

getclr

```
PROCEDURE getclr (clr : IN OUT integer);
```

getclr reads a color from the keyboard. It displays the color menu and asks the user to select from it.

getclr returns the color selected. clr should be initialized before calling getclr.

getdis

```
PROCEDURE getdis (dis : IN OUT real);
```

getdis reads a distance from the user. It displays the default distance menu and then accepts input from either the function keys or the keyboard.

The parameter dis is in DataCAD drawing units. The function keys and input strings depend on the currently-selected scale type, but the variable is always in the same units. The parameter must be initialized first since it is an IN OUT parameter and used for the default.

getesc

```
PROCEDURE getesc (key : OUT integer);
```

getesc reads in a function key from the user. Use this procedure when the user's only option is to select a function key, and has nothing to point to. This procedure is used in DataCAD's Edit and Utility menus. The user can enter walls when getesc is called.

An example piece of code that lets the user select from two different options or toggle a variable is:

```
done := false;
REPEAT
  lblsinit;
  lblset ( 1, 'Method 1');
  lblset ( 2, 'Method 2');
  lblsett ( 3, 'Fast', fast);
  lblset (20, 'Exit');
  lblson;
  wrtlvl ('Widgets');
  wrtmsg ('Enter widget options.');
  getesc (key);
  IF key = f1 THEN
    doMethod1;
  ELSIF key = f2 THEN
    doMethod2;

  ELSIF key = f3 THEN
    fast := NOT fast;
  ELSIF key = s0 THEN
    done :=true;
  END;
UNTIL done;
```

getflname

```
FUNCTION getflname (fname, path : IN OUT string;
                    ext : string;
                    entire : OUT string;
                    addext : boolean) : boolean;
```

getflname reads a filename from the user. You can scroll through the existing filenames which appear on the function keys, select a filename, or type the filename (and path if necessary).

- fname is an IN OUT variable that is the eight- character filename being read. On input it is the default filename, on exit it is the user- selected filename. fname is only the eight- character filename without the extension or path.

- path is the path where the file is located. It is this path that the default files are listed from. The user can change the path, and upon exit path may change from its input value.

- ext is the file extension of files to list; it includes the dot. Therefore, .dc3 (not dc3) is used to read in a drawing name.

- entire is the entire name, including the path, of the file that was read in. This may be a relative or absolute filename.

- addext is true when entire includes the extension ext, otherwise it is not.

- getflname returns true when the user enters a filename or selects one from the function keys. false is returned when the user selects exit, (S10). getflname does not check to see if a file exists, it merely reads in a filename.

getint

```
PROCEDURE getint (i : IN OUT integer);
```

getint is used to read an integer from the keyboard. The default integer menu appears and the user can type the number or use the function keys.

When getint returns to the macro, the parameter i contains the value that was entered. Because this parameter is an IN OUT parameter it must be initialized before being passed to getint.

getlyr

```
PROCEDURE getlyr (lyr : OUT layer);
```

getlyr reads a layer from the keyboard. Input by function key or by typing the name of a layer. You can scroll through the default layers which appear on the function keys.

NOTE: Exit, (S0), does not work in getlyr, see "fgetlyr".

getmode

```
FUNCTION getmode (action : string;
                  mode : IN OUT mode_type;
                  key : integer) : integer;
```

getmode is the function that implements the entity, group, area, fence, selection set user interface.

action is the verb that determines what to do to selected entities, for example, action might be move at a Move menu.

mode is the variable that reads the user- selected entities. This is explained in greater depth in a following section. For example:

```
done := false;
REPEAT
```

```
  lblsinit;
  lblset ( 7, 'Undo');
  lblset ( 8, 'Partial');
  lblset (10, 'Clr Undo');
  { lblson should NOT be used here }
  wrtlvl ('Erase');
  result := getmode ('erase', mode,
                     key);
  IF result = res_escape THEN
    IF key = f7 THEN
      undo;
    ELSIF key = f8 THEN
      partial;
    ELSIF key = f0 THEN
      clrUndo;
    ELSIF key = s0 THEN
      done := true;
    END;
  ELSIF result = res_normal THEN
    { note this loop is the same if the
     user picked entity, group, area,
     or selection set }
    addr := ent_first (mode);
    WHILE ent_get (ent, addr) DO
      addr := ent_next (ent, mode);
      deleteIt (ent);
    END;
  END;
UNTIL done;
```

There are several important points to note about the above example:

- When setting the labels, do not use (F1) through (F6) or (S0). These are used by getmode for the entity, group, area, fence, selection set, layer search, and exit keys.

- Do not call lblson before calling getmode. getmode does this itself, and if you call it, the functions keys blink.

- The return value and the parameter key are used as in getpoint. When the user correctly selects entities, res_normal returns. When res_normal is returned, mode reads the selected entities out of the database.

- If one of the function keys (F7) to (S9) are pressed, res_escape returns and the appropriate value is returned in key.

getpoint

```
FUNCTION getpoint (pt : OUT point; key OUT integer) : integer;
```

getpoint is used for reading the current cursor position using the mouse or keyboard.

When a valid point is entered, getpoint returns res_normal. When an escape code is

read from the keyboard getpoint returns res_escape. The escape code is returned in the variable key. key determines which function key was pressed. getpoint returns when any of the following happen:

- The left mouse button is pressed (a point is entered).
- The middle mouse button is pressed and a point is snapped to.
- The right mouse button is pressed. Pressing the right mouse button is ALWAYS the same as pressing (S10), regardless of where the cursor is pointing or what is assigned to (S10).
- The user enters a point with (Spacebar).
- A function key is pressed. See getchar for more information.

getrll

```
PROCEDURE getrll (rl : IN OUT real);
```

getrll reads a real number from the keyboard.

The default real numbers (decimal numbers) appear on the function keys. The user types a number or uses the appropriate function key.

The parameter rl must be initialized before calling getrll.

getstr

```
PROCEDURE getstr (str : IN OUT string; len : integer);
```

getstr reads a string from the keyboard. Since there is no default string menu, the function keys are not used by this procedure.

- The parameter str must be initialized.
- The parameter len is the maximum allowable length of the string. len can be less than or equal to (but not greater than) the maximum length of str.

globalesc

```
PROCEDURE globalesc (key : IN integer);
```

globalesc handles keyboard toggles, keyboard interrupts, and user-defined keyboard macros. Ordinarily, these toggles and interrupts are handled automatically by input routines which take a keystroke as input. During a call to getpoint or getmode, when a key is pressed which is not trapped by the routine itself, the key is implicitly passed on to globalesc for further processing.

For example, if the user presses the forward slash key, the implicit call to globalesc presents the WindowIn menu. globalesc determines what action to take, if any, and executes the action corresponding to the keystroke if any is required.

These calls to globalesc are automatic and implicit to all DCAL input routines except getchar and getpointp. getchar always returns the integer key code of the key pressed no matter what key it may be. All Alt, Control, Function, and high-bit keys are returned by getchar; however, no processing is done on the key code by globalesc. When the user enters the forward slash key as input, the WindowIn menu does not appear unless the DCAL program tests the input for "/" and correspondingly calls menuwindowin.

getpointp gives you the option of whether or not to automatically process keys through globalesc.

If the doesc toggle passed to getpointp is false when getpointp is called, globalesc is not called automatically and key is handled the same as in a call to getchar. In this case, getpointp has a return value of res_escape, and key contains the key code of the pressed key. You can then examine the value of key prior to passing it to globalesc. Thus you can determine what action the user intends prior to calling globalesc in your program. This command is useful with keyforceexit to clean up before forced, or early termination of a macro occurs, for example:

```
...
iores := getpointp (pnt, key, vmode_all,
false);
IF iores = res_normal THEN
  { process point }
ELSIF iores = res_escape THEN
  { process key }
  IF key = f1 THEN
    { do your action }
  ...
  ELSIF keyforceexit THEN
    { do cleanup before termination }
    globalesc (key);
  ELSE

    globalesc (key);
  END;
  globalesc (key);
END;
```

NOTE: getchar and getpointp always trap the key combination (Ctrl)-(C). This is the only key combination that can never be trapped from within a DCAL macro. When (Ctrl)-(C) is pressed, the macro immediately stops executing under all conditions. See getpointp, getchar, and keyforceexit.

Reading from the Database

Use the routines in this section to read entities from the DataCAD database. With these routines you can examine the drawing file. The routines are flexible enough to read the same information in several ways.

The entities that read the database use a variable of type mode_type. This variable controls the type of entity that is being read and where in the database the current search is occurring.

draw_mode

```
FUNCTION draw_mode (mode : IN OUT mode_type;
                    clear, dobrk, todl : IN boolean;
                    drmode : IN integer) : integer;
```

draw_mode draws a collection of entities described by the mode variable mode.

- mode must be initialized by either the procedure mode_init or mode_init1, or be a return value from the procedure getmode. draw_mode gives you complete control over how to draw the entities using any of four parameters.

- clear is a Boolean flag which, when true, indicates to clear the screen prior to drawing the collection of entities. When clear is false, the screen is not cleared first

- dobrk is a Boolean flag which, if true, indicates to DataCAD that the (Del) and (End) keys should be monitored. When dobrk is true, and either the (Del) or (End) keys are pressed, redrawing the data is interrupted. When dobrk is false, (Del) and (End) are ignored and the data is drawn to completion.

- todl is a Boolean flag which, when true, indicates to draw_mode that the hardware display list (if applicable) should be appended to during the screen refresh. When todl is false, the data is refreshed on the screen only and not appended to the display list. The display list is dependent on your graphics card and driver.

- drmode may take on one of three constant values: drmode_white, drmode_flip, or drmode_black. See ent_draw.

The return value from draw_mode indicates the status of the break keys. When the toggle dobrk is false, draw_mode returns 0. When dobrk is true, draw_mode may return -1, 0, or 1. A return value of 0 indicates that draw_mode executed to completion and neither (Del) nor (End) was pressed. A return value of -1 indicates that (Del) was pressed and drawing was interrupted. A return value of 1 indicates that (End) was pressed and drawing was interrupted. With this information, draw_mode may be placed in a loop, and (Del) and (End) monitored accordingly.

ent_first

```
FUNCTION ent_first (mode : IN OUT mode_type) : entaddr;
```

ent_first returns the address of the first entity that mode specifies.

When mode is set to read from the current layer (lyr_curr), ent_first returns the address of the first entity on the current layer. When no entities are specified by mode, ent_first returns nil.

ent_get

```
FUNCTION ent_get (ent : OUT entity;
                  adr : entaddr) : boolean;
```

ent_get reads the entity located at address adr from the database and returns true when an entity is read from that location. false returns when the address is nil or if DataCAD is unable to read an entity from that address.

The following loop is used to read the database:

```
{ first, set the mode variable up }
mode_init (mode);
mode_ss (mode, 10);

{ this part of the loop is INVARIANT
  with respect to the mode. No matter
  how many entities you are reading, be
  it one or all, the following piece of code is
  THE SAME }
addr := ent_first (mode);
WHILE ent_get (ent, addr) DO
  { do something to the entity here }
  { if the entity is deleted, call
    ent_next before it is deleted }
  addr := ent_next (ent, mode);
END;
```

ent_near

```
FUNCTION ent_near (ent : OUT entity;
                   x, y : real;
                   mode : IN OUT mode_type;
                   errmsg : boolean) : boolean;
```

ent_near is used to search the database for the entity that is nearest to the point (x, y).

- When no entity is within the miss distance of the point (x, y), false returns.
- When an entity is found, ent is set to that entity and ent_near returns true.
- The database is searched according to the variable mode, which must be initialized before calling ent_near.
- When errmsg is true, an error message prints if no entities are found within the

current miss distance. When errmsg is false, no error message prints. The value of errmsg does not affect the value that is returned by ent_near. The following code example finds the nearest entity that is a line or arc on any layer that is turned on:

```
mode_init (mode);
mode_lyr (mode, lyr_on);
mode_enttype (mode, entlin);
mode_enttype (mode, entarc);
IF ent_near (ent, x, y, mode,
    true) THEN
  ent_draw (ent, drmode_flip);
  pause (0.1);
  ent_draw (ent, drmode_flip);
END;
```

ent_next

```
FUNCTION ent_next (ent : IN OUT entity;
                   mode : IN OUT mode_type) : entaddr;
```

ent_next returns the address of the next entity specified by mode.

The entity parameter to ent_next is the last entity read from the database with mode. See also ent_get.

ent_setunused

```
PROCEDURE ent_setunused (ent : IN OUT entity; ignore : boolean);
```

ent_setunused marks or unmarks an entity as being unused when the database is read by mode_ignore. An entity should always be marked as unused and then immediately marked as used after the database is read.

When ignore is true, ent is ignored. When ignore is false, the entity is read even if mode_ignore reads the database. The following code is used to find the two nearest entities to a given point.

```
mode_init (mode);
IF ent_near (ent1, x, y, mode, true) THEN
  mode_init (mode);
  mode_ignore (mode);
  { do not read ent1 during this scan }
  ent_setunused (ent1, true);
  b = ent_near (ent2, x, y, mode, true);
  { reset ent1 as able to be read }
  ent_setunused (ent1, false);
  IF b THEN
    { look at the two entities }
  END;
END;
```

extents_mode

```
PROCEDURE extents_mode (mode : IN OUT mode_type;
                        min, max : OUT point);
```

extents_mode calculates the x, y, and z extents of any collection of entities described by the mode variable mode.

- mode must be initialized by either the procedure mode_init or mode_init1, or be a return value from the procedure getmode. Any valid mode may be used.

- min and max describe the opposite corners of the smallest three-dimensional box which contains the extents of the data. min and max are the minimum and maximum extents of this box

mode_1lyr

```
PROCEDURE mode_1lyr (mode : IN OUT mode_type; lyr : layer);
```

mode_1lyr instructs DataCAD to read the entities from only one layer, which is specified by the parameter lyr.

To set a mode variable to read all entities from the layer that a particular entity is on, use:

```
mode_init (mode);
mode_1lyr (mode, ent.lyr);
```

mode_atr

```
PROCEDURE mode_atr (mode : IN OUT mode_type; aname: string);
```

mode_atr sets the mode variable to return all entries in the drawing with an attribute attached named aname.

mode_box

```
PROCEDURE mode_box (mode : IN OUT mode_type; x1, y1, x2, y2 : real);
```

When mode_box is called, mode reads only entities that are completely inside the box specified by (x1, y1), (x2, y2). DataCAD automatically determines which corners of the box are given, so the coordinates do not have to be in any particular order.

mode_enttype

```
PROCEDURE mode_enttype (mode : IN OUT mode_type; enttype : integer);
```

By default, a mode variable returns all entities on a given layer or combination of layers. By calling mode_enttype once or more times, you can control which types of entities are read.

For example, the following code changes all lines and circles on the current layer to red.

```
mode_init (mode);
mode_enttype (mode, entlin);
mode_enttype (mode, entcrc);
addr := ent_first (mode);
WHILE ent_get (ent, addr) DO
  addr := ent_next (ent, mode);
  ent_draw (ent, drmode_black);
  ent.color := clrred;
  ent_update (ent);
  ent_draw (ent, drmode_white);
END;
```

mode_fence

```
PROCEDURE mode_fence (mode : IN OUT mode_type;
                      pnts : IN pntarr;
                      npnt : IN integer);
```

mode_fence is similar to mode_box, but is used to specify entities that are inside a polygonal fence.

The array of points is pnts. The number of points is given by npnt.

mode_group

```
PROCEDURE mode_group (mode : IN OUT mode_type; ent : IN OUT entity);
```

mode_group specifies that a group (linked collection of entities) is to be read.

ent is any entity that is in the group. The first entity read will not necessarily be ent, but ent will be one of the entities read.

mode_ignore

```
PROCEDURE mode_ignore (mode : IN OUT mode_type);
```

mode_ignore sets mode to ignore any entities in the database that have been marked unused by ent_setunused. This is useful for snapping to a point and finding the next nearest entity.

See "ent_setunused" for an example.

mode_init

```
PROCEDURE mode_init (mode : IN OUT mode_type);
```

mode_init must be called to initialize the mode variable. This procedure should be called before the mode variable is used. The default setting of the mode variable is to read all entities from the current layer.

mode_init1

```
PROCEDURE mode_init1 (mode : IN OUT mode_type);
```

mode_init1 is similar to mode_init, except the layer search type is set to the value currently being used by DataCAD. That is, when Layer Search is on in DataCAD, an automatic mode_lyr (mode, lyr_on) is performed, otherwise a mode_lyr (mode, lyr_curr) is performed on the mode variable.

mode_lyr

```
PROCEDURE mode_lyr (mode : IN OUT mode_type; lyr : integer);
```

mode_lyr sets the mode variable to read from layers in one of three ways, depending upon the value of lyr:

- When lyr is lyr_curr, entities are read from the current layer only.
- When lyr is equal to lyr_on, entities are read from all layers that are on.
- When lyr is equal to lyr_all, entities are read from the entire database.
- When lyr is 12, all all entities on locked layers are read from the database.
- When lyr is equal to 22, entities on all unlocked layers are read.
- When lyr is equal to 11, entites are read from locked layers that are on only.
- When lyr is 21, entities are only read from unlocked layers that are on.
- When lyr is equal to 10, entities are read from the current locked layre only.
- When lyr is 11, entities are only read from the current unlocked layer.

mode_ss

```
PROCEDURE mode_ss (mode : IN OUT mode_type; ssnum : integer);
```

mode_ss sets mode to read from the database those entities that are in selection set number ssnum.

mode_sym

```
PROCEDURE mode_sym (mode : IN OUT mode_type; sym : IN OUT symbol);
```

mode_sym reads the entities that are in a symbol. The entities read from a symbol are read only.

NOTE: Do not attempt to change the entities with ent_update.

Additional Input Routines

The declarations for the routines in this section are not built-into the DCAL compiler, but require the inclusion of the file _input.inc.

The routines in this section are used to enter data or objects in more complex ways than by using getpoint alone. These routines complement the functions getpoint and getmode. They are similar in operation, but significantly increase the functionality of a macro's user interface. See also getpolyline.

getarc

```
FUNCTION getarc (msg : IN string;
                 init : IN OUT boolean;
                 center : OUT point;
                 radius, bang, eang : OUT real;
                 key : OUT integer) : integer;
```

getarc allows the user to input an arc using one of seven different methods with a single call to a DCAL routine. This is the same function used throughout DC-Modeler for entering circular shapes such a cylinders and domes. getarc automatically handles dragging as required, clockwise/ counterclockwise toggling for two-point arc entry, changing of input modes, and the effects of changing the current color upon the input sequence.

- init is a Boolean flag indicating whether getarc should initialize itself or not. Since getarc is reentrant, init should be set to true prior to the capture of each new arc. init immediately sets to false after the first invocation of getarc and uses this fact to determine whether or not any particular call to getarc is reentrant or not.

- msg is the message with which getarc prompts the user during the input sequence. It is the string which follows the prompt, "Enter ... point of ".

- getarc returns res_normal when a valid arc is entered. When res_normal returns, the description of the arc is contained in the fields center, radius, bang and eang. center is the center point of the arc. The z-coordinate of center is the value of Z-Base upon exit from the routine. radius is the radius of the arc. bang and eang are the beginning and ending angles of the arc respectively. begang and endang have been properly normalized prior to exit. Normalized angles are always positive, and bang is always smaller than eang.

- When a function key is pressed, res_escape returns. In this case, key should be processed the same as it would with a call to getpoint and contains the keycode of the function key pressed. getarc uses function keys (Fl) through (F8), and (S0). The calling routine may use the remaining function keys. As with getmode, you need only execute lblsinit; lblson is performed automatically by getarc. The following example acts as a template for the use of getarc:

```
VAR

  key     : integer;
  result  : integer;
  done    : boolean;
  center  : point;
  radius  : real;
  bang    : real;
  eang    : real;
...

  init := true;
  REPEAT
  ...
    result := getarc ('Enter arc.',
      init, center, radius,
      bang, eang, key);
    IF result = res_escape THEN
      IF key = s0 THEN
        done := true;
      END;
    ELSIF result = res_normal THEN
      { A valid arc has been captured
        and is described by center,
        radius, bang, and eang. }
    END;
  UNTIL done;
```

getpointp

```
FUNCTION getpointp (pnt : OUT point;
                    key : OUT integer;
                    vmode : IN integer;
                    doesc : IN boolean) : integer;
```

getpointp is used in place of the function getpoint when you want to explicitly control the viewing projection or to trap any calls to globalesc. The rest of getpointp operation is identical to that of getpoint.

- getpointp returns an integer value equal to either res_normal or res_escape. When a valid point is entered the return value is res_normal.

- pnt contains the point entered in world coordinates.

- When one of the function keys is pressed, the return value is res_escape. When a keyboard interrupt is pressed doesc is false. In this case, key contains the value of the function key which was pressed ((Fl) thru (F0) or (Sl) thru (S0)), or the keycode of the key pressed.

- vmode may be one of the constants vmode_orth, vmode_para, vmode_pers, or vmode_oblq indicating that the allowable viewing projection for the call to getpointp is orthographic, parallel, perspective, or oblique, respectively. When the current viewing projection does not correspond to this projection, the screen automatically refreshes in the most current view in the appropriate projection.

Effectively, getpointp executes a call to view_checkmode prior to allowing input.

- vmode may be set to either of the constants vmode_edit or vmode_all. vmode_edit indicates that either an orthographic or a parallel projection is allowable. When editing in three dimensions, you can use either of these two projections for many input and editing operations. When the current projection is either orthographic or parallel, no action is taken. If the current projection is not either of these, then the most recent view in these two projections is used. vmode_all indicates that any viewing projection is allowable.

- doesc is a flag which indicates whether or not getpoint should handle keyboard interrupts automatically. When doesc is true, keyboard interrupts are handled from within getpointp just as they are with getpoint. When doesc is false, keyboard interrupts are passed back to the calling macro. In this case, key contains the key code of the key which would normally be passed to the procedure globalesc. You can examine the value of this key and take action accordingly, or pass the key explicitly to the procedure globalesc.

The following example calls getpointp allowing only for orthographic or parallel projections. Handling of globalesc by getpointp is disabled. When (Esc) is pressed, a message prints and no further action is taken. Otherwise, control is explicitly passed to globalesc.

```
CONST
  esc = 27;

VAR
  key     : integer;
  result  : integer;
  curs    : point;
  done    : boolean;
...

  result := getpointp (curs, key, vmode_edit, false);
  IF result = res_escape THEN
    IF key = s0 THEN
      { Exit control loop. }
      done := true;
    ELSIF key = esc THEN
      wrterr ('ESC pressed.
        No action taken.');
        { Do nothing. }
    ELSE
      { Call globalesc explicitly. }
      globalesc (key);
    END;
  ELSIF result = res_normal THEN
    { Process input here the same as when using getpoint. }
  END;
```

getpoly

```
FUNCTION getpoly (msg : IN string;
                  init : IN OUT boolean;
                  pnt : IN OUT polyarr;
                  npnt : IN OUT integer;
                  key : OUT integer) : integer;
```

getpoly allows the user to input a polygon by a single call to a DCAL routine. Since polygons have a wide variety of uses, getpoly simplifies and standardizes a user interface for polygon input.

getpoly automatically handles dragging as required, partial drawing of the polygon when the screen is refreshed or the current color changed, and going backwards through the input process at the user's request.

init is a Boolean flag indicating whether or not getpoly should initialize itself. Since getpoly is reentrant, set init to true prior to the capture of each new polygon. init immediately sets to false after the first invocation of getpoly and uses this fact to determine whether any particular call to getpoly is reentrant or not. msg is a string which is concatenated to the end of the input request, "Enter the ... point of the ", by getpoly.

getpoly returns res_normal when a valid polygon is entered. When res_normal is returned, the variable pnt contains the array of vertices for the polygon. The z-coordinates of the polygon are set to Z-Base, and may not all be the same depending upon whether the user changed Z-Base during the operation. npnt is the number of vertices the polygon contains. When a function key is pressed or the user selects (Cancel), res_escape returns. In this case, key should be processed identically to a call to getpoint and contains the keycode of the function key pressed. getpoly uses function keys (S7), (S8), and (S0). The calling routine may use the remaining function keys. As is the case with the function getmode, you need only execute lblsinit; lblson is executed automatically by getpoly. The following example acts as a template for the use of getpoly:

```
VAR
  key     : integer;
  result  : integer;
  done    : boolean;
  pnt     : polyarr;
  npnt    : polyarr;
...

  init := true;
  REPEAT
  ...
    lblsinit;
    result : = getpoly ( ' to enter ',
      init , pnt , npnt , key);
    IF result = res_escape THEN
      IF key = s 0 THEN
        done := true;
      END;
    ELSIF result = res_normal THEN
```

```
      { A polygon has been captured.
        pnt and npnt contain the
        description of the polygon. }
    END;
    UNTIL done;
```

inputat

```
PROCEDURE inputat ( col, row : IN integer );
```

inputat controls the screen location of input for the functions dgetstr, dgetint, dgetrl, dgetdis, and dgetang.

Ordinarily, these input functions operate only on the message line (the bottom line of the screen) and the location of input is at the end of the last issued wrtmsg procedure. inputat makes it possible to obtain input using these functions at any location on the screen. col is the column (in characters) in which the input field begins. row is the row in which the input field exists. Currently, col must be an integer between l and 80 inclusively, and row must be an integer between 1 and 25 inclusively.

inputat can create formlike input screens when used with the procedure printstr, and the input functions described above.

In the following example, dgetstr is called, but input is directed towards the center of the screen using inputat:

```
VAR
  str     : str80;
  key     : integer;
  result  : integer;
...

  str := '';
  printstr ( 'Enter distance: ',
    15, 10, clrblue, O, false);
  inputAt (31, 10);
  result : = dgetstr ( str, len, key);
  IF result = res_normal THEN
    {Input string captured. }
  ELSIF result = res_escape THEN
    { Function key pressed. }
END;
```

inputwhere

```
PROCEDURE inputwhere ( col, row : OUT integer );
```

inputwhere returns the current input location which is used during the next call to dgetstr, dgetdis, etc.

See "inputat".

keyforceexit

```
FUNCTION keyforceexit (key : IN integer) : boolean;
```

keyforceexit determines when a keyboard interrupt should cause control to pass out of the calling macro to some other point in the program.

keyforceexit is used with getpointp. When getpointp is called with doesc set to false, the key passed back to the calling macro may be examined using keyforceexit. keyforceexit returns true when (;) is pressed. Pressing (;) exits the user from the current process and passes control to the main Edit menu.

Knowing when a macro is to be exited, allows the calling macro to clean up prior to releasing control. If you drew temporary data on the screen, keyforceexit allows you to know if the data should be undrawn before passing control to globalesc and subsequently out of the macro.

keyforceexit does not return true when the user presses (Ctrl)-(C). When (Ctrl)-(C) is pressed, the macro immediately stops executing under all conditions.

Dragging Routines

The following routines may optionally be used during input operations which use the functions getpoint and getpointp to drag an object on the screen. Many of DataCAD's standard editing functions either drag existing data as it is being manipulated, or drag a shape which is representative of the operation being performed. The routines described here provide for much of this functionality from within a DCAL macro.

- All coordinates (points) are in absolute world coordinates unless specified otherwise.

- Some of the dragging routines provide for defining the location of the cursor relative to the geometry of the dragged shape independently so that relative world coordinates may be used.

- All of the dragging routines take a color parameter. The color parameter should be specified using one of the 15 predefined constants for DataCAD's standard colors. Alternatively, the color may be specified with the built-in variable linecolor so that the current line color is used. Note that linecolor varies on a layer by layer basis.

- Many of the dragging routines also take an ortho parameter. This parameter is a Boolean flag which indicates whether or not the dragging routine should consider orthographic snapping. When the ortho parameter is true, orthographic snapping is

enabled only if the user enabled orthographic snapping via the O keyboard interrupt. When the ortho parameter is false, orthographic snapping is not enabled regardless of whether the user enabled orthogaphic snapping via the O keyboard interrupt or not.

drag2pt

```
PROCEDURE drag2pt (ptl, pt2 : IN point;
                   ortho: IN boolean;
                   clr: IN integer);
```

drag2pt drags two lines with the cursor. drag2pt is used in DataCAD's standard user interface by the Fence function for selecting entities during an editing operation.

- One line extends from ptl to the cursor and the other line extends from pt2 to the cursor. ptl and pt2 are defined in absolute world coordinates.

- clr is the color of the lines.

- ortho indicates when dragging is sensitive to the orthographic keyboard interrupt.

dragbar

```
PROCEDURE dragbar (pt : IN point;
                   lftofs, rhtofs, pntofs, curofs : IN real;
                   ortho : IN boolean;
                   clr : IN integer);
```

dragbar drags a rectangle by fixing one end of the rectangle and moving the other end of the rectangle with the cursor.

- pt is the point about which the rectangle is rotated. The dimensions of the rectangle are defined using offsets relative to the point pt and the cursor.

- pntofs is the distance between pt and the end of the bar nearest pt. When pntofs is zero, the end of the bar coincides with pt.

- curofs is the distance between the cursor and the end of the bar nearest the cursor. When the end of the bar coincides with the cursor, curofs is zero.

- lftofs is the distance from a line extending from pt to the cursor to the left side of the rectangle.

- rhtofs is the distance from a line extending from pt to the cursor to the right side of the rectangle.

- clr is the color of the polygon; and

- ortho indicates when dragging is sensitive to the Orthographic keyboard interrupt.

dragboxmove

```
PROCEDURE dragboxmove (pt, min, max : IN point; clr : IN integer);
```

dragboxmove drags a rectangular box whose sides are parallel with the x and y axes of the screen. dragboxmove is used in DataCAD's standard user interface by the Move/Drag function when the number of entities to drag exceeds Maxlines.

- min and max are the lower left and upper right corners of the box respectively.

- pt is a point in the same coordinate system as min and max and describes the location of the cursor for dragging the box. When the coordinates of pt are identical to min, for example, the box is dragged by its lower left corner.

- clr is the color of the box.

dragcrc3

```
PROCEDURE dragcrc3 (ptl, pt2 : IN point;
                    doline, ortho : IN boolean;
                    clr : IN integer);
```

dragcrc3 drags a circle defined by three points. dragcrc3 is used in DataCAD's standard user interface by the Curves/3Pt Circ function.

- Two of the points are specified by ptl and pt2; the third point is defined by the location of the cursor. ptl and pt2 are defined in absolute world coordinates. When doline is true, a line is drawn extending from ptl to pt2. When doline is false, this line is not drawn.

- clr is the color of the circle.

- ortho indicates when dragging is sensitive to the Orthographic keyboard interrupt.

dragdia

```
PROCEDURE dragdia (pt : IN point;
                   doline, ortho: IN boolean;
                   clr: IN integer );
```

dragdia drags a circle by extending a line across its diameter. dragdia is used in DataCAD's standard user interface by the Curves/Dia Circ function.

- pt is the point from which to drag the line. The circle passes through the point pt and the cursor. When doline is true, a line is drawn from pt to the cursor bisecting the circle. When doline is false, this line is not drawn.

- clr is the color of the circle.

- ortho indicates whether dragging is sensitive to the Orthographic keyboard interrupt.

dragmodemove

```
PR0CEDURE dragmodemove (mode : IN OUT mode_type; pt : IN point);
```

dragmodemove drags any collection of entities which exist in the drawing database. dragmodemove is used in DataCAD's standard user interface in the Move/Drag function.

- mode is a mode_type variable which must be initialized and set using one or more of the mode routines. The definition of the mode variable is important to the performance of dragmodemove; some modes scan the database more efficiently than others. Set mode using the procedure mode_ss. Modes which are defined as selection sets are one of the more efficient ways to read the database, and result in maximum efficiency of dragmodemove. Adversely, modes defined using mode_fence or mode_box are more computationally intensive and result in poorer dragging performance by dragmodemove. Because dragging is dependent upon the speed and performance of each particular computer system, it's not a good idea to specify a mode which refers to too many entities in the database. Certain entities such as mesh surfaces and surfaces of revolution may require the generation of many line segments, seriously degrading the performance of dragmodemove. When specifying mode, use care as to how the mode variable is defined as well as the type of entities to which mode refers.

- pt is a point in world coordinates which represents the location of the cursor relative to the data represented by the mode variable.

dragmoderot

```
PROCEDURE dragmoderot (mode : IN OUT mode_type;
                       cent, ofs, ref : IN point);
```

dragmoderot drags a collection of entities which exist in the drawing database by rotating them about a point.

- mode is a mode_type variable which must be initialized and set using one or more of the mode routines.

- cent is a point in absolute world coordinates which defines the center of rotation relative to the entities in the database described by the mode variable.

- ofs is a relative offset from cent describing a point to which the center of rotation maps. The entities are rotated about a point which is computed by adding the

offset ofs to the point cent.

- ref is a point in absolute world coordinates about which the cursor rotates and which is used as a reference point for calculating the rotation angle of the collection of entities. cent, ofs, and ref may be identical in some applications. For example, if you rotated a collection of entities about a specified point named pt, cent and ref would be set equal to pt, and the coordinates of ofs would be set to zero.

Refer to "dragmodemove" regarding the specification of the mode_type variable. These notes apply equally to procedure dragmoderot. dragmoderot is not currently used in DataCAD's standard user interface but is provided for use by DCAL applications.

dragply

```
PROCEDURE dragply (pnt : IN pntarr;
                   startidx, endidx : IN integer;
                   ref : IN point;
                   close, ortho : IN boolean;
                   clr : IN integer);
```

dragply drags a polygon of arbitrary shape containing up to 36 vertices.

- pnt is a variable of type pntarr and contains the vertices of the polygon. The actual values of the z-coordinates of the polygon are not critical since dragging is a two-dimensional operation, but they must be initialized to valid real numbers.

- startidx and endidx are the starting and ending indices of the polygon respectively. startidx is typically equal to 1, but does not have to be.

- ref is a point in world coordinates which indicates the location of the cursor relative to the points contained in array of vertices pnt. l When the polygon is closed, close is true. When the beginning and ending vertices of the polygon are not joined, close is false.

- clr is the color of the polygon.

- ortho indicates when dragging is sensitive to the Orthographic keyboard interrupt.

dragply is used in DataCAD's standard user interface when inserting symbols with the DynmRot toggle turned off.

dragplyrot

```
PR0CEDURE dragplyrot (pnt : IN pntarr;
                      startidx, endidx : IN integer;
                      cent, ofs, ref : IN point;
                      close, ortho, doline : IN boolean;
```

```
                        clr : IN integer);
```

dragplyrot drags a polygon of arbitrary shape containing up to 36 vertices by rotating the polygon about a point.

- pnt is a variable of type pntarr and contains the vertices of the polygon. The actual value of the z-coordinates of the polygon is not critical since dragging is a two-dimensional operation, but they must be initialized to valid real numbers.

- startIdx and endIdx are the starting and ending indices of the polygon respectively. startIdx is typically equal to 1, but does not have to be.

- cent is a point relative to the coordinates of the polygon about which the polygon rotates.

- ofs is the relative distance between cent and the point in world coordinates to which cent maps, and about which the polygon rotates.

- ref is a point in absolute world coordinates about which the cursor rotates and is used as a reference for calculating the rotation angle of the polygon. Cent, ofs, and ref may be identical in some applications. If you rotate a polygon which currently exists in the database about its first vertex, cent and ref equal the coordinates of the first vertex of the polygon, and the coordinates of ofs are set to zero.

- ortho indicates when dragging is sensitive to the Orthographic keyboard interrupt.

- When doline is true, a line is drawn between the point ref and the cursor. When doline is false, this line is not drawn.

- clr is the color of the polygon.

dragplyrot is used in DataCAD's standard user interface when inserting symbols with the DynmRot toggle turned on.

<u>dragrad</u>

```
PROCEDURE dragrad (pt : IN point;
                   doline, ortho : IN boolean;
                   clr : IN integer);
```

dragradis drags a circle by extending a line from the center point of the circle to a point on the circle's circumference.

- pt is the center point of the circle; and the circle passes through the cursor.

- When doline is true, a line is drawn extending from the center of the circle to the cursor. When doline is false, this line is not drawn.

- clr is the color of the circle.

- ortho indicates when dragging is sensitive to the Orthographic keyboard interrupt. dragRad is used in DataCAD's standard user interface by the Curves/Rad Circ function.

Output Routines

Display Routines

Use the routines in this section to write to the DataCAD command line area, to control the screen mode, and to control which parts of the drawing appear in the drawing viewport.

currwndw

```
PROCEDURE currwndw (lowleft, upright : IN OUT point);
```

This procedure sets lowleft and upright to be the lower left and upper right corners, respectively, of the current window. The two points are in world coordinates.

drawcursor

```
PROCEDURE drawcursor (pt : point);
```

drawcursor is used for debugging.

It draws a cursor at the given world coordinates using drmode_flip, so that calling drawcursor twice with the same point restores the screen to its original state.

extents

```
PROCEDURE extents (min, max : OUT point; recalc : IN boolean);
```

extents calculates the extents of the current drawing. extents returns a set of values that reflect only the layers which are currently turned on.

- min is a point which contains the x, y, and z- coordinates of the lower left corner of the bounding box. max is a point which contains the x, y, and z-coordinates of the upper right corner of the bounding box.

- When the flag recalc is true, the extents of the drawing are recalculated by examining each entity, and updating the extents which are saved with each layer prior to calculating the overall extents of those layers which are turned on. When the flag recalc is false, these calculations are not made, and the extents stored with each layer are assumed to be correct. DataCAD constantly updates the extents of each layer when entities are added to the drawing database, or during editing operations which cause the extents of a layer to become larger. When an editing or erase operation causes the extents of a layer to become smaller, the extents stored for each layer may not correctly reflect the actual extents of the entities on those layers. In this case, recalculation of the drawing extents is required to correctly update the extents for each layer.

grafmode

```
PROCEDURE grafmode;
```

grafmode sets the display into graphics mode. The display is confusing, however, until redrawall is called.

icons_draw

```
PROCEDURE icons_draw (turnon : boolean;
                      redraw : boolean); BUILTIN 232;
```

Set turnon variable to true to turn toolbar on, false to turn it off. If redraw is true, which you would want in most cases, then the toolbar is refreshed as it is turned on/off, otherwise it is not refreshed.

Note that you will need to declare the procedure (including ‘BUILTIN 232’) in your code as it is not currently included in any of the standard include files.

icons_present

```
FUNCTION icons_present : boolean; BUILTIN 231;
```

Returns true if the icon toolbar is on, false if it is off.

Note that you will need to declare the procedure (including ‘BUILTIN 231’) in your code as it is not currently included in any of the standard include files.

pixsize

```
FUNCTION pixsize : real;
```

pixSize returns the size, in world coordinates, of a pixel at the current display scale.

popview

```
PROCEDURE popview;
```

popview pops a view off DataCAD's internal stack of 2D views. This is identical to pressing (P).

printstr

```
PROCEDURE printstr (str : string;
                    col, row, color, cursor : integer;
                    inverse : boolean);
```

printstr draws a string anywhere on the screen in graphics mode. This is the same procedure that is used internally to write all of the function keys and messages.

- str is the string to be written.
- col and row are the character coordinates of the position where the string is written. col is in the range 1 to 80 and row is in the range 1 to 25, with (1, 1) being the upper left of the screen.
- color is the color in which the string appears.
- cursor is the position in the string where the inverse cursor appears. When cursor is zero no cursor appears in the string (the typical case). Not all display drivers support a cursor in the string.
- When inverse is true, the string is in inverse video.

See also "inputat" in this chapter.

pushview

```
PROCEDURE pushview;
```

pushview pushes the current view onto DataCAD's internal stack of 2D views. This is the procedure DataCAD calls before it performs windowin.

redraw

```
PROCEDURE redraw;
```

redraw is like regen, but on boards with a display list, the drawing is redrawn from the display list and not from the database. On cards without a display list, redraw is the same as regen, that is, the drawing is redrawn from the database. In general, when you want to redraw the drawing, call redraw.

redrawall

```
PROCEDURE redrawall;
```

redrawall redraws everything on the graphics screen after clearing the screen, including the messages, function keys, and the drawing.

regen

```
PROCEDURE regen;
```

regen redraws the entire drawing (all layers that are on) from the database. On boards with a display list, this is equivalent to the (U) key. On other boards, this is the same as the (Esc) key.

textmode

```
PROCEDURE textmode;
```

textmode sets the graphic display into text mode. Any of the input routines with the exception of getpoint (and its derivatives) work while in text mode.

toview

```
PROCEDURE toview (num : integer);
```

toview switches to one of DataCAD's ten saved views. num must be between 1 and 10 inclusively.

vwptclear

```
PROCEDURE vwptclear;
```

vwptclear erases the portion of the screen where the drawing appears. When the template viewport is on, the screen is not cleared.

windowin

```
PROCEDURE windowin (lowleft, upright : point);
```

windowin zooms as close as possible to the two points passed to it using the available scales. The points are in world coordinates.

- lowleft and upright can be any two diagonally-opposite corners of the new window.

  wrterr

  ```
  PROCEDURE wrterr (str : string);
  ```

  wrterr writes its argument on the error line of the monitor, overwriting any previous error message. The error line is the second line from the bottom of the screen.

  wrtltype

  ```
  PROCEDURE wrtltype;
  ```

  wrtLtype updates the line type descriptor at the lower left corner of the screen. Changing the value of the built-in variable linetype does not automatically update this descriptor; wrtltype must be called explicitly to perform this function.

  wrtlvl

  ```
  PROCEDURE wrtlvl (str : string);
  ```

  wrtlvl writes out str as the current program level. This is the eight-character string

on the bottom line next to the message string. When str is more than eight characters long, only the first eight characters are used. When str is less than eight characters long, it is padded on the right with blanks. The previous level string is overwritten.

wrtlyr

```
PROCEDURE wrtlyr;
```

wrtlyr writes out the name of the current layer in the lower left corner of the screen. On a color monitor, the layer name appears in the current color.

wrtmsg

```
PROCEDURE wrtmsg (str : string);
```

wrtmsg writes its argument on the bottom line of the display, overwriting the previous message.

Use this procedure to write prompts to the screen for input routines:

```
  wrtmsg ('Enter number of divs: ');
  getint (divs);

  getout := false;
  REPEAT
    wrtlvl ('Stairs');
    wrtmsg ('Point to center.');
    result := getpoint (cent, key);
    IF result = res_escape THEN
      IF key = s0 THEN
        getout := true;
      END;
    END;
  UNTIL (result = res_normal) OR
        getout;
```

wrtscl

```
PROCEDURE wrtscl;
```

wrtscl writes out the current scale in the lower left corner of the monitor.

wrtss

```
PROCEDURE wrtss;
```

wrtss updates the selection set indicator in the lower left hand corner of the screen to indicate the currently active selection set and whether or not selection set appending is active. wrtss is typically used immediately after a call to a selection set routine.

See "Selection Set Routines" for more information.

wrtstat

```
PROCEDURE wrtstat;
```

wrtstat updates the SWOTHLUD status indicator at the lower left corner of the screen.

Changing the value of one of the built-in variables such as ortho snapping, or walls on/off does not automatically update this indicator; wrtstat must be called explicitly to do this.

Plotting Routines

The declarations for the routines in this section are not built-into the DCAL compiler, but require the inclusion of the file _plot.inc.

The following routines invoke DataCAD's plotting system from within a DCAL macro. The plotting interface to DCAL is extremely flexible without compromising the performance or usability of the system. In fact, the plotting system available through DataCAD is a subset of the plotting system available through DCAL.

plot_close

```
PROCEDURE plot_close; (plot : IN OUT plot_type);
```

plot_close closes the plotting subsystem.

* plot must be the same variable passed to plot_open.

plot_mode

```
PROCEDURE plot_mode (plot : IN OUT plot_type;
mode : IN OUT mode_type;
multipen : IN boolean;
pensort : IN boolean;
colors : IN boolean;
vwptMin : IN point;
vwptMax : IN point;
wndwMin : IN point;
wndwMax : IN point;
center : IN point;
ang : IN real);
```

plot_mode does the actual plotting of entity data to the plotter or file. plot_mode may be called multiple times between calls to plot_open and plot_close.

- plot is the variable initialized by plot_open. The entities to plot are determined by mode, just as in any reading of the database.

- multipen controls whether or not a multiple pen plot is being performed.

- pensort controls the sorting by pen option.

- colors controls whether or not a color plot is being performed.

- vwptmin and vwptmax control the area on the plotter where the plotted output appears. Only the x and y coordinates of these points are used; however, z-coordinates should be initialized to a real value.

- wndwmin and wndwmax are the world coordinates that are plotted into the area defined by vwptmin and vwptmax (again, only the x and y values are used). Notice that the ratio of x and y values in vwptmin and vwptmax does not have to match the ratio of x and y values in wndwmin and wndwmax, but typically you want the ratios to be the same.

- center is the center of rotation for a rotated plot. center should always be initialized to some value, even when no rotation is specified. center is specified in world coordinates (that is, relative to wndwmin and wndwmax).

- ang is the angle of rotation for a rotated plot.

plot_mode1

```
PROCEDURE plot_mode (plot : IN OUT plot_type;
                     mode : IN OUT mode_type;
                     multipen : IN boolean;
                     pensort : IN boolean;
                     colors : IN boolean;
                     vwptMin : IN point;
                     vwptMax : IN point;
                     wndwMin : IN point;
                     wndwMax : IN point;
                     center : IN point;
                     ang : IN real);
```

plot_mode1 works just like plot_mode except that the pen is not put away when the plot is completed.

plot_open

```
FUNCTION plot_open (penwidth : IN integer;
                    penspeed : IN integer;
                    maxx : IN real;
                    maxy : IN real;
                    tofile : IN boolean;
                    numplots : IN integer;
                    fname : IN string;
```

```
                    plot  : IN OUT plot_type) : integer;
```

plot_open opens the plotting subsystem. It must be called before a plot is started.

- penwidth is the pen width (in plotter units) DataCAD uses to plot multiple weight lines.
- penspeed is the pen speed DataCAD passes to the plotter driver. Not all plotter drivers support pen speeds; the units are plotter defined. When in doubt, pass the built-in variable pltpenspeed to plot_open.
- maxx and maxy are the sizes, in world coordinates, of the paper in the plotter.
- tofile is true when the plotter output is going to a file, and false when output is sent to the plotter.
- numplots is usually passed as 1, but for HPGL-II if passed as a number greater than 1 the repeat plot command will be sent to the plotter.
- fname is only used when tofile is true, and is the name of the plot file where the plotter data is sent.
- The variable plot is initialized by plot_open and used in plot_close and plot_mode. The return value is zero if successful and non- zero if unable to open the plotter.

File Output Routines

Use the routines in the following section to manipulate DOS files. See the "File I/O Routines" section for reading and writing files. Any routines that return integers use the DOS file error constants described in the "Constants" chapter.

File Manipulation Routines

chdir

```
FUNCTION chdir (path : string) : integer;
```

chdir changes the current directory to path.

This procedure should NEVER be called, but is included for completeness.

file_copy

```
FUNCTION file_copy (oldname, newname : string) : integer;
```

file_copy copies a file from one location to another.

newname can be in the same directory as oldname, as long as the filename is different.

file_del

```
FUNCTION file_del (fname : string) : integer;
```

file_del deletes a file. fname is the pathname of a unique file (no wild cards). The return value is the DOS return code.

file_exist

```
FUNCTION file_exist (fname : string) : boolean;
```

file_exist checks for existence of a file. It returns true when the file fname exists, false when it does not.

file_find

```
FUNCTION file_find (str : OUT string;
                    attr : OUT integer;
                    buf : IN OUT dosbuf) : boolean;
```

file_find searches for files specified in the previous call to file_pattern. The name of the file found (without the name of the path) is returned in str, and its attribute is returned in attr. For example, the following example of code:

```
file_pattern ('c:/mtec/*.exe', 0, buf');
WHILE file_find (str, attr, buf) DO
  wrterr (str);
END;
```

could produce the output:

```
  DCAD.EXE
  CONFIG.EXE
  CONVTPL.EXE
```

file_pattern

```
PROCEDURE file_pattern (pat : string;
                        attr : integer;
                        buf : IN OUT dosbuf);
```

file_pattern is used with file_find to search directories for given files.

file_pattern initializes the variable buf for use by file_find. attr is the file attribute to search for (for normal files, this number is zero - for more information on file attributes, see your DOS Reference Manual).

file_ren

```
FUNCTION file_ren (oldname, newname : string) : integer;
```

file_ren renames a file from oldname to newname. This function can also be used to move files from one directory to another, as long as they remain on the same disk drive.

For example:

```
iores := file_ren ('c:/dcad/abc.def', 'c:/abc.def');
```

moves the file abc.def to the root directory, as long as the return code is fl_ok.

mkdir

```
FUNCTION mkdir (path : string) : integer;
```

mkdir creates (makes) directories. Similar to the DOS mkdir command, mkdir can create only one directory at a time.

For example, when c:/dcad does not exist, `'iores := mkdir (c:/dcad/dwg);` fails.

rmdir

```
FUNCTION rmdir (path : string) : integer;
```

rmdir is used to remove directories. Similar to the DOS keyboard command rmdir, the directory being removed must be empty.

File I/O Routines

The functions presented in this section allow DCAL macros to read or write to files or devices. All functions in this section (with the exception of f_eof) return one of the file i/o constants, described in the "Constants" chapter. An operation with no errors returns fl_ok. Once an error occurs in a file, no further operations are performed on that file, and all subsequent calls to any of the file i/o functions return the same error number.

DCAL understands two types of files: text files and binary (or data) files. Specify the file type in the f_open or f_create function calls. You can use the routines that read and write strings, reals, integers, and characters with either text files or data files. For text files, these routines convert the arguments to and from strings. For data files, the parameter is written to the file as a data item (not converted to a string).

f_close

```
FUNCTION f_close (fl : IN OUT file) : integer;
```

f_close closes the file fl.

f_close must be called before leaving the macro to ensure that the RAM buffer is flushed to the disk and the file handle is released.

f_create

```
FUNCTION f_create (fl : OUT file;
                   fname : string;
                   text : boolean) : integer;
```

f_create creates a file with the name fname.

When the file is a text file, text is true, otherwise text is false. The file is opened for writing (fmode_write).

f_eof

```
FUNCTION f_eof (fl : IN OUT file) : boolean;
```

f_eof returns true when the current state of file fl is the end of file; otherwise, it returns false.

f_open

```
FUNCTION f_open (fl : OUT file;
                 fname : string;
                 text : boolean;
                 mode : integer) : integer;
```

f_open opens an existing file or device.

- The name of the file (including the path) is given by fname.
- The file variable fl is initialized to read, write, or read and write the file depending on whether mode is fmode_read, fmode_write, or fmode_read_write, respectively.
- When text is true, the file is opened as a text file, and mode must be fmode_read. When text is false, the file is opened as a data file.

f_rdchar

```
FUNCTION f_rdchar (fl : IN OUT file; ch : OUT char) : integer;
```

f_rdchar reads a character from the specified file.

f_rdint

```
FUNCTION f_rdint (fl : IN OUT file; i : OUT integer) : integer;
```

f_rdint reads an integer from the file fl.

In a text file, an integer is specified as a string. This function skips leading white space, and converts the string that follows to an integer. In a data file, two bytes are read from the file and put in the variable i.

f_rdln

```
FUNCTION f_rdln (fl : IN OUT file) : integer;
```

This function reads past the next new line sequence. This function is important when you read a text file for strings.

The first call to f_rdstr returns the first line (string) of the file. The next call to f_rdstr returns a string of zero length, because the file is sitting at the new line mark, which terminates a string. To advance to the next line, call f_rdln between the calls to f_rdstr. This call is only valid for text files, and returns an error when used on a data file.

f_rdreal

```
FUNCTION f_rdreal (fl : IN OUT file; rl : OUT real) : integer;
```

This function reads a real from the file fl.

f_rdstr

```
FUNCTION f_rdstr (fl : IN OUT file; str : OUT string) : integer;
```

This function reads a string from the file fl. The string is terminated by either reaching the maximum length of the file or by hitting a new line character.

See the discussion of f_rdln.

f_seek

```
FUNCTION f_seek (fl : IN OUT file; recnum : integer) : integer;
```

f_seek can only be used on data files. It positions the file pointer to the recnum record of file fl, based on the record length set for that file in f_setrec. Record numbers start at 0.

For example, to seek to the fourth record of a file, write:

```
iores := f_seek (fl, 3);
```

This positions the pointer for the file at the 18th byte of the file fl. f_seek and f_setrec allow a programmer to build and read a random access data file.

f_setrec

```
FUNCTION f_setrec (fl : IN OUT file; len : integer) : integer;
```

f_setrec sets the record length of a file in bytes. This record length is used only in the function f_seek.

The parameter fl must be a data file. f_setrec must be called after f_open or f_create. If f_setrec is not called, the record length is assumed to be one byte. The function sizeof is useful with this function.

For example, if every record of a data file consisted of two fields, one a real and the other an integer, you might write:

```
 iores := f_setrec (fl,       sizeof (0.0) + sizeof (0));
```

This sets the record size to 6 bytes. Because this number may change in future versions of DCAL and DataCAD, it is a good idea to use sizeof and f_setrec.

f_wrchar

```
FUNCTION f_wrchar (fl : IN OUT file; ch : char) : integer;
```

This function writes a single character to the file.

f_wrln

```
FUNCTION f_wrln (fl : IN OUT file) : integer;
```

f_wrln writes a new line to the file. This tells the file to start a new line at the current position.

f_wrint

```
FUNCTION f_wrint (fl : IN OUT file; int : integer) : integer;
```

This function writes an integer to the file.

integer is converted to a string first when using a text file. Notice that the integer is not padded on either side with spaces, so you may want to write leading or trailing spaces (or both) to the file using f_wrstr or f_wrchar.

f_wrreal

```
FUNCTION f_wrreal (fl : IN OUT file; rl : real) : integer;
```

f_wrreal is similar to f_wrint, except that it writes out a real instead of an integer. The real number is converted to a string before writing to the file when fl is a text file.

f_wrstr

```
FUNCTION f_wrstr (fl : IN OUT file; str : string) : integer;
```

f_wrstr writes a string to the file fl. The string is not padded with spaces or new line characters.

CHAPTER 6 Data Routines

Character and String Routines

The routines in this section allow you to convert between several data types and strings, manipulate strings, and perform useful operations on characters.

cvangst

```
PROCEDURE cvangst (ang : real; str : OUT string);
```

cvangst converts its first argument (an angle measured in radians) to a string and which it returns in its second argument.

The returned string depends upon the current angle type. An error occurs when the string is not long enough to hold the result.

cvdisst

```
PROCEDURE cvdisst (dis : real; str : OUT string);
```

cvdisst converts a distance to a string. The distance is in DataCAD units as described earlier.

The returned string depends upon the current scale type (metric, architectural, etc.). An error occurs when the string is not long enough to hold the result.

cvintst

```
PROCEDURE cvintst (i : integer; str : OUT string);
```

cvintst converts an integer to a string.

An error occurs when the string is not long enough to hold the result.

For example, in the statements:

```
 i := 45;   cvintst (-1 * i, str);
```

str is equal to ’-45’.

cvlntst

```
PROCEDURE cvlntst (li : longint; str : OUT string);
```

cvlntst converts a long integer to a string.

cvrllst

```
PROCEDURE cvrllst (rl : real; str : OUT string);
```

cvrllst converts a real number to a string. An error occurs when the string is not long enough to hold the result.

The statement:
```
 cvrllst (45.56, str);
```
results in str being equal to '45.56'.

cvstint

```
FUNCTION cvstint (str : string; i : OUT integer) : boolean;
```

cvstint converts the string str to an integer and returns the result in i.

When cvstint is able to convert the string, it returns true. When there is an error in conversion, (for instance, str = '45x3') false returns.

cvstlnt

```
FUNCTION cvstlnt (str : IN string; li : OUT longint) : boolean;
```

cvstlnt converts a string to a long integer (longint).

cvstrll

```
FUNCTION cvstrll (str : string; rl : OUT real) : boolean;
```

cvstrll converts the string to a real.

true returns when there is no error in conversion; false returns when there is a conversion error.

islower

```
FUNCTION islower (ch : char) : boolean;
```

islower returns true when ch is a lowercase letter. When ch is an uppercase letter or not a letter islower returns false.

islower (a) returns true islower (B) returns false islower ($) returns false

isupper

```
FUNCTION isupper (ch : char) : boolean;
```

isupper returns true when its argument is an uppercase character. When its argument is a lowercase character or not a character isupper returns false.

isupper (a) returns false isupper (B) returns true isupper ($) returns false

strassign

```
PROCEDURE strassign (dest : IN OUT string; source : string);
```

strassign copies source into dest.

This procedure is needed because you can't assign two string variables of different maximum or dynamic lengths to one another using ':='. A string literal, however, can be assigned to a string variable with ':=' or strassign.

strcat

```
PROCEDURE strcat (str1 : IN OUT string; str2 : string);
```

strcat returns in its first argument the first argument concatenated with the second argument.

An error occurs when the maximum length of the first argument cannot hold the result.

If str1 = 'Good ' and str2 = 'Morning', then:

strcat (str1, str2);

results in str1 = 'Good Morning'. str2 will be unchanged. Similarly:

strcat (str2, str1);

results in str2 = 'MorningGood ' and str1 is unchanged.

strcomp

```
FUNCTION strcomp (str1, str2 : string; len : integer) : boolean;
```

strcomp compares two strings to see if they are equal.

When len is -1, the entire strings are compared, that is, their dynamic lengths must be the same for them to be equal. When len is a positive number, only the first len characters are compared.

strcomp ('Hello', 'Hello', -1) = true strcomp ('Hello', 'Hello1', 5) = true strcomp ('Hello', 'Hello1', -1) = false

strdel

```
PROCEDURE strdel (str : IN OUT string; start, len : integer);
```

strdel deletes len characters from str starting at position start.

When there are not len characters in the string after position start, a run-time error occurs.

For example if str = 'abcdefghij', then

strdel (str, 4, 3);

results in str = 'abcghij'.

strinc

```
PROCEDURE strinc (str : IN OUT string);
```

strinc increments a string.

The characters on the right end of a string that form a number are incremented to the next number. The string can have any number of digits on the right end of the string.

For example if str = 'abc123', then after calling strinc four times, str = 'abc127'.

strins

```
PROCEDURE strins (dest : IN OUT string;
                  source : string;
                  pos : integer);
```

strins inserts source into dest before position pos.

For example if dest = '12345', then
`strins (dest, 'hi', 3)`
results in dest = '12hi345'.

strlen

```
FUNCTION strlen (str : string) : integer;
```

strlen returns the current dynamic length of its argument.

For example, if str = 'hello, world', then:
`strlen (str)`
returns 12. Also,
`strlen (test)`
returns 4.

strpad

```
PROCEDURE strpad (str : IN OUT string; len : integer; ch : char);
```

strpad pads the string str on the right with character ch until its length (as returned by strlen) is equal to len.

When str is longer than len, it is truncated to be len characters long.

For example if str = '123', then
`strpad (str, 6, "*")`
leaves str = '123***'.

strpos

FUNCTION strpos (pat, str : string; start : integer) : integer;

strpos searches for pat in str, starting at position start. It returns the position in str where pat first occurs, or 0 if pat does not occur in str after start characters. For example:

`strpos ('12', '123123', 2)`
returns 4, the position of the first occurrence of '12' after position 2.

strsub

```
PROCEDURE strsub (source : string; start, len : integer; dest : IN
                  OUT string);
```

strsub extracts a piece of a string. On exit, dest contains len characters from source starting at start.

For example:
`strsub ('12345678', 2, 4, dest)`
results in dest = '2345'.

strupcase

```
PROCEDURE strupcase (str : IN OUT string; toupper : boolean);
```

strupcase converts a string to either all uppercase or all lowercase.

When the parameter toupper is true, the string is converted to uppercase. When toupper is false, the string is converted to lowercase. strupcase uses the same rules as the functions toupper and tolower.

tolower

```
FUNCTION tolower (ch : char) : char;
```

tolower returns its argument converted to lowercase if it is an uppercase character; otherwise, tolower returns its argument unchanged.

`tolower (a)` returns 'a' `tolower (A)` returns 'a' `tolower ($)` returns '$'

toupper

```
FUNCTION toupper (ch : char) : char;
```

toupper returns its argument converted to uppercase if it is a lowercase character, otherwise, toupper returns its argument unchanged.

`toupper (a)` returns 'A' `toupper (A)` returns 'A' `toupper ($)` returns '$'

Working with Entities

Use the routines in this section to perform operations on entities that have been read from the database. See the following section on ways to read entities from the database.

ent_add

```
PROCEDURE ent_add (ent : IN OUT entity);
```

ent_add adds the entity ent to the database on the current layer. All fields of an entity specific to the particular entity type (line, slab, dome) must be set prior to calling ent_add. The macro stops executing when there is not enough room to add the entity to the database.

During this procedure, the currently-set values of the variables that define line type, entity color, etc., are used for the added entity. When the entity is added to the database, it is not automatically drawn on the graphics screen. This must be done separately by the macro.

See the "Variables" chapter for more information. See ent_draw.

ent_copy

```
PROCEDURE ent_copy (oldent, newent : IN OUT entity;
                    toCurr, doAttr : boolean);
```

ent_copy duplicates an entity in the database. Use this procedure instead of assigning entities and then adding the new one to the database.

For example:

```
ent_copy (oldent, newent, false, true);
```

- The parameter toCurr controls whether newent is copied to the current layer or to the layer that oldent is on. When toCurr is true, newent is copied to the current layer. When toCurr is false, newent is copied to the same layer as oldent.

- When doAttr is true, the attributes of oldent are copied to newent. When doAttr is false, the attributes are not copied. It is recommended that doAttr always be true,

unless you have a specific reason not to copy attributes. There is no need to call ent_init (newent,...); prior to ent_copy or to call ent_update (newent); after a call to ent_copy.

ent_del

```
PROCEDURE ent_del (ent : IN OUT entity);
```

ent_del deletes ent from the database. ent must have been read from the database using one of the database routines. If it was not, a fatal internal error occurs.

ent_draw

```
PROCEDURE ent_draw (ent : IN OUT entity; drmode : integer);
```

ent_draw draws ent on the screen using the specified drawing mode.

- drmode can be either drmode_white, drmode_black, or drmode_flip.
- ent does not have to have been read from the database with ent_get, as long as all the information needed to draw the entity has been specified (color, end points, radius, etc). This is useful for creating temporary images on the screen.

ent_draw_2d

```
PROCEDURE ent_draw_2d (ent : IN entity; drmode : IN integer);
```

ent_draw_2d is similar to procedure ent_draw except that in any viewing projection other than orthographic, ent_draw_2d draws an entity disregarding the 3D to 2D viewing projection.

ent_draw_2d is typically used during input and editing operations which are performed in parallel projections to allow for the drawing of entity geometry without accounting for the current viewing matrix, but correctly handling the current window-to-viewport transformation.

ent_draw_dl

```
PROCEDURE ent_draw_dl (ent : IN entity; drmode : IN integer; todl :
                       boolean);
```

ent_draw_dl draws ent on the screen using the specified drawing mode.

- drmode can be either drmode_white, drmode_black, or drmode_flip. l ent does not have to have been read from the database with ent_get, as long as all the information needed to draw the entity has been specified (color, end points, radius, etc).
- When todl is true, the entity is added to the display list. When todl is false, the

entity is not added to the display list. todl should be false for temporary screen data. This is useful for creating temporary images on the screen.

ent_drawmode

```
FUNCTION ent_drawmode (mode : IN mode_type;
                       drmode : IN integer;
                       todl, clearscr : IN integer) : integer;
```

ent_drawmode is similar to ent_draw_dl, except that instead of drawing only one entity, every entity returned by mode is drawn.

- drmode can be either drmode_white, drmode_black, or drmode_flip.
- ent does not have to have been read from the database with ent_get, as long as all the information needed to draw the entity has been specified (color, end points, radius, etc).
- When todl is true, the entity is added to the display list. When todl is false, the entity is not added to the display list. todl should be false for temporary screen data.
- When clearscr is true, the screen is cleared before the entities are drawn.

See "mode".

ent_enlarge

```
PROCEDURE ent_enlarge (ent : IN OUT entity;
                       cent : point;
                       Xsc, Ysc, Zsc : real);
```

ent_enlarge enlarges entity ent by the factors Xsc, Ysc, and Zsc about the point cent.

Perform an ent_update after the entity is enlarged to update the database.

ent_extent

```
PROCEDURE ent_extent (ent : IN OUT entity;
                      lowleft, upright : OUT point);
```

ent_extent finds the minimum and maximum extents of ent. This is the smallest box that completely encloses the entity.

ent_init

```
PROCEDURE ent_init (ent : IN OUT entity; enttype : integer);
```

Whenever an entity is declared, ent_init should be called to initialize the entity. This call initializes all fields of an entity so that it may be drawn and otherwise operated

on. Failure to call this procedure before using an entity can result in an entire range of errors.

enttype is the constant for the type of entity, such as entlin or entply.

ent_move

```
PROCEDURE ent_move (ent : IN OUT entity; dx, dy, dz : real);
```

ent_move moves entity ent by dx, dy, and dz.

Perform an ent_update after moving an entity to replace the entity in the database.

ent_relink

```
PROCEDURE ent_relink (ent1, ent2 : IN OUT entity);
```

ent_relink moves the logical location of an entity in the database.

ent1 is not relocated in the database. The relinking operation is performed upon ent2. The logical location in the database of ent2 is altered so that ent2 resides on the same layer as ent1 and so that ent2 immediately follows ent1 in the chain of entities on that layer. ent1 and ent2 must both be valid entities that currently exist in the drawing database. Both entities must have been read from the database using the function ent_get, or just added to the database using ent_add with no intervening operations.

An effect of using ent_relink is that ent2 will be placed in the same group as ent1.

The following example demonstrates how to use ent_relink to effectively insert an entity into a drawing after another entity by first adding the entity and then relinking it using ent_relink:

```
VAR
  oldent : entity;
  newent : entity;

  ent_add (newent);
  ent_relink (oldent, newent);
```

ent_rotate

```
PROCEDURE ent_rotate (ent : IN OUT entity; cent : point; ang : real);
```

ent_rotate rotates entity ent by ang radians about the point cent.

The rotation is around the Z axis, that is, in the XY plane. This is the normal DataCAD 2D rotation. Perform ent_update to update the database.

ent_tran

```
PROCEDURE ent_tran (ent : IN OUT entity;
                    mat : IN modmat;
                    doatr, update : IN boolean);
```

ent_tran transforms an entity by the transformation described in a modeling matrix. ent_tran may not be used on all of DataCAD's entities, but only those which may be arbitrarily translated, scaled, mirrored, and rotated in space.

The function ent_tranok determines if the entity will respond to the procedure ent_tran. Entities which can be described by the following constants can be used with ent_tran and cause the function ent_tranok to return true:

entLn3	entAr3	entDom	entMrk
entCon	entTor	entSym	entCyl
entSrf	entPly	entTrn	entBlk
entSlb	entCnt		

- ent is the entity to transform.

- mat is the modeling matrix to apply to ent. Set mat using one or more of DCAL's matrix calculation routines.

- When the flag doatr is true, any visible attributes attached to the entity ent are transformed as well.

- When the flag update is true, an ent_update is performed automatically by ent_tran. When the flag update is false, an ent_update is not performed automatically.
  Entities of type polygon or slab which contain voids, have their voids transformed and updated in the database regardless of the value of the flag update.

ent_tranok

```
FUNCTION ent_tranok (ent : IN OUT entity) : boolean;
```

ent_tranok determines when an entity responds to a transformation applied directly via a modeling matrix.

ent_tranok is typically used with the procedure ent_tran to determine if ent_tran can be applied to an entity of a particular type. When the entity passed to ent_tranok can be transformed using ent_tran, ent_tranok returns true. When the entity cannot be transformed ent_tranok returns false.

- ent is the entity to check.

ent_update

```
PROCEDURE ent_update (ent : IN OUT entity);
```

ent_update updates the database with the current description of ent.

For example, you can read entities from the database, undraw them, change their colors, put them back into the database, and then redraw them using the following code:

```
  mode_init (mode);
  mode_lyr (mode, lyr_curr);
  adr := ent_first (mode);
  WHILE ent_get (ent, adr) DO
    ent_draw (ent, drmode_black);
    ent.color := clrred;
    ent_update (ent);
    ent_draw (ent, drmode_white);
    adr := ent_next (ent, mode);
  END;
```

stopgroup

```
PROCEDURE stopgroup;
```

As entities are added to the database, they are considered part of the same group. stopgroup sets the last entity entered as the last entity belonging to a group. stopgroup works only on the current layer. When the current layer is changed, an implicit stopgroup is performed since a group can not span layers.

Selection Set Routines

Use the routines in this section to add to, delete from, and scan selection sets. DataCAD provides eight selection sets for the macro writer.

Every entity 'knows' which selection set(s) it belongs to. Therefore, when an entity is deleted, its selection set is automatically corrected to reflect that the entity is no longer present. This ensures that selection sets never contain invalid data by pointing to entities that no longer exist.

Every selection set routine takes a selection set number as a parameter. This number must be in the range of 0 to 7, specifying one of the eight selection sets available. There are other selection sets within DataCAD, but they are used internally and should not be used by the macro programmer. Use of a selection set other than those numbered zero through seven can cause unpredictable results in both the execution of the macro and the subsequent use of DataCAD.

ssAdd

```
PROCEDURE ssAdd (ssNum : integer; ent : IN OUT entity);
```

ssAdd adds its second argument, ent, to the selection set number ssNum. When ent is already a member of ssNum, no operation is performed.

ssClear

```
PROCEDURE ssClear (ssNum : integer);
```

ssClear clears out selection set number ssNum. All entities that were in the selection set are deleted from the selection set. These entities are removed from this selection set only. They remain in the database and are still members of other selection sets.

ssDel

```
FUNCTION ssDel (ssNum : integer; ent : IN OUT entity) : boolean;
```

ssDel deletes ent from selection set number ssNum.

When ent is a member of ssNum, true is returned. When ent is not a member of ssNum, false is returned and no other operation is performed.

ssGetName

```
PROCEDURE ssGetName (ssNum : IN integer; ssname : OUT string);
```

ssGetName reads the name of one of the first eight selections sets (0 through 7).

ssLength

```
FUNCTION ssLength (ssNum : integer) : integer;
```

ssLength returns the number of entities that are in selection set number ssNum.

ssMember

```
FUNCTION ssMember (ssNum : integer; ent : IN OUT entity) :
boolean;
```

ssMember determines when an entity is a member of a particular selection set.

When ent is a member of selection set ssNum, ssMember returns true, otherwise false returns.

ssSetName

```
PROCEDURE ssSetName (ssNum : IN integer; ssname : IN string);
```

ssSetName sets the name of one of the first eight selection sets.

Layers

Every entity in a drawing is on a layer. The routines in this section manipulate layers. Notice that some of the built-in variables described in the "Variables" chapter also affect the current layer.

getlyrcurr

```
FUNCTION getlyrcurr : layer;
```

getlyrcurr returns the current (or active) layer.

getlyrname

```
PROCEDURE getlyrname (lyr : layer; name : OUT string);
```

getlyrname returns the name of the layer lyr in name. When lyr is not a valid layer, an error occurs.

lyr_clear

```
PROCEDURE lyr_clear (lyr : layer);
```

lyr_clear erases every entity from layer lyr and deletes them from the database.

lyr_create

```
FUNCTION lyr_create (lname : string; lyr : OUT layer) : boolean;
```

lyr_create adds a new layer to the current drawing.

- The layer is created with the name lname.
- The layer variable that describes the new layer is returned in lyr. lyr_create returns true when DataCAD creates the layer, and false when DataCAD is unable to create the layer (too many layers or out of disk space).

lyr_del

```
PROCEDURE lyr_del (lyr : IN layer);
```

lyr_del deletes a layer from a drawing database.

lyr must be a valid layer in the drawing database and must be a part of the standard layer/entity data structure. All entities on the layer, as well as the layer itself, are deleted and completely removed from the database. Unless the drawing is cancelled, the layer and its entities are completely deleted and may not be undone or recovered in any way.

Do not attempt to use lyr_del on a layer which was added to the drawing database using the function lyr_init. lyr_init adds a layer to the database which is not a part of the standard layer/entity structure. In this case, use the procedure lyr_term. lyr_term is specifically designed to terminate and delete layers which are not part of the standard layer/entity database structure.

lyr_find

```
FUNCTION lyr_find (lname : string; lyr : OUT layer) : boolean;
```

lyr_find looks for a layer with the name lname in the database.

When successful, true returns and lyr is set to be the layer that was found. When there is no layer named lname, false returns.

lyr_first

```
FUNCTION lyr_first : layer;
```

lyr_first returns the first layer that is in the database.

lyr_islocked

```
FUNCTION lyr_islocked (lyr : layer) : BOOLEAN;  BUILTIN 235;
```

lyr_islocked returns true if lyr is locked, false if it is not locked.

Note that you will need to declare the procedure (including 'BUILTIN 235') in your code as it is not currently included in any of the standard include files.

lyr_ison

```
FUNCTION lyr_ison (lyr : layer) : boolean;
```

lyr_ison returns true when the layer lyr is on (displayed). When lyr is off (not displayed), lyr_ison returns false.

lyr_lock

```
PROCEDURE lyr_lock (lyr : layer; lock : boolean); BUILTIN 234;
```

Use lyr_lock to lock or unlock the nominated layer. Lyr is the layer to be locked or unlocked. Lock is either true to lock the lyr, or false to unlock lyr.

Note that you will need to declare the procedure (including 'BUILTIN 234') in your code as it is not currently included in any of the standard include files.

lyr_next

```
FUNCTION lyr_next (lyr : layer) : layer;
```

lyr_next returns the next layer in the database after lyr. When lyr is the last layer in the drawing, lyr_next returns a layer that is nil. Test this with lyr_nil.

lyr_nil

```
FUNCTION lyr_nil (lyr : layer) : boolean;
```

lyr_nil returns true when the layer lyr does not point to anything.

The following code writes the name of every layer in the database that is on:

```
  lyr := lyr_first;
  WHILE NOT lyr_nil (lyr) DO
    IF lyr_ison (lyr) THEN
      getlyrname (lyr, str);
      wrterr (str);
      pause (0.5);
    END;
    lyr := lyr_next (lyr);
  END;
```

lyr_read

```
FUNCTION lyr_read (lyr : IN layer;
                   filename : IN string;
                   clearlyr, drawlyr : IN boolean) : integer;
```

lyr_read reads a layer from a layer file into the drawing database.

lyr_read returns fl_ok when the layer file successfully loads or returns one of the other file operation constants. See the "Constants" chapter for more information .

- lyr is the layer to which the contents of the layer file are added. lyr must exist in the drawing database, and must have been added to the drawing using lyr_create, lyr_init, or DataCAD's standard layer user interface. lyr is renamed to match the name of the file which was read in. This could result in two different layers in a drawing with the same name.

- filename is the name of the layer file to read. filename must be a correct layer filename including a complete or relative pathname of the file but without the .lyr extension. This function only looks for files with .lyr extensions.

- clearlyr is a flag that is true when the layer lyr is to be cleared of all entities prior to appending the contents of the layer file. When clearlyr is false, the contents of the layer file are appended to the layer and any existing entities on the layer are not deleted.

- drawlyr is a flag which is true when the layer is to be drawn immediately after it is loaded. When drawlyr is false, the layer is loaded into the drawing database, but is not drawn. If lyr is on when new entities are concatenated to it by lyr_read and

drawlyr is false, although the new layer file information is not visible, that information is scanned during the next editing or snapping operation. The next operation which causes the screen to refresh redraws the new data as well.

lyr_set

```
PROCEDURE lyr_set (lyr : layer);
```

lyr_set sets the current layer to the parameter lyr.

When lyr is not a valid layer, an error occurs.

lyr_seton

```
PROCEDURE lyr_seton (lyr : layer; onoff : boolean; redraw : boolean);
```

- lyr_seton toggles layers on or off.
- lyr is the layer to act upon.
- onoff is true when the layer is to be turned on and false when it is to be turned off.
- The parameter redraw is true when the layer is to be drawn, and false when the layer is not to be redrawn. To remove entities (which reside on a particular layer) from the screen, set onoff to false and redraw to true. The current layer cannot be turned off.

lyr_viewfile

```
PROCEDURE lyr_viewfile (filename : IN string;
                        refresh, extents : IN boolean;
                        imagesize : IN real);
```

lyr_viewfile duplicates the functionality of Layer/ViewFile in DataCAD allowing the user to quickly view images or pictures as though they were slides. The layer file appears on the screen only; the file is not loaded into the drawing database.

- filename is the name of the layer file to view. filename must be a complete pathname to the file, or a valid pathname relative to the current directory. filename must contain the .lyr filename extension, when applicable.
- When the flag refresh is true, the screen is cleared prior to drawing the contents of the layer file. When refresh is false, the screen is not refreshed prior to displaying the contents of the layer file.
- When the flag extents is true, a WindowIn based on the extents of the layer file is performed prior to the display of the contents of the layer file. When extents is false, the layer file appears at the current scale and window-to-viewport mapping.

- imagesize controls the size of the image relative to the current viewport. imagesize has an effect when extents is true. When extents is false, imagesize is ignored. imagesize must be a real number between 1.0 and 100.0 and indicates the percentage of the screen area which the image should fill. For example, if imagesize is set to 80.0, the actual size of the image is reduced by 20 percent of the screen width or height (as appropriate) so that the image does not bleed to the edges of the screen.

lyr_write

```
FUNCTION lyr_write (lyr : IN layer; filename : IN string) : integer;
```

lyr_write writes a layer out to a layer file.

- lyr is the layer to be written. lyr must exist in the database.

- filename is the name of the file to which the layer is written. filename should contain either the full pathname of the file, or a valid pathname relative to the current directory. The filename should not contain the .lyr extension. This is added by the routine. When a layer file of the same name exists, the layer file is overwritten and no warning is generated. To first check for the existence of a layer file with the same name prior to writing the layer using lyr_write, use the function file_exist.

numlayer

```
FUNCTION numlayer : integer;
```

numlayer returns the number of existing layers in the drawing. This number is always at least one.

setlyrname

```
PROCEDURE setlyrname (lyr : layer; name : string);
```

setlyrname sets the name of lyr to name. When lyr is not a valid layer, an error occurs. The layer name on the screen is not updated when lyr is the current layer. You must call wrtlyr to force the layer name to be updated.

Symbol Routines

A symbol is often a name given to a variable like pt1. With the routines in this section you can add and manipulate symbols in the current drawing. Symbol instances are treated just like any entity. Use these routines to manipulate the definition of a symbol.

sym_clearref

```
PROCEDURE sym_clearref;
```

Every symbol has a Boolean field in its header, .refflag, that determines if the symbol has been referenced by the procedure sym_ref. sym_clearref clears (sets to false) all .refflag fields for all symbols in the database.

sym_count

```
PROCEDURE sym_count (mode : IN OUT mode_type);
```

sym_count looks through the entities returned by mode which are instances of symbols (entsym) and counts the number of instances of each symbol. This count can then be examined by looking at the num field of each symbol.

A macro should never write a value to num.

For example, to count the number of times each symbol has an instance on the current layer, use:

```
mode_init (mode); {look on this layer}
sym_count (mode);
saddr := sym_first;
WHILE sym_get (sym, saddr) DO
  writeCount (sym.num);
  saddr := sym_next (sym);
END;
```

sym_create

```
PROCEDURE sym_create (sym : OUT symbol;
                      mode : IN OUT mode_type;
                      ref : point;
                      sname : string;
                      delEnts, undraw : boolean);
```

sym_create creates or redefines a symbol.

- When a symbol of name sname already exists, it is redefined; when it does not exist, it is created.
- The entities that make up the symbol are read by mode.
- The reference point of the symbol is ref.
- When delEnts is true, the entities read from mode are deleted.
- When undraw is true, the entities are undrawn as the symbol is created.
- On exit, sym refers to the new (or updated) symbol. As an example, the following

code converts everything in selection set 0 to a symbol:

```
mode_init (mode);
mode_ss (mode, 0);
setpoint (pt, 0.0);
sym_create (sym, mode, pt, 'newSym', true, true);
```

sym_find

```
FUNCTION sym_find (sname : string; sym : OUT symbol) : boolean;
```

sym_find looks in the database for a symbol named sname.

When sname is found, sym is set to the symbol and true is returned. If the symbol is not found, then sym_find returns false.

sym_first

```
FUNCTION sym_first : symaddr;
```

sym_first returns the address of the first symbol in the database. This, along with sym_next and sym_get, are used to scan through the symbols defined in the drawing.

sym_get

```
FUNCTION sym_get (sym : OUT symbol; saddr : symaddr) : boolean;
```

sym_get reads a symbol sym given its address.

If the address is nil, sym_get returns false, otherwise it returns true.

For example, to write the name of every symbol in the database, use:

```
saddr := sym_first;
WHILE sym_get (sym, saddr) DO
  wrterr (sym.name);
  pause (0.5);
  saddr := sym_next (sym);
END;
```

sym_get_atr

```
FUNCTION sym_get_atr (sym : OUT symbol;
                      saddr : symaddr;
                      atrName : string;
                      atr : OUT attrib) : Boolean;
```

sym_get_atr is similar to sym_get, but returns the next symbol that has an attribute with the given name. If no more such symbols exist in the database, false is returned.

This example shows how to use sym_get_atr. This piece of code reads every symbol that has an attribute named '*mtec'.

```
saddr := sym_first;
WHILE sym_get_atr (sym, saddr, '*mtec', atr) DO
  wrtAtr (atr);
  saddr := sym_next (sym);
END
```

sym_next

```
FUNCTION sym_next (sym : symbol) : symaddr;
```

sym_next returns the address of the next symbol in the database. When there is no next symbol, sym_next returns nil.

See the example under sym_get.

sym_purge

```
PROCEDURE sym_purge; BUILTIN 240;
```

Use sym_purge to remove all unused symbos from the drawing database.

Note that you will need to declare the procedure (including 'BUILTIN 240') in your code as it is not currently included in any of the standard include files.

sym_read

```
FUNCTION sym_read (fName, symName : string;
                   sym : OUT symbol) : integer;
```

sym_read reads a symbol from a symbol file.

- The filename fName may not contain the file extension; .sm3 is automatically added.
- symName is the internal name to use for the symbol. On exit, the function returns the file i/o constant from reading the symbol file. fl_ok indicates success.
- On exit, sym refers to the symbol that was read.

sym_ref

```
PROCEDURE sym_ref (mode : IN OUT mode_type);
```

sym_ref sets the .refflag fields to true for all symbols that have instances returned by mode, including nested symbols. The main use of this routine is internal to DataCAD to determine when a redefinition of a symbol is self-referencing. It is also used when saving a symbol to a file.

sym_write

```
FUNCTION sym_write (sym : symbol; fName : string) : integer;
```

sym_write writes an existing symbol sym to a file. As with sym_read, the filename must not contain the file extension.

The return constant of fl_ok indicates success. The symbol and all of the nested symbols that it references as well as all of its attributes are written to the file.

Attribute Routines

Use the routines in this section to handle attributes. Attributes consist of two parts, the name and the value. The name is a string of maximum length atr_name_len characters. The value can be a real, integer, distance, angle, point, string, or logical address.

Attributes can be attached to symbols, entities, symbol instances, layers, and the drawing (referred to as system attributes). Most of the routines in this section deal with attributes attached to any of these, but some are specific as to what the attribute is attached to. These have ent, sym, lyr, or sys in their name.

atr_2str

```
PROCEDURE atr_2str (atr : IN OUT attrib; str : OUT string);
```

atr_2str converts the value of atr to a string str, for any type of attribute. This procedure correctly converts any attribute of type integer, real, distance, angle, point, or string to a string.

atr_add2ent

```
PROCEDURE atr_add2ent (ent : IN OUT entity; atr : IN OUT attrib);
```

atr_add2ent adds an attribute to an entity.

The attribute atr must be initialized with atr_init and the name and value of the attribute should be assigned. The entity atr is added to ent. When the attribute field visible is true, the value of atr appears when the entity is drawn. The remaining fields of the attribute type describe how the attribute is displayed. For example, to add an integer attribute to an entity:

```
atr_init (atr, atr_int);
atr.name := '*EVS';
atr.int := 3;
atr_add2ent (ent, atr);
```

atr_add2lyr

```
PROCEDURE atr_add2lyr (lyr : IN OUT layer; atr : IN OUT attrib);
```

atr_add2lyr adds an attribute to the chain of attributes attached to a layer in the drawing database.

- lyr is the layer to which the attribute is added. lyr must exist in the drawing database before calling atr_add2lyr.

- atr is the attribute to add to the layer. atr should be initialized using the procedure atr_init, and should contain a valid name and data.

- The attribute field visible may be set to true, but the layer attribute does not appear. Visible attributes appear only for entities.

atr_add2sym

```
PROCEDURE atr_add2sym (sym : IN OUT symbol; atr : IN OUT attrib);
```

This procedure is virtually identical to atr_add2ent, except it adds an attribute to a symbol.

- The attribute atr must be initialized with atr_init and the name and value of the attribute should be assigned.

- The entity atr is added to ent.

- The attribute field visible may be set true, but the symbol attribute will not appear. Visible attributes are displayed only for entities. The remaining fields of the attribute type describe how the attribute appears.

atr_add2sys

```
PROCEDURE atr_add2sys (atr : IN OUT attrib);
```

atr_add2sys adds an attribute to the chain of attributes attached to a drawing at the system level.

- atr is the attribute to add. atr should be initialized using the procedure atr_init, and should contain a valid name and data.

- The attribute field visible may be set to true, but the system attribute does not appear. Visible attributes appear only for entities.

atr_delent

```
PROCEDURE atr_delent (ent : IN OUT entity; atr : attrib);
```

atr_delent deletes an attribute from an entity. The parameters are the entity to delete from, ent, and the attribute to delete, attrib.

atr_dellyr

```
PROCEDURE atr_dellyr (lyr : IN OUT layer; atr : IN OUT attrib);
```

atr_dellyr deletes an attribute from the chain of attributes attached to a layer.

- lyr is the layer from which to delete the attribute.

- atr is the attribute to delete. atr must be read from the chain of attributes attached

to the layer using the function atr_get.

atr_delsym

```
PROCEDURE atr_delsym (sym : IN OUT symbol; atr : attrib);
```

This procedure deletes an attribute from a symbol definition. The parameters are the entity to delete from, ent, and the attribute to delete, attrib.

atr_delsys

```
PROCEDURE atr_delsys (atr : IN OUT attrib);
```

atr_delsys deletes an attribute from the chain of system attributes. Space in the drawing file taken up by the attribute is freed up for later use.

atr is the attribute to delete. atr must have been read from the chain of system attributes using the function atr_get. The attribute is deleted from the drawing database.

atr_entfind

```
FUNCTION atr_entfind (ent : IN OUT entity;
                      aName : string;
                      atr : OUT attrib) : boolean;
```

atr_entfind finds an attribute atr of name aName belonging to entity ent.

When an attribute with the given name exists, atr_entfind returns true and sets atr equal to the attribute. If the attribute does not exist, false is returned.

atr_entfirst

```
FUNCTION atr_entfirst (ent : IN OUT entity) : atraddr;
```

This function returns the address of the first attribute associated with an entity.

See the example under atr_get.

atr_get

```
FUNCTION atr_get (atr : OUT attrib; addr : atraddr) : boolean;
```

atr_get reads an attribute atr given its address addr.

When addr is nil, false returns; otherwise true is returned. This function is similar to ent_get and sym_get.

To read all of the attributes belonging to an entity, use:

```
aAddr := atr_entfirst (ent);
WHILE atr_get (atr, aAddr) DO
  processAtr (atr);
  aAddr := atr_next (atr);
END;
```

To read all of the attributes belonging to a symbol, use:

```
aAddr := atr_symfirst (sym);
WHILE atr_get (atr, aAddr) DO
  processAtr (atr);
  aAddr := atr_next (atr);
END;
```

Notice these two pieces of code are identical except for the function used to find the address of the first attribute. This code shows how all attributes are the same, regardless of what they are attached to.

See also atr_lyrfirst and atr_sysfirst.

atr_init

```
PROCEDURE atr_init (atr : IN OUT attrib; atrtype : IN integer);
```

atr_init initializes a variable of type attrib so the attribute can then be added to the database.

Always call atr_init before adding an attribute to any data object.

- atr is the attribute to initialize.
- atrtype is an integer constant indicating the type which the attribute assumes (integer, distance, real, angle, string, or address).

atr_lyrfind

```
FUNCTION atr_lyrfind (lyr : IN layer;
                      aname : IN string;
                      atr : OUT attrib) : boolean;
```

atr_lyrfind searches for an attribute of name aname belonging to layer lyr.

When an attribute with the given name exists, atr_lyrfind returns true and sets atr equal to the attribute. If the attribute does not exist, false returns.

lyr must exist as a valid layer in the drawing database.

atr_lyrfirst

```
FUNCTION atr_lyrfirst (lyr : IN layer) : atraddr;
```

atr_lyrfirst returns the address of the first attribute attached to a given layer.

atr_lyrfirst is usually used with the routines atr_get, and atr_next to linearly scan the chain of attributes attached to a layer.

lyr is the layer to examine for attributes.

The following example scans the list of attributes attached to a layer and counts those which are a distance:

```
VAR   atr  : attrib;   addr : atraddr;   count : integer;   lyr  :
layer; ...
```

```
VAR
  atr  : attrib;
  addr : atraddr;
  count : integer;
  lyr  : layer;
...

  count := 0;
  addr := atr_lyrfirst (lyr);
  WHILE atr_get (atr, addr) DO
    addr := atr_next (atr);
    IF atr.atrtype = atr_dis THEN
     count := count + 1;
    END;
  END;
```

atr_next

```
FUNCTION atr_next (atr : IN OUT attrib) : atraddr;
```

atr_next returns the address of the next attribute after atr. This procedure is used regardless of whether the attributes are coming from entities, symbols, layers, or the system.

When there are no more attributes after atr, nil is returned.

atr_symfind

```
FUNCTION atr_symfind (sym : IN OUT symbol;
                      aName : string;
                      atr : OUT attrib) : boolean;
```

atr_symfind finds an attribute atr of name aName belonging to symbol sym.

When an attribute with the given name exists, atr_symfind returns true and sets atr equal to the attribute. If the attribute does not exist, false returns.

atr_symfirst

```
FUNCTION atr_symfirst (sym : IN OUT symbol) : atraddr;
```

atr_symfirst returns the address of the first attribute belonging to a symbol.

atr_symfirst is usually used with the routines atr_get, and atr_next to linearly scan the chain of attributes attached to a symbol.

sym is the symbol to examine for attributes.

atr_sysfind

```
FUNCTION atr_sysfind (aName : IN string; atr : OUT attrib) : boolean;
```

atr_sysfind searches for an attribute atr of name aname belonging to symbol sym.

When an attribute with the given name exists, atr_sysfind returns true and sets atr equal to the attribute. If the attribute does not exist, false returns.

atr_sysfirst

```
FUNCTION atr_sysfirst : atraddr;
```

atr_sysfirst returns the address of the first system attribute.

atr_sysfirst is usually used with the routines atr_get and atr_next to linearly scan the chain of system attributes.

The following example scans the list of system attributes and counts those which are a distance.

```
VAR
  atr  : attrib;
  addr : atraddr;
  count : integer;

  count := 0;
  addr := atr_sysfirst;
  WHILE atr_get (atr, addr) DO
    addr := atr_next (atr);
    IF atr.atrtype = atr_dis THEN
     count := count + 1;
    END;
  END;
```

atr_update

```
PROCEDURE atr_update (atr : IN OUT attrib);
```

atr_update updates an attribute in the database after it is changed. The type of an attribute may change, as well as whether or not the attribute is visible.

Polyline and Polyvert Routines

The declarations for the routines in this section require the inclusion of the file _pline.inc because they are not built-into the DCAL compiler.

The following types, constants, and routines manipulate the geometric data associated with entities of type polyline and surface of revolution. These two entities, unlike most other entities, contain what is referred to as copious data. In a typical entity such as an arc, all of the data describing the arc can be contained completely within a variable of type entity; attributes are stored in a separate list attached to each entity. With polylines and surfaces of revolution, not all the data for the entity is contained within the entity record structure.

An entity of type polyline or surface of revolution consists of header information (like color, linetype, etc.) which is exactly like that for any other entity. These entities also contain additional information in the entity record structure like z-base and z-height, with polylines, or a modeling matrix, with surfaces of revolution. The data describing the vertices, lines, and arcs of a polyline or surface of revolution profile is contained in a list of objects attached to the entity. This list, comprised of objects called polyverts, is accessed through the address of the first and last polyvert in the list.

The basic entity also contains the addresses of the first and last polyverts in the list associated with each entity. The remaining polyverts are accessed by linearly scanning the list. With the exception of where these two addresses are stored, access to and manipulation of polyverts is identical for both polylines and surfaces of revolution.

Additional steps must be taken to access the geometry of this copious data. Therefore, DataCAD and DCAL have a variety of built-in types, constants, routines, and functions for managing and manipulating copious data.

Each vertex of the polyline requires that a polyvert be attached to the entity. A polyvert contains the (x,y,z) coordinates of a single point (the vertex) and a real number which describes the type of arc between the vertex and the following vertex. This real number is usually referred to as the bulge.

arc_to_bulge

```
PROCEDURE arc_to_bulge (center : IN point;
                        radius, begang, endang : IN real;
                        ccw : IN boolean;
                        pnt1, pnt2 : OUT point;
                        bulge : OUT real);
```

arc_to_bulge converts an arc which is defined by the same method as entities of type entarc to the form used by polylines and surfaces of revolution.

- The arc is defined using five parameters:

- center is the center point of the arc
- radius is the arc's radius.
- begang and endang are the arc's beginning and ending angles
- ccw is a Boolean flag which is true when the arc goes counterclockwise from begang to endang; and false when the arc goes clockwise from begang to endang.
- The results of the computation are stored in pnt1, pnt2, and bulge:
- pnt1 contains the coordinates of the polyvert vertex
- pnt2 contains the coordinates of the next polyvert in the list
- bulge contains the value of the bulge parameter for the polyvert.
  When the value of bulge is zero the polyvert is a line and should be assigned a shape of pv_vert; otherwise it is an arc segment and should be assigned a shape of pv_bulge.

For example, to convert an ordinary entity of type entarc to a polyvert, use the following code:

```
arc_to_bulge (ent.arccent, ent.arcradius, ent.arcbang, ent.arceang,
              true, pv.pnt, pv.nextpnt, pv.bulge);
```

In actual use, ccw must first be calculated, but its computation is typically dependent on the application. true is used here for simplicity.

bulge_to_arc

```
PROCEDURE bulge_to_arc (pnt1, pnt2 : IN point;
                        bulge : IN real;
                        center : OUT point;
                        radius, begang, endang : OUT real;
                        ccw : OUT boolean);
```

bulge_to_arc performs the complement function of arc_to_bulge. It takes a polyline vertex of shape pv_bulge and converts it into the standard description of an arc segment.

- The input to this routine consists of three parameters:
  pnt1, the coordinates of the polyvert vertex
  pnt2, the coordinates of the next polyvert vertex in the list
  bulge, the value of the bulge parameter for the polyvert.
- The output of bulge_to_arc consists of five parameters:
  center is the center point of the arc

radius is the arc's radius
begang is the beginning angle of the arc segment
endang is the ending angle of the arc segment
ccw is a Boolean flag which is true when the arc goes counterclockwise from begang to endang, and false when the arc goes clockwise from begang to endang.

NOTE: When working with arc segments as related to polyverts, maintain sufficient information indicating the direction of the arc segments because a polyline or surface of revolution maintains the connectivity of polyverts. A polyvert maintains this information by maintaining the order of the vertices in a polyline and, implicitly, via the bulge parameter. For the standard description of an arc, an additional flag, ccw, indicates the direction of the arc relative to the center and beginning of arc.

disfrompolyvert

```
FUNCTION disfrompolyvert (pv : IN OUT polyvert;
                          tstpnt : IN point) : real;
```

disfrompolyvert returns the square of the distance (note that this is not the actual distance) from a point to a polyvert. Use this function to determine exactly which polyvert in a polyline a user is pointing to.

In the following example, the variable ent has already been read using ent_near or a similar operation and is a valid entity of type entpln. The following code finds the nearest polyvert to a point:

```
VAR
  pv      : polyvert;
  addr    : pvtaddr;
  ent     : entity;
  pnt     : point;
  nearest : pvtaddr
  dis     : real
  neardis : real
  first   : boolean


  first := true;
  addr := ent.plnfrst;
  WHILE polyvert_get (pv, addr) DO
    addr := pv.next
    dis := disfrompolyvert (pv, pnt);
    IF first THEN
      first := false;
      closedis := dis;
      nearest := pv.addr;
    ELSE
      IF dis < closedis THEN
        closedis := dis;
        nearest := pv.addr;
      END;
    END;
```

```
  END;

  IF NOT first THEN
    { At least one polyvert was scanned and nearest now contains the
      address of the polyvert which is nearest to the point pnt. }
  END;
```

ent2polyvert

```
PROCEDURE ent2polyvert (ent : IN entity; pv : OUT polyvert);
```

ent2polyvert performs the complement function of polyvert2ent.

*An entity of type entlin or entarc converts to a polyvert of the appropriate type. When the entity is of type entarc and has a valid geometry, a polyvert of shape pv_bulge returns. When the entity is of type entlin a polyvert of shape pv_vert returns.

getpolyline

```
FUNCTION getpolyline (msg : IN str80;
                      viewmode : IN integer;
                      dodraw, doclosed : IN boolean;
                      init, isclosed : IN OUT boolean;
                      frst, last : IN OUT pvtaddr;
                      key : OUT integer) : integer;
```

getpolyline allows user input of a complete polyline by a call to a single DCAL routine. This routine takes most of the work out of designing a user interface for entering a series of polyverts which make up a polyline or surface of revolution.

The many parameters of getpolyline allow as much control over its operation as possible. With getpolyline you can use some function keys (those not used by getpolyline itself). getpolyline automatically allows the user to enter chains of polyverts which contain bulges using two-point arc, three-point arc, and tangent arc inputs, with all dragging handled automatically.

- msg is a string constant or variable containing the message you want to appear on the message line during the entering of normal vertices.

- viewmode takes a value of one of the viewing projection constants.

- Set dodraw to true when you want getpolyline to draw the polyline as the user enters each polyvert. When dodraw is false, the chain of polyverts are not drawn as they are entered.

- Set doclosed to true when you want getpolyline to automatically present the Closed label key to the user and handle the toggling of this flag; then the variable isclosed reflects the status of this toggle.
  When doclosed is false, getpolyline does not handle the toggling of the closed field. In this case, whether or not the chain of polyverts is closed is determined by

the current value of the variable isclosed.

- init is a Boolean variable which indicates to getpolyline that you are starting a new input sequence. When init is true, the variables frst and last are automatically set to nil. After the first call to getpolyline, init is set to false. In this way, getpolyline is reentrant. You can trap the function keys, process them accordingly, call getpolyline again, and you are at the same place as before. Simply reset init to true when you want to initiate a new input sequence.

- getpolyline returns an integer value which should be interpreted in the same way as a call to getpoint. The return value of getpolyline is either of the constants res_normal or res_escape. When the return value is res_normal, a valid chain of polyverts has been captured and entered into the database. This chain may be processed accordingly. The parameters frst and last contain the addresses of the first and last polyverts in the chain. The parameter isclosed is true when the chain of polyverts is closed (as indicated by the user), or false when the chain of polyverts is open.

- When one of the function keys not used by getpolyline is pressed, the return value from getpolyline is res_escape. getpolyline uses keys (F1), (F2), (F3), (F5), (S7), (S8) and (S0). Don't assign labels to these keys because they will be overwritten.

- The variable key contains the keycode of the pressed function key. You can test this variable and process it accordingly. As is the case with the function getpoly, you need only execute lblsinit; lblson is performed automatically by getpolyline.

The following portion of DCAL code provides a skeleton for the use of getpolyline. In this example, the viewing projection may only be orthographic as indicated by the vmode_orth parameter. In this example getpolyline controls the drawing of the polyline during its operation, and allows the user to toggle the closed option on or off.

When the polyline is closed, the variable isclosed is returned with a value of true, otherwise it is false. frst and last return the address of the first and last polyverts created during the call to getpolyline.

```
VAR
  done,
  init,
  isclosed  : boolean;
  key,
  result    : integer;
  frst,
  last      : pvtaddr;

...

done := false;
init := true;
closed := true;
REPEAT
```

```
  lblsinit;
  { lblson done by getpolyline }

  result := getpolyline ('Enter the
    next point of polyline.',
    vmode_orth, true, true, init,

    isclosed, frst, last, key);

  IF result = res_normal THEN
    { A valid chain of polyverts
     was entered. Process the
     chain of polyverts. }
  ELSIF result = res_escape THEN
    { A function key was pressed. }
    IF key = s0 THEN
      done := true;
    END;
  END;
UNTIL done;
```

pline_area

```
FUNCTION pline_area (frst, last : IN pvtaddr) : real;
```

pline_area calculates and returns the area enclosed by a polyline. The area returned may be positive or negative depending upon the sense of the polyline. Usually you want the absolute value of the resulting area so an appropriate call to absr is included in the expression. pline_area assumes that the polyline is closed regardless of the status of the entity to which the chain of polyverts is attached.

frst and last are the addresses of the first and last polyverts defining the polyline.

pline_centroid

```
PROCEDURE pline_centroid (frst, last : IN pvtaddr;
                          totarea : IN OUT real;
                          frstmoment, centroid : IN OUT point;
                          first, add : IN boolean);
```

pline_centroid calculates the first moment and centroid of an area described by a polyline. This procedure calculates centroids of areas made of multiple subareas, each described by a polyline, and allows for both positive and negative areas.

- frst and last are the addresses of the first and last polyverts in the chain making up each loop.

- When the flag first is true, pline_centroid automatically assumes that this is the first or only call of a sequence, and initializes the parameters totarea, frstmoment, and centroid prior to initiating calculations. first should be set to false when pline_centroid is called the second or later time in a sequence of calls in order to accumulate calculations for multiple areas.

- totarea is the cumulative total area of the polyline and used for calculating the centroid of areas containing multiple subareas and/or voids.

- frstmoment is the cumulative first moment of the area of the polyline and used for calculating the centroid of areas containing multiple subareas and/or voids.

- centroid is the centroid of the polyline or composite polyline when multiple subareas and/or voids are considered.

- When the flag add is true, the area of the polyline passed to pline_centroid is assumed to be a positive area in the calculations. When the flag add is false, the area of the polyline passed to pline_centroid is assumed to be negative or a void in the calculations.

The following example assumes that mode is a mode_type variable containing a selection set of entities of type polyline. The first entity in the selection set is assumed to be a positive area encompassing a series of voids. The remaining polylines are assumed to be voids within this area. This example computes the centroid of the resulting area:

```
VAR
  ent   : entity;
  addr   : entaddr;
  mode   : mode_type;

  first  : boolean;
  addit  : boolean;
  area   : real;
  frstmom : point;
  centroid : point;
...

  first := true;
  addit := true;
  { First time through, initialize
   and add result--subsequent times
   subtract the result. }
  addr := ent_first (mode);
  WHILE ent_get (ent, addr) DO
    addr := ent_next (ent, mode);

    pline_centroid (ent.plnfrst,
     ent.plnlast, totarea,
     frstmom, centroid,
     first, addit);

    first := false;
    addit := false;
  END;

  { centroid now contains the centroid
   of the polyline with voids. }
```

pline_perim

```
FUNCTION pline_perim (frst, last : IN pvtaddr) : real;
```

pline_perim calculates the perimeter of a polyline.

pline_perim returns the perimeter of the polyline. pline_perim assumes that the polyline is closed regardless of the status of the entity to which the chain of polyverts is attached.

frst and last are the addresses of the first and last polyverts describing the polyline.

polyvert_add

```
PROCEDURE polyvert_add (pv : IN OUT polyvert;
                        frst, last : IN OUT pvtaddr);
```

polyvert_add takes the polyvert contained in pv and adds it to the end of the chain of polyverts for which frst and last are the addresses of the first and last polyvert in the chain.

When frst and last are part of an entity you must perform ent_update to ensure that the entity itself is properly updated in the database.

For example:

```
  polyvert_init (pv);
  pv.shape := pv_vert;
  pv.bulge := 0.0;
  pv.pnt.x := 10.0;
  pv.pnt.y := 50.0;
  pv.pnt.z := 0.0;
  polyvert_add (pv, ent.plnfrst, ent.plnlast);
  ent_update (ent);
```

polyvert_copy

```
PROCEDURE polyvert_copy (
                oldfrst, oldlast, newfrst, newlast : IN OUT pvtaddr);
```

polyvert_copy makes a complete copy of a chain of polyverts.

Because polyvert_copy needs only the addresses of the first and last polyverts in both the existing (old) and new chains, polyvert_copy works correctly regardless of what the chains are attached to.

Initialize newfrst and newlast before calling polyvert_copy. Typically, newfrst and newlast are initialized to nil. Alternatively, newfrst and newlast may represent an existing chain of polyverts, in which case the old chain is appended to the new chain. To copy an entire entity which may contain a chain of polyverts, use ent_copy rather than polyvert_copy. ent_copy properly copies the chains of polyverts which make up entities of type polyline or surfaces of revolution. For complete control over which

polyverts are copied, use polyvert_copy.

For example, the following code copies an entity of type polyline without using ent_copy. Note that any attributes associated with the old entity are not copied to the new entity.

```
VAR
  oldent    : entity;
  newent    : entity;


  ent_init (newent);
  setnil (entaddr (newent.plnfrst));
  setnil (entaddr (newent.plnlast));
  newent.plnbase := oldent.plnbase
  newent.plnhite := oldent.plnhite
  newent.plnclose := oldent.plnclose
  polyvert_copy (oldent.plnfrst,
    oldent.plnlast, newent.plnfrst,
    newent.plnlast);
  ent_update (newent);
```

polyvert_count

```
FUNCTION polyvert_count (frst : IN pvtaddr) : integer;
```

polyvert_count returns the number of polyverts in a chain beginning with the polyvert pointed to by frst.

For example, to know how many polyverts are attached to a surface of revolution, use the following code:

```
numverts := polyvert_count  (ent.revfrst);
```

polyvert_del

```
PROCEDURE polyvert_del (pv : IN OUT polyvert;
                        frst, last : IN OUT pvtaddr);
```

polyvert_del deletes a polyvert from the chain of polyverts indicated by the fields frst and last. The polyvert is deleted from the database.

When frst and last are part of an entity the entity itself must be updated after the delete operation. For example, if pv had previously been read out of the database and belongs to an entity of type surface of revolution, then the following two calls must be made to delete the polyvert from the database and update the entity appropriately:

```
VAR
  ent   : entity;
  pv    : polyvert


  polyvert_del (pv, ent.revfrst,
```

```
                ent.revlast);
  ent_update (ent)
```

polyvert_get

```
FUNCTION polyvert_get (pv : OUT polyvert;
                       addr, frst, last : IN pvtaddr) : boolean;
```

polyvert_get reads a polyvert out of the database and places its contents into the variable pv.

- addr is the address of the polyvert to read.
- frst and last are the addresses of the first and last polyverts in the chain. frst and last must be valid addresses of polyverts in the database or an error occurs.

To read the entire list of polyverts associated with an entity of type polyline, use the following code:

```
VAR
  pv        : polyvert;
  addr      : pvtaddr;
  ent     : entity;
  frst, last  : pvtaddr


  addr := ent.plnfrst;
  WHILE polyvert_get (pv, addr) DO
    addr := pv.next

    { Operate upon the polyvert here. }

  END;
```

polyvert_init

```
PROCEDURE polyvert_init (pv : IN OUT polyvert);
```

polyvert_init initializes a polyvert variable before using it for a subsequent polyvert_add or polyvert_ins call.

Similar to ent_init and atr_init, polyvert_init initializes all the fields of the polyvert, in particular those which might affect the way the polyvert is added to or read from the database.

polyvert_ins

```
PROCEDURE polyvert_ins (pv : IN OUT polyvert;
                        locaddr : IN pvtaddr;
                        frst, last : IN OUT pvtaddr);
```

polyvert_ins is similar to polyvert_add but with polyvert_ins you can add a polyvert at

a particular location in the list of polyverts for a particular entity, not just at the end of the list.

- * locaddr is the address of the polyvert after which to insert the new polyvert. Typically, locaddr is determined from an editing operation which immediately precedes the adding of a polyvert to the chain.

- frst and last are the addresses of the first and last polyverts in the chain of polyverts for this particular entity.

polyvert_update

```
PROCEDURE polyvert_update (pv : IN OUT polyvert);
```

polyvert_update updates a polyvert in the database after it is changed.

This is only a valid operation when the polyvert was read from the database via polyvert_get or added to the database via polyvert_add or polyvert_ins.

NOTE: As is the case with most database operations, simply changing the values of a variable does not update the database; an appropriate update call must be performed.

polyvert2ent

```
PROCEDURE polyvert2ent (pv : IN polyvert; ent : OUT entity);
```

polyvert2ent converts a polyvert into an entity of the appropriate type. polyvert2ent automatically accounts for the directionality of the arc.

When the polyvert has a shape of pv_vert, or the value of the bulge parameter for the polyvert is zero, the entity is of type entlin. When the polyvert has a shape of pv_bulge and the value of bulge is non-zero, the entity is of type entarc. When the distance between the vertex of the polyvert and the vertex of the next polyvert is zero, the entity is of type entmrk.

NOTE: The nextpnt field is used by polyvert2ent, and therefore must contain valid information.

Void Database Routines

The following routines manage voids in entities of type polygon or slab (entply or entslb). Voids are handled much like the polyverts which are attached to entities of type polyline and surface of revolution in that voids constitute copious data. Copious data is information which is required to completely describe an entity's geometry, but which is not fully contained in the record structure of a variable of type entity. Unlike polyverts, voids do not have their own record type, but instead use the type entity for their description. Voids do, however, have their own set of database routines. You must take care when using voids not to use any of the standard entity database management routines even though a void uses the same variable type as a standard entity.

A polygon or slab may have zero or more voids attached to it. There is no limitation (except for drawing file size) on the number of voids which may be attached to an entity of type polygon or slab. The routines described here are used only for the database management of voids attached to entities of type polygon and slab. The geometric calculation, manipulation, and integrity of voids is the responsibility of the programmer, and may be managed using DCAL's standard library of arithmetic and geometric routines.

For more information, see "Voids" in the DataCAD Reference Manual which contains a discussion of voids and the operations which may be performed upon them. Of particular importance are the concepts regarding geometric integrity, the geometric relationship of the master polygon or slab to the void, the relationship of the reference and offset face when working with slabs, and the rules regarding convex and degenerate polygons, slabs, and voids.

void_add

```
FUNCTION void_add (ent, void : IN OUT entity); boolean;
```

void_add adds a void to an entity of type polygon or slab. void_add returns true when a void is added.

- ent is the entity to which the void is added.
- void is an entity variable which is initialized using void_init, and contains the valid geometric description of a void corresponding to that particular entity.

The geometric integrity of the void relative to the entity is the responsibility of the programmer. No geometric checking is performed by void_add.

void_del

```
PROCEDURE void_del (ent, void : IN OUT entity);
```

void_del deletes a void from an entity of type polygon or slab.

- ent must be previously read from the database using the function ent_get and must contain a valid description of an entity of type polygon or slab.

- void must be previously read from the database using the function void_get or procedure void_get_di and be a void belonging to ent. The void is deleted from the entity and removed from the database

void_del_all

```
PROCEDURE void_del_all (ent : IN OUT entity);
```

void_del_all deletes all voids from an entity of type polygon or slab when any voids currently belong to the entity. When the entity does not have any voids associated with it, no action is taken.

void_del_all is functionally identical to the following code, but is more efficient:

```
VAR

  ent   : entity;
  void  : entity;
  addr  : entaddr;
...

addr := ent.plyfrstvoid;
WHILE void_get (void, addr) DO
  addr := void.next;
  void_del (ent, void);
END;
ent_update (ent);
```

void_get

```
FUNCTION void_get (void : OUT entity; addr : IN entaddr) : boolean;
```

void_get reads a void from the drawing database.

void_get is usually used in a WHILE loop to read the chain of voids attached to an entity of type polygon or slab. After each void is read, the data for that void may be interrogated or updated. void_get returns true when the void is successfully read out of the database. void_get returns false when there are no voids attached to the entity, or when the end of the chain of voids is reached (addr is equal to nil).

- void is an entity variable containing the contents of the void which is read from the database.

- addr is the address of the void.

The following code demonstrates this principle. In this example, ent is known to be an

entity of type polygon (entply):

```
VAR
  void    : entity;
  ent     : entity;
  addr    : entaddr;
...

  addr := ent.plyfrstvoid;

  WHILE void_get (void, vaddr) DO
     addr := void.next;

     { Operate on the void here. }

  END;
```

void_get_di

```
PROCEDURE void_get_di (void : OUT entity; addr : IN entaddr);
```

void_get_di works similarly to the function void_get except void_get_di is used when addr is not equal to nil, but points to a valid void.

void_init

```
PROCEDURE void_init (ent, void : IN OUT entity);
```

void_init initializes a variable of type entity for use as a void.

void_init should always be called prior to adding a void to an entity. It is good practice to call void_init before using an entity for any void database management operation.

ent must be an entity of type polygon or slab, and should be the entity to which the void is added.

void_update

```
PROCEDURE void_update (void : IN OUT entity);
```

void_update updates a void in a drawing database.

void_update is usually called after a void is read from the database using void_get and its geometry is altered. As is the case for entities, simply altering the contents of the void variable do not update the description of the void in the drawing database. void_update makes the drawing database reflect the changes made to the variable void.

CHAPTER 7

Processing Routines

Math Routines

The following routines are routines for handling arithmetic data, geometry, and modeling matrices.

absi

```
FUNCTION absi (i : integer) : integer;
```

absi returns the absolute value of an integer expression.

absr

```
FUNCTION absr (rl : real) : real;
```

absr returns the absolute value of a real expression.

acos

```
FUNCTION acos (rl : real) : real;
```

acos returns the inverse cosine of its argument rl (the angle whose cosine is rl).

asin

```
FUNCTION asin (rl : real) : real;
```

asin returns the angle whose sine is the argument rl (the inverse sin of rl).

atan

```
FUNCTION atan (rl : real) : real;
```

atan returns the angle whose tangent is the argument rl. Note that this angle is always in the first or fourth quadrant. A more useful function is atan2.

atan2

```
FUNCTION atan2 (dx, dy : real) : real;
```

atan2 returns the angle given by the delta values dx and dy. This is similar to atan (dy / dx), but the angle is in the correct quadrant and you don't need a special test for dx = 0.

chr

```
FUNCTION chr (i : integer) : char;
```

chr is a type conversion procedure. It takes an integer and returns the corresponding character value. The high eight bits of the integer are ignored.

cos

```
FUNCTION cos (angle : real) : real;
```

cos returns the cosine of the angle that is its argument.

exp

```
FUNCTION exp (x : real) : real;
```

exp returns e raised to the x power (the inverse of natural logarithm).

expt

```
FUNCTION expt (a, x : real) : real;
```

expt returns a raised to the x power. x can be fractional but not negative.

float

```
FUNCTION float (i : integer) : real;
```

float is a type conversion function. It returns a real value that is the same as its integer argument.

For example, float (-4) returns -4.0.

This function is needed because DCAL does not provide automatic type conversions.

intand

```
FUNCTION intand (int1, int2 : integer) : integer;
```

intand returns the bitwise AND of its two arguments.

For instance, intand (326, 211) returns 66. (326 = Binary 101000110,
211 = Binary 011010011, and
066 = Binary 001000010)

intor

```
FUNCTION intor (int1, int2 : integer) : integer;
```

intor returns the bitwise OR of its two arguments.

For example, intor (326, 211) returns 471. (326 = Binary 101000110,
211 = Binary 011010011, and

471 = Binary 111010111)

intxor

```
FUNCTION intxor (int1, int2 : integer) : integer;
```

intxor returns the bitwise exclusive OR of its two arguments.

For example, intxor (326, 211) returns 405. (326 = Binary 101000110,
211 = Binary 011010011, and
066 = Binary 110010101)

log

```
FUNCTION log (x : real) : real;
```

log returns the natural logarithm (base e) of its argument.

max

```
FUNCTION max (a, b : real) : real;
```

max returns the maximum of its two arguments.

min

```
FUNCTION min (a, b : real) : real;
```

min returns the minimum of its two arguments.

odd

```
FUNCTION odd (i : IN integer) : boolean;
```

odd returns true when the integer i is odd. When the integer i is even, odd returns false.

ord

```
FUNCTION ord (ch : char) : integer;
```

ord is a type conversion function. It takes a character and returns the integer ASCII value.

order

```
PROCEDURE order (a, b : real; min, max : OUT real);
```

order returns in min the minimum of a and b, and returns in max the maximum of a and b.

round

```
FUNCTION round (rl : real) : integer;
```

round takes a real and returns the nearest integer to it.

round4

```
FUNCTION round4 (rl : IN real) : longint;
```

round4 takes a real number and rounds it to the nearest long integer.

sin

```
FUNCTION sin (ang : real) : real;
```

sin returns the sine of the angle that is passed to it.

sqr

```
FUNCTION sqr (rl : real) : real;
```

sqr returns the square of its argument. This routine is slightly faster than multiplying something by itself (especially when the argument is an expression).

sqrt

```
FUNCTION sqrt (rl : real) : real;
```

sqrt returns the square root of its argument.

A run-time error occurs when the argument is negative.

tan

```
FUNCTION tan (ang : IN real) : real;
```

The function tan returns the tangent of the angle ang.

When ang is a multiple of +pi/2 or -pi/2, tan returns a value, but the value is undefined.

trunc

```
FUNCTION trunc (rl : real) : integer;
```

trunc is similar to round, except the returned value is truncated toward zero.

For example, trunc (3.8) returns 3.

trunc4

```
FUNCTION trunc4 (rl : IN real) : longint;
```

trunc4 takes a real number and truncates it to the nearest long integer.

Geometric Routines

The routines in this section let you manipulate geometric data, such as the geometry of lines and arcs. This refers to geometric concepts, not necessarily to DataCAD entities.

addpnt

```
PROCEDURE addpnt (pt1, pt2 : point; respnt : OUT point);
```

addpnt (pt1, pt2, respnt) is equivalent to:

```
 respnt.x := pt1.x + pt2.x;
 respnt.y := pt1.y + pt2.y;
 respnt.z := pt1.z + pt2.z;
```

angle

```
FUNCTION angle (pt1, pt2 : point) : real;
```

angle returns the angle (in radians) between two lines. The first line is defined by the two IN parameter points. The second line goes through pt1 and is parallel to the x axis. The angle is measured from zero degrees in a counterclockwise direction. The return value is equivalent to that given by atan2 (pt2.x - pt1.x, pt2.y - pt1.y).

Although angle ignores the z coordinate, the z coordinate should be initialized to some valid value.

For example, when:

```
  pt1.x := 0.0;
  pt1.y := 0.0;
  pt2.x := 2.6;
  pt2.y := 5.7;
  x := angle (pt1, pt2);
```

x is equal to 1.142848 radians, or about 65.5 degrees.

angnormalize

```
PROCEDURE angnormalize (ang : IN OUT real);

The return value of ang will be less than TwoPi and greater than or
equal to zero.
```

between

```
FUNCTION between (pt1, pt2, testpt : point) : integer;
```

between determines when a point testpt is on, between, or beyond the line segment defined by pt1 and pt2.

between has the following return values:

-2	beyond pt2
-1	beyond pt1
0	between pt1 and pt2
1	on pt1
2	on pt2

The following conclusions can be drawn based on the return value:

< 0	The point is beyond either end point.
<> 0	The point is on or beyond either end point.
= 0	The point is between both end points.
>= 0	The point is between or on both end points.
> 0	The point is on either end point.
<= 0	The point is not on an end point.

betweenang

```
FUNCTION betweenang (tstang, ang1, ang2 : real);
```

betweenang returns true when tstang is between ang1 and ang2, otherwise, false returns. Between is defined in a counterclockwise direction.

For example:

betweenang (0.0, -pi, pi) returns true betweenang (0.0, 1.0, pi) returns false

cart_cylind

```
PROCEDURE cart_cylind (x, y, z : real;
                       rad, planang, zdis : OUT real);
```

cart_cylind converts from cartesian to cylindrical coordinates.

For example, the point (x, y, z) converts to (rad, planang, zdis).

cart_sphere

```
PROCEDURE cart_sphere (x, y, z : real;
                       rad, planang, riseang : OUT real).
```

cart_sphere converts from cartesian to spherical coordinates.

For example, the point (x, y, z) converts to (rad, planang, riseang).

circ3pt

```
FUNCTION circ3pt (pt1, pt2, pt3 : point;
                  cent : OUT point;
```

```
                    rad : OUT real) : boolean;
```

circ3pt calculates the circle that goes through any three points, pt1, pt2, and pt3.

- The center of the circle is returned in cent.
- The radius of the circle is returned in rad.
- When the three points do not define a circle (they are colinear), circ3pt returns false, otherwise true returns.

clip

```
FUNCTION clip (pt1, pt2 : IN OUT point;
               min, max : point;
               doz boolean) : boolean;
```

clip is used to clip a line segment to a box. clip works either in three dimensions (x, y, and z) or in two dimensions (x and y) based on the parameter doz.

- When doz is true, clip works in three dimensions, otherwise it works in two.
- min and max are the bounding corners of the box.
- pt1 and pt2 are the two points defining the ends of the line segment.

When clip returns false, the line segment defined by pt1 and pt2 is not in the bounding box. When true returns, pt1 and pt2 change to the portion of the line segment inside the box.

crossprod

```
PROCEDURE crossprod (pt1, pt2, pt3 : point; result : IN OUT point);
```

crossprod calculates the vector cross product of two input vectors.

The notation used is somewhat non-standard and is defined to make classifying polygons simple.

The first vector is defined as going from pt1 to pt2. The second vector goes from pt2 to pt3. The resulting vector is returned as result. Think of result as a 3D vector, with the x, y, and z values representing the x, y, and z components of the vector.

crossz

```
FUNCTION crossz (pt1, pt2, pt3 : point) : real;
```

crossz is similar to crossprod, except it returns the z value of the resulting cross product. crossz is useful to determine which side of a two-dimensional line an

arbitrary point lies on.

The first vector is defined as going from pt1 to pt2. The second vector goes from pt2 to pt3. When a line goes from pt1 to pt2, crossz returns a positive number if pt3 lies to the left of the line and a negative number if pt3 lies to the right of the line. If pt3 lies on the line, zero is returned.

cylind_cart

```
PROCEDURE cylind_cart (rad, planang, zdis : real;
                       x, y, z : OUT real);
```

cylind_cart converts from cylindrical to cartesian coordinates.

For example, (rad, planang, zdis) is converted to (x, y, z).

degrees

FUNCTION degrees (angle : real) : real;

degrees converts its argument (which is given in radians) to degrees. All DCAL functions (except radians, see below) expect their arguments in radians.

dis_from_arc

```
FUNCTION dis_from_arc (center : IN point;
                       radius, begang, endang : IN real;
                       testpnt : IN point) : real;
```

dis_from_arc returns the square of the shortest distance from an arc specified by its center, radius, beginning angle, and ending angle to a point.

- center is the center point of the arc.
- radius is the radius of the arc.
- begang and endang are the beginning and ending angles of the arc respectively.
- testpnt is the point from which to calculate the distance to the arc.
- dis_from_arc returns the square of the distance between the point and the arc.

dis_from_line

```
FUNCTION dis_from_line (pt1, pt2, tstpt : point) : real;
```

dis_from_line returns the square of the shortest distance from the point tstpt to the infinite line defined by pt1 and pt2.

dis_from_seg

```
FUNCTION dis_from_seg (pt1, pt2, tstpt : point) : real;
```

dis_from_seg returns the square of the shortest distance from the point tstpt to the line segment defined by pt1 and pt2.

distance

```
FUNCTION distance (pt1, pt2 : point) : real;
```

distance returns the distance between the two points, pt1 and pt2.

Although distance ignores the z coordinates, the z coordinate should be initialized to a valid value.

For example, when:

```
  pt1.x := 0.0;
  pt1.y := 0.0;
  pt2.x := 2.6;
  pt2.y := 5.7;
  x := distance (pt1, pt2);
```

x is equal to 6.265.

dotprod

```
FUNCTION dotprod (pt1, pt2, pt3 : point) : real;
```

dotprod computes the scalar dot product of two vectors.

See "crossprod" for the definition of the parameters pt1, pt2, and pt3.

fixangs

```
PROCEDURE fixangs (bang, eang : IN OUT real);
```

Forces (0 <= begang AND begang < twoPI) AND (begang < endang).

gridcalc

```
PROCEDURE gridcalc (origin : IN point;
                    ang : IN real;
                    gridIn, gridout : OUT modmat);
```

gridcalc calculates two modeling matrices which are the transformations required for snapping a point to a grid which is defined by an origin and an angle of rotation. The procedure gridcalc is typically called once prior to using gridsnapto procedure for snapping a point to a grid. By using modeling matrices for the definition of the grid orientation, gridsnapto is efficient in its calculations, particularly when used inside

repetitive loops.

Procedures gridcalc and gridsnapto are coordinate system independent when used together and may be used in any real number, cartesian coordinate system.

- ang is the angle of rotation of the grid coordinate system about its specified origin.
- origin is the origin of the grid in the grid coordinate system. The grid need not be defined in the global world coordinate system.
- gridIn is the modeling transformation from the base coordinate system to the translated and rotated grid coordinate system.
- gridout is the modeling transformation from the translated and rotated grid coordinate system to the base coordinate system.

gridsnapto

```
PROCEDURE gridsnapto (pnt : IN OUT point;
                      gridsize : IN point;
                      gridin, gridout : IN modmat);
```

gridsnapto snaps a point to a grid whose origin and angle of rotation are specified by two modeling matrices, and whose grid size is specified by an x and a y offset.

- pnt is the point that is snapped to the grid.
- gridsize is a point whose x and y-coordinates describe the size of the grid in the translated and rotated coordinate system described by the modeling matrices gridin and gridout.
- gridin and gridout are modeling matrices which are ordinarily computed using the procedure gridcalc. See "gridcalc" for a discussion of the definition of these two matrices.

intr_arcarc

```
FUNCTION intr_arcarc (cent1, cent2 : point;
                      rad1, rad2, bang1, eang1, bang2, eang2 : real;
                      int1, int2 : OUT point) : integer;
```

intr_arcarc computes the intersections of two arcs.

The first arc is centered at cent1, its radius is rad1, and its beginning and ending angles are bang1 and eang1. The second arc is similarly defined.

intr_crcarc

```
FUNCTION intr_crcarc (cent1, cent2 : point;
```

```
                      rad1, rad2, bang, eang : real;
                      int1, int2 : OUT point) : integer;
```

intr_crcarc is similar to intr_crccrc except that cent2 is the center of an arc whose beginning and ending angles are bang and eang, respectively.

intr_crccrc

```
FUNCTION intr_crccrc (cent1, cent2 : point;
                      rad1, rad2 : real;
                      int1, int2 : OUT point) : integer;
```

intr_crccrc computes the intersections (if any) between two circles.

- int1, int2, and the return value have the same meaning as they do in intr_linarc.
- The first circle has its center at cent1 and its radius is rad1.
- The second circle has its center at cent2 and its radius is rad2.

intr_linarc

```
FUNCTION intr_linarc (center : point;
                      radius, bang, eang : real;
                      pt1, pt2 : point;
                      int1, int2 : OUT point;
                      segment : boolean) : integer;
```

intr_linarc computes the intersection of either a line or a line segment with an arc.

- intr_linarc returns the number of intersections found.
  When intr_linarc returns 0, the line (or line segment) does not intersect the arc.
  When the return value is 1, int1 contains the only intersection.
  When the return value is 2, int1 and int2 contain both of the intersections.
- center is the center of the arc, radius is its radius, and bang and eang are its beginning and ending angles.
- When segment is true, pt1 and pt2 are considered the end points of a line segment, otherwise they are points defining an infinite line.

intr_lincrc

```
FUNCTION intr_lincrc (center : point;
                      radius : real;
                      pt1, pt2 : point;
                      int1, int2 : OUT point;
                      segment : boolean) : integer;
```

intr_lincrc is similar to intr_linarc, but computes the intersection of a line (or line segment) and a circle.

- The circle being tested for intersection has its center at center and its radius is radius.

- When segment is true, pt1 and pt2 are considered the end points of a line segment, otherwise they are points defining an infinite line.

intr_linlin

```
FUNCTION intr_linlin (pt1, pt2, pt3, pt4 : point;
                      intr : OUT point;
                      segments : boolean) : boolean;
```

intr_linlin computes the intersection point of two lines or two line segments.

- The two lines (or segments) are defined from pt1 to pt2 and from pt3 to pt4.

- When the parameter segments is true, the intersection must be on or between the end points. When segments is false, the intersection can be anywhere on the two lines.

- If the lines do intersect (and the intersection is between the two end points if segments is true), then the function returns true, otherwise false returns.

- When true is returned, intr is the intersection point.

linelen3

```
FUNCTION linelen3 (pt1, pt2 : point) : real;
```

linelen3 returns the square of the distance from pt1 to pt2. Note that this is the three-dimensional distance in space, not the two dimensional distance returned by distance.

meanpnt

```
PROCEDURE meanpnt (pt1, pt2 : point; meanpt : IN OUT point);
```

meanpnt finds the point midway between 2 points. It is equivalent to:

```
meanpt.x := (pt1.x + Pt2.x) / 2.0;
meanpt.y := (pt1.y + Pt2.y) / 2.0;
meanpt.z := (pt1.z + Pt2.z) / 2.0;
```

mulpnt

```
PROCEDURE mulpnt (pt : point; scale : real; respnt : OUT point);
```

`mulpnt (pt, scale, respnt)` is equivalent to:

```
respnt.x := pt.x * scale;
respnt.y := pt.y * scale;
respnt.z := pt.z * scale;
```

pnt_in_poly

```
FUNCTION pnt_in_poly (testpnt : IN point;
                      pnt : IN pntarr;
                      npnt : IN integer;
                      min, max : IN point) : integer;
```

pnt_in_poly determines if a point lies within, on, or outside of the boundaries of a polygon. The polygon is described as an array of up to 36 points. The polygon may be arbitrarily concave or convex, but may not be degenerate.

Only the x and y-coordinates of the test point and the polygon vertices are considered, but the z-coordinates of the test point and the polygon vertices must be initialized to valid real numbers.

- pnt_in_poly returns the value 1 when testpnt lies within the boundary of the polygon, 0 when testpnt lies on the boundary of polygon, and -1 when testpnt lies outside the boundaries of the polygon.

- testpnt is the point to test.

- pnt is the array of vertices of the polygon, up to a maximum of 36.

- npnt is an integer indicating the number of vertices in the polygon. npnt must be at least 3, but no more than 36.

- min and max contain the extents of the polygon. min and max are used for trivial rejection to increase performance when pnt_in_poly is used in a repetitive manner.

pntscolinear

```
FUNCTION pntscolinear (pt1, pt2, pt3 : point;
                       epsilon : real) : boolean;
```

pntscolinear will return true if pt1, pt2 and pt3 are colinear + or - epsilon..

polar

```
PROCEDURE polar (pt1 : point; ang, dist : real; pt2 : OUT point);
```

polar takes a point pt1 and returns in pt2 the point that is at a distance dist and angle ang from that point.

poly_fix

```
PROCEDURE poly_fix (pnt : IN OUT pntarr;
                    npnts : IN integer;
                    min, max : IN OUT point);
```

poly_fix performs three operations upon a polygon to normalize the polygon for use in some algorithms. poly_fix tests to insure that the points of the polygon are specified in a counterclockwise order. When they are not, the order of points is reversed.

Secondly, poly_fix insures that pnt [1] of the polygon is the point which has the minimum x-coordinate among the points which have the minimum y-coordinate. This normalization allows some algorithms which test for inclusion in a polygon to be used when they would not otherwise operate properly. Third, poly_fix computes the extents of the polygon.

- pnt is the array of points which are the vertices of the polygon, up to a maximum of 36.

- npnts is an integer indicating the number of vertices in the polygon. npnts should be a number between 3 and 36 inclusive.

- min is the minimum extents of the polygon and max is the maximum extents of the polygon.

project

```
PROCEDURE project (pt1, pt2 : point; tstpt : IN OUT point);
```

project projects tstpt onto the line formed by pt1 and pt2.

On entry, tstpt is any point. On exit, tstpt is on the line (not line segment) formed by pt1 and pt2 perpendicular to the original value of tstpt.

radians

```
FUNCTION radians (ang : real) : real;
```

radians takes an angle ang in degrees and converts it to radians.

```
radians (180.0) returns a number that is equal to pi.
```

setpoint

```
PROCEDURE setpoint (pt : OUT point; rl : real);
```

setpoint sets the three coordinates of pt to be equal to rl. This is identical to:

```
pt.x := rl;
pt.y := rl;
pt.z := rl;
```

but executes faster, generating less code.

sphere_cart

```
PROCEDURE sphere_cart (rad, planang, riseang : real;
                       x, y, z : OUT real);
```

sphere_cart converts the point (rad, planang, riseang) from spherical coordinates to its cartesian equivalent (x, y, z).

subpnt

```
PROCEDURE subpnt (pt1, pt2 : point; respnt : OUT point);

subpnt (pt1, pt2, respnt) is equivalent to:  respnt.x := pt1.x-pt2.x;
                                             respnt.y := pt1.y-pt2.y;
                                             respnt.z := pt1.z-pt2.z;
```

swappnt

```
PROCEDURE swappnt (pt1, pt2 : IN OUT point);

This procedure sets p1 = p2 and p2 = p1;
```

Modeling Matrix Routines

The routines in this section manipulate modeling matrices. A modeling matrix is a four- by-four matrix of real numbers. With it, any series of arbitrary three-dimensional scalings, rotations, and translations can be applied to any entity that has a modeling matrix in its definition, such as a symbol or a dome. In addition, using the procedure xformpt, you can apply these same transformations to any point. These scalings, rotations, and enlargements can be applied one after the other, or concatenated.

For examples of the usage of modeling matrices, see the "Sample Macros" chapter.

catenlrel

```
PROCEDURE catenlrel (mat : IN OUT modmat;
                     xsc, ysc, zsc : real;
                     pt : point);
```

catenlrel is very similar to setenlrel, but concatenates the relative enlargement instead of setting it.

catmat

```
PROCEDURE catmat (mat1 : IN OUT modmat; mat2 : modmat);
```

catmat concatenates two modeling matrices, mat1 and mat2.

catrotate

```
PROCEDURE catrotate (mat : IN OUT modmat;
                     ang : real;
                     axis : integer);
```

catrotate adds a rotation of ang around axis to the current transformation of mat.

See the description of setrotate for the usage of ang and axis.

catrotrel

```
PROCEDURE catrotrel (mat : IN OUT modmat;
                     ang : real;
                     axis : integer;
                     pt : point);
```

catrotrel is similar to setrotrel, but the rotation is concatenated onto the existing transformation of mat.

catscale

```
PROCEDURE catscale (mat : IN OUT modmat; xsc, ysc, zsc : real);
```

catscale concatenates a scaling onto the current transformation assigned to mat.

See "setscale".

cattran

```
PROCEDURE cattran (mat : IN OUT modmat; dx, dy, dz : real);
```

cattran concatenates, or adds, a translation of (dx, dy, dz) to the current transformation assigned to mat.

invert

```
FUNCTION invert (matin : modmat;
                 matout : OUT modmat;
                 det : OUT real) : boolean;
```

invert calculates the inverse transformation for matin, if one exists.

false returns when no inverse exists, otherwise true returns. When the inverse does exist, the determinate of the matrix returns in det and the inverse matrix returns in matout.

matmmat

```
PROCEDURE matmmat (mat1, mat2 : modmat; result : OUT modmat;
```

matmmat multiplies two modeling matrices, mat1 and mat2, and places the result in result. Multiplying two matrices concatenates the transformations of mat2 onto mat1.

setenlrel

```
PROCEDURE setenlrel (mat : IN OUT modmat; xsc, ysc, zsc : real; pt :
                     point);
```

setenlrel sets mat to produce an enlargement relative to pt. This procedure is similar to setscale but the enlargement is not relative to the origin.

setident

```
PROCEDURE setident (mat : OUT modmat);
```

setident sets mat to have the identity transformation, that is, no affect at all. A point transformed by the identity matrix yields the original point.

setrotate

```
PROCEDURE setrotate (mat : OUT modmat; ang : real; axis : integer);
```

setrotate sets mat to have a rotation of ang about the axis axis.

The axis should be one of the built-in constants x, y, or z. Notice that the center of this rotation is around the origin.

setrotrel

```
PROCEDURE setrotrel (mat : IN OUT modmat; ang : real; axis : integer;
                     pt : point);
```

setrotrel is similar to setrotate but the rotation is relative to the point pt.

setscale

```
PROCEDURE setscale (mat : OUT modmat; xsc, ysc, zsc : real);
```

setscale sets mat to apply a scaling of xsc, ysc, and zsc along the x, y, and z axes, respectively. The scaling is relative to the origin.

settran

```
PROCEDURE settran (mat : OUT modmat; dx, dy, dz : real);
```

settran sets mat to have a translation of (dx, dy, dz). This is used to move a point or entity.

transpose

```
PROCEDURE transpose (mat : IN OUT modmat);
```

transpose swaps the rows and columns of the matrix mat. Row one becomes column one, row two becomes column two, etc.

xformpt

```
PROCEDURE xformpt (ptin : point; mat : modmat; ptout : OUT point);
```

xformpt transforms the point ptin according to the transformations that have been assigned to mat, and places the result in ptout. This is most often used to set up a transformation and then transform a series of points to determine their new values.

Viewing Routines

Use the routines in this section to manage and manipulate saved and displayed views in DataCAD. The basic unit of information used by DataCAD's viewing system is the view. A view, represented in DCAL as a variable of type view_type, represents two kinds of information:

A view can represent the information necessary to completely specify or calculate a viewing projection. In this case, a view is any one particular orientation of a drawing or model relative to the screen in a particular projection. For example, with a perspective projection, a view contains the location of the viewer, the direction in which the viewer is looking, clipping information, and window to viewport mapping information, as well as a variety of supporting information.

A view can represent the basic unit of information for the storage and retrieval of viewing parameters in a drawing database. In this case, a variable of type view_type is the data object used for maintaining a linked data structure consisting of sets of viewing parameters. DataCAD supports four types of projections: orthographic, parallel, perspective, and oblique.

Refer to the DataCAD Reference Manual for a discussion of the differences between views and projections, as well as the differences between each of the four types of projections. The DataCAD Reference Manual also includes instructions on how to define and create different views in each of the projection types. Many of the DCAL view calculation routines are used by DataCAD's viewing system, therefore the explanations in the DataCAD Reference Manual also apply to these routines.

Viewing Database Management Routines

The following routines manage views in the drawing database.

detail_get

```
FUNCTION detail_get (view  : IN OUT view_type,
                     addr  : IN viewaddr) : BOOLEAN; BUILTIN 236;
```

Works the same as the view_get routine (see details further down in this section).

Note that you will need to declare the procedure (including 'BUILTIN 236') in your code as it is not currently included in any of the standard include files.

view_add

```
PROCEDURE view_add (view : IN OUT view_type);
```

view_add adds a view to the list of saved views in a drawing database.

The variable view must be initialized using view_init, or previously read from the

database using view_get. view must contain a valid set of viewing parameters. The view is added to the chain of saved views at the end of the list.

Adding a view to the list of saved views does not make a view the current view, nor does it set the current viewing parameters to those of the view variable, nor does it refresh the screen. view_add is strictly a database function which adds the contents of the view variable to the list of saved views. view_add automatically stores a list of those layers which are on at the time with the view. In this way, a view is sensitive to the LayrSet toggle in the DataCAD 3D GotoView menu.

view_del

```
PROCEDURE view_del (view : IN OUT view_type);
```

view_del deletes a view from the list of saved views in a drawing database.

The view must be previously extracted from the database by a call to view_get. Do not attempt to delete a view which has not been previously read from the database using view_get.

view_first

```
FUNCTION view_first : viewaddr;
```

view_first returns the address of the first view in the list of saved views in a drawing database.

When the list of saved views is empty, view_first returns the value nil. view_first is used with view_get so you can linearly scan the list of saved views in a drawing database.

See "view_get".

view_flread

```
view_flread (view : IN OUT view_type; fl : IN OUT file); BUILTIN 238;
```

Reads the ASCII text file 'fl' into 'view'.

Note that you will need to declare the procedure (including 'BUILTIN 238') in your code as it is not currently included in any of the standard include files.

view_flwrite

```
PROCEDURE view_flwrite (view : IN OUT view_type;
                        fl   : IN OUT file); BUILTIN 239;
```

Writes 'view' to the ASCII text file 'fl'.

Note that you will need to declare the procedure (including 'BUILTIN 239') in your code as it is not currently included in any of the standard include files.

view_get

```
FUNCTION view_get (view : IN OUT view_type; addr : IN viewaddr) :
                   boolean;
```

view_get reads a view from the drawing database located at address addr.

- When addr is nil the list is either empty or the end of the list has been reached and view_get returns false. In this case, the view returned is undefined.

- When view_get is successful, true is returned and the variable view contains the view located at address addr. view_get is most often used with view_first to linearly scan the list of saved views.

For example, to scan the list of saved views and test for those which are orthographic projections, use the following code:

```
VAR
  view    : view_type
  addr    : viewaddr
...

  addr := view_first
  WHILE view_get (view, addr) DO
    addr := view.next
    IF view.projtype = vmode_orth THEN
      { Process orthographic
        projections here.}
    END;
  END;
```

view_init

```
PROCEDURE view_init (view : IN OUT view_type);
```

view_init initializes the fields of a variable of type view_type.

Call view_init before using a view_type variable, particularly when adding or deleting views from the database.

view_last

```
FUNCTION view_last : viewaddr;
```

view_last returns the address of the last view in the list of saved views in a drawing database.

When the list of saved views is empty, view_last returns a value of nil. Use view_last

to scan the list of views in reverse order or to test against the address of the last view.

view_update

```
PROCEDURE view_update (view : IN OUT view_type);
```

view_update updates a view in the list of saved views in a drawing database.

The view must be previously extracted from the database with a call to view_get. The view must also contain a valid set of viewing parameters. view_update is strictly a database function. To make the view the current view, use view_setcurr. view_update does not alter the current viewing parameters, nor will it refresh the screen.

Viewer Control and Interrogation Routines

The routines described in this section control the current viewing projection in various ways, or interrogate the current viewing parameters. The routines are most-frequently used during three-dimensional editing operations or during the specification of views.

inmat_curr

```
PROCEDURE inmat_curr (inmag : OUT modmat);
```

inmat_curr returns the value of the current input matrix. This matrix is applicable only in parallel projections, and contains the inverse of the complete viewing transformation.

inmat_curr is typically used with the function getpointp when entering or editing data in parallel projections as is done in the DC- Modeler. The matrix returned by inmat_curr represents the transformation required to change from world coordinates to the screen coordinate system used by getpointp.

scale_curr

```
FUNCTION scale_curr : integer;
```

scale_curr returns the number of the current two-dimensional window-to-viewport display scale.

The value that returns is an integer in the range of 1 to 18.

To determine the window-to-viewport scale factor corresponding to this integer number, use the following code:

```
VAR
  scale   : real;
  name    : string (8);

  scale_get (scale_curr, scale, name);
```

scale_get

```
PROCEDURE scale_get (num : IN integer;
                     scale : OUT real;
                     name : OUT string);
```

scale_get returns the display scale and name for a given scale number.

- scale is the ratio of the world coordinate system to the display coordinate system. DataCAD maintains up to 18 discrete drawing display scale factors.
  The user can edit these scale factors via the Settings/EditDefs/ Scales function. The DataCAD support file dcad.scl may also be edited directly so that any new drawing reflects these new scales. Therefore, the 18 numbered scales are not necessarily unique for any particular drawing nor from drawing to drawing.

- name is a string, up to eight characters in length, which is the name of the scale used in DataCAD's ToScale menu.

view_checkmode

```
PROCEDURE view_checkmode (projtype : IN integer);
```

view_checkmode checks the current viewing projection to determine if it is the same as projtype.

view_checkmode takes an integer parameter which must be equal to one of the constants vmode_orth, vmode_para, vmode_pers, vmode_oblq, or vmode_edit. The first four constants check for the projection which they represent as described above. vmode_edit indicates to view_checkmode that either an orthographic or a parallel projection is acceptable.

With vmode_edit, when the current projection is either orthographic or parallel no action is taken. vmode_edit is typically used for three- dimensional editing operations similar to the 3D Move function or the entering of slabs and polygons. These functions are valid in either an orthographic or parallel projection, but not in a perspective or oblique projection.

When the current viewing projection is equal to projtype, no action is taken. When the current viewing projection is not equal to projtype, the most recent view of that projection type is made current and the screen is refreshed.

view_currmode

```
FUNCTION view_currmode : integer;
```

view_currmode returns an integer value indicating the current viewing projection type.

view_currmode is typically used for determining the current viewing projection without having to call view_getcurr and examine the individual fields of a view_type variable. For this purpose, view_currmode is faster and easier to use.

The return value is equal to one of the constants vmode_orth, vmode_para, vmode_pers, or vmode_oblq. These values represent orthographic, parallel, perspective, or oblique projections.

The following example prints a message indicating the current viewing projection:

```
VAR
  vmode   : integer;
...

  vmode := view_currmode;
  IF vmode = vmode_orth THEN
     wrterr (Orthographic.);
  ELSIF vmode = vmode_para THEN
     wrterr (Parallel.);
  ELSIF vmode = vmode_pers THEN
     wrterr (Perspective.);
  ELSIF vmode = vmode_oblq THEN
     wrterr (Oblique.);
  END;
```

view_getcurr

```
PROCEDURE view_getcurr (view : IN OUT view_type);
```

view_getcurr sets the view_type variable view to the current viewing parameters. view_getcurr does not read from or affect the list of saved views.

view_getcurr does not alter or initialize any of the following fields in a view_type variable: addr, next, prev, name, currlyr, frstlyr, lastlyr, or togglelyr.

Use the following code to load the current viewing parameters into a view variable, and then add this view to the list of saved views in a drawing database:

```
VAR
  view  : view_type;
...

  view_init (view);
  view_getcurr (view);
  view_add (view)
```

To read, update, add to, or delete from the list of saved views, refer to "Viewing Database Management Routines" in this chapter.

view_setcurr

```
PROCEDURE view_setcurr (view : IN OUT view_type;
                        redraw : IN boolean);
```

view_setcurr sets the current view to reflect the parameters and projection contained in the variable view.

The view need not be read out of the database, but must contain a valid set of viewing parameters. When the view is not read from the database, it should be initialized using view_init prior to modifying the fields of the view.

When redraw is true, the screen is refreshed after setting the current viewing parameters.

When redraw is false, the current viewing parameters are altered, but the screen is not refreshed. In this case, if the user presses (Esc), or any subsequent operation is performed which implicitly forces a refresh of the screen, the screen is refreshed using the new viewing parameters.

Typically, the view variable is initialized using view_init, then one of the view calculation routines is called, followed by a call to view_setcurr with refresh set to true. During a call to view_setcurr, some of the fields of the view may be changed depending upon the projection specified and the status of certain system variables. This is because some of the view parameters may be derived from these system parameters depending upon the type of projection. In any case, the fields addr, next, prev, name, lyrcurr, lyrfrst, lyrlast, flag1 and flag2 are not altered by view_setcurr.

view_setmode

```
PROCEDURE view_setmode (projtype : IN integer; refresh : IN boolean);
```

view_setmode explicitly sets the current viewing projection to the type passed by projtype.

- When the parameter refresh is true, the screen is refreshed (even if the current viewing projection is the same as projtype).

- When refresh is false, the screen is not refreshed, but the current viewing projection is still updated. In this case, if the user presses (Esc), or performs any other operation which automatically forces a refresh, the viewing projection contained in projtype is used. view_setmode explicitly changes among the four current viewing projections.

- Unlike view_checkmode, view_setmode takes an action even when the current viewing projection is equal to projtype.

Viewing Calculation Routines

The routines described in this section calculate a view of a specific projection based

upon a given set of input parameters. These routines operate only upon the view_type variable passed to them. These routines do not affect the current viewing parameters (use view_setcurr) nor do these routines affect any view which was saved in the drawing database (use view_add or view_update).

view_calcoblq

```
PROCEDURE view_calcoblq (view : IN OUT view_type;
                         center : IN point;
                         planang, shearang, shearfact : IN real;
                         oblqtype, scalenum : IN integer);
```

view_calcoblq calculates an oblique viewing projection. Calculate oblique viewing projections using one of two conventions: plan oblique or elevation oblique.

- When the calculation is made using the plan oblique convention, oblqtype is set to the constant oblqplan. When the calculation is made using the elevation oblique convention, oblqtype is set to the constant oblqelev.

- For plan oblique calculations, the parameters to view_calcoblq have the following meanings:

    - center becomes the center point of the display and is the point relative to which the viewing shear is calculated.
    - planang is the angle by which the drawing is rotated about the z-axis before applying the oblique shear.
    - shearang is ignored.
    - shearfact is the factor by which lines parallel to the z-axis are enlarged (nominal value is 1.0).
    - scalenum is an integer, from 1 to 18 inclusively, that indicates which of the currently-loaded display scales is used to calculate the window to viewport mapping for the view.
- For elevation calculations, the parameters to view_calcoblq have the following meanings:

    - center is the center point of the display and is the point relative to which the viewing shear is calculated.
    - planang is the angle about the z-axis by which the drawing is rotated to orient a particular vertical plane towards the viewer.
    - shearang is the angle up from the positive x- axis by which lines extending out of the screen are rotated as the shear is applied.
    - shearfact is the factor by which lines parallel to the axis of shear are enlarged (nominal value is 1.0).

- scalenum is an integer, from 1 to 18 inclusively, that indicates which of the currently-loaded display scales is used to calculate the window to viewport mapping for the view.

view_calcorth

```
PROCEDURE view_calcorth (view : IN OUT
view_type;
center : IN point;
scalenum : IN integer);
```

view_calcorth calculates an orthographic projection.

- center is the point in world coordinates which lies at the center of the screen in the resulting view.

- scalenum is an integer, from 1 to 18 inclusively, that indicates which of the currently-loaded display scales is used to calculate the window-to-viewport mapping for the view.

The absolute position of the drawing in the viewport is dependent upon the user's display device. Some display devices have a much larger resolution than others and as a consequence may display more or less of a drawing accordingly.

view_calcpara

```
PROCEDURE view_calcpara (view : IN OUT view_type;
                         center : IN point;
                         planang, riseang : IN real;
                         scalenum : IN integer);
```

view_calcpara calculates a parallel viewing projection.

- center is the center point of rotation for the view and indicates where the center of the screen should be located.

- planang is the angle by which the drawing is rotated about the z-axis to orient the view.

- riseang is the angle up from the xy-plane (horizon) by which the drawing is rotated to orient the view.

- scalenum is an integer, from 1 to 18 inclusively, that indicates which currently-loaded display scale to use to calculate the window-to-viewport mapping for the view.

view_calcpers

```
PROCEDURE view_calcpers (view : IN OUT view_type;
```

```
                    eyepnt, centpnt : IN point;
                    coneang : IN real);
```

view_calcpers calculates a perspective viewing projection.

- eyepnt is the location of the viewer's eye.

- eyepnt is in absolute world coordinates and the z-coordinate of eyepnt determines the height from the xy-plane (at z = zero) of the viewer's eye.

- centpnt is the point in the drawing or model at which the user is looking. centpnt defines the location of the picture plane relative to eyepnt. The picture plane is perpendicular to a line passing from eyepnt to centpnt, and passes through centpnt. centpnt as passed to view_calcpers is not necessarily equal to the field view.centpnt since the cone-of-vision angle may alter this parameter.

- coneang is the angle of the cone of vision and is the subtended angle between the right and left sides of the frustum of vision. coneang must be within the range 0.0 to 180.0 degrees, but may not be equal to 0.0, or 180.0. coneang determines the values of the window to viewport mapping parameters for the view. The cone of vision in the y-direction is typically less than in the x-direction since nearly all display devices have an aspect ratio which is less than 1.0. Unlike the other view calculation routines, the window-to-viewport mapping parameters for the view are derived from the cone of vision, instead of being explicitly stated.

Hidden Line Removal Routines

The following routine allows a DCAL application to call DataCAD's hidden line removal routine directly.

hide

```
FUNCTION hide (mode : IN mode_type;
               lyr : IN OUT layer;
               view : IN view_type;
               addimag, drawhidden : boolean) : boolean;
```

hide performs hidden line removal on the entities described by the mode_type variable mode.

- The mode variable must be properly initialized, and may contain any valid combination of mode parameters.

- view is a view_type variable which must contain a valid set of viewing parameters. view does not necessarily need to exist in the drawing database, but must be valid.

- When drawhidden is true, hidden lines are drawn. When drawhidden is false, hidden lines are not drawn.

- When addimag is true, the result of the Hidden Line Removal process is retained and stored on the layer passed to hide. When addimag is false, the result of the hidden line removal process is not retained, but is displayed on the screen only.

- The layer represented by lyr must be previously created and exist in the database prior to calling hide with addimag set to true; otherwise an error occurs. If you create a temporary layer in the database using lyr_init, make sure you subsequently remove this layer using lyr_term. Note that lyr_init and lyr_term are used only for free layers (layers not part of the main layer/entity structure), and are distinctly different from layers created by the routines lyr_create and lyr_del which add and delete layers from the main layer/entity structure.

The following example performs hidden line removal on all layers which are currently on, using the current view and projection, and stores the result on the active layer:

```
VAR
  lyr   : layer:
  mode  : mode_type;
  view  : view_type;
...
  lyr := getlyrcurr;
  view_get_curr (view);
  mode_init (mode);
  mode_lyr (mode, lyr_on);
  IF hide (mode, lyr, view, true, false) THEN
```

```
  { Successful run. Resulting image resides on active layer. }
ELSE
  { Process broken via DEL or END keys, or out of memory. }
END;
```

Hatching Routines

The declarations for the routines in this section are not built into the DCAL compiler, but require the inclusion of the file _hatch.inc.

The following routines invoke DataCAD's hatching system from a DCAL macro. The hatching interface to DCAL is extremely flexible without compromising the performance of the system. Hatch patterns are defined as a series of passes consisting of parallel scan lines. The scan lines can either be solid (continuous) or dashed.

hatch_mode may be called many times to create complex hatching patterns. For instance, the DataCAD hatch pattern brick consists of two calls to hatch_mode, one with a horizontal solid pattern and the second with a 90 degree rotated, dashed line pattern.

hatch_mode

```
PROCEDURE hatch_mode   (mode   : IN OUT mode_type;
                        sl     : IN OUT scanLineType;
                        zbase  : IN real;
                        zhite  : IN real;
                        origin : IN point;
                        ang    : IN real;
                        scale  : IN real;
                        htype  : IN integer;
                        lyr    : IN layer;
                        min    : IN point;
                        max    : IN point;
                        draw   : IN boolean;
                        brk    : IN OUT integer;
                        dobrk  : IN boolean);
```

(Note : I am not sure if the draw parameter eis in the right position. It was originally outside the brackets in the doco)

hatch_mode hatches a collection of entities.

hatch_mode takes a large number of parameters so you can have as much control as possible over the hatching process. This also provides you with the flexibility to create virtually any hatching pattern.

hatch_mode calls the same set of procedures that are used by the standard Hatch menu, and therefore executes at virtually the same speed.

- mode is a mode_type variable that indicates which entities to hatch. mode must be initialized using procedure mode_init and refers to a valid collection of entities in the drawing database.

- sl is a variable of type scanlinetype and describes the dash/space pattern for each scan line. Refer to "scanlinetype" for the definition of each field of this record.

- zbase is the z-base coordinate of any lines added to the database by hatch_mode.

- zhite is the z-height coordinate of any lines added to the database by hatch_mode.

- origin is the origin of the overall hatch pattern. Be careful not to confuse the parameter origin with the .origin field of a scanlinetype. In this case, origin is the overall origin of the hatch pattern whereas the .origin field of a scanLineType is the local origin for each scan line.

- ang is the angle of the hatch pattern and is measured using the standard DataCAD angle conventions.
  When ang is non-zero, rotation of the hatch pattern is performed about the point origin (in any event, ang should be initialized).

- scale is the overall scaling factor for the hatch pattern. A value for scale of 1.0 indicates that the values contained in the .dash field of sl are in world coordinates and are used directly.

- htype is a flag which may take on one of the three values htype_normal, htype_outer, or htype_ignore and indicates the manner in which hatch_mode handles interior objects in the hatching boundary. See the hatching constants in the "Constants" chapter.

- lyr is a variable of type layer and is the layer to which entities of type line (entlin) are added during the hatching process. lyr must be a valid layer which exists in the drawing database or an error occurs.

- min and max are the extents of the collection of entities to hatch as indicated by mode. The hatching entities' extents must be calculated before calling hatch_mode or hatching does not proceed correctly. The procedure ent_extent may be used in a loop for calculating the extents of a collection of entities described by a mode variable.

- draw is a Boolean flag which is true when the resulting lines of the hatching process are to be drawn as they are added to the drawing database. When draw is false, lines are not drawn as they are added to the drawing database.

- dobrk is a flag which indicates when the user can interrupt the hatching process by pressing (Del) or (End). When dobrk is true, interruption of the hatching process is allowed, otherwise it is not.

- brk is an integer containing the current status of the brkpress state as would be returned by the function brkpress. brk should be initialized to zero. brk is passed to hatch_mode as a formal parameter so that hatch_mode can be placed in a loop when more than one set of scan lines are required for a given hatch pattern. In this case, the calling macro knows if the process has been broken and by which key, through the function brkpress. When (Del) is pressed brk returns a value of -1; when (End) is pressed brk returns a value of 1.

Menu Routines

The routines in this section call the appropriate DataCAD menu. The user is taken to the menu and stays there until exiting the menu, at which time the macro continues. These are convenient for digitizer menus as well as for incorporating the default menus into your own macros.

menuarc2pt	menuarc3pt	menuarccentang	menuarccentarc
menuarccentchrd	menuarctan	menuarraycirc	menuarrayrect
menuchamfer	menuchange	menucleanup	menuclipcube
menucontrols	menucopy	menucrc3pt	menucrcdia
menucrcrad	menucurves	menucutwall	menuDataCAD3
menudirectry	menudisplay	menudivide	menudmension
menudoorswng	menueditplane	menuelevation	menuellipse
menuenlarge	menuerase	menufileio	menufillets
menufreehand	menugoodies	menugotoview	menugotoview3d
menugrids	menuhatch	menuhide	menuidentify
menuintrsect	menulayers	menulinetype	menulinkline
menulintsct	menumeasures	menumirror	menumove
menumovedrag	menuobjsnap	menuobjsnap	menuplanesnap
menuplotter	menupolygons	menurotate	menusaveimage
menusetobliq	menusetpersp	menusettings	menusettings3d
menuss	menustretch	menutangents	menutemplate
menutext	menutintsct	menutoscale	menuviewer
menuwalkthru	menuweldline	menuwindowin	menuwindows

The following menu calls all require the DC- Modeler package, otherwise, no action is taken and control returns immediately to the macro.

menublocks	menuchange3d	menucone	menucopy3d
menucylnhori	menucylnvert	menudome	menuedit3d
menuenlarge3d	menuentity3d	menuerase3d	
menuexplode3d	menumarker	menumeshsurf	menumove3d
menupartial	menupolygon	menupolyhori	menupolyincl
menupolyrect	menupolyvert	menurevsurf	menurotate3d
menuslab	menuslabhori	menuslabincl	menuslabrect
menuslabvert	menustretch3d	menutorus	menutrunccone
menuvoids			

Miscellaneous Routines

The routines in this section do not fall into any other category presented in this manual.

addr2ints

```
PROCEDURE addr2ints (addr : entaddr; int1, int2 : OUT integer);
```

addr2ints converts an address to two integers.

Use this when you write the address out to a file and must write out type integer. While the address of an entity does not change, you have no way of knowing when an entity was deleted and its location used for another entity.

assignProc

PROCEDURE assignProc (proc : procedure; x, y : integer);

assignProc assigns a procedure given by proc to a given box on the digitizer menu. The procedure must have no parameters. It must be declared at the outermost lexical level -- it cannot be a nested procedure.

l x and y are the location of the box to which the procedure is assigned. When the user picks the box (x, y), procedure proc executes. For example, look at the following code fragment.

```
PROCEDURE callMove;

BEGIN
  menumove;
END callMove;

BEGIN { main program }
  configTab (11.0 * 32.0, 11.0 * 32.0,
    22, 22, 9, 10, 17, 16);
  assignProc (callMove, 0, 0);
END main.
```

The procedure proc given as a parameter cannot be one of the built-in procedures, so a small wrapper is needed, such as callMove.

beep

```
PROCEDURE beep;
```

This procedure causes a beep or warning tone.

bitclear

```
FUNCTION bitclear (int, bitnum : integer) : integer;
```

- bitclear clears bit number bitnum in int.
- bitnum is is a value from 0 to 15.
- The return value is int with the specified bit cleared.
  `num := bitclear (num, 5);`
  clears the fifth bit of num.

`bitclear (100, 5)` returns 68 (100 base 2 = 1100100, 68 base 2 = 1000100)
`bitclear (100, 4)` returns 100 (100 base 2 = 1100100)

bitset

```
FUNCTION bitset (int, bitnum : integer) : integer;
```

bitset is identical in usage to bitclear, but the corresponding bit number is set, not cleared.

bittest

```
FUNCTION bittest (int, bitnum : integer) : boolean;
```

bittest is similar to bitclear and bitset, but it tests to see if a particular bit is set. When the number bitnum bit is set in int, bittest returns true; otherwise, bittest returns false.

`bittest (100, 6)` returns true `bittest (100, 5)` returns false

brkpress

```
FUNCTION brkpress : integer;
```

DataCAD defines two break keys, (End) and (Del).

brkpress returns one of three possible values. It returns 0 when neither break key is pressed, 1 when (End) is pressed, and -1 when (Del) is pressed. Note that the keyboard buffer is cleared whenever brkpress is called, regardless of what the return value is.

calctext

```
PROCEDURE calctext (ent : IN OUT entity; which : integer);
```

Calculates a text entity based upon the current datacad setting indicated by which. which may take the following values:

0	regular text
1	dimension text
2	forms text

calcdim

```
PROCEDURE calcdim (ent : IN OUT entity; pt1, pt2, pt3 : point);
```

Calculates an associative dimension entity based upon current datacad settings

clrGetName

```
PROCEDURE clrGetName (clr : IN integer; clrname : OUT string);
```

clrGetName reads the name of one of the 15 built-in colors.

In the standard (English) version of DataCAD, color 1 returns White, color 2 returns Red, etc. This procedure is useful to build menus that use the color names.

clrGetPen

```
FUNCTION clrGetPen (clr : IN integer) : integer;
```

clrGetPen returns the pen to use with a certain color when the drawing is plotted.

clrSetName

```
PROCEDURE clrSetName (clr : IN integer; clrname : IN string);
```

clrSetName sets the name of one of the 15 built- in colors.

clrSetPen

```
PROCEDURE clrSetPen (clr, pennum : IN integer;
```

clrSetPen sets the pen pennum that color clr is plotted in.

configTab

```
PROCEDURE configTab (xsize, ysize : real;
                     xmax, ymax, xlft, ybot, xrht, ytop : integer);
```

This procedure, along with assignProc, sets up digitizer menus within DataCAD.

- The physical dimensions of the overall tablet are given by xsize and ysize, in 32nds of an inch.
- xmax is the total number of boxes in the x dimension, and ymax is the total number of boxes in the y dimension.
- The lower left corner of the area reserved for the screen is xlft, ybot; the upper right corner of the screen area is xrht, ytop.

For example, to divide an 11 x 11 inch digitizer into half inch squares, you could use:

configTab (11.0 * 32.0, 11.0 * 32.0, 22, 22, 9, 10, 17, 16);

cutout

```
FUNCTION cutout (pt1, pt2 : IN OUT point;
                 pt3, pt4 : OUT point;
                 addr1, addr2, addr3, addr4, addr5, addr6 : OUT entaddr;
                 doCut : boolean) : boolean;
```

cutout cuts an opening in a wall.

- The three input parameters are pt1, pt2, and doCut.
- pt1 and pt2 are the two points that define the edges of the cut in the wall. As in DataCAD, these points do not have to be on the wall.
- When doCut is true, the two lines that form the wall are broken into two pieces with an opening between them. This is useful for inserting a window or door into an existing wall. When doCut is false, the wall lines are not cut, but cutout finds the walls lines anyway. When cutout finds a wall to put an opening in, it returns true.
- On exit, pt1, pt2, pt3, and pt4 define the four corners of the opening in the wall. These points are on the wall that was found. pt3 is opposite pt1, and pt4 is opposite pt2. The meanings of the addresses depend on whether the wall was cut, that is, if doCut was true or false.
- When the wall is cut, addr1 is the address of the line that ends at pt1. addr2 is the address of the line that ends at pt2.
- addr3 and addr4 are the addresses of the lines ending at pt3 and pt4, respectively.
- addr5 is the address of the line that was added between pt1 and pt3.
- addr6 is the address of the line that was added between pt2 and pt4.

As an example, this piece of code deletes the two small lines added at the jamb:

```
IF cutout (pt1, pt2, pt3, pt4, addr1,  addr2, addr3, addr4, addr5,
addr6, true) THEN
  IF ent_get (ent, addr5) THEN
    { this is always true }
    ent_draw (ent, drmode_black);
    ent_del (ent);
  END;
  IF ent_get (ent, addr6) THEN
    { this is always true }
    ent_draw (ent, drmode_black);
    ent_del (ent);
  END;
END;
```

If the wall was not cut, addr1 is the address of the line that pt1 and pt2 are on, and addr2 is the address of the line that pt3 and pt4 are on. In this case, addr3, addr4, addr5, and addr6 are undefined.

dwgname

```
PROCEDURE dwgname (str : OUT string);
```

dwgname sets its parameter to the current drawing name without the path or extension.

For example, if you are in a drawing named c:\dcad\dwg\spiral.dwg, dwgname sets str to spiral.

ent_explode

```
FUNCTION ent_explode (ent : IN OUT entity; lyr : layer; expMode :
                      integer); boolean;
```

ent_explode explodes an entity into lines or polygons.

- The resulting lines or polygons are created on layer lyr, which can be a regular or temporary layer as described under lyr_init. Any existing entities on lyr are deleted first.

- When expMode is 0, the entity is exploded into lines; when expMode is 1, the entity is exploded into polygons. ent_explode works for any type of entity, including an instance of a symbol. When this operation is completed successfully, it returns true.

ent_incirc

```
FUNCTION ent_incirc (ent: IN OUT entity;
                     centre : IN point;
                     radius : IN real) : boolean;
```

Returns true if ent is wholly within the circle defined by centre and radius.

envget

```
PROCEDURE envget (find : string; val : OUT string);
```

envget reads the DOS environment.

This procedure looks for an environment variable with a name like find.

When find is found, the parameter val is set to its value. When find is not found, val is set to an empty string (length zero)

execprog

```
FUNCTION ExecProg (Path, Name, Params : string;
                   Child : OUT integer) : integer;
```

This Function allows a macro to call and run another DOS program or batch file. The function returns 1 if successful.

- Path is the fully qualified path to a program (or batch file) without the trailing backslash.
- Name is the name of the program or (batch file.)
- Params is any parameter list to be passed to the program (or batch file.)
- Child is a value returned by the child process (or batch file.)

NOTE: *The ExecProg procedure is "unsupported". It works in DOS versions of DataCAD, and in Windows versions 8.07 and later.*

getcurrfont

```
PROCEDURE getcurrfont (font : IN OUT string);
```

getcurrfont reads the current font that DataCAD is using. This is a string of up to eight characters that represents the font file without path or extension. The string may be null.

See the DataCAD Reference Manual for further information on fonts.

getCurrInMat

```
PROCEDURE getCurrInMat (mat : OUT modmat);
```

getCurrInMat gets a modeling matrix which is the mathematical inverse of the current viewing matrix. getCurrInMat returns a valid matrix only when the current projection is either orthographic or parallel.

mat is the inverse of the viewing matrix in these cases. getCurrInMat is typically used when entering and editing entities in a parallel projection and is faster and simpler than using the procedure view_getcurr and extracting the field inptmat from the view_type variable.

getcurrlbl

```
PROCEDURE getcurrlbl (keynum : IN integer; lbl : IN OUT string);
```

Returns in lbl the label currently on function key at position keynum.

getpath

```
PROCEDURE getpath (str : OUT string; num : integer);
```

getpath reads one of DataCAD's internal pathnames.

DataCAD maintains several pathnames internally and uses them when it opens or creates several types of files. Examples are the pathnames for symbol files and the path where layer files are kept. Notice that the user can change most of these at the get filename prompts. The constants used for the various paths (used for the parameter num) are given in the "Constants" chapter.

getrefpnt

```
PROCEDURE getrefpnt (pt : OUT point);
```

getrefpnt reads the point from which DataCAD is displaying distances. This point is used in relative distance output mode.

getrubpnt

```
PROCEDURE getrubpnt (pt : OUT point);
```

getrubpnt reads the point from which DataCAD is rubberbanding. This is typically the last point entered. This point is valid whether or not rubln or rubbx is true.

high

```
FUNCTION high (param : ARRAY) : integer;
```

high, and the corresponding procedure low, handle arrays of indeterminate bounds (conformant arrays).

For example:

```
FUNCTION sum (a : ARRAY OF integer) :
            integer;

VAR
  i     : integer;
  sum1  : integer;

BEGIN
  sum1 := 0;
  FOR i := low (a) TO high (a) DO
    sum1 := sum1 + a [i];
  END;
  RETURN sum1;
END sum;
```

This function can be called with any array of integers. At run-time, low and high, return the bounds of the actual parameter. Many of the string routines do essentially this, taking an array of characters.

ints2addr

```
FUNCTION ints2addr (int1, int2 : integer) : entaddr;
```

ints2addr converts two integers into an address. This procedure is the reverse of addr2ints.

The two integer parameters must be in the same order as in addr2ints.

isnil

```
FUNCTION isnil (adr : entaddr) : boolean;
```

isnil tests for the address of adr. It returns true when adr points to nothing, that is, it is nil. isnil returns false when adr points to something.

NOTE:Even when adr is pointing to invalid data isnil returns true.

keypress

```
FUNCTION keypress : boolean;
```

keypress returns true when a key is waiting in the keyboard buffer, and false when one is not.

light

```
PROCEDURE light (onoff : boolean);
```

light toggles the asterisk in the lower left corner of the screen.

DataCAD typically uses this to let the user know some time-consuming task is underway.

When onoff is true, the asterisk is visible, otherwise it is off.

low

```
FUNCTION low (param : ARRAY) : integer;
```

low, and the corresponding procedure high, handle arrays of indeterminate bounds (conformant arrays).

lyr_init

```
PROCEDURE lyr_init (lyr : OUT layer);
```

lyr_init is similar to lyr_create, creating a layer in the database.

The layer, however, is not one of the standard DataCAD layers that the user has access

to. The layer is considered a temporary layer. The layer can be used for a short period of time, but should never exist while the user has control, that is, during any of the get or dget procedures because the user can terminate the macro with (Ctrl)-(C) leaving the temporary layer in the database. It must never be the current layer when the user has control.

The macro can set lyr as the current layer to add entities to it, as long as the user is not given control to make changes to lyr. Set some other layer as the current layer before returning control to the user.

lyr_term

```
PROCEDURE lyr_term (lyr : layer);
```

lyr_term destroys a layer that was created using lyr_init.

This procedure should never be called with a layer that DataCAD created or one the macro created with lyr_create. All entities must be deleted from the layer with lyr_clear before lyr_term is called.

pause

```
PROCEDURE pause (secs : real);
```

pause stops the system for the specified number of seconds.

readclock

```
PROCEDURE readclock (year, month, day, hours, minutes, seconds,
                                        hundredths : IN OUT integer);
```

Returns the system date and time.

sercheck

```
FUNCTION serCheck (str : IN string) : boolean;
```

serCheck returns true when str is the serial number of the copy of DataCAD the macro is running under.

setcurrfont

```
PROCEDURE setcurrfont (font : string);
```

setcurrfont sets the current font to font.

font is the filename, without path or extension, for a font file. The font file specified should exist in the path specified by the system constant pathchr. See the DataCAD Reference Manual for further information on fonts.

setnil

```
PROCEDURE setnil (addr : IN OUT entaddr);
```

setnil sets addr to the special value nil, therefore, the pointer points to nothing.

See isnil and ent_get.

setpath

```
PROCEDURE setpath (str : string; num : integer);
```

setpath is related to getpath. It is used to set DataCAD's internal paths.

Exercise great care in changing these paths
Some paths, such as the pathnames for drawing files, should not be changed.

setrefpnt

```
PROCEDURE setrefpnt (pt : point);
```

setrefpnt sets the point from which DataCAD displays distances.

setrefpnt is useful when the point being rubberbanded from (see setrubpnt) is not the same as the point you want distance information from, as displayed across the bottom of the window.

setrubpnt

```
PROCEDURE setrubpnt (pt : point);
```

setrubpnt sets the point from which DataCAD is rubberbanding when either rubbx or rubln is true.

sizeof

```
FUNCTION sizeof (variable) : integer;
```

sizeof returns the size, in bytes, of its parameter.

The parameter can be any variable or constant, but not a type.
For example:

```
VAR
  rl      : real;
  addr    : entaddr;
  str   : str80;

sizeof (1)        returns 2
sizeof (rl)       returns 4
```

`sizeof (addr)` returns 4
`sizeof (str)` returns 81
{NOTE: A string has an extra byte which contains the dynamic length of the string.}

ssdelall

```
PROCEDURE ssdelall (ent : IN OUT entity);
```

ssdelall delets ent from all selection sets.

txtbox

```
PROCEDURE txtbox (ent : IN OUT entity;
                  pt1, pt2, pt3, pt4 : IN OUT point);
```

txtbox finds the four corners of an entity of type enttxt. Use this procedure to justify text.

On exit: pt1 is the lower left corner of the text, that is, it is the same as ent.txtpnt.
pt2 is the lower right corner of the text.
pt3 is the upper left corner of the text.
pt4 is the upper right corner of the text.

- The entity ent must be of type enttxt.

CHAPTER 8 Sample Macros

The following is a listing of the several sample macros included with your DCAL development kit. A short description is included for each macro so that you may more easily find those which apply to your development task.

Sample Macro Directory

ARROW	Specifically designed as an introduction to DCAL for the beginner. The macro enters a simple arrow into a drawing made up of lines. The macro is extensively commented, explaining virtually every line of code in detail.
ATR	Demonstrates how to create symbols in a drawing, and how to add attributes to entities, symbol descriptions, and to a drawing at the system level.
CONCRETE 3D	parametric macro using entities of type slab as a basic building block. The macro also contains some good examples of dragging cases.
CROP	Demonstrates how the function CLIP may be used as an editing tool, as well as how to perform a basic editing operation on the database.
DAFILE	Demonstrates all basic techniques necessary for operating upon a direct access, binary datafile. It uses FORMLIB for capturing and editing input data.
DOORLABL	Complete source code to theDataCAD DOORLABL macro.
DRAG	Demonstrates nearly all dragging routines available in DCAL.
ENTER3D	Demonstrates the techniques used for entering 3D entities from parallel projections (such as elevations).
FORMLIB	A reusable toolkit for entering and editing collections of data via a forms type screen input system This library is used by the sample macros FORM, FORMATR, and DAFILE.
FORMATR	Allows the editing of sytem, symbol, and entity attributes via a forms-like interface.
FORM	Outlines the steps required for using FORMLIB to create custom input screens.
GETARC	Demonstrates the use of the function getarc which is used in many DC-Modeler menus.
HATCH	Demonstrates how to use the hatch_mode function.

LYRUTIL	Allows saving and reloading collections of layers Primarily designed for packing large drawing files or recovering corrupted data files.
MENU3D	Demonstrates all menu calls to the 3D Viewer and DC-Modeler.
PLOT	Demonstrates the use of the plot_mode procedure.
POLYLINE	Demonstrates polyvert procedures and functions as well as polyline input.
REVSURF	Demonstrates techniques for creating surfaces of revolution.
READTEXT	Demonstrates techniques for reading text files and adding their contents to a drawing.
SPIRAL	The parametric SPIRAL stair macro included with DataCAD.
STAIR	The parametric STAIR macro included with DataCAD.
STUD	The parametric STUD macro for creating drawings of stud wall construction.
SYMEXP	Takes the instance of a symbol in a drawing and explodes it into individual entities.
SYMTMPLT	Examines all of the symbols in a drawing and creates a template file from the information. Particularly useful for DXF-IN operations.
VIEWMAST	Complete source code to the VIEWMAST macro included with the DC- Modeler Demonstrates nearly all of the functions and procedures related to viewing and saved views.
WNDWLABL	The WNDWLABL macro included with DataCAD.
WRITE	Demonstrates how to scan a database and write out the entire contents of each entity to a text file. May be used as a skeleton for macros which interrogate the database and write its contents to a file in one or another form for post processing.
WRTUTL	A set of utilities for creating messages. It is used by many of the sample macros in this collection

Each macro is in a separate sub-directory. Many macros include a .lnk or a .bat file. The .lnk file contains the linker statement used by the dcl.exe linker. When there are more than one object file, the .lnk file simplifies the linking process. Take for example the CONCRETE macro, to link it, use the command:

dcl <concrete.lnk

The .bat files are like MAKE files in that they recompile all modules of the macro and then relink the macro. To recompile and relink the entire CONCRETE macro, type:

CONCRETE

The batch and linker files assume that each sample macro is in a separate sub-directory, and that the system includes files are in a sub- directory at the same directory level. As well, macros which use the WRTUTL library also assume that the WRTUTL subdirectory is at the same directory level.

```
MTEC ----|-- DCX ----|-- INC   System includes files
|    |-- WRTUTL   Message utilities
|    |-- ARROW   Arrow macro
|    |-- CONCRETE  Concrete macro
|      .
|      .   etc.
|      .
|-- DWG
|-- SUP
|-- SYM
|-- TPL
```

In each macro and in each .lnk file, the pathnames for system includes files, and for object files are designed with the above directory structure in mind. All pathnames are relative, such that the name of the DCX subdirectory is unimportant, only the location of the INC and WRTUTL directories relative to the macro source code.

Hatch

This program is a simple example of the routine hatch_mode. It uses a hardwired brick hatch pattern.

```
PROGRAM hatch;

#include '../wrtutl/wrtutl.inc'
#include '../inc/_hatch.inc'

CONST
  ssNum = 15;

PROCEDURE getOrigin (org : IN OUT point);

VAR
  done    : boolean;
  result  : integer;
  key     : integer;
  pt      : point;

BEGIN
  done := false;
  REPEAT
    lblsinit;
    lblset (20, 'Exit);
    lblson;
  wrtlvl (Hatch);
  wrtmsg (Enter hatch origin.);
```

```
  result := getpoint (pt, key);
  IF result = res_escape THEN
    IF key = s0 THEN
      done := true;
    END;
  ELSIF result = res_normal THEN
     org := pt;
     done := true;
  END;
  UNTIL done;
END getOrigin;

PROCEDURE getHtype (htype : IN OUT integer);

VAR
  done  : boolean;
  key   : integer;

BEGIN
  done := false;
  REPEAT
    lblsinit;
    lblset ( 1, 'Normal);
    lblset ( 2, 'Outer);
    lblset ( 3, 'Ignore);
    lblset (20, 'Exit);
    lblson;

    wrtlvl (Hatch);
    wrtmsg (Select hatch type.);

    getesc (key);
    IF key = f1 THEN
      htype := htype_normal;
      done := true;
    ELSIF key = f2 THEN
      htype := htype_outer;
      done := true;
    ELSIF key = f3 THEN
      htype := htype_ignore;
      done := true;
    ELSIF key = s0 THEN
      done := true;
    END;
  UNTIL done;
END getHtype;

PROCEDURE hatch_main;

VAR
  sl      : scanLineType;
  done      : boolean;
  result  : integer;
  brk     : integer;
  key     : integer;
  mode      : mode_type;
  mode1   : mode_type;
  ent     : entity;
  addr      : entaddr;
  minPt   : point;
```

```
  maxPt   : point;
  min1      : point;
  max1      : point;
  frst      : boolean;
  ang     : real;
  scale   : real;
  org     : point;
  htype   : integer;

BEGIN
  done := false;
  setpoint (org, 0.0);
  ang := 0.0;
  htype := htype_normal;
  scale := 1.0;

  REPEAT
    REPEAT
      lblsinit;
      lblset (10, 'HtchType);
      lblset (12, 'Scale);
      lblset (13, 'Angle);
      lblset (14, 'Origin);
      lblset (20, 'Exit);

      wrtlvl (Hatch);

      result := getmode (hatch', mode, key);
      IF result = res_escape THEN
      IF key = f0 THEN
        getHtype (htype);
      ELSIF key = s2 THEN
        wrtmsg (Enter hatch scale: );
        getrll (scale);
      ELSIF key = s3 THEN
        wrtmsg (Enter hatch angle: );
        getang (ang);
      ELSIF key = s4 THEN
        getOrigin (org);
      ELSIF key = s0 THEN
        done := true;
      END;
      END;

    UNTIL done OR (result = res_normal);
    IF NOT done THEN
      { find the extents of the entities and place them into
        a selection set }

     ssClear (ssNum);

     frst := true;
     addr := ent_first (mode);
     WHILE ent_get (ent, addr) DO
     addr := ent_next (ent, mode);

     ssAdd (ssNum, ent);

     ent_extent (ent, min1, max1);
     IF frst THEN
```

```
      minPt := min1;
      maxPt := max1;
      frst := false;
    ELSE
      minPt.x := min (min1.x, minPt.x);
      minPt.y := min (min1.y, minPt.y);
      minPt.z := min (min1.z, minPt.z);

      maxPt.x := max (max1.x, maxPt.x);
      maxPt.y := max (max1.y, maxPt.y);
      maxPt.z := max (max1.z, maxPt.z);
     END;
     END;

     brk := 0;

     mode_init (mode);
     mode_ss (mode, ssNum);
    { get the entities out of the selection set }
    { draw the horizontal lines 2 2/3 inches apart }
     sl.ang := 0.0;
     setpoint (sl.origin, 0.0);
     sl.delta.x := 0.0;
     sl.delta.y := 2.6667 * 32.0;
     sl.numdash := 0;
     hatch_mode (mode, sl, zbase, zhite, org, ang, scale, htype,
        getlyrcurr, minPt, maxPt, true, brk, true);

     { draw the vertical (dashed) lines every 8 inches }
     sl.ang := halfpi;  { oriented vertically }
     setpoint (sl.origin, 0.0);
     sl.delta.x := 2.6667 * 32.0;
     sl.delta.y := 4.0 * 32.0;
     sl.numdash := 2;
     sl.dash [1] := 2.6667 * 32.0;
     sl.dash [2] := 2.6667 * 32.0;
     sl.dashDraw [1] := true;
     sl.dashDraw [2] := false;

     hatch_mode (mode, sl, zbase, zhite, org, ang, scale, htype,
        getlyrcurr, minPt, maxPt, true, brk, true);

     ssClear (ssNum);
  END;
 UNTIL done;
END hatch_main;

BEGIN
  hatch_main;
END hatch.
```

Plot1

This program gives examples of the plotter routines available through a DCAL macro.

PROGRAM plot1;

```
#include '../wrtutl/wrtutl.inc'
#include '../inc/_plot.inc'

VAR
  done      : boolean;
  key     : integer;
  cent      : point;
  wndwMin   : point;
  wndwMax   : point;
  vwptMin   : point;
  vwptMax   : point;
  tofile  : boolean;
  fname   : str80;
  paperx  : real;
  papery  : real;
  init      : boolean;
  ang     : real;
  ManyPens  : boolean;

PROCEDURE getPaper (size : integer; custx,
custy : real;
       x, y : IN OUT real);

BEGIN
  IF size = 1 THEN
    x := 10.5;
    y := 8.0;
  ELSIF size = 2 THEN
    x := 16.0;
    y := 10.0;
  ELSIF size = 3 THEN
    x := 21.0;
    y := 16.0;
  ELSIF size = 4 THEN
    x := 33.0;
    y := 21.0;
  ELSIF size = 5 THEN
    x := 43.0;
    y := 33.0;
  ELSE
    x := custx;
    y := custy;
  END;
  x := x * 32.0;  { convert from inches to world coords }
  y := y * 32.0;  { convert from inches to world coords }
END getPaper;

PROCEDURE wrtPaperErr;

BEGIN
  errStart;
  errDis (paperx);
  errStr ( );
  errDis (papery);
  errShow;
END wrtPaperErr;

PROCEDURE doplot;
```

```
VAR

  plot  : plot_type;
  mode  : mode_type;
  key   : integer;

BEGIN
  IF plot_open (pltpenwidth, pltpenspeed, paperx, papery,
      tofile, 'test.plt', 1, plot) = fl_ok THEN

    wrterr (ok);
    wrtmsg (Plotting...);

    mode_init (mode);
    mode_lyr (mode, lyr_on);

    plot_mode (plot, mode, ManyPens, pltpensort, pltcolor,
      vwptMin, vwptMax, wndwMin, wndwMax, cent, ang);

    plot_close (plot);
  END;
END doplot;

PROCEDURE getVwpt (min, max : IN OUT point);

VAR
  minPt,
  maxPt     : point;
  screenRatio : real;
  paperRatio  : real;
  minBox,
  maxBox      : point;
  done      : boolean;
  key         : integer;
  result      : integer;
  pt1         : point;
  pt2         : point;
  pt3         : point;

BEGIN
  IF (absr (wndwMax.x - wndwMin.x) < 0.01) OR
    (absr (wndwMax.y - wndwMin.y) < 0.01) THEN
    wrterr (Invalid window, select window first.);
  ELSE
    { draw the sheet of paper on the screen }
    vwptClear;

    currWndw (minPt, maxPt);
    screenRatio := (maxPt.x - minPt.x) / (maxPt.y - minPt.y);

    paperRatio := paperx / papery;

    IF paperRatio < screenRatio THEN
    ELSE
    END;

  done := false;
  REPEAT
    REPEAT
```

```
  lblsinit;
  lblset (20, 'Exit);
  lblson;

  wrtmsg (Enter first corner of viewport.);
  result := getpoint (pt1, key);
  IF result = res_escape THEN
    IF key = s0 THEN
      done := true;
    END;
  END;
  UNTIL done OR (result = res_normal);
  IF NOT done THEN
  REPEAT
    lblsinit;
    lblset (20, 'Exit);
    lblson;

    wrtmsg (Enter second corner of viewport.);
      ratioBox := true;
    ratioRatio := (wndwMax.x - wndwMin.x)/(wndwMax.y - wndwMin.y);
    rubbx := true;
    result := getpoint (pt2, key);
    ratioBox := false;
    IF result = res_escape THEN
      IF key = s0 THEN
        done := true;
      END;
     END;
  UNTIL done OR (result = res_normal);
   IF NOT done THEN
    REPEAT
      lblsinit;

      lblset (20, 'Exit);
      lblson;

      wrtmsg (Position viewport.);

      minBox.x := pt1.x - pt2.x;
      minBox.y := pt1.y - pt2.y;
      setPoint (maxBox, 0.0);
      setpoint (pt3, 0.0);

      dragBoxMove (pt3, minBox, maxBox, linecolor);

      result := getpoint (pt3, key);
      IF result = res_escape THEN
        IF key = s0 THEN
        done := true;
        END;
      END;
    UNTIL done OR (result = res_normal);
    IF NOT done THEN
    END;
  END;
  END;
UNTIL done;

wrtmsg (Enter vwptMin.x: );
```

```
  getdis (min.x);
  wrtmsg (Enter vwptMin.y: );
  getdis (min.y);
  wrtmsg (Enter vwptMax.x: );
  getdis (max.x);
  wrtmsg (Enter vwptMax.y: );
  getdis (max.y);
  END;
END getVwpt;

PROCEDURE getWndw (min, max : IN OUT point);

VAR
 done     : boolean;
 key    : integer;
 result   : integer;
 pt1    : point;
 pt2    : point;

BEGIN
  done := false;

  REPEAT
    REPEAT
    lblsinit;
    lblset (20, 'Exit);
    lblson;

    wrtmsg (Enter first corner of window.);
    result := getpoint (pt1, key);
    IF result = res_escape THEN
    IF key = s0 THEN
      done := true;
    END;
    END;
  UNTIL done OR (result = res_normal);
  IF NOT done THEN
    REPEAT
      lblsinit;
      lblset (20, 'Exit);
      lblson;
      wrtmsg (Enter second corner of window.);
      rubbx := true;
      result := getpoint (pt2, key);
      IF result = res_escape THEN
        IF key = s0 THEN
          done := true;
        END;
      END;
      UNTIL done OR (result = res_normal);
      IF NOT done THEN
      order (pt1.x, pt2.x, min.x, max.x);
      order (pt1.y, pt2.y, min.y, max.y);

      { if we entered both points, get out }
      done := true;
      ELSE
      { if we pressed exit, go back to the previous loop }
      done := false;
      END;
```

```
    END;
  UNTIL done;
END getWndw;

PROCEDURE extra;

VAR
 done   : boolean;

 key  : integer;

BEGIN
  done := false;
  REPEAT
    lblsinit;
    lblset ( 1, 'Plot);
    lblsett ( 2, 'To File', tofile);
    lblset ( 3, 'Vwpt);
    lblset ( 4, 'Wndw);
    lblset ( 5, 'Paper);
    lblset (20, 'Exit);
    lblson;

    wrtmsg (Select plotter function.);
    wrtPaperErr;
    getesc (key);

    IF key = f1 THEN
      doPlot;
    ELSIF key = f2 THEN
      tofile := NOT tofile;
    ELSIF key = f3 THEN
      getVwpt (vwptMin, vwptMax);
    ELSIF key = f4 THEN
      getWndw (wndwMin, wndwMax);
    ELSIF key = f5 THEN
    ELSIF key = s0 THEN
      done := true;
    END;
  UNTIL done;
END extra;

PROCEDURE layout;

VAR
 done     : boolean;
 pt     : point;
 key    : integer;
 result   : integer;
 min    : point;
 max    : point;
 str    : str80;
 fac    : real;

BEGIN
  done := false;
  REPEAT

    lblsinit;
    lblset (20, 'Exit);
```

```
    lblson;

    wrtlvl (Layout);
    wrtmsg (Enter center of plot.);

    getPaper (pltPSize, pltPCustx, pltPCusty, paperx, papery);
    scale_get (pltScaleNum, fac, str);

    setpoint (pt, 0.0);
    min.x := -paperx / 2.0 / fac;
    min.y := -papery / 2.0 / fac;
    max.x := -min.x;
    max.y := -min.y;
    dragBoxMove (pt, min, max, linecolor);

    result := getpoint (pt, key);

    IF result = res_escape THEN
      IF key = s0 THEN
      done := true;
      END;
    END;
  UNTIL done OR (result = res_normal);
  IF NOT done THEN
    pltcentx := pt.x;
    pltcenty := pt.y;
  END;
END layout;

PROCEDURE doplotData;

VAR
  vwptMin   : point;
  vwptMax   : point;
  wndwMin   : point;
  wndwMax   : point;
  str     : str80;
  fac     : real;
  plot      : plot_type;
  mode      : mode_type;
  cent      : point;
  ang     : real;

BEGIN
  getPaper (pltPSize, pltPCustx, pltPCusty, paperx, papery);
  scale_get (pltScaleNum, fac, str);

  vwptmin.x := 0.0;
  vwptmin.y := 0.0;
  vwptmax.x := paperx;
  vwptmax.y := papery;

  wndwMin.x := pltCentx - paperx / 2.0 / fac;
  wndwMin.y := pltCenty - papery / 2.0 / fac;
  wndwMax.x := pltCentx + paperx / 2.0 / fac;
  wndwMax.y := pltCenty + papery / 2.0 / fac;

  setpoint (cent, 0.0);
  ang := 0.0;
```

```
  IF plot_open (pltpenwidth, pltpenspeed, paperx, papery,
                false, 'test.plt', 1, plot) = fl_ok THEN

    wrterr (ok);
    wrtmsg (Plotting...);

    mode_init (mode);
    mode_lyr (mode, lyr_on);

    plot_mode (plot, mode, ManyPens, pltpensort, pltcolor, vwptMin,
    vwptMax, wndwMin, wndwMax, cent, ang);

    plot_close (plot);
  END;
END doplotData;

PROCEDURE scale;

VAR
  i   : integer;
  str   : str80;
  done  : boolean;
  fac   : real;

BEGIN
  done := false;
  REPEAT
    lblsinit;
    FOR i := 1 TO 18 DO
      scale_get (i, fac, str);
      lblset (i, str);

      cvrllst (fac, str);
      lblmsg (i, str);
    END;
    lblset (20, 'NoChange);
    lblson;

    wrtlvl (Scale);
    wrtmsg (Select plotter scale.);
    getesc (key);

    i := fnKeyConv (key);
    IF ((1 <= i) AND (i <= 18)) THEN
      pltScaleNum := i;
      done := true;
    ELSIF key = s0 THEN
      done := true;
    END;
  UNTIL done;
END scale;

PROCEDURE doPaper;

VAR
  done    : boolean;
  key     : integer;
  result  : integer;
  r     : real;
```

```
BEGIN
  done := false;

  REPEAT
    lblsinit;
    lblsett ( 1, '8.5x11A', pltPsize = 1);
    lblsett ( 2, '11x17 B', pltPsize = 2);
    lblsett ( 3, '18x24 C', pltPsize = 3);
    lblsett ( 4, '24x36 D', pltPsize = 4);
    lblsett ( 5, '36x48 E', pltPsize = 5);
    lblsett ( 7, ' Custom', pltPsize = 6);
    lblset (20, 'NoChange);
    lblson;

    wrtlvl (PaperSiz);
    wrtmsg (Select paper size.);
    getPaper (pltPSize, pltPCustx, pltPCusty, paperx, papery);
    wrtPaperErr;

    getesc (key);

    IF key = f1 THEN
      pltPsize := 1;
    ELSIF key = f2 THEN
      pltPsize := 2;
    ELSIF key = f3 THEN
      pltPsize := 3;
    ELSIF key = f4 THEN
      pltPsize := 4;
    ELSIF key = f5 THEN
      pltPsize := 5;
    ELSIF key = f7 THEN
      wrtmsg (Enter plotting width ("X" axis): );
      r := pltPcustx * 32.0;
      getdis (r);
      pltPcustx := r / 32.0;

      wrtmsg (Enter plotting height ("Y" axis): );
      r := pltPcusty * 32.0;
      getdis (r);
      pltPcusty := r / 32.0;
       pltPsize := 6; { custom }
    ELSIF key = s0 THEN
      done := true;
    END;
  UNTIL done;
END doPaper;

VAR
  str   : str80;
  str1 : str80;
  i   : integer;
  fac   : real;

BEGIN
  done := false;
  IF NOT init THEN
    getPaper (pltPSize, pltPCustx, pltPCusty, paperx, papery);
    tofile := false;
    fname := '';
```

```
  setpoint (vwptMin, 0.0);
  vwptMax.x := paperx;
  vwptMax.y := papery;

  setpoint (wndwMin, 0.0);

  setpoint (wndwMax, 0.0);
  setpoint (cent, 0.0);
  ang := 0.0;
  ManyPens := false;
  init := true;
END;
REPEAT
  lblsinit;
  lblset ( 1, 'Plot);
  lblmsg ( 1, 'Plot all layers that are turned on.);

  lblset ( 2, 'Backgrnd);
  lblmsg ( 2, 'Background plotting.);

  lblset ( 3, 'To File);

  lblset ( 4, 'Scale);
  str := 'Current scale = ';
  scale_get (pltScaleNum, fac, str1);
  strcat (str, str1);
  lblmsg ( 4, str);

  lblset ( 5, 'PaperSiz);

  lblset ( 6, 'PenSpeed);
  lblmsg ( 6, 'Set plotter pen speed.);

  lblset ( 7, 'PenWidth);
  lblmsg ( 7, 'Set plotter pen width.);

  lblset ( 8, 'Partial);
  lblset ( 9, 'Layout);
  lblset (10, 'Lyout Sz);
  lblset (11, 'LyoutDiv);
  lblsett (12, 'Rotate', pltRot);
  lblsett (13, 'ClrPlot', pltColor);
  IF ManyPens AND pltColor THEN
    lblset (14, 'Set Pens);
    lblsett (15, 'PenSort', pltpensort);
  END;
  lblsett (16, 'ManyPens', ManyPens);
  lblset (19, 'extra);
  lblset (20, 'Exit);
  lblson;

  wrtlvl (Plotter);
  wrtmsg (Select plotter function.);

  getesc (key);

  IF key = f1 THEN
    doplotData;
  ELSIF key = f2 THEN
  ELSIF key = f3 THEN
```

```
    ELSIF key = f4 THEN
      scale;
    ELSIF key = f5 THEN
      doPaper;
    ELSIF key = f6 THEN
      wrtlvli (146 {'PenSpeed'});
      wrtmsgi (172);
      i := pltpenspeed;
      getint (i);
      pltpenspeed := absi (i);
    ELSIF key = f7 THEN
    ELSIF key = f8 THEN
    ELSIF key = f9 THEN
      layout;
    ELSIF key = f0 THEN
    ELSIF key = s1 THEN
    ELSIF key = s2 THEN
    ELSIF key = s3 THEN
      pltColor := NOT pltColor;
    ELSIF key = s4 THEN
      IF pltColor AND ManyPens THEN
      END;
    ELSIF key = s5 THEN
      pltpensort := NOT pltpensort;
    ELSIF key = s6 THEN
      ManyPens := NOT ManyPens;
    ELSIF key = s9 THEN
      extra;
    ELSIF key = s0 THEN
      done := true;
    END;

  UNTIL done;
END plot1.
```

Stair

```
PROGRAM stair;

VAR
  str1      : string (80);
  str       : string (20);

  ch      : char;
  init,
  getout,
  done1,
  done      : boolean;
  result,
  clr,
  oldcolor,
  stairtype,
  key       : integer;
  stairnum,
  numrisers : integer;
  angle1,   { angle up stairs }
  angle2    : real;    { angle across stairs }
```

```
  cent1,
  cent,
  pt1,
  pt2,
  pt3,
  pt4       : point;
  zstart,
  riserheight,
  depth,
  width   : real;

{
************************************
*         *
*   <- depth ->    *
*         *
*   pt4-------pt3   *
*   | ^ |    *
*   | | |    *
*   | | |    *
*   | w |    *
*   | i |    *
*   | d |    *
*   | t |    *
*   | h |    *
*   | | |    *
*   | | |    *
*   | v |    *
*   pt1-------pt2   *
*         *
************************************
}

PROCEDURE addriser (pt1, pt2, pt3, pt4 : point; num : integer);

CONST
  zero = 0.0;

VAR
  ent : entity;

BEGIN
  ent_init (ent, entblk);
  ent.blkpnt [ 1].x := pt1.x;
  ent.blkpnt [ 1].y := pt1.y;
  ent.blkpnt [ 1].z := zstart + (float (num) * riserheight);

  ent.blkpnt [ 2].x := pt2.x;
  ent.blkpnt [ 2].y := pt2.y;
  ent.blkpnt [ 2].z := zstart + (float (num) * riserheight);

  ent.blkpnt [ 3].x := pt4.x;
  ent.blkpnt [ 3].y := pt4.y;
  ent.blkpnt [ 3].z := zstart + (float (num) * riserheight);

  ent.blkpnt [ 4].x := pt1.x;
  ent.blkpnt [ 4].y := pt1.y;
  ent.blkpnt [ 4].z := zstart + (float (num) * riserheight)
                          + riserheight;
```

```
  ent_add (ent);
  ent_draw (ent, drmode_white);
END addriser;

PROCEDURE doesc (key : integer; getout : OUT boolean);

BEGIN
  IF key = f1 THEN
    wrterr (Drawing stairs by left side.);
    stairtype := 1;
  ELSIF key = f2 THEN
    wrterr (Drawing stairs by right side.);
    stairtype := 2;
  ELSIF key = f3 THEN
      wrterr (Drawing stairs by center line.);
    stairtype := 3;
  ELSIF key = f5 THEN

    wrtmsg (Enter the number of risers: );
    getint (numrisers);
  ELSIF key = f6 THEN
    wrtmsg (Enter the width of the stairs: );
    getdis (width);
  ELSIF key = f7 THEN
    wrtmsg (Enter the height of the risers: );
    getdis (riserheight);
  ELSIF key = f8 THEN
    wrtmsg (Enter the depth of tread: );
    getdis (depth);
  ELSIF key = f9 THEN
    wrtmsg (Select color of stairs.);
    getclr (clr);
  ELSIF key = f0 THEN
    wrtmsg (Enter the starting height of the stairs: );
    getdis (zstart);
  ELSIF key = s0 THEN
    getout := true;
  END;
END doesc;

PROCEDURE setkeys;

BEGIN
  lblsinit;
  lblsett ( 1, ' Left', (stairtype = 1));
  lblsett ( 2, ' Right', (stairtype = 2));
  lblsett ( 3, ' Center', (stairtype = 3));
  lblset ( 5, 'NumRisrs);
  lblset ( 6, 'Width);
  lblset ( 7, 'Height);
  lblset ( 8, 'Depth);
  lblset ( 9, 'Color);
  lblset (10, 'Start Ht);
  lblset (20, 'Exit);
  lblson;
END setkeys;

PROCEDURE get_point1;

BEGIN
```

```
  REPEAT
    setkeys;

    IF stairtype = 3 THEN
      str := 'center';

    ELSIF stairtype = 1 THEN
      str := 'left side';
    ELSE
      str := 'right side';
    END;

    str1 := 'Enter ';
    strcat (str1, str);
    strcat (str1, ' of stairs at bottom.);
    wrtmsg (str1);

    result := getpoint (cent, key);

    IF result = res_escape THEN
      doesc (key, getout);
    END;
  UNTIL (result = res_normal) OR getout;
END get_point1;

PROCEDURE get_point2;

BEGIN
  REPEAT
    lblsinit;
    lblset (20, 'Exit);
    lblson;

    str1 := 'Enter another point on the ';
    strcat (str1, str);
    strcat (str1, ' of the stairs.);
    wrtmsg (str1);

    rubln := true;
    result := getpoint (cent1, key);
    IF result = res_escape THEN
      IF key = s0 THEN
      getout := true;
      END;
    END;
  UNTIL (result = res_normal) OR getout;
END get_point2;

PROCEDURE setitup;

BEGIN
  light (true);
  lblsinit;
  lblset ( 8, '********);
  lblset (10, 'WORKING);

  lblset (12, '********);
  lblson;
  wrtmsg (Constructing stairs, please wait.);
```

```
  angle1 := angle (cent, cent1);
  angle2 := angle1 - radians (90.0);

  { calculate the first point, from now on, all of the cases
    are the same }
  IF stairtype = 3 THEN
    { we have the center, move to the left half the width }
    polar (cent, angle2, -width / 2.0, pt1);
  ELSIF stairtype = 1 THEN
    { we already have the left, don't need to calculate anything }
    pt1 := cent;
  ELSE
    { we have the right side, move to left by an entire stair width }
    polar (cent, angle2, -width, pt1);
  END;

  oldcolor := linecolor;
  linecolor := clr;
  stairnum := 1;
END setitup;

PROCEDURE doit;

BEGIN
  stopgroup;

  WHILE stairnum <= numrisers DO
    polar (pt1, angle1, depth, pt2);
    polar (pt1, angle2, width, pt4);
    polar (pt2, angle2, width, pt3);

    addriser (pt1, pt2, pt3, pt4, stairnum - 1);

    pt1 := pt2;
    stairnum := stairnum + 1;
  END;
  stopgroup;
  light (false);

END doit;

PROCEDURE initialize;

BEGIN
  stairtype := 1;
  clr := linecolor;
  numrisers := 12;
  width := 1536.0;
  riserheight := 192.0;
  depth := 256.0;
  zstart := 0.0;
  init := true;
END initialize;

BEGIN
  { set up defaults }
  IF NOT init THEN
    initialize;
  END;
```

```
  getout := false;

  wrtlvl (Stair);

  REPEAT
    get_point1;
    IF NOT getout THEN
      get_point2;
      IF NOT getout THEN
        setitup;
        doit;
        linecolor := oldcolor;
      END;
    END;
  UNTIL getout;
END stair.
```