Parasolid V34.1

Kernel Interface Driver Manual

January 2022

Important Note

Francis House
112 Hills Road
Cambridge CB2 1PH
UK
Tel: +44 (0)1223 371555
email: parasolid.support.plm@siemens.com
Web: www.parasolid.com

Table of Contents

Kernel Interface Driver (KID) – an Overview

1

1.1 Introduction

Document purpose

This manual, for the Kernel Interface Driver (KID), is for programmers who wish to use KID to drive PARASOLID. It assumes knowledge of the kernel interface (KI) and the PK interface, LISP, and modeling in PARASOLID, though some basic elements are explained in the following chapters.

Notation used in this manual

Throughout this manual, the notation `-->` is used in examples to indicate when information is returned by a call.

What is KID?

KID is recommended as an introduction to Parasolid which enables you to quickly create solid objects, manipulate and display them.

- KID is a stand-alone program, that accesses the Parasolid KI / PK independently of any application program, incorporating:
  - the Parasolid library;
  - a basic graphics library;
  - its own Frustrum for file and memory management;
  - a command-line user interface.
- The KID user interface is a LISP interpreter that allows the Parasolid KI / PK to be called directly and interactively.

  A command is a complete LISP expression, enclosed in brackets, which is read in and evaluated by the driver.

  The LISP interpreter is implemented in C and is included within the supplied executable image of KID.
- The user-interface has two levels, which can be combined:
  - object-oriented KID
  - a lower level, more direct interface to the KI / PK called FLICK
- As quite complex programs can be built up in LISP, without the need for a compiling/ relinking cycle, KID is an ideal tool for learning about Parasolid and for the prototyping of ideas before they are coded into an application.
- KID is used by customers and internally for investigating and reporting faults in Parasolid.

1.1.1 LISP

What is LISP?

The programming language LISP, or “LISt Processing language”, is one of the oldest programming languages, dating from 1960. Though primarily intended for symbolic processing with applications in artificial intelligence, algebraic computation and theorem proving, in KID it is used as a general purpose interface.

LISP is highly interactive and is thought of as an interpreter as it evaluates symbolic expressions, or s-expressions, which you pass to it. You do not need to compile or link LISP programs.

The language supports a range of programming styles from “Fortran with brackets” to functional programming. Modern implementations are very complex and different from one another.

PARASOLID LISP is provided with a set of standard functions.

1.1.2 Facilities within KID

Graphics

KID is supplied with its own graphical system and device drivers. Functions are available to render items from the kernel in a number of ways. Picking from the screen is used extensively as a method of selection.

Rollback

It is possible to roll back the kernel via KID functions. A rollmark can be set at the beginning of each command or sequence of commands. If a command fails to complete successfully, the kernel can then be rolled back to the state it was when the rollmark was set.

It is not possible to roll back the driver. After a kernel rollback, the driver and the kernel may be inconsistent.

Help

The on-line help often provides a convenient way of finding out about objects in KID. It provides information about an object and its properties.

1.2 Concepts

1.2.1 Object-oriented KID

In any object oriented programming environment, an object is used to represent a collection of data and functions. An object can be used to represent a real thing or an abstract idea.

Objects and classes

Objects in KID are arranged in a pre-defined class structure and an object can own other sets of objects. An owning object is referred to as a class. The objects in a class represent the same kind of component, e.g. bodies, faces.

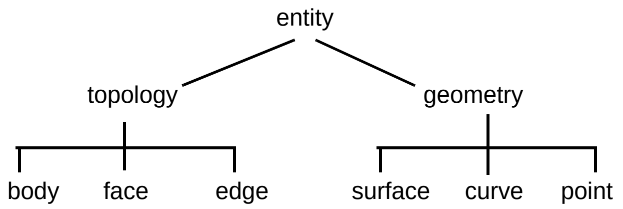

Figure 1–1 Class tree structure (objects owning objects)

All the information concerning an object, including the functions necessary to manipulate it, are kept within the object. PARASOLID LISP has been extended by the inclusion of object oriented functions and these are used extensively within KID.

Item

An item exists as a tagged (i.e. uniquely numbered) entity in the kernel. It has a tag which may or may not be known about by KID.

```
> (define f1 face)  -- define f1 as a member of the face class
> (f1 pick)  -- pick is a function inherited by the object face
                and enables a cursor pick from a model drawing.
```

Objects are used to refer to an item in the kernel. Only an object in a class below the entity class may refer to an item in this way. An object may refer to more than one item.

One of the main tasks of KID is to maintain the correspondence between items and objects.

Primitives

Before an item can be created by the kernel it is necessary to supply data to define it. This data is stored in one of the classes below the primitive class. The data may simply be a set of geometric data (point, direction, radius, etc.) or it may also include the names of existing objects.

Most primitive classes have a corresponding create function, their names are usually the name of the type of entity which is created preceded by p_.

Tags

The tag of an object is one of its properties. This property can be one of the following:

- nil – the object does not have an associated item;
- an atom – the object refers to a single item;
- a list of tags which refer to a set of items. To maintain version independence, the set is not ordered and it is not possible to use individual items from the set. For example,

it is possible to define a face object which consists of all the faces in a body, and to move all the faces with a single function call.

All communication with the kernel interface is via tags, so you use an object which refers to an item to perform a kernel function on the item.

Dead Tags

Objects which refer to items which have been deleted, for example after a rollback, contain dead tags. Some caution should be exercised in this case, and “undefine” used to remove the dead object. It is possible that a dead tag may be unintentionally passed to the kernel. This is always trapped by the kernel and an ifail error message returned.

Errors

Missing mandatory parameters give an error to the user.

To rectify you should repeat the KID command after first setting the properly named parameter on the object. For example:

```
> (define b0 p_cone)
> (b0 lrad 10; height 20; create)
            --> "b0 should first have parameter urad specified"
            ...(followed by lisp error 42)
> (b0 urad 2; create)
```

Function arguments

The majority of object functions use predefined properties of the object as their arguments. Some functions use both predefined properties and a single argument given at the time of the call, for example:

```
> ( <object> transmit '<file_name>)
```

1.3 Starting/terminating a KID session

Starting

To run the KID program, for example, on VMS platform, type:

```
$ run PARASOLID:kid.exe
```

where PARASOLID is a logical name defining the pathname to kid.exe.

When the program is ready the > prompt is shown:

```
restore: finished in 2.149999999999s

*** KID version >v60<

switch to journal file kid.jou
>
```

For further information on running KID on all supported platforms, see Chapter 5, “Using Parasolid”, of the *Installation Notes*.

Terminating

To terminate a KID session type:

```
> (quit)
```

LISP in KID 2

2.1 Introduction

As previously discussed in the opening chapter we have assumed that users of KID have knowledge of LISP, but for those who are a little rusty or unsure of the concepts used, the following section is designed to get you started.

2.1.1 LISP evaluation

You should think of LISP as an interpreter. It evaluates or attempts to evaluate messages which you pass to it. The messages which you pass are called symbolic expressions (s-expressions).

S-expressions

In the following examples of s-expressions notice that numbers evaluate to themselves:

```
> ( times 3 4 )
12
> 3.1417
3.1417
> ( times 3 ( plus 2 2 ) )
12
```

S-expressions are composed of lists and atoms.

Atoms

Atoms are entities which LISP treats as whole items, i.e. they cannot be broken down further. Examples are:

- integers
- reals, e.g. 3.1417
- strings, e.g. a; b; plus. Strings are commonly used as either variable or function names.

Lists

Lists are chains of elements bounded by parentheses, where elements are either atoms or lists themselves. For example:

```
(3 4 5 )
(a d f )
( plus 2 ( times 4 3 ) )
( )         --- empty list
```

List evaluation

When evaluating lists LISP applies the following criteria:

- the first element of the list is treated as a function or operator name, and the subsequent elements are arguments, for example:

```
    > ( plus 2 4 )
```

- as lists can be embedded, the innermost lists are evaluated first, and their values are taken as arguments in the next innermost list, etc. For example:

```
    > ( times 6 ( plus 1 ( plus 2 2 ) ) )
    30
```

Quotation

S-expressions which are preceded by a quote (') are NOT evaluated, for example:

```
> ' ( 3 4 )
( 3 4 )

> ( quote ( 3 4 ) )--- is equivalent to the above
( 3 4 )
> 'a
a
```

2.1.2 Atomic Symbols

SETQ

As previously discussed, strings can be used as variables. They can be bound to values using the `setq` operator, for example:

```
> ( setq a 3 )
3
> a         --- 'a' now evaluates to 3
3
> ( setq a ( add1 a ) )
4
> a
4
```

A side effect of these type of operations is that the complete s-expression always evaluates to a result.

The following examples are equivalent forms:

```
> ( setq a 2 )

> ( set 'a 2 )

> ( set ( quote a ) 2 )
```

Predefined symbol-strings

A number of symbol-strings are predefined by the system, for example:

- `plus`, `times`, `add1` (operator names)
- `nil` (the empty list, or logical false)
- `t` (the logical true)

2.2 List operators

Symbolic operations on lists consist primarily of taking lists apart and building them up. LISP provides two basic functions for taking lists apart, these are `car` and `cdr`. Both are functions of one argument, which should be a list, and they always cause their argument to be evaluated.

2.2.1 CAR and CDR

car

`car` returns the first element of this list, for example:

```
> (car '(a b c))
a
```

cdr

`cdr` returns the list with its first element missing, for example:

```
> (cdr '(a b c))
(b c)
```

`car` and `cdr` are considered non-destructive as they do not actually change the lists on which they operate, for example:

```
> ( setq x '(a b c))
(a b c)
> x
(a b c)
> (car x)
a
> x
(a b c)
> (cdr x)
(b c)
> x
(a b c)
```

Embedded car and cdr calls

`car` and `cdr` can be embedded in a single call, for example:

```
> (cdr (car '((a b c) (d e f)))
(b c)
> (car (cdr '((a b c) (d e f)))
(d e f)
> (car (cdr (car (cdr '((a b c) (d e f))))))
e
```

Code containing long strings of cars and cdrs is hard to follow. Alternatively, the same calls can be made by the single function that corresponds to the sequence of calls used. For example, the previous examples would use these single calls to achieve the same results:

```
> (cdar '((a b c) (d e f)))
(b c)
> (cadr '((a b c) (d e f)))
(d e f)
> (cadadr '((a b c) (d e f)))
e
```

element

The function `element` is a shorthand for embedded `car` and `cdr` calls, for example, in the following example `element` returns the third element of the given list:

```
> ( setq x '(1 4 6 7 ) )
(1 4 6 7)
> ( element 3 x )
6
```

2.2.2 CONS

Just as `car` and `cdr` take lists apart, `cons` builds lists up. `cons` is a function of two arguments where the second argument should always evaluate to a list.

`cons` evaluates both of its arguments, and then returns as its value the list obtained by taking the second argument and placing the first one in front of it, for example:

```
> (cons ’a ’(b c))
(a b c)
```

`cons` can be considered to be the inverse function of `car` and `cdr`, as `cons` always produces a list whose `car` is the first argument to `cons`, and whose `cdr` is the second argument.

Like `car` and `cdr`, `cons` is non-destructive.

Dotted pairs

If the second argument to `cons` is an atom then the result is a dotted pair rather than a list. In most cases this is not a desirable result and the use of `list` would produce preferable results. However, the syntax for the input of PK option structures requires the use of dotted pairs for which `cons` should be used. For example:

```
(cons 'a 'b) -> (a . b)
(cons 'a '(b c)) -> (a b c)
(cons '(a b) 'c) -> ((a b) . c)
(list '(a b) 'c) -> ((a b) c)
```

2.2.3 LIST and APPEND

`cons` can be used to build up complicated s-expressions, for example, to create the lists (1 2 3) and (a (b c) d), using `cons` we would:

```
> (cons ’1 (cons ’2 (cons ’3 nil)))
(1 2 3)
> (cons ’a (cons (cons ’b (cons ’c nil)) (cons ’d nil)))
(a (b c) d)
```

As this is obviously cumbersome, the `list` and `append` functions are simpler ways to build new lists.

list

`list` takes any number of arguments, evaluates them, and builds a new list containing each value as an element. For example:

```
> (list ’1 ’2 ’3)
(1 2 3)
> (list ’a ’(b c) ’d)
(a (b c) d)
```

append

`append` takes two arguments, which should both evaluate to lists, and creates a new list by concatenating the given lists. For example:

```
> (append '(a b) '(c d))
(a b c d)
```

append can also produce dotted pairs:

```
(append '(a b) 'c) -> (a b . c)
```

2.3 Predicates

A predicate is a symbolic expression which evaluates to true (t) or false (nil), i.e. it is a test.

Logical operators

The following logical operators are defined: not, and, or.

atom

atom determines whether or not its argument is an atom, for example:

```
> (atom 'a)
t
> (atom '(a b c))
nil
```

listp

listp determines whether something is a list. For example:

```
> (listp 'a)
nil
> (listp '(a b c))
t
```

2.4 Conditionals

Predicates can be used to make choices, but to do this the equivalent of a conditional branch is needed. For this the cond (for conditional) function is provided. cond is similar to the “if; then; and else” statements.

A cond s-expression can have any number of arguments (clauses), which consist of a series of expressions. The first element of a cond clause is treated as a condition to be tested for; the rest consists of things to do should the condition prevail.

```
> (cond
   (predicate1 action1a action1b action1c ...)
   (predicate2 action2a action2b action2c ...)
   .
   .
   (t default_action1 default_action2 ...)
)
```

meaning:

```
if
    predicate1 is true, then evaluate action1a,
    action1b, etc. in sequence
else if
    predicate2 is true, then evaluate action2a ...
    .
    .
else
    evaluate default_action1, ...
```

A `cond` clause is only fully evaluated providing that the first element of the clause evaluates to true `t`.

For example, if you want to be sure that something is a list before you take its `car`, do:

```
> (cond ((listp x) (car x)))
```

This previous example is a `cond` of one clause, the s-expression `((listp x) (car x))`. Where the first element of the clause is `(listp x)` which is the condition of the clause. It is only when this evaluates to true that LISP evaluates the rest of the clause, the expression `(car x)`. For example:

```
> (setq x '(a b c))
(a b c)
> (cond ((listp x ) (car x)))
a
> (setq x 'y)
y
> (cond ((listp x) (car x)))
nil
```

Like other LISP functions, `cond` always returns a value. In the previous example, when the test in the `cond` clause evaluates to true, LISP evaluates the next expression, whilst returning the value of that expression as the value of `cond`. When the test failed, the `cond` returned `nil`.

2.5 User-defined functions

A LISP user can create functions using the function `defun` (define function).

defun

defun takes as its arguments the name of the function to be defined, a list of formal parameters (literal atoms), and some bodies of code (s-expressions). defun does not evaluate any of these arguments, it associates, for future reference, the formal parameter list and bodies of code with the given function name.

For example, to create the simple function addthree:

```
> (defun addthree (x) (plus x 3))
addthree
> (addthree 4)
7
```

More generally, the syntax of a call to defun would look like:

```
> (defun function_name (param1 param2 ... paramn)
    (... s-expression1 ...)
    (... s-expression2 ...
    (... s-expression2 ...)
    .
    .
)
```

2.6 Recursion vs. iteration

At times we want to repeat an operation an indefinite number of times, each time with different inputs. This can be achieved through the use of iteration or recursion.

2.6.1 Recursion

By using recursion we can accomplish the equivalent of indefinite repetition; a function is said to be recursive if it refers to itself in its definition. It is necessary to make sure that the function checks first for a termination condition, to avoid an infinite loop. For example:

```
> (defun fac (n)
    (cond
        ((eq n 0) 1)
        (t (times n (fac (sub1 n))))
    )
    )
fac
> (fac 4)
24
```

2.6.2 Iteration

In iterative code, indefinite repetition is designated by explicit instructions to do something repeatedly. In LISP there are several functions that enable you to write an explicit loop

mapc

`mapc` is a “LISP-like” way of doing an iteration, in which it takes two arguments; the first being a list and the second a function. For example:

```
> ( mapc list_name function_name )
```

`mapc` maps each element of `list_name` by applying `function_name` (which must be a single argument) to it, for example:

```
> ( mapc '( 2 5 7 ) add1 )
( 3 6 8 )
```

2.7 Special features of Parasolid LISP

Parasolid LISP provides a more extensive set of LISP facilities than those which are generally found in other LISP dialects, important points worth noting are listed below:

- atomic types include integer, real, string, and function
- arithmetic operations – `plus`, `difference`, `times`, `quotient`, `equal`, `greaterp`, `lessp` are overloaded

  For example, `plus` works with character strings as well as reals.
- the input token reader:
  - regards **underscore** _ as part of a symbol, rather than **minus** -
  - recognizes strings in double quotes **“....”** as quoted atoms
  - recognizes **double hyphen,** `--`, as beginning a comment
- **HELP** allows retrieval of system and user information
- **LOAD** permits execution of journal files or separately developed code
- variables which are not defined, and properties which are not available default to **UNDEFINED** (note that **UNDEFINED** has the value true for the purposes of conditionals etc.)
- system functions may not be redefined and can only be handed to other functions by quoting, e.g. eval and plus in:

```
    ( apply 'eval ( list plus 1 1 ) )
```

- **CAR**, **CDR** on **nil** or an atom are not permissible.
- **COND** raises an error if no true condition is encountered.

Quick reference summary

For a quick reference table summary of the functions available in PARASOLID LISP, see Appendix B, “Parasolid LISP Functions”.

Error codes

The error codes in PARASOLID LISP are given in Appendix D, “List of Parasolid LISP Functions”.

2.8 Object oriented LISP

Parasolid LISP has been extended by the inclusion of object-oriented functions and these are used extensively within KID.

Object-oriented expressions have the form:

```
(object function argument1 argument2 ...)
```

The functions listed first are extensions to functional LISP rather than object oriented themselves:

- `define` – creates objects within structures.

```
   > ( define <object> <class> )
```

- `undefine` – deletes an object (and any of its subclasses) whether or not it has been previously defined. The syntax is shown below; the argument can be one object or many, separated by spaces:

```
   > ( undefine <objects> )
```

- `redefine` – redefines an object as another class.

Object oriented LISP provides many more useful functions to add to those described for standard LISP. Below is a summary of such functions which are properties of the universe class. Try using (universe help _<name_>) for more information.

- `help` – returns help about the object.

```
   > ( fred help [property] )      --> useful information
```

- `detach` – detaches an object from its owning class or parent.
- `attach` – attaches an object to a new class.
- `is` – returns the owning class of an object. If an argument is given it must be the name of a class and the function returns the subclass of the one given which leads to the object in the class structure, e.g.

```
   > ( define limb fred )        --> limb
   > ( define left_leg limb )    --> left_leg
   > ( define right_leg limb )   --> right_leg
   > ( fred is )                 --> man
   > ( left_leg is )             --> limb
   > ( left_leg is man )         --> fred
   > ( left_leg is fred )        --> limb
```

- `superclass` – returns the owning class of an object
- `subclass` – returns the objects owned by the given class
- `sibling` – returns all objects in the same owning class as the given object
- `supertree` – returns the direct ancestors of the object, e.g.

```
> ( left_leg supertree ) --> (left_leg limb fred man universe)
```

- `subtree` – returns the descendants of a given object
- the symbol minus `-` removes a property from an object. If the property is not found, this message is not passed to the owning object.

```
    > ( fred age 33 )          --> 33
    > ( fred age )             --> 33
    > ( fred - age )           --> t
    > ( fred age )             --> undefined
```

- `defun` – defines a function

The following object functions are used internally in KID and need not concern the user. They are reserved words and should not be overwritten.

```
PROPERTY FUNCTIONP INHERIT SYSTEM SUBJECT LAZY GUARD LISTENER
UNGUARD OWN OWNER RESUME ABANDON
```

Object-Oriented KID 3

3.1 Object oriented programming

In any object oriented programming environment, an object is used to represent a collection of data and functions. An object can be used to represent a real thing or an abstract idea.

All the information concerning an object, including the functions necessary to manipulate it, are kept within the object. The principles of object orientated programming have been exploited in many languages, PARASOLID LISP has been extended by the inclusion of object oriented functions and these are used extensively within KID.

3.1.1 Objects and message passing

Objects are LISP entities. They consist of some data and a set of operations. The nature of an object’s operations depends on the nature of the component it represents. An object representing data structures might store and retrieve information, an object representing a solid body might answer enquiries about its relationships to its component faces and edges, or might perform operations on itself to modify its shape or position in space.

In KID objects are arranged in a predefined hierarchical class structure, see Appendix A, “KID Class Structure”.

A message is a request for an object to carry out one of its operations, it specifies which operation is desired, but not how that operation is carried out. The object to which the message was sent determines how that operation should be carried out. A crucial property of messages is that they are the only way to invoke an object’s operations.

Operations in KID are generally carried out by passing to the command-line interface messages of the following form:

```
( object function argument1 argument2 ... )
( object property argument )
```

3.1.2 Parasolid PK functions and KI routines

Many LISP functions and object operations call a Parasolid PK Function or KI Routine. This provides a convenient, version-independent method of using Parasolid in KID. If timing data is being output (the default action), the name of the function or routine is also output.

- As the Parasolid PK Interface is developed, many operations are being changed to use the PK function equivalent of the KI routine they previously called, but this change is transparent to the user.

3.2 KID journal file

KID opens a journal file with the name `kid.jou` and journals all KID commands in this file, which can be renamed before another session is started. It should be possible to reproduce the original operations by loading this file in a subsequent KID session. If the file extension is changed to .lsp then it is not necessary to specify it in the argument to load.

```
> (load "bug_27.jou")        -- file is bug_27.jou
> (load ’bug_27)             -- file is bug_27.lsp
> (load "bug_27" ’reflect)   -- commands used are shown
> (load ’bug_27 ’verify)     -- commands and returns are shown
```

When a graphical pick is made with the cursor, a point and a direction are recorded. These vectors are written to the journal file and are used when replaying the journal file so no user interaction is needed.

3.3 Starting and stopping the kernel

The kernel is started and stopped using functions of the object modeller, which are pre-defined within the KID class structure.

In the following example, KID is instructed to perform the function `start` (which requires no arguments) on the object `modeller`. This has the effect of calling the Parasolid KI routine STAMOD.

Normally the start function should be called before doing anything else.

Object	Function
modeller	start, stop

start function

```
> (modeller start)
```

and to stop the modeller:

stop function

```
> (modeller stop)
```

3.4 Parasolid journal file

When the kernel is started using (modeller start), a Parasolid journal file is opened, with the default name kid.jnl_txt. This records all the Parasolid PK functions and KI routines called in a KID session, with their received and returned arguments.

The journal file is useful if unexpected errors occur, as it can be inspected to see what functions have been called and when the error occurred.

All GO (Graphical Output) functions are now logged in the journal file. It should be noted that this:

- generates very large files
- increases the KID session elapsed time

Therefore, it is recommended that Parasolid journalling is turned off unless it is required in a particular modeling session.

Journalling options available before (modeller start)

```
> (option journal t)           -- enables journalling (default)
> (option journal_file "f1_b0")  -- enables journalling, using
                                   this journal file
> (option journal nil)           -- disables journalling

> (option journal)            --> returns whether or not a
                                journal is being kept
> (option journal_file)       --> returns the journal file name
```

Journalling options available after (modeller start), if journalling is enabled

```
> (option journal t)          -- toggles journalling on
> (option journal nil)        -- toggles journalling off

> (option journal)            --> returns whether or not a
                                journal is being kept
> (option journal_file)       --> returns the journal file name
```

3.5 Rollback during a modeling session

The function `mark` provides the means to set marker points in the modeling session, to which it is possible to:

- roll back using roll, thereby undoing everything performed since the last mark was set
- roll forward, after you have previously rolled back, to retrace your steps between operations

mark

```
> (modeller mark)
     -- generates a mark name, e.g. mark5, mark6, etc.
```

roll

```
> (modeller roll) -- returns to the most recent mark)
```

When setting a `mark` you can specify a name for it, and then you can roll to that specifically named mark (or a system defined name, e.g. mark6).

```
> (modeller mark 'start_here )
   ....
> ( modeller roll 'start_here )
```

Live KID objects can be rolled back in synchrony with the kernel. Define body, face, edge to be types of KID objects you want managed as follows:

```
> (modeller roll_class '( body face edge ) )
```

No KID object is created or undefined during a rollback, but those objects in the defined `roll_class` have their tag values updated.

Graphics is kept in step by honoring the current tag values of the objects it is being asked to display.

Only the kernel (without affecting KID objects) is rolled backward/forward, therefore, when a rollback is to a state before the creation of a body in the kernel, KID still acknowledges its existence, while the kernel does not – an error results if any modeling operation referring to such a body is attempted. Conversely if a kernel item has been deleted since the rollmark was set and its corresponding KID object undefined, after a rollback the kernel acknowledges the item's existence, but KID does not.

Object	Function
modeller	mark, roll

```
> (modeller mark)
> (define b0 p_block)
> (b0 x 10; y 20; z 30; create)
> (b0 is)            --> returns body
> (modeller roll)    --> rolls back the kernel
> (b0 is)            --> returns body
> (b0 enquire)       --> returns an error from the kernel
```

3.6 Defining KID objects

define

In LISP an identifier has a value given it by `setq` or `defun`. In object oriented LISP a special function, `defun`, is used to create an object which returns the identifier of the object.

```
> ( define b0 body )  -- defines a KID object with the 'b0'
                         which belongs to the class 'body'
```

In the above example, no Parasolid entity has been created, only a KID entity.

undefine

To undefine a previously defined KID object:

```
> ( undefine b0 )      -- undefines the KID object ‘b0’
```

redefine

To redefine a previously defined KID object as another class:

```
> ( redefine b0 assembly )  -- redefines the KID object ‘b0’ in
                                the class ‘assembly’
```

3.7 Combining tags of KID objects

include, remove

The function `include` combines the tags contained in the specified KID objects, so that the KID object refers to more Parasolid entities. The function `remove` removes the specified entities from a KID object. The KID objects must be of the same type.

Object	Function
entity	include, remove

```
> (e0 tag)
( 46 50 54 )
> (e1 tag)
( 83 80 )
> (e0 include e1)
( 83 80 46 50 54 )
> (e0 remove e1)
( 46 50 54 )
```

3.8 Receive and transmit

receive, transmit and state functions

The part class and its subclasses, assembly and body, can be received and transmitted. Text files are used unless changed by the appropriate option setting.

Object	Function
part	receive, transmit

```
> (define b0 body)      -- the object must be defined before
                        -- before the function can be used
> (define b1 body)
> (b0 receive "b0")          -- receives the file b0.xmt_txt
> (b1 receive "flt4453.xmt") -- receives the file flt4453.xmt
                  -- the extension must be given for .xmt files
> (b0 transmit "b0")    -- transmits the object to the file
                     -- b0.xmt_txt,overwriting original version
> (b0 transmit "file1") -- transmit body to new file
> (define pump body)
> (pump receive "p9423") -- receives an existing body
```

Both the `receive` and `transmit` functions take a single argument, the key of the relevant part.

3.9 Help

help function

`help` returns useful information about an object and its functions.

Object	Function
modeller	help

```
> (modeller help resabs) --> linear resolution in the modeller
> (body help) --> list of valid messages and descriptions
> (b0 help) --> list of valid messages and descriptions
> (p_cone help create)
            --> create function properties for primitive p_cone
> (face help rmfaso )
            --> function rmfaso of object face and arguments
> (b1 help transmit)
            --> how to use transmit function for body b1
```

3.10 Options

The option class contains functions which control the interface to the kernel. These functions:

- affect the options when the modeller is started;
- change them (if it can) while the modeller is running or store them until they can be changed.

The current setting of any option may be enquired by calling the function without any arguments. If the option has not been set then the default value is returned. Text files are the default for part files and binary files for snapshots. The use of these options is shown below.

Object	Function
option	bb, bb_user, bspline_io, bspline_geometry, bspline_splitting, check, continuity_checking, data_checking, enquire, get_snapshot, journal, journal_file, logging, logging_size, logging_number, logging_forward, parameter_checking, receive, rec_user, save_snapshot, self_checking, transmit, user_field
	pk_session_tolerance, pk_session_receive, pk_session_transmit, pk_session_local_checking

```
> (option journal "f1_b0") -- if this is done before starting,
                     the kernel will start with this journal file
> (option journal)
               --> returns whether or not a journal is being kept
> (option journal_file) --> returns the journal file name
> (option logging [nil | 'ki | 'pk | +ve | -ve )
               -- switch off/on logging with required options set
> (option enquire) --> <list of current settings>
> (option check t)
         -- switch on local checking, this is the default setting
```

Options for receive and transmit

Part receive and transmit use the values in the appropriate flags:

```
(option help receive) --> information
(option receive 'binary) -- binary receive
(option transmit 'text) -- text transmit, default setting
(option transmit 'neutral -- machine independent, binary format
```

These are initialized in (modeller start) to the global value set using (option receive) and (option transmit):

```
> (option pk_session_receive)
> (option pk_session_transmit 'binary)
```

Option check for local operation

When a local operation is performed, there are sometimes a number of possible solutions. When the option check is set to `t`, each solution is reviewed and checked in the resolution of ambiguous cases. If not set, the first solution is picked.

```
> (option check t)      -- default setting for local checking on
> (option check nil)    -- local checking off
```

If local checking has been turned off, an illegal solution is not detected until a complete body check is subsequently done, thus losing a possible legal solution for the local operation, and leaving the body in an invalid state.

```
> (option pk_session_local_checking)
```

This local checking flag is initialized in (modeller start) to the global value set using (option check). It is used in various KID functions where the underlying PK function requires the local checking flag.

Options for rollback

The switching on/off of the roll mark facility is controlled by the logging properties. The logging properties that can be set when switching on the logging option are:

- ki or pk or proll (the default) – to select the appropriate rollback system.
- +integer (+ve) – to select KI rollback with the given file size. The file size is specified using the option `logging_size`.
- -integer (-ve) – to select the PK rollback with the given maximum number of live marks. The maximum number of live rollmarks can be set using the `logging_number` option (default number is 20).

The roll mark facility creates session marks only. To model in partitions and create partition marks, the appropriate PK functions must be called directly, using the PK FLICK interface. For further details on how to do this, see Chapter 4, “Calling the KI/PK Using KID (FLICK)”.

option user_field

The `option user_field` function is used to set up the user field length which is passed to STAMOD via `modeller start`. The user field length can not be changed while the modeller is running, however, option `user_field` does store the current setting

.

Expression	Description
`> (option user_field 0)`	default setting
`> (option user_field 12)`	set user field length to 12
`> (option user_field)`	current setting <and pending change>

Options reset by new modeling session

It is now possible to specify some options which only persist for a modeling session, i.e. they are reset by `(modeller stop;start)`. Currently there are only four such options:

```
(option pk_session_tolerance)
(option pk_session_receive)
(option pk_session_tansmit)
(option pk_session_local_checking)
```

If the general values are modified these changes are passed on to the session values immediately, for example:

Expression	Description
`(option receive ’binary)`	sets global binary receive
`(option pk_session_receive)`	enquire receive type, binary

Expression	Description
`(option pk_session_receive 'text)`	resets receive to text for session
`(option pk_session_receive)`	enquire receive type, text
`(option receive)`	check that global setting unchanged
`(modeller stop;start)`	
`(option pk_session_receive)`	enquire receive type, now binary
`(option receive 'text)`	sets global text receive
`(option pk_session_receive)`	enquire receive type, now text

Tolerance setting

The tolerance setting is required by various PK boolean and local operation functions and defaults to 0.000001.

Expression	Description
`(option pk_tolerance)`	enquire tolerance setting
`(option pk_tolerance 0.001)`	set PK tolerance
`(option pk_tolerance)`	enquire tolerance setting
`(option pk_tolerance nil)`	reset to default value (0.000001)
`(option pk_session_tolerance)`	enquire session tolerance
`(option pk_session_tolerance 0.1)`	set session tolerance
`(option pk_session_tolerance)`	enquire session tolerance
`(option pk_tolerance 0.001)`	this also resets session value
`(option pk_session_tolerance)`	enquire session tolerance
`(option pk_session_tolerance nil)`	reset to default value(0.000001)
`(option pk_session_tolerance)`	enquire session tolerance
`(option pk_tolerance)`	enquire tolerance setting

Calling the KI/PK Using KID (FLICK) 4

4.1 Introduction

It is possible to use the Kernel Interface Driver (KID) to call the functions in the PARASOLID kernel interface in two different ways. This chapter describes how to do this. It is assumed that the reader is familiar with LISP and with object oriented LISP.

4.2 Functional low-level interface to the C-kernel (FLICK)

FLICK is a subsystem within KID which provides functions to interface directly with the KI/PK. It consists of a small number of support functions and a large number of functions to call the KI/PK; 2 per KI routine and 1 per PK function.

A typical KID user need not refer to any of these functions since the higher levels of KID provides access to much of the KI/PK indirectly anyway. Those who wish to use the FLICK routines have a choice of two modes of use:

Using upper case Parasolid KI routines

Upper case routine names (STAMOD, CRSOFA ...) map directly onto the KI routines argument by argument. These routines merely pass simple data values between FLICK and the KI/PK with no interpretation or simplification. They are therefore rather unhelpful to use and require several calls to additional support functions to interpret the results. They are made available to enable the user to overcome any restrictions imposed by the more convenient FLICK routines.

```
> ( STAMOD 1 3 “KID” 0 )                --> ( 1 600382 0 )
> ( CRBXSO ’( 0 0 0 ) ’( 0 0 1 ) 5 5 5 ) --> ( 7 0 )
> ( IDCOEN 7 ( token ’TYTOFA ))         --> ( 69 6 0 )
> ( setq workspace ( alloc 6 ))         --> @12303420
> ( GTTGLI 69 1 6 workspace )           --> ( 0 )
> ( empty 6 int workspace )             --> ( 9 23 37 48 55 72 )
```

Details of the support functions available can be found in Appendix D, “Flick Function Descriptions”, of the Parasolid KI Reference Manual.

Using lower case Parasolid KI routines

Lower case KI routine names (stamod, crsofa ...) and PK functions (pk_body_ask_faces, ...), while primarily providing access to the equivalent KI/PK function, go to greater lengths to provide convenient output in LISP data formats. They convert integers to symbolic tokens where possible, expand arrays and KI lists as LISP lists, suppress unused arguments and ifails, and sometimes provide optional arguments with defaults.

Note: Since only KI routines and PK function are supported, FLICK in isolation does not support graphics devices, windows or view ports. It must be used in conjunction with KID to access these facilities.

```
> ( stamod )                                -->V6.00.382
> ( crbxso '( 0 0 0) '( 0 0 1 ) 5 5 5 )    -->7
> ( idcoen 7 'tytofa )                      -->( 9 23 37 48 55 72 )
```

4.3 Calling KI routines

All the LISP KI routines are documented individually in Appendix D, “Flick Function Descriptions”, of the Parasolid KI Reference Manual.

Option lists

Option lists in FLICK, including associated data, are passed as an argument to the routine in a list.

Option tokens

A simple list of option tokens can be passed in either of these formats:

```
( token token token ... )

( ( token ) ( token ) ( token ) ... )
```

Note that simple tokens in a list must either all be bracketed or all be unbracketed, these forms cannot be mixed.

Option tokens with associated data

Associated data is passed in a sublist with the relevant token:

```
( ( token real real ... ) ( token tag ... ) ( token vector ...)
 ... )
```

- the list can include either bracketed or unbracketed simple tokens:

```
( token ( token real ... ) token ( token tag ... ) )

( ( token ) ( token real ... ) ( token ) ( token tag ... ) )
```

“options” arguments passed as lists of lists where each sub-list contains an option token together with any associated data:

```
> ( rrvdep '(( RROPCT 0.000259873 1000.0 3.0 )
                ( RROPSI )
                ( RROPTR ) )
                ( b0 tag ) nil view_matrix )
```

4.4 Calling PK functions

The documentation of the PK functions in the Parasolid PK Interface Programming Reference manual should be referred to when calling PK functions in KID.

In general, there is a LISP function for every PK function:

- The name of the LISP function is the same as the PK function **except that all characters are lower case**. For example, the PK function PK_TOPOL_find_box is called from LISP as pk_topol_find_box.
- The arguments to the LISP function are the received arguments of the PK function.
- Arrays are represented as a list of their elements. When a PK function passes an array as an integer length and an array as separate arguments, the LISP function just uses a list.

Options argument

Options are passed to PK functions as options structures, and these are represented in LISP as a list of dotted pairs. The elements of each dotted pair are the name of a field of an options structure and the value it is to be given. For example:

```
> (pk_face_contains_vectors ( f0 tag )
       '(( n_vectors . 3 )
         ( vectors . (( 0 0 0 ) ( 0 4 0 ) ( 0 5 0 )) )) )
```

- Each time a PK function with an options structure is called from LISP, the options macro is called to set all the defaults before the LISP argument is processed to possibly change some of them.
- The option names for each field in the options structures are documented in the Parasolid PK Interface Reference Manual.

The two following examples contain valid option structures:

```
> (setq *vectors '((0 0 0)(10 0 0)(0 10 0)))
> (pk_face_contains_vectors (f0 tag)
      (list
          (cons 'n_vectors (abs *vectors))
          (cons 'vectors *vectors)
      )
  )
> (pk_face_contains_vectors (f0 tag)
      (list
          (cons 'n_vectors 3)
          (cons 'vectors '((0 0 0)(10 0 0)(0 10 0)))
      )
  )
```

However, these two examples do not:

```
> (pk_face_contains_vectors (f0 tag)
      (list
          (cons 'n_vectors 3)
          (list 'vectors '((0 0 0)(10 0 0)(0 10 0)))
      )
  )
> (pk_face_contains_vectors (f0 tag)
      (list
          (list 'n_vectors 3)
          (cons 'vectors '((0 0 0)(10 0 0)(0 10 0)))
      )
  )
```

Optional received arguments

For all the PK_BODY_create_... functions, the last received argument is a pointer to a PK_AXIS2_sf_t function which defines the local coordinate system. This may be NULL, indicating that the local and world coordinates are the same. This is indicated in LISP by leaving out the argument.

```
> (pk_body_create_solid_block 30 20 10
     '((100 0 0)(0.8 0.6 0)(0 0 1)))
            --- solid block defined in local coordinate system
> (pk_body_create_solid_block 30 20 10)
            --- solid block defined in world coordinate system
```

Optional returned arguments

Many PK functions have return arguments of the form:

```
int           *const n_things   --- number of things
PK_THING_t  **const things      --- things (optional)
```

meaning that `things` can be set to NULL by an application if it does not want the array of things returned.

Any PK function that has such optional returns has a corresponding LISP function with optional logical arguments saying whether the array is to be returned or not. These arguments default to `t`, meaning the array is returned. However, if the argument is supplied as nil, then only the number of `things` – the length of the array – is returned:

```
> (pk_body_ask_faces (b0 tag)) --- returns a LISP list of faces
> (pk_body_ask_faces (b0 tag) nil)
                           --- just returns the number of faces
```

Where there are several optional returns, the optional logical arguments to the LISP function are in the same order as the optional returns from the PK function. To supply an optional logical argument as nil, any others preceding it must be explicitly supplied as t or nil:

```
> (pk_shell_ask_oriented_faces (sh0 tag) t nil)
```

In the journal file, optional logical arguments supplied as NULL or nil are journalled as @0.

Structures

Structures are represented as a list of their fields. This means that each struct adds another pair of brackets. Note the following examples:

Structure	Example
PK_AXIS2_sf_t	`′((0 0 0)(0 0 1)(1 0 0))`
PK_CIRCLE_sf_t	`′(((0 0 0)(0 0 1)(1 0 0)) 10)`
PK_POINT_sf_t	`′((10 20 30))` `--- a struct containing a PK_VECTOR_t`

Primitives

The following primitives are used by the LISP PK functions:

PK	LISP
`int`	numeric atom without decimal point
`double`	numeric atom with or without decimal point
`char*`	`′abcd` or `“abcd”` or `(quote abcd)`
PK_LOGICAL_t	`t` or `nil` (strictly: if it’s not `nil` it’s `t`)
PK_VECTOR_t	`(list x_comp y_comp z_comp)`
PK_INTERVAL_t	`(list low high)`
PK_BOX_t	`(list x_low y_low z_low x_high y_high z_high)`
PK_UV_t	`(list u v)`
PK_UVBOX_t	`(list u_low v_low u_high v_high)`

4.5 Using the quote (′)

Passing lists directly to the KI/PK

As for s-expressions, lists which are to be passed directly to the PK/KI (without first being evaluated by LISP) should be preceded by a quote.

```
> ( crknpa ′( 20 22 24 ) )
```

Therefore, the quote must be omitted when elements are to be evaluated.

```
> ( crknpa ( list ( b0 tag ) ( b1 tag ) ( b2 tag ) ) )
```

Examples of calls which do *not* work because of misuse of the quote are:

```
> ( crknpa ( 20 22 24 ) )
> ( crknpa ( ( b0 tag ) ( b1 tag ) ( b2 tag ) ) )
> ( crknpa '( ( b0 tag ) ( b1 tag ) ( b2 tag ) ) )
```

4.6 KI ifail checking

Like KID, FLICK performs ifail checking and raises a LISP error if a returned ifail is considered invalid. Currently this mechanism is quite different from and independent of the KID ifail processing, which is rather limited. Just which ifails are allowed may be adjusted using the support function allow_ifails. A simple example of which is:

```
( allow_ifails ( KI_missing_geom ) ( IDSOFF 99 ) )
```

allow_ifails function

The function allow_ifails takes any number of arguments. The first is a specification of the ifails which are valid while the remainder, which should be LISP expressions, are evaluated. The function returns the value obtained by evaluating the last argument.

Initially the only valid ifail is KI_no_errors (i.e. zero), any other ifail returned generate a LISP error. The set of valid ifails can be extended by including those which are to be allowed in the valid ifail specification either for all KI calls or – if a KI function name is given as the first element of the valid ifail list – only for that named KI function. Calls to `allow_ifails` can be nested. Once an ifail has been made valid it cannot subsequently be disallowed.

- To allow all ifails the (pseudo) token KI_all_ifails is provided.
- To indicate all but specified ifails use a negative sign (e.g. – KI_not_a_tag).
- Any KI call which returns an ifail returns all its other arguments as nulls and zeros.

```
> ( allow_ifails ( KI_missing_geom ) ( IDSOFF 99 ))
                        -- call IDSOFF but don't produce an
                            error if the face lacks geometry
> ( allow_ifails ( STOMOD KI_modeller_not_started ) ( STOMOD ))
   -- stop the modeller but don't complain if it isn't started.
> ( allow_ifails ( KI_roll_is_off ) ( my_strict_programme ))
                -- allow KI_roll_is_off errors, but no others
> ( allow_ifails ( - KI_corrupt_file ) ( my_liberal_programme))
                -- allow all errors except KI_corrupt_file
> ( allow_ifails ( KI_all_ifails ) ( my_careless_programme ))
                -- don't complain about anything
```

For greater flexibility of use the valid ifail specification may be a list of valid ifail specifications, as in this example.

```
> ( allow_ifails(( IDSOFF KI_missing_geom KI_not_a_tag )
         ( IDCOEN KI_missing_geom KI_not_a_tag ))
         ( setq v1 ( IDSOFF 99 ))
         ( setq v2 ( IDCOEN 100 (token 'TYTOFA) )))
```

Some lower case FLICK functions declare ifails valid like this. For example stamod permits the ifail KI_modeller_not_stopped.

PK Errors `valid_ifails` tries to take PK errors and KI ifails and convert the ifails into PK errors where possible, this is to try and ensure that code which uses KID functionality has similar behavior when KID calls the KI and the PK.

```
((define b0 p_block) x -10)
(valid_ifails '(KI_distance_le_0) '(b0 create))
```

should trap the distance_le_0 error whether (b0 create) tries to use CRBXSO or PK_BODY_create_solid_block.

4.7 PK error checking

The function `valid_pk_errors` is similar to `valid_ifails`. It raises and reports the number of the first LISP error which occurs within the supplied test code:

```
(valid_pk_errors '(PK_ERROR_wrong_entity)
'(pk_curve_ask_interval 12))
(valid_pk_errors '(PK_ERROR_wrong_entity) '(car 1))
```

Other types of error are not trapped, for example this still raises an error:

```
(valid_pk_errors '(PK_ERROR_wrong_entity)
'(pk_curve_ask_interval s))
```

For multi s-expression code, use progn to prevent evaluation of the returns:

```
(valid_pk_errors '(PK_ERROR_wrong_entity)
  '(
  progn
  (define b0 p_block) create)
  (pk_curve_ask_interval (b0 tag))
  )
)
```

However, this raises an error because the PK error is PK_ERROR_not_an_entity, not PK_ERROR_wrong_entity:

```
(valid_pk_errors '(PK_ERROR_wrong_entity)
  '(
  progn
  (undefine b0)
  ((define b0 p_block) create)
  (pk_curve_ask_interval (plus (b0 tag) 2000))
  )
)
```

To allow both KI ifails and PK errors wrap the code in a valid_ifails function:

```
(valid_ifails '(KI_not_a_tag)
  '(valid_pk_errors '(PK_ERROR_wrong_entity)
    '(
    progn
    (undefine b0 l0)
    ((define b0 p_block) create)
    (chcken (plus (b0 tag) 2000))
    (pk_curve_ask_interval (b0 tag))
    )
  )
)
```

4.8 Timing

Like KID, FLICK also generates a timing_line message to track each KI routine or PK function called. Some, none or all KI routine or PK functions can be traced.

The function timing controls the output of timing data for KI routine and PK function calls. It takes one argument, the timing level and returns the new level.

- The lowest level, 0 or nil, causes no timing data at all to be printed.
- The highest level, 2 or t, causes timing data to be printed for all KI routines and PK functions.
- Level 1 (the initial default level) results in statistics being printed for important KI routines and PK functions, but not for ancillary ones such as KI list handling routines.

With no argument the current level is returned unchanged.

Creation of Primitives 5

5.1 Introduction

The following sections deal with the creation of primitives which include assemblies, points, acorns, wires, sheets, solids, curves and surfaces. The primitive classes are temporary storage for data about an object before a kernel item is created. When created, the objects are transferred to an appropriate subclass of entity i.e. assembly, vertex, body, curve or surface.

5.1.1 Solid Primitives

Various types of solid can be created from primitive objects. Note that all primitive bodies inherit pre-defined properties from the class p_body. These are `point`, with a value of `’(0 0 0)`, and `direction`, with a value of `’(0 0 1)`. These properties are used as defaults for the creation of some of the primitives unless they are superseded by locally defined properties.

Object	**Function**
p_acorn, p_block, p_cone, p_cylinder, p_prism, p_sphere, p_torus, p_paracurve, p_parasurf, p_sheet, p_wire, p_profile, p_pyramid	create

Creating a block

```
> (p_block help create)  -- information on property names and
                            defaults
> (define b0 p_block)    -- define the primitive
> (b0 x 10; y 20; z 30)  -- and direction gets default values
> (b0 create)
> (b0 is)                --> body
```

Creating a sphere

```
> (define b1 p_sphere)
> (b1 help create)                              --> information
> (b1 radius 20; point ’(10 10 10); create
                                         -- change default point
```

Each newly created primitive body carries properties, top and bottom, which can be used to position further primitives.

```
> ( define cube p_block )
> ( define bar  p_cylinder )
> ( cube z 50; direction '( 1 0 0 ); create )
> ( bar height 30; radius 5; point ( cube top ); create )
```

5.2 Additional Primitive Options

Additional primitive options which create non-convex bodies, self obscuring bodies, or those with many edges meeting at a vertex, and profiles are supported as follows for:

5.2.1 p_pyramid

The primitive type `p_pyramid` parameters are sides, radius, height (mandatory) point, direction (default table). The radius may be omitted if the parameter “length” specifying length of side is provided instead.

5.2.2 p_block, p_cone and p_cylinder

The `p_block`, `p_cone` and `p_cylinder` recognize the defining parameter “thickness”, which defines the wall thickness of the primitive. Hollow pipes of various cross section can be defined using this.

```
> ( define pipe p_cylinder )
> ( pipe height 10; radius 5; thickness 1.5 ; create)
```

5.2.3 p_sphere and p_torus

If the thickness parameter is applied to `p_sphere` or `p_torus` a simple sphere or torus is created containing an internal void.

5.2.4 Profiling

p_profile class

Primitive contains a class `p_profile`. This class allows the user to define a facial profile on a body, from a given list of vector points. The user can create a minimum body from a single point, a wire body from a list of unconnected points, or a sheet body from a list of connected points (points are connected if they define a closed loop.) An attempt to create a sheet body may fail if the points are not co-planar. All the edges of the wire body are straight. The examples below illustrate each case:

Object	Function
p_profile	create

To create a minimum body and move it to coordinates (1 2 3):

```
> (define b0 p_profile)
> (b0 coordinate '(1 2 3))
> (b0 create)
```

To create a sheet body with a triangular profile:

```
> (define b1 p_profile)
> (b1 coordinate '( (0 0 0) (0 1 0) (1 1 0) (0 0 0) ))
> (b1 create)
```

Note: Had the set of coordinates not been closed then a wire body would have resulted.

scribe function

The scribe function may be used to scribe the bounded portion of the curve onto a specified face, region or body.

Object	Function
p_bounded_curve	scribe

```
> (( define b0 p_acorn) create )
> (( define c0 p_line ) point '( 0 0 0 );
   direction '( 0 0 1 ); create )
> (( define bc0 p_bounded_curve )
        body 'b0;
        curve 'c0;
        startp ( c0 deparameterise 0 );
        endp ( c0 deparameterise 1 );
        scribe)
```

Geometric Primitives

Simple geometric properties such as points, vectors, curves and surfaces can also be created from primitives. Primitive curves and surfaces inherit pre-defined properties from the class `p_geometry`.

Object	Function
p_circle, p_ellipse, p_intersection, p_line	create

Creating a circle:

```
> (p_circle help create)     --> information
> (define c1 p_circle)
> (c1 point '(3 2 0))        -- center of circle
> (c1 radius 10)
> (c1 direction '(1 1 1))    -- axis direction
> (c1 create)
```

Creating an ellipse:

```
> (define c2 p_ellipse)
> (c2 point '(0 10 0))      -- center of ellipse
> (c2 direction '(1 0 0)) -- normal, i.e. ellipse in YZ plane
> (c2 majrad 10;minrad 5)
> (c2 majaxi '(0 1 0))      -- major axis along Y axis
> (c2 create)
```

Creating a p_wire from a p_line

`p_wire` creates a wire body from a bounded region of a curve. The range property is optional if the curve is bounded.

Object	Function
p_wire	create

```
> (( define c0 p_line ) point '( 0 0 0 );
   direction '( 0 0 1 ); create )
> (( define b0 p_wire ) curve 'c0; urange '( 0 1 ); create )
```

Object	Function
p_planar, p_cylindrical, p_conical, p_spherical, p_toroidal, p_swept, p_spun, p_offset	create

Creating a planar surface:

```
> (p_planar help create)        -->information
> (define s1 p_planar)
> (s1 point '(0 0 0); direction '(1 0 0))
> (s1 create)
```

Creating a swept surface from a given curve:

```
> (( define c0 p_line ) point '( 0 0 0 );
   direction '( 0 0 1 ); create )
> (( define s0 p_swept ) curve 'c0;
   direction '( 1 0 0 ); create )
```

Creating a spun surface from a given curve:

```
> (( define c0 p_line ) point '( 0 0 0 );
   direction '( 0 0 1 ); create )
> (( define s0 p_spun ) curve 'c0; point '( 0 2 0 );
   direction '( 1 0 0 ); create )
```

Creating an offset from a given surface, which if possible, is simplified:

```
> (( define s0 p_planar ) point '(0 0 0);
   direction '(0 0 1); create)
> (( define off0 p_offset ) surface 's0; distance 5; create )
```

Creating a p_sheet from a p_planar surface: `p_sheet` creates a sheet body from a bounded region of a surface. The range properties are optional bounded parameters.

Object	Function
p_sheet	create

```
> (( define s0 p_planar ) point '( 0 0 0 );
   direction '( 0 0 1 ); create)

> (( define b0 p_sheet ) surface 's0;
   urange '( 0 1 ); vrange '( 0 1 ); create )
```

5.3 Transformation Primitives

For all of the following, the transform is first defined and then applied to the given entity.

Object	Function
p_equal_scaling, p_reflection, p_rotation, p_translation, p_general_transform	apply

5.3.1 p_equal_scaling

```
> ( ( define b0 p_block ) create )
> ( ( define t0 p_equal_scaling )
    scale 1.5; centre '( 0 0 5 ); create )
> ( graphics ske 'b0; ar )
> ( t0 apply 'b0 )
> ( graphics sketch 'b0; ar )
```

5.3.2 p_reflection

```
> ( ( define b0 p_block ) create )
> ( ( define t0 p_reflection )
    point '( 11 0 0 ); normal '( 1 0 0 ); create )
> ( graphics ske 'b0; ar )
> ( t0 apply 'b0 )
> ( graphics sketch 'b0; ar )
```

5.3.3 p_rotation

```
> (( define b0 p_block ) create )
> (( define t0 p_rotation ) point '( 11 0 0 );
   direction '( 1 0 0 ); angle 3.1415926; create )
> ( graphics ske 'b0; ar )
> ( t0 apply 'b0 )
> ( graphics sketch 'b0; ar )
```

5.3.4 p_translation

```
> ( ( define b0 p_block ) create )
> (( define t0 p_translation )
   direction '(0 0 1); distance 30; create)
> ( graphics ske 'b0; ar )
> ( t0 apply 'b0 )
> ( graphics sketch 'b0; ar )
```

5.3.5 p_general_transform

With this function the transformation is specified explicitly by the transform matrix. Therefore, this primitive may be used to create more complicated transforms, e.g. general affine. Note, this can only be applied to a limited subset of the usual entities.

A combination translation and general affine deformation:

```
> ( ( define b0 p_block ) create )
> ( ( define t0 p_general_transform )
    matrix ' ( 1 0 0 1 0 2 0 1 0 0 3 1 0 0 0 1 ); create )
> ( graphics ske 'b0; ar )
> ( t0 apply 'b0 )
> ( graphics sketch 'b0; ar )
```

5.4 Assemblies and Instances

assembly and instance functions

Assemblies and instances can be created from primitives.

Object	Function
p_assembly, p_instance	create

Creating an empty assembly:

```
> (define a0 p_assembly)
> (a0 create)
```

Creating an instance within an assembly:

```
> (define i1 p_instance)
> (i1 assembly 'a0; part 'b0)
> (i1 create)
```

disassemble function

To convert a flat assembly (a0) to the class body, where a0's tag list contains all those Parasolid bodies in the assembly, use:

```
> (a0 disassemble)
```

Operations on Bodies, Curves, Surfaces, etc. 6

6.1 Introduction

This section covers the operations which can be performed on bodies etc. which have been created as primitives or, in the case of bodies, have been received.

6.2 Booleans

A number of functions exist for operations which are only possible on bodies, such as the boolean functions unite and subtract. A list and a few examples of these functions appear below.

Object	Function
body	check, unite, subtract, intersect, section, merge, unfix

Assume two overlapping bodies b1 and b2 are created or received prior to each of the following examples. In these examples b1 is the target and b2 is the tool body. These commands preserve the tag of the target body and also return an appropriately named object of type body containing the complete body set of the resulting boolean operation (i.e. its tag property is a list of tag values):

- subtract_temp – for subtract
- unite_temp – for unite
- intersect_temp – for intersect
- section_temp – for section

One of the resultant bodies has the same tag as the target body and this is the item that the target object refers to, it is not in general defined which of the result bodies this is. The `section` operation is slightly different: the tool object must be a **s**urface, not a body, and the target object, on completion, refers to the set of items which lie on the front (i.e. in the direction of the sectioning surface normal) of the sectioning surface, while `section_temp` refers to the set of objects at the back.

As a side effect of the boolean operation the tool body is deleted from the kernel, its tag is dead and attempts to refer to it again produce an error.

When intersecting a sheet (target) with a solid (tool) body, the result is another sheet. The result of an `intersect` must be the same as the target body.

check function

```
> (b1 check)        -- check the body is valid
```

unite function

```
> (b1 unite 'b2)  -- b1 is now b1 + b2
                     (if required, b2 can be a list of bodies)
```

subtract function

```
> (b3 subtract 'b4)    -- b3 is now b3 - b4
```

intersect function

```
> (b5 intersect 'b6)   -- b5 is now the intersecting volume
```

6.2.1 Multiple bodies

Multiple tool bodies are supported in KID Booleans using:

```
> (b0 unite '( b1 b2 ))
> (b0 intersect '( b1 b2 b3 ))
> (b0 subtract '( b1 b2 ))
```

6.2.2 Sectioning primitives

halve and quarter functions

Two additional sectioning functions that assist the rapid shaping of primitives are:

```
> (body halve <body axis> )
> (body quarter <body axis> )     -- think of cutting a cake
```

These functions section symmetric bodies with respect to the given axis, through their center of gravity. If the axis direction is missed out it defaults to `'( 0 0 1 )`.

6.2.3 Operations on the single class

The single class consists of the topological items face, edge and vertex. The functions `merge` and `unfix` are inherited by these topological items.

merge function

`merge` removes redundant faces, edges and vertices.

unfix function

`unfix` detaches, geometry from a face, edge or vertex.

6.3 Sewing

When an object of class `body` tag list contains a set of sheet bodies, these sheet bodies can, wherever possible, be knitted together to form a single sheet or solid body using the function `sew`.

sew function

```
> (b0 sew 'solid)
```

As real parts often need manual intervention to set their edge tolerances before the sewing operation completes successfully, the function `tolerance` supplied.

tolerance function

`tolerance` operates on edges and can be used to either set the tolerance on a Parasolid edge or to enquire its tolerance.

```
> (e0 tolerance 0.000254)
                      -- set the tolerance of edge e0 to 0.000254
> (e0 tolerance)      -- enquire the tolerance of edge e0
```

6.4 Transforming bodies

All subclasses of the transformable class (body, surface, face etc.) can be moved and rotated. The functions require that the objects have properties which define the transformation.

move function

`move` has two properties; a vector (direction), which is required and a scalar (distance), which is optional. The object is moved in the direction given by the direction vector, through a distance given by the distance property if it is set, or by the magnitude of the direction vector if the distance property is not set.

rotate function

Properties for `rotate` are direction, point and angle, all are required.

Object	**Function**
transformable	move, rotate

Move a body a specified distance and direction:

```
> (define b0 p_block)
> (b0 help create)
> (b0 x 10; y 10; z 10; create)
> (b0 help move)                          --> information
> (b0 direction '(0 1 0); distance 12)     -- b0 is a body
> (b0 move)        -- this moves b0 12 units in the Y direction
```

`rotate` a body about the X axis by a specified angle:

```
> (b3 point '(0 0 0); direction '(1 0 0) ; angle 45)
> (b3 rotate)       -- rotate b3 about the X axis by 45 degrees
```

`move` a face(set) a specified distance and direction:

```
> (f1 direction '(0 0 1); distance 2)
> (f1 move)          -- move f1 2 units in the Z direction
```

`move` a face(set) by the supplied direction vector:

```
> (f2 direction '(3 4 5))
> (f2 move)        -- if no distance is supplied then the item
                      will be moved by the direction vector
```

6.5 Blends

It is possible to create unfixed blends. The primitive for a general blend class `p_blend` is set by `itype` to a default of a true rolling ball blend. The class `p_chamfer` has local properties of `type` corresponding to true chamfer blends while `p_fillet` duplicates the `p_blend` default to true rolling ball blends. Defaults for properties such as `range` and `type` can be found with the help facility.

p_vrb and p_ff_blend are instances of p_blend. (p_vrb help) and (p_ff_blend help) give the properties specific to these instances; anything else relevant they inherit from p_blend.

6.5.1 Creating unfixed blends

p_blend, p_fillet, p_chamfer

The first steps in attaching a blend are:

- to define a primitive of the correct type to hold the blend data, and
- set the appropriate properties for the blend primitive, which are:

  r1 and r2, rib, type, idraw and irib

All of these properties have default values, except for r1 and r2, which are the ranges of the blend on the two faces adjacent to the edge the blend is to be applied to. Only r1 is appropriate for a `p_fillet`.

If it is required to reverse the sense of the blend, a property of the primitive blend named rev can be set to true before using the function `apply` to attach the blend to the edge.

Having defined a `p_blend`, `p_fillet` or `p_chamfer` the values to be used for the blend parameters can be set or changed. Only a single value of the range properties r1 and r2 needs to be given in order to define the blend, in this case the blend is symmetric. The thumbweight value for all blends is set to 1.0 and cannot be changed. Although it is possible to change the type this could lead to confusion as it does not change the type of the KID object concerned, so it is not recommended.

blending properties

Blending properties, when required, must be set before the blend is applied:

Property	**Description**
smooth	t => BLECSM option
propagate	t => BLECPR option
cliff_edge	takes edge object and passes it in with the BLECCL option
irib	t => pass rib value in with BLECRI option
draw	t => BLECDF option

p_vrb

This allows you to specify a `positions` property (either a list of values or the string `'ends`) and a `ranges` property of the same length (except for `'ends` for which it should be of length 2) which is passed into variable radius blend creation. For example:

```
(undefine b0 e0 bl0)
((define b0 p_block) create)
((define e0 edge) pick_from 'b0;
 pick_using '(e0 clash '(5 0 10)))
((define bl0 p_vrb) ranges '(1 4); positions 'ends; apply 'e0)
(b0 blend_fix; check)
(undefine b0 e0 bl0)
((define b0 p_block) create)
((define e0 edge) pick_from 'b0;
 pick_using '(e0 clash '(5 0 10)))
((define bl0 p_vrb)
   ranges '(1 3 4);
   positions '((5 5 10)(5 0 10)(5 -5 10));
   apply 'e0)
(b0 blend_fix; check)
```

p_ff_blend

This requires the specification of the two walls of faces to be blended, then the blending options if changes from the default values are required. The face sets are defined as

- left wall of faces
- right wall of faces
- if reversed, then set the sense flag to true

```
((define ff0 p_ff_blend)
left_wall 'f0;
left_sense t;
right_wall 'f1;
)
```

The options settings can be examined by:

```
(ff0 options)
```

There is then a simple way to set up all the relevant properties using KI tokens and the function convert_ki_options:

```
((define ff0 p_ff_blend)
  convert_ki_options
    (list
      '(FXFTCB 0.005)
      '(FXFTTL 0.00003)
      '(FXFTPR)
      '(FXFTMS)
      '(FXFTAT)
      (list 'FXFTCE (list (e0 tag)(e1 tag)))
    )
)
```

Alternatively, each property required can be specified by name:

```
(ff0 cliff_edges 'e0; --- FXFTCE
r1 0.005;             --- FXFTCB
tolerance 0.00003;    --- FXFTTL
propagate t;          --- FXFTPR
multiple_sheets t;    --- FXFTMS
walls 'attach;        --- FXFTAT
create)
```

The `create` reports an error if one occurs, and raises a LISP error if `(option raise_blending_errors t)` has been set.

The functions which require 1D curve data extract the relevant data (x, y and z components) from the given curve. For example, if c0 is a 3D B-curve with tag 28:

```
((define ff0 p_ff_blend)
  range1_curve 'c0; --- extract the x component
  range2_curve 'c0; --- extract the y component
  rho_curve 'c0; --- extract the z component
  ...
  create)
((define ff0 p_ff_blend)
  range1_curve c0; --- extract the x component
  range2_curve c0; --- extract the y component
  rho_curve c0; --- extract the z component
  ...
  create)
((define ff0 p_ff_blend)
  range1_curve 28; --- extract the x component
  range2_curve 28; --- extract the y component
  rho_curve 28; --- extract the z component
  ...
  create)
```

If c0, c1 and c2 are 1D B-curves (tags 29, 30, 31):

```
((define ff0 p_ff_blend)
  range1_curve 'c0;
  range2_curve 'c1;
  rho_curve 'c2;
  ...
  create)
((define ff0 p_ff_blend)
  range1_curve c0;
  range2_curve c1;
  rho_curve c2;
  ...
  create)
((define ff0 p_ff_blend)
  range1_curve 29;
  range2_curve 30;
  rho_curve 31;
  ...
  create)
```

The data can also be input explicitly:

```
((define ff0 p_ff_blend)
  range1_curve '( 3 1 nil ( 7.0 8.0 9.0 9.0 9.0 8.0 7.0 )
  ( 4 3 4 ) ( -10.0  0.0 10.0 )
  PK_knot_piecewise_bezier_c nil t );
  ...
  create)
```

apply function

To attach the blend attribute to the model, the function apply is used.

Note: The name of the edge to attach the blend to must be quoted. The default setting of `itype` for `p_blend` is 3, the value for a true rolling ball blend.

Object	Function
p_blend, p_fillet, p_chamfer, p_vrb, p_ff_blend	apply

```
> (p_blend help apply)     --> information
> (define f1 p_blend)
> (f1 r1 3.2)                --  reset range values
> (f1 apply 'e0)    -- apply offset blend to edge e0
                        (e0 has already been defined and picked)
```

6.5.2 Checking, enquiring and removing unfixed blends, and picking blends

Object	Function
edge	pick_blends, blend_remove, blend_check, blend_enquire

blend_check function

`blend_check` checks the validity of a blend on a particular edge, or set of edges.

```
> (e0 blend_check)      -- check unfixed blend on edge(s) e0
```

blend_enquire function

`blend_enquire` returns blend information.

```
> (e0 blend_enquire)    -- information on unfixed blends on
                           edge(s) e0
```

blend_remove

The function `blend_remove` can be used to remove any unfixed blend attribute from an edge. Once blends have been fixed, `blend_remove` can not remove a blend from an edge.

```
> (e0 blend_remove)     -- remove unfixed blend from edge(s) e0
```

pick_blends function

`pick_blends` identifies the edge(s) of a body which has unfixed blends attached.

```
> (e0 pick_blends 'b0) -- pick unfixed blend on edge(s) e0 from
                           body b0
```

6.5.3 Fixing blends

blend_fix function

The `blend_fix` function fixes all unfixed blends on a specified body.

Object	Function
body	blend_fix

```
> (b0 blend_fix)     -- all unfixed blends are fixed in body b0
```

6.5.4 Extracting blend information from a blended body

extract function

The function `extract` **e**xtracts the blend information from a blended edge into the properties of a `p_blend` primitive. This leaves the edge without a blend. Blend extraction is from one edge only, and returns `t` if the extraction has been successful.

Object	Function
p_blend	extract

```
> (define bl1 p_fillet)   -- define blend
> (bl1 r1 5; apply 'e1)   -- and apply to edge e1
> (define bl2 p_blend)    -- define bl2 as a p_blend
> (bl2 extract 'e1)
             -- extract blend information from e1 into bl2
> (bl2 r1 1;apply 'e1)   -- change parameter and re-apply to e1
```

6.5.5 Creating a cliff-edge blend

cliff_edge blend function

The function `cliff_edge` creates a cliff edge, but does require that the edge that is the “cliff to” edge is specified.

```
> (define bl1 p_fillet)           -- define blend
> (bl1 r1 5)                      -- set range values
> (bl1 cliff_edge 'e0; apply 'e1) -- apply blend, `cliff to' e0
```

6.5.6 Defining and fixing a blend in a single operation

To define and fix a blend in a single operation use either of the relevant `fillet` or `chamfer` functions.

```
> ( topology fillet <radius> )
> ( topology chamfer <radius> )
```

Fillet and chamfer work :

- on a solid body – all edges are blended
- on a solid face – all the edges of the face
- at a solid vertex – all the edges at the vertex

Fillet also works on sheet and wire bodies :

- on a non-solid body – all vertices are blended
- on a sheet face – all the vertices of the face
- at a non-solid vertex – the list of vertex tags

6.5.7 Blending on vertices

Support for variable radius blends is by filleting a vertex object with two tags and 2 radii.

```
> ( define v0 vertex )
> ( v0 pick )
> ( v0 pick )                -- two vertices on a single edge
> ( v0 fillet ’( 1.5 5.5 ))  -- varying radii
```

6.6 Sweeping and swinging

Many subclasses of the topology class can be modified with the functions `sweep` and `swing`.

- minimum bodies can be swept/swung into wire bodies;
- wire bodies can be swept/swung into sheet bodies;
- sheet bodies can be swept/swung into solid bodies.

sweep function

The `sweep` function takes a vector as its argument.

swing function

The `swing` function takes as its arguments:

- the direction of the rotation axis,
- a point on the direction axis, and
- an angle through which the body is to be swung

Object	**Function**
topology	sweep, swing

Using sweep to create a solid body from a minimum body:

```
> (define b1 p_acorn)
> (b1 create)                -- b1 is an acorn at the origin
> (b1 help sweep)            -- for information on sweeping b1
> (b1 sweep '(10 0 0))       -- b1 is now a wire body
> (b1 sweep '(0 20 0))       -- b1 is now a sheet body
> (b1 sweep '(0 0 30))       -- b1 is now a solid body
```

Using `sweep` and `swing` to create a semi-circular sheet body from a minimum body which is then subsequently swung to form a hemisphere:

```
> (define b2 p_acorn)
> (b2 create)
> (b2 sweep '(10 0 0))
> (b2 point '(0 0 0); direction '(0 0 1); angle 180)
> (b2 swing)          -- b2 is now a semi-circular sheet body
                      -- center at the origin, radius 10, arcing
                      -- from (10 0 0) to (-10 0 0)

> (b2 point '(0 0 0); direction '(1 0 0); angle 180)
> (b2 swing)
                      -- b2 is now a hemisphere, center at
                       -- the origin, radius 10 in positive Z
```

Some care has to be taken when creating a sheet body using `sweep` and `swing` on wire vertices. The example below has the same effect as the method shown above for creating a semi-circular sheet.

Note: Scribe must be the last function used, if the wire is to be closed, creating a body without geometry on the faces.

The function `fix` is finally used to attach a planar surface to one of the faces consisting of surfaces geometry.

```
> (define b3 p_acorn)
> (b3 create; direction '(10 0 0); move)
> (b3 point '(0 0 0 ); direction '(0 0 1); angle 180; swing)
> (define n1 p_line)
> (n1 point '(0 0 0 ); direction '(-1 0 0); create)

> (define b4 p_bounded_curve)
> (b4 body 'b3; curve 'n1)
> (b4 startp '(10 0 0 ); endp '(-10 0 0 ))
> (b4 scribe)

> (define f1 face)
> (f1 pick_from 'b3)
> (f1 fix )                  -- planar surface fitted to face
```

6.6.1 Sweeping faces

Faces of sheet and solid bodies can both be swept.

```
> ( f0 sweep '( 0 0 10 ) )
```

6.7 Hollowing, offsetting and imprinting

Object	Function
body	hollow, offset, imprint

6.7.1 Hollowing

hollow function

The function `hollow` creates a hollowed part from a solid body taking the single argument of the required wall thickness, (passing a negative distance causes the hollow to work outwards):

```
> ( b0 hollow 1 )          -- thickness of walls
```

This operation does not necessarily return a non-zero ifail in the event of a failure, it tries to return diagnostic information. Therefore, to make the `hollow` function fail when hollowing is unsuccessful, set the following option before attempting the operation:

```
> ( option raise_hollowing_errors t )
```

hollowing properties

Hollowing properties, when required, must be set on the body before the hollowing operation:

Property	Description
check_hollow	`t` or `nil`: ■ off and face checking translates to `nil`, ■ full checking translates to `t`
pierce_faces	list of faces not to be offset
offset_faces	list of faces with specific offsets
tolerance	maximum applied tolerance
max_faults	maximum number of entities on badtaglist

pierce_faces function

An option parameter may be set to pierce some of the faces of the resulting body opening up the interior void.

Hollowing a body and opening up two faces:

```
> ( b0 pierce_faces ( f0 tag ))        -- faces to remove
> ( b0 hollow 0.1 )                    -- hollow & pierce
```

In the above example:

- first the body is copied,
- offset surfaces are then created from the faces which are NOT to be opened up,
- the corresponding faces on the copied body are tweaked to these offset surfaces,
- and finally the copy is subtracted from the original, leaving the hollow body.

6.7.2 Offsetting

offset function

The `offset` function offsets the faces in a body by a specified distance:

```
> ( b0 offset 1 )         -- makes a bigger body
> ( b0 offset -1.5 )      -- makes a smaller body
> ( f0 offset 10 )        -- just do a faceset
```

Note: Since this exploits local operations, the offset distance must be small enough so that the topology of the body is not changed.

offsetting properties

Offsetting properties, when required, must be set on the body before the offsetting operation:

Property	Description
check_offset	`t` or `nil`: ■ off and face checking translates to `nil`, ■ full checking translates to `t`
pierce_faces	list of faces not to be offset
offset_faces	list of faces with specific offsets
tolerance	maximum applied tolerance
max_faults	maximum number of entities on badtaglist

6.7.3 Imprinting

imprint function

The `imprint` function

```
> ( a imprint b)
```

works, where either of a or b are a faceset or a body.

Local Operation Functions 7

7.1 Introduction

Functions which are grouped as local operations operate only on that part of the body represented by the given face set.

Object	Function
face(s)	tweak, ntweak, twefac, create_sheet, remove_faces, create_solid, delete_faces, taper

tweak function

`tweak` is a local operation that can be used to change the existing surface of a face to a given surface. The topology would be unaltered. `tweak` takes as an argument a list of of faces and a corresponding list of surfaces.

```
> (define b1 p_block)
> (define b2 p_cylinder)
> (b1 x 10;y 10;z 10;create)
> (b2 radius 1; height 10; point '(0 5 5);
   direction '(0 1 0); create)
> (b1 unite 'b2)      -- unite small cylinder onto side of block
> (graphics sketch 'b1; silhouette; autowindow; redraw)

> (face help tweak)    -- for information on how to tweak a face
> (define f1 face)
> (f1 pick)            -- pick block top face using the cursor

> (define s1 p_planar)
> (s1 point '(0 0 50); direction '(0 0 1); create)
                   -- s1 is plane at Z = 50 parallel to XY plane
> (f1 tweak 's1) -- this will raise the face up to the surface
         s1, and now body b1 has the dimensions x 1O, y 10, z 50
```

ntweak function

`ntweak` is similar to `tweak`, but changes the existing surface of a face to a given surface but is reversed. The topology must be unaltered.

twefac function

`twefac` is a local operation that can be used to modify the surface of a face by a given transformation. It can also apply a transformation or a list of transformations to either a list of faces or a list of lists of faces. In all cases the topology must be unaltered.

```
> (define  b1 p_block)
> (b1 x 10;y 10;z 10;create)
> (face help twefac)            -- for information on twefac
> (define f1 face)
> (f1 pick_from b1)             -- pick the faces from the block

> (define t1 p_translation)
> (t1 direction '(1 -1 1); distance 30; create)
> (graphics clear)
> (graphics sketch 'b1; autowindow; redraw)

> (f1 twefac 't1)               -- translate all the faces
> (graphics sketch 'b1; autowindow; redraw)

> (f1 twefac 't1)               -- translate all the faces again
> (graphics sketch 'b1; autowindow; redraw)
```

create_sheet function

This face function copies the geometry and topology of the face and uses them to make a new sheet body `create_sheet_temp`. It only works for single items, not sets.

```
> ( define f0 face)
> (f0 pick)
> (f0 create_sheet)
```

remove_faces, create_solid and delete_faces functions

This sub group of face functions create new offspring bodies from a set of faces.

- **create_solid** - Makes a copy of the faces involved and creates a new body from the copy.
- **remove_faces** - Removes the faces from their parent body, heals the parent and uses the face set to make a new body.
- **delete_faces** - Deletes the face set from the parent body and heals the wound. An argument, one of the three listed below, determines how the wounds on parent and offspring bodies are mended.
  - **cap** - Fits the simplest surface to the bounding edges.
  - **grow** - Only for `create_solid` and `remove_faces`. Grows surrounding edges and faces, extending them with their surfaces to cover the wound.
  - **growp** - Makes an offspring body, using parent surfaces and curves of edges to mend the wound.
  - **shrink** - If extending faces does not yield a solution, then shrinking the faces is tried.

The new `child` bodies created are given names of the form `create_solid_c1`, `create_solid_c2` (etc)/ `remove_faces_c1`, `remove_faces_c2` (etc) depending on which function created/removed them, and the number of child bodies created/removed.

In the case of `remove_faces`, where the parent is modified, the resulting parent body is returned as `remove_faces_p1`. `remove_faces` is the only function which can take up to two optional arguments.

- When no arguments are given, cap is used to mend both parent and offspring bodies.
- When only one argument is given, this is used to mend the parent. The default is used to mend the offspring.
- Where two arguments are given, the first is used to mend the parent, the second to mend the offspring.

```
> (define f2 face)
> (f2 pick2)              -- pick two faces of cylindrical boss
                             from a body previously drawn
> (f2 create_solid 'growp) -- create new body with copied faces
                              using growp to mend wound.
> (define f2 face)
> (f2 pick2)              -- pick two faces of cylindrical boss
                             from a body previously drawn
> (f2 remove_faces 'grow)
      -- remove faces and  mend parent with grow and create new
         body with copied faces, using default cap for mend
> (define f2 face)
> (f2 pick2)              -- pick two faces of cylindrical boss
                             from a body previously drawn
> (f2 remove_faces 'growp 'grow)
  -- remove faces and mend parent with grow using parent
     surfaces to mend wound, create offspring with copied faces
> (define f2 face)
> (f2 pick2)              -- pick two faces of cylindrical boss
                             from a body previously drawn
> (f2 delete_faces 'cap)  -- delete these faces from b1 and
                             cap the 'wound'
```

taper function

`taper` drafts a set of faces, which can be any combination of planar, cylindrical and conical surfaces. It has taper properties of point, direction and angle, which define the taper plane and angle (see the description of TAPFAS for details of the effects of this operation and the meaning of the properties involved).

The following tapers a previously defined set of faces by 2 degrees about a taper plane, defined by the point and direction parameters:

```
> ( f1 point '(0 0 10); direction '(0 0 1); angle 2; taper )
```

Miscellaneous Useful Functions 8

8.1 Replication of objects within the modeller

replicate function

The `replicate` function can be used to make replicas of objects and copies of items within the modeller.

Object	Function
modeller	replicate

The following example shows the effect upon objects which do not refer to a kernel item and those the kernel is not capable of copying, in this case before a primitive has been turned into a body.

```
> (define b0 p_sphere)
> (b0 centre '(0 0 1); radius 10 ; colour 'red )
                            -- local properties of b0
> (define b1 p_sphere)      -- the object must be defined first
> (b1 replicate 'b0)        -- b0 is copied into b1
> (b1 colour)               --> red
> (b0 create )              -- create sphere body b0
> (b1 create )              -- create sphere body b1
```

The function copies all the local properties of one object to another object, and overwrites any existing properties with the same names.

If the objects refer to an item in the kernel which can be copied:

```
> (define b0 p_torus)
> (b0 majrad 20; minrad 5; colour 'blue; create)
                          -- create torus b0, property blue
> (define b3 body)
> (b3 replicate 'b0)       -- copy b0 and property into b3
```

This also copies all local properties of b0 to b3, except for the `tag` property. When the `replicate` function encounters the tag property, it calls the KI routine COPYEN to create a copy of the kernel item. The tag of this new item is put into the `tag` property of the copied object.

The kernel objects which cannot have their tags copied in this way are edges, faces, vertices, loops, shells and attributes; a call to `replicate` for these objects creates a new object with the same tag value.

8.2 Renaming a modeller item

rename function

The function `rename` enables a name change of a modeller item.

Object	Function
modeller	rename

```
> (define b0 p_block)
> (b0 x 10; y 20; z 30; create)
> (b0 rename ’p7112)           --> renames b0 to p7112
> (p7112 is)                   --> body
> (b0 is)                      --> error as b0 is now undefined
```

8.3 Selecting an entity using its identifier

identify function

To uniquely select a face from a body in a way that is not session dependent you can use their identifiers.

```
> ( f0 identify )            --> ( FA2 FA33 FA100 )
> ( e0 identify )            --> ( ED12 ED24 )
```

Identifiers are the same after the body is transmitted or received, and they are the same in copies of the part also.

Getting to the tags again from the identifiers requires specifying which part the identifiers refer to:

```
> ( f0 identify_in b0 )       -- mandatory parameter
> ( f0 identify  ’( FA2 FA33 F100 ))
```

This locates the tags of these faces and saves them in the object f0.

This method is much faster and more robust than doing a pick on a complex body. It may be useful to locate faces in test scripts.

8.4 Magnifying, reflecting and mirroring a body

magnify function

The function `magnify` scales a body about it’s center of gravity.

```
> ( body magnify <scale> )
```

reflect function

The function `reflect` images the body in a plane.

```
> (  body reflect <plane> )
```

A plane can conveniently be created by picking from the model. for example:

```
> ( define s1 surface )
> ( s1 pick )               -- & use graphics picking
```

mirror function

The `mirror` function copies a body, reflects it and unites with the mirror image, merging out redundant topology.

```
> (  body mirror <plane> )
```

8.5 Mass properties

Mass property enquiries are driven from general functions associated with the topology object:

```
(topology mass_amount)
(topology mass)
(topology cofg)
(topology mofi)
(topology periphery)
```

The interpretation of the mass_amount (“amount” in the PK documentation) and periphery depend upon the specific type of topology. Bodies, faces and edges have specific enquiries for some of this information:

```
(body volume)
(body area)
(face volume)       --- Treating the face set as a single solid
(face cofg 0.95 t)  --- Treating the face set as a single solid
(face cofg)
(face area)
(face periphery)
(edge length)
```

All the above functions take two optional arguments. The first is used to control the accuracy of the calculations and defaults to 0.95 (this is not especially precise), for example:

`(e0 length 0.9)` would provide a less accurate result.

The second is just a logical which controls whether facesets are considered as a single solid or not. This affects the meaning of the amount and periphery values, e.g.

`(f0 mass_amount 0.95 t)` returns the volume of the enclosed space, whereas

`(f0 mass_amount 0.95)` returns the surface area of the faces.

8.6 KI/PK Functions

It is possible to combine low level calls to individual KI/PK functions,

```
> ( <KI routine name> <arguments separated by spaces>)
```

with KID commands as have been described in this chapter. This may be necessary in cases where only a low level call to a particular KI/PK routine is possible. This is described in Chapter 4, “Calling the KI/PK Using KID (FLICK)”.

Enquiries 9

9.1 Introduction

This chapter describes the enquiries available in KID.

enquire function

It is possible to enquire about topological and geometrical items. The `enquire` function is applicable to points, curves, surfaces, vertices, edges and faces. The function can be used to print out either all available information or just specific details about the geometry of the item. For instance, assuming c1 (a curve), f1 (a face) and a1 (an assembly) have already been picked:

Object	Function
modeller	enquire

```
> (c1 enquire)
> (f1 enquire)
> (a1 enquire)
```

This prints out information about the geometry of the item on the screen.

- For the curve, the type is given, and also information about its geometry. For example, if the curve is a circle, the center point, axis and radius are printed out.
- For the face, the geometry of the attached surface is printed out, together with the values of the sense and reverse flags for the surface and face respectively, and the tolerance. For example, the outside face of a cone has a conical surface, whose surface sense is 1, i.e. positive.
- For the assembly, the number of components it consists of; how many of those components are sub-assemblies; and its box vector are printed out.

Using enquire on a one layer assembly

In addition, for assemblies of only one layer of substructure, a statement to that effect and the number of sheet and solid bodies the assembly consists of are printed out.

Using the level function to create a single layer assembly

To convert an “unlevelled” assembly to a single layer assembly, type:

```
> ( a1 level )
```

Other enquiries on an assembly are:

```
> ( a1 instances )    -- a list of the first layer of instances
> ( a1 bodies )       -- a list of the first layer of bodies
> ( a1 transforms )
                 -- a list of the first layer of transform tags
```

It is also possible to get more specific information from the enquire function. An argument identifying the type of information required is given to the `enquire` function, and the function thens return the value of the information, or `nil` if it is not found.

Assuming c1 is a circular curve:

```
> (c1 enquire 'point)      -->(0 0 0)
> (c1 enquire 'direction) -->(0 0 1)
> (c1 enquire 'radius)     --> 42
> (c1 enquire 'maj_axis)   --> nil
> (c1 enquire 'type)       --> circle
```

9.2 Enquiring/setting the tag property

If a primitive is created, or if an object has been picked from the screen, it has a property which contains the value of the kernel tag which represents the item. KID uses this when it calls the KI / PK. It is possible to access the value of the tag, or to set it to a particular value. For instance, if it is known that tag 99 refers to a face, the following creates an object which may then be manipulated by KID:

```
> (define f1 face)
> (f1 tag 99)         -- set tag property to 99
> (f1 enquire)        --> information
> (f1 tag)            --> gives tag value 99
```

It may prove useful to manipulate tag values in this way to perform operations which are not possible with the normal KID functions.

9.3 Using enquire to construct complex functions

Some KID functions result in the tag property of an entity having a LISP list of tags. An example of this is the function `pick_from`. It is often possible to manipulate an object with a list of tags using the same functions as if it had a single tag. This facility is not provided by all KID functions.

```
> (define b1 p_block)
> (b1 x 10; y 20; z 10; create)    -- create block b1
> (define f1 face)
> (f1 pick_from 'b1)         -- f1 is a list of b1's 6 faces
> (f1 pick_using '(f1 clash '(5 5 10)))
   -- leave only those faces which clash with this point f1
      now refers to a set of three faces from body b1

> (f1 enquire )                    --> information
> (f1 direction '(10 9 8); move)
                     -- this will move all three faces
```

This form of the `enquire` function can be used to construct complex functions where the geometry of one surface is used to define the size of another.

```
> (define b0 p_cone)
> (b0 lrad 10; urad 0; height 30; create)
> (define f1 face)
> (define s1 surface)
> (f1 pick_from b0)
> (f1 pick_using '(f1 clash '(0 0 30)))
          -- see later for clash definition
> (s1 pick_from f1)
> (f1 enquire)
> (s1 enquire)
> (s1 enquire 'sense)
```

The argument type can be used to select specific curve or surface types, which are named as follows:

Curve	**Surface**
line	planar
circle	cylindrical
ellipse	conical
intersection	spherical
B-curve	toroidal
SP-curve	blend
foreign_geometry	B-surface
constant_parameter	swept
	spun
	offset
	foreign_geometry

Use of the `enquire` function within the pick facility using type is described in the section “Picking directly from other objects” in Chapter 14, “Picking”.

9.4 Accessing the KI routine IDCOEN for topological entities

A convenient way of accessing the KI routine IDCOEN for topological entities is:

```
> ( b0 faces )     -- returning a list of b0's (body) faces
> ( f0 edges )     -- returning a list of f0's (face) edges
> ( a0 bodies )    -- returning a list of a0's (assembly) bodies
```

9.5 Enquiring coordinates of box enclosing single item

The function box returns two vectors defining the extremes of a minimal rectangular box aligned with the axis system, and enclosing the single topology item, or list of topology items of the same type. This is true for assemblies, bodies, faces, edges and vertices. For a single vertex point, the box extremes are identical.

Object	**Function**
topology	box

```
> (define b1 p_block)
> (b1 x 10;y 20;z 30; create)
> (b1 box)         --> ( ( -5.0 -10.0 0.0 ) ( 5.0 10.0 30.0 ) )

> (define v1 vertex)
> (v1 pick_from 'b1)
> (v1 box)         --> ( ( -5.0 -10.0  0.0 ) ( 5.0 10.0 30.0 ) )

> (v1 pick_using '(v1 clash '(5 10 30)))
> (v1 box)         --> ( ( 5.0 10.0 30.0 ) ( 5.0 10.0 30.0 ) )
```

9.6 Enquiring on a supplied point

Function clash provides a means of testing whether a supplied point is contained in, on or outside a body, face, edge or vertex. Clash uses the KI routine ENCONT.

Object	**Function**
topology	clash

```
> (define b1 p_block)
> (b1 x 20; y 20; z 10; create) -- creates block
> (b1 help clash)               --> information on clash for b1
> (b1 clash '(0 0 5))           --> in
> (b1 clash '(0 0 0))           --> on
> (b1 clash '(0 0 -5))          --> nil (i.e. outside)
```

Attributes in KID 10

10.1 Using attributes

An example of the function of an attribute `colour`, given a face set f0 with tags is:

```
> (colour enquire)       --> output information about attribute
                             definition
> (f0 colour)            --> delete all colour attributes in f0
> (f0 colour '(0.3 0.3 0.3))
                          --> create colour attributes on all f0
```

10.2 Constructing attributes

There are 5 ways to create a working attribute.

Activate all current attributes

```
> ( attribute update )
```

Interrogates the model for all active attributes and build corresponding objects in KID with matching names. Only attributes new to the World since the last call to this are re-created.

Attach to an existing attribute

```
> ( colour create )
```

The effect is as in the first method, but since only a single attribute is processed this is faster.

Specify a full attribute definition

```
> ( define spin_axis attribute )
> ( spin_axis owners '( face body assembly ))
                                  -- or inherit the default
> ( spin_axis class 5 )
> ( spin_axis prefix "CUSTOMER/" )    -- optional
> ( spin_axis name   "rotation_axis" ) -- mandatory
> ( spin_axis create )
```

Create just from the tag

```
> ( attribute create 21 )
```

This path is used to do the work for the first option.

Post the attribute name but delay creation until actual use

```
> ( define colour system_attribute )
> ( colour lazy )      -- only supported for system defined
attributes
```

10.3 Defining attribute structures

Attribute structures are defined from a list of names, each field in the structure can be of type: real, integer, string, vector, coordinate, direction or axis.

```
> ( define myatt attribute )
> ( myatt structure '( string vector real vector real ))
> ( myatt create )
```

The structure must always be given as a list, even if only one field is needed.

Additionally an attribute with NO fields is valid. This is indicated in KID by defining the structure to be `t`.

```
> ( define marked attribute )
> ( marked structure t )
> ( marked create )
```

If a structure is not defined, KID interrogates the kernel when the attribute is created to try and find an existing structure definition for an attribute of that name.

10.4 Reading from attributes

Values are returned in the order implied by the structure definition, formatted into sublists to reflect the implied structure of axis, coordinate, vector and direction subfields. Since an object may have several tags, each with an attribute attached, or since class 6 and 7 attributes may be attached to a single owner multiple times, each set of attribute data is returned in a separate list.

```
> ( f0 hatching )   --> ( ( 0.1 0.2 0.3 4 ) ( 0.2 0.2 0.2 1 ) )
                    -- data for two tag lists

> ( e0 blend_v5 )
       --> ( ( 0.3 1 ( 0.1 0.2 0.3 ) ( 4.4 5.5 6.6 ) ) )
       -- an attribute structure with 2 vector fields
```

Attributes with a single field return that field without enclosing it in another list.

```
> ( f0 name )      --> ( George )one tag with a name attached
> ( g0 name )      --> ( Peter Bob two tags have a name attached
```

Attributes with no fields return a (t) for each tag in the object taglist which has the attribute attached and nil if there are none at all.

```
> ( f0 marked )    --> ( t t t t ) 4 of the faces are marked
> ( f0 marked )    --> nil - none of the faces are marked
```

10.5 Writing to attributes

The attribute values should be supplied in the order which corresponds to the structure definition.

```
> ( b0 density ’( 135.4 kg/m3 ))
            -- real before string in this case
```

Integer values may be provided in positions in which real values are expected.

```
> ( b0 density ’( 135 kg/m3 ))
```

Reflecting embedded structure of the data values using brackets is optional.

Structures with a single field need not be enclosed in a list. So both the following work:

```
> ( f0 translucency ’( 0.5 ))
> ( f0 translucency 0.5 )
```

Structures with no fields can always be set using the value t. All structures can be unset (deleted) by using the value nil.

```
> ( f0 marked t )            -- set a logical attribute
> ( f0 marked nil )          -- delete for any attribute
```

10.6 Controlling attribute names

If the class of attributes has a standard prefix then it would be convenient to take this for granted whilst working with the attributes. This system is already used for system_attributes which have a prefix of SDL/TYSA_.

```
> ( define site_attribute attribute )
> ( site_attribute prefix "EDS/UG_" )
> ( define system_id  site_attribute )
> ( system_id structure '( integer ))
> ( system_id create )          -- New site_attribute system_id:
                                   full name EDS/UG_SYSTEM_ID
```

Alternatively the full attribute name can be overridden manually before it is first created.

```
> ( define id attribute )
> ( id name "EDS/UG_SYSTEM_ID" )
> ( id structure '( integer ))
> ( id create )   -- New attribute id:full name EDS/
UG_SYSTEM_ID
```

KID Graphics: Overview 11

11.1 Introduction

Within KID there is a graphical support library known as GRA. This module is only contained in KID and not in the PARASOLID kernel library. Calls are made to GRA to define a viewing environment, and then calls may be made to the Parasolid rendering functions. These also call GRA (via the Frustrum) and this results in the appearance of a picture on the screen, or transmission of graphical output to a file.

Opening an Xwindow

To open an X window for graphical display, type:

```
> (graphics open_device ’x)
```

Re-using an existing graphics window

To re-use an existing open graphics window, allowing a lisp script to be re-run multiple times without spawning multiple windows, type:

```
> (graphics reopen_device ’x)
```

Using the class structure, which provides inherited functions and properties, the user may define many different views with great ease. A new view may be defined to show the same objects from a different viewpoint (e.g. for orthographic views), or to show different objects from the same view (e.g. for assembly viewing).

For instance, if it is required to zoom in on an area of the picture a new view can be defined as a subclass of the current view. The window sizes can be changed and all other properties are inherited. The picture can then be drawn for the new view without altering the default graphics view. This scrap view can be deleted or kept for later use.

KID only allows one view to be active, although any number can be defined. When KID is started by the command (modeller start), a default view is defined and selected.

Functions are provided which allow the user to easily change the definition of the view, to send output to files or the screen and to pick items from the display.

11.2 The Class Structure

The pre-defined class structure for KID graphics is very simple, see the “Graphics substructure” section in Appendix A, “KID Class Structure”. All graphics functions and default view information are held in a class called `xgraphics`. `graphics` is a subclass of xgraphics and inherits all its properties.

Altering Defaults

It is possible to alter the defaults which are defined in the `xgraphics` class directly, but it is preferable to leave these alone. There are two methods that can be used, the first is to work in the subclass `graphics`, so that default values can be restored from `xgraphics` (or by removing the properties from graphics – which has the same effect). The other method is to define subclasses in which defaults can be changed and restored to the graphics default in the same way.

```
> ( graphics help)               -- information
> (graphics view_to '(10 2 5))   -- default value changed
> (graphics view_to (xgraphics view_to))
                                -- default retrieved from xgraphics
> (define my_view graphics)
   -- define subclass of graphics with all graphics functions
> (my_view select)          -- select my_view
> (my_view help view_to)    -- help on graphics property view_to
> (my_view view_to '(1 1 1 ))     -- reset view_to
> (my_view - view_to )           -- removes local setting
```

Note: In the following KID examples any subclass of the graphics object can be substituted for the graphics object `graphics`.

Current View

KID allows one class to be the current view. All interaction with GRA is in terms of the attributes (local or inherited) of this current view.

```
> (graphics current_view)
    --> returns the name of the current view
```

11.3 Output Devices

When KID creates and selects the first view (i.e. graphics) it opens a null device to send its output to. It is possible to open and close other devices, and to send output simultaneously to a number of devices. This and other device dependent KID functions are described in Appendix F, “Machine Dependency in KID”.

Framemaker, Interleaf, Laser, Plot and Postscript

These graphics functions are designed to write graphical output to a file in a specified format. Each function takes a text argument which is used for the output filename. The output is equivalent to that which would result from a (graphics redraw) command.

Object	Function
graphics	interleaf, laser, plot, postscript, framemaker

```
> (define b0 p_block)          -- define a block
> (b0 create)                  -- create it
> (graphics sketch 'b0; ar)    -- sketch it
> (graphics laser "block.ln3") -- output the sketch to the file
                                  block as a pixel file
> (graphics zoom 0.5)          -- expand the view
> (graphics postscript "post.pst")
                         -- output the postscript commands for
                            the expanded view to the file post
```

The `postscript` output files are, at least, minimally conforming to the postscript file structuring conventions.

If all output is to be sent to a file then one of the `plotter`, `laser`, `postscript`, `interleaf` and `framemaker` devices can be used. These are opened and closed with the commands:

```
> (graphics open_device '<device_name>)
> (graphics close_device '<device_name>)
```

Once a device has been opened the graphical output is sent, in addition to any open display devices, to a file (plot_file, pixel_file, postscript_file, interleaf_file and framemaker_file respectively). The default files (which have no extension) may be changed, prior to opening the device, by a command of the form:

```
> (graphics postscript_file "output.pst")
```

It is important to close the device before leaving the KID session to ensure that the correct termination commands are appended to the output file. It should also be noted that all images sent to the file are superimposed.

Viewing Environment and Definition 12

12.1 Introduction

The viewing environment is held in properties which are either inherited from the graphics class or are defined locally. Defaults are provided for all graphics classes.

view_to, view_from view_direction view_vertical, perspective functions

These properties allow the user to define how the model is viewed. Examples of the use of these properties, and their default values, are given below:

```
> (graphics view_to '( 0 0 0 ) )     -- look at origin
> (graphics view_from '(3000 1732 1999))
        -- view from specified point
> (graphics view_direction '(-0.3 -0.1732 -0.2))
        -- define specific view direction
> (graphics view_direction)
        -- returns current view direction '-3.0 -0.1732 -0.2'
> (graphics view_vertical '(0 0 1))
        -- make Z axis vertical in view
> (graphics perspective nil)        -- switch off perspective
```

If `graphics perspective` is `nil` (the default), then,

- `graphics view_direction` is used instead of `graphics view_from`, and

If `graphics perspective` is `t`, then'

- the view direction is the vector from the `view_from` point to the `view_to` point.

view function

The following pre-defined view directions can be set using their associated commands:

```
> (graphics view 'top)
> (graphics view 'bottom)
> (graphics view 'left)
> (graphics view 'right)
> (graphics view 'front)
> (graphics view 'back)
> (graphics view 'trimetric)
> (graphics view 'isometric)
```

12.2 Windowing

view_window_xmin/xmax/ymin/ymax

Windowing defines that area of the model image which is in view. The “view_window_” functions are the most basic ones. There default settings are:

```
> (graphics view_window_xmin -100)
> (graphics view_window_xmax 100)
> (graphics view_window_ymin -100)
> (graphics view_window_ymax 100)
```

12.2.1 Using the cursor for redefining the window

The following functions provide an easier way to redefine the window by using the cursor.

pick_window

When a picture has been drawn, “pick_window” enables the cursor for two picks, to define a window diagonal.

```
> (graphics sketch ’b0)
> (graphics pick_window )    -- cursor enabled for two picks
> (graphics redraw )
```

pick_centre

`pick_centre` enables the cursor for a pick which defines the centre of the new window. If more detail of the new window centre is required this operation can be followed by a “zoom”. Be careful not to use “autowindow” immediately after these commands, unless the new window is to be purposely overwritten.

```
> (graphics sketch ’b0)
> (graphics pick_centre)
> (graphics zoom 2)
> (graphics redraw)
```

autowindow

It is possible to “autowindow” the view so that the window is set to the smallest size possible for the objects which are currently drawn.

redraw

“redraw” clears the screen and draws the current GRA graphics data structure. This means that if the graphics drawing list is altered, followed by a “redraw”, the GRA graphics data is output, not reflecting the change in the drawing list.

ar

“ar” is the combination of an “autowindow” and a “redraw”.

```
> (graphics sketch ’b0)     -- b0 sketched with current view
> (graphics autowindow)     -- no visible change
> (graphics redraw)
                    -- sketch of b0 with smallest possible
window
```

or

```
> (graphics ar)             -- autowindow and redraw
> (graphics drawing_list ’( b0 b1)) -- drawing list changed
> (graphics redraw)         -- but only b0 will be drawn
```

Note: Using “autowindow” on its own does not have any visible effect, although it does altered the GRA viewing environment. Its effect is only visible after a “redraw”.

centre

The “centre” function uses a model space pointer to specify the centre of the current view window. To refresh the view, and therefore see the view in relation to the specified “centre”, it is necessary to do a “redraw”.

```
> ( graphics ske ’b0 )
> ( graphics centre ’( 0 0 10 ); redraw )
```

zoom

The “zoom” function takes a real number as its argument. It changes the current window sizes so that the image is magnified or reduced about the “centre” of the current view window.

A factor greater than 1.0 magnifies the image on the screen.

To see the zoomed image it is necessary to refresh the screen using “redraw”.

```
> ( graphics zoom <factor> )
> ( graphics redraw )
```

12.3 View manipulation

pan_left/right/up/down

It is possible to change the view definition by altering the properties previously described. However, a number of functions are provided to allow easy manipulation of the view. The pan functions take a real number as their argument. The effect is to move the edges of the window by the given distance in the given direction.

```
> (graphics pan_left <distance>)
> (graphics pan_right <distance>)
> (graphics pan_up <distance>)
> (graphics pan_down <distance>)
```

rotate_left/right/up/down

The rotate functions also take a real number as their argument. They rotate the viewing direction and the 'from point' about the image by the given amount. The angle is specified in degrees.

```
> (graphics rotate_left <angle>)
> (graphics rotate_right <angle>)
> (graphics rotate_up <angle>)
> (graphics rotate_down <angle>)
```

12.4 Selecting a view

Only one view or graphics subclass can be active at a time, and all graphics output are sent to that view until another one is selected. The view must first be defined, then this view can be selected:

Object	Function
graphics	select

```
> (define view_x graphics)
> (view_x select)
```

Note: Selecting a view does not clear the screen, or draw the frame and axes.

12.5 Clearing the screen and drawing the current view

Object	Function
graphics	clear, axes, frame

clear

The function `clear` clears the graphics screen of the terminal, and empty the GRA stream(s) used by the current view:

```
> (graphics clear)
```

In addition to the `clear` function, use of the abbreviated forms of `sketch` and `hidden` clear the screen before drawing the entity.

```
> (graphics ske 'b0)
> (graphics hid 'b0)
```

frame and axes functions

To display the limits of the current view on the screen, the “frame” function is provided, “axes” draws axes in the current view.

```
> (graphics frame [<t or nil>])
> (graphics axes [<t or nil>])
```

Using these functions without any arguments draws a frame or axes in the view. If an argument is given, this is interpreted as a logical value (t or nil). If the value given is `t` (i.e. anything but nil) a frame or axes are automatically drawn when ever the view is redrawn.

12.6 Use of the drawing list

drawing_list

Each view has a drawing list property, e.g. `view_1 drawing_list`. This is used to contain a list of the objects which have been drawn in this particular view. Initially, the drawing list is empty, but when objects are rendered their names are added to this list.

Some functions redraw all the items in the drawing list. If the view has no drawing list itself, they look up the class tree until a non empty drawing list is found. Therefore, one class may be used to hold the drawing list and its subclasses may be views which are used to render the objects.

It is possible to add items to the drawing list without rendering them. This could be used to create a sketch of a number of objects.

```
> (define my_view graphics)
> (my_view select)
> (my_view drawing_list '(b0 e1 f2))
> (my_view sketch)   -- objects b0, e1 and f2 will now be drawn
```

Note: 1) The object which is the argument should be a list of KID objects, which represent kernel items with tags, and it must be quoted (**'**) to avoid it being evaluated.

2) `graphics clear` does not clear the `drawing_list` which is used for “Picking”. Therefore, it is possible to pick objects from what appears to be a clear display. The `drawing_list` must be reset independently.

```
> (graphics drawing_list '(b0))-- drawing_list only contains b0
> (graphics drawing_list nil)    -- empty the drawing_list
```

12.7 Enquiry

It is possible to enquire about a view:

Object	Function
graphics	enquiries

```
 > (view_x enquire)   -- prints information about the view
```

KID Rendering 13

13.1 Introduction

The KID rendering functionality uses the rendering functions in Parasolid’s PK interface. The KID rendering options have retained their six-character naming convention (as used in the KI interface), but are applied to the options used by the PK functions.

To render items in a given view, the view must be the current one, i.e. the last one to be selected. The following functions are available.

13.1.1 Wire frame pictures

sketch

Using the function “sketch”, bodies, faces, edges and B-surfaces can be sketched.

Object	Function
graphics	sketch

```
> (graphics sketch ’[<object>])
```

If the optional argument is not given, all the items in the `drawing_list` property are rendered in the appropriate style. If an argument is given (which can be a single object or a list of objects), a new drawing list is defined which contains only these objects.

Another way to sketch objects is by the use of functions which are properties of the object itself.

Object	Function
topology	sketch

```
> (define view_1 graphics)
> (view_1 sketch ’b0)
> (view_1 zoom 2)
> (view_1 redraw)
```

This draws the object according to the current view and adds the object to the drawing list of the current view, unless it is already there.

13.1.2 Hidden line pictures

hidden

Object	Function
graphics	hidden

```
> (graphics hidden '[<object>])
```

If the optional argument is not given, all the items in the `drawing_list` property are rendered in the appropriate style. If an argument is given (which can be a single object or a list of objects), a new drawing list is defined which contains only these objects.

13.1.3 Shaded pictures

shade

The procedure for opening an Xwindow for a shaded graphical display differs from that described previously. When using the “shade” function the following calls are used to open an Xwindow:

```
> (graphics open_device 'xcolour)
> (graphics shading_output ['device | 'file] )
                         -- use 'device to output to the screen
                             and 'file to output to a file
```

Object	Function
graphics	shade

```
> (graphics shade '[<object>])
```

If the optional argument is not given, all the items in the `drawing_list` property are shaded according to the given shading options. If an argument is given (either a single object, or a list of objects), the `drawing_list` property is redefined, and the given objects are shaded.

Note: Note that the shade function is only available on UNIX, it does not work on NT.

13.1.4 Faceted pictures

facet

Object	Function
graphics	facet

```
> (graphics facet '[<object>])
```

If the optional argument is not given, all the items in the `drawing_list` property are faceted according to the given faceting options. If an argument is given (either a single object, or a list of objects), the `drawing_list` property is redefined, and the given objects are faceted and output through the GO.

13.2 Rendering options

For all of the rendering options the option can be called by either its long name (as used in all the following option examples) or its code.

```
> (graphics drafting t)
    OR
> (graphics rropdr t)
```

anti_aliasing (RROPAN)

This option controls the anti_aliasing of entities passed to the shade function.

```
> (graphics anti_aliasing [ t | nil ])
```

If no argument is given, the current value of the `anti_aliasing` option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPAN is set
nil	option RROPAN is unset (default)

background_colour (RROPBK)

This option controls the background colour that is passed to the shade function.

```
> (graphics background_colour [argument])
```

If no argument is given, the current value of the background colour option is returned. If an argument is given, it is interpreted as follows:

Value	Description
list of three values	option RROPBK is set with the given values (in the order below)
anything else	option RROPBK is unset (default)

If the option RROPBK is not set, the following defaults are used for background color:

Color	Value
Red	0.0
Green	0.0
Blue	0.0

blend (RROPUB)

This option specifies that the rendering operation takes account of all unfixed blends in the entity passed to the sketch function.

```
> (graphics blend [ t | nil ])
```

If no argument is given, the current value of the unfixed blend option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPUB is set
nil	option RROPUB is unset (default)

convexity (RROPCV)

This option controls the convexity of the facets, that are to be output as convex polygons, by the facet function.

```
> (graphics convexity [<anything but nil OR nil>])
```

If no argument is given, the current value of convexity is used if it has been set; otherwise a default is used. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPCV is set
nil	option RROPCV is unset (default)

curve_tolerance (RROPCT)

This option controls the faceted representation by considering the curved edge approximation of the entity that is passed to the sketch, facet and hidden functions.

```
> (graphics curve_tolerance [argument])
```

If no argument is given, the current values of the curve tolerances are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
list of three values	option RROPCT is set with the given values (in the order below)
anything else	option RROPCT is unset (default)

The curve tolerances are:

- Chord tolerance in model units
- Maximum chord length in model units
- Angular tolerance in radians

depth_modulation (RROPDM)

This option controls depth modulation of an entity that is passed to the shade function.

```
> (graphics depth_modulation [argument])
```

If no argument is given, the current value of the depth modulation option is returned. If an argument is given, it is interpreted as follows:

Value	Description
list of one value	option RROPDM is set with the given value
anything else but nil	option RROPDM is set with the value 0.3 (default)
nil	option RROPDM is unset

drafting (RROPDR)

This option controls the output of drafting-style lines of entities that are passed to the hidden function by distinguishing between lines which are blocked by other lines and those which are obscured by other faces of the body.

```
> (graphics drafting [ t | nil ])
```

If no argument is given, the current state of the drafting-style lines option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPDR is set
nil	option RROPDR is unset (default)

Note: The “drafting (RROPDR)” and “perspective (RROPPS)” options are mutually exclusive and turn each other off.

edge_data (RROPED)

This option controls the drawing of edge data of the entities passed to the sketch function.

```
> (graphics edge_data [ t | nil ])
```

If no argument is given, the current state of the edge data option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPED is set
nil	option RROPED is unset (default)

edge_tags (RROPET)

This option controls output of the edge tag for those facet edges derived from face edges for entities passed to the facet function.

```
> (graphics edge_tags [ t | nil ])
```

If no argument is given, the current values of the edge tags are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPET is set
nil	option RROPET is unset (default)

face_colour (RROPFC)

This option controls the face colour of entities passed to the shade function.

```
> (graphics face_colour [argument])
```

If no argument is given, the current value of the face colour option is returned. If an argument is given, it is interpreted as follows:

Value	Description
list of three values	option RROPFC is set with the given values (in the order below)
anything else	option RROPFC is unset (default)

If the option RROPFC is not set, the following defaults are used for face color:

Color	Value
Red	1.0
Green	1.0
Blue	1.0

facet_infinite (RROPFI)

This option controls the non-generation of facets for faces which can quickly be identified as back-facing in the view from infinity direction of entities passed to the facet function.

```
> (graphics facet_infinite [ t | nil ])
```

If no argument is given, the current state of the `facet_infinite` option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPFI is set
nil	option RROPFI is unset (default)

Note: The “facet_infinite (RROPFI)”,“facet_perspective (RROPFP)” and “vertex_matching (RROPVM)” options are exclusive and turning one on turns the others off.

facet_minimum_size (RROPMF)

This option controls the minimum size of the facets of entities passed to the facet function.

```
> (graphics facet_minimum_size [argument])
```

If no argument is given, the current value of the minimum facet size, if set, is returned. If an argument is given, it is interpreted as follows:

Value	Description
n	option RROPMF is set with the given value
anything else	option RROPMF is unset (default)

facet_perspective (RROPFP)

This option controls the non-generation of facets for faces which can quickly be identified as back-facing in the perspective view direction of entities passed to the facet function.

```
> (graphics facet_perspective [ t | nil ])
```

If no argument is given, the current state of the `facet_perspective` option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPFP is set
nil	option RROPFP is unset (default)

Note: The “facet_infinite (RROPFI)”,“facet_perspective (RROPFP)” and “vertex_matching (RROPVM)” options are exclusive and turning one on turns the others off.

facet_size (RROPFS)

This option controls the faceted representation of entities passed to the facet function by considering the size of the facet.

```
> (graphics facet_size [argument])
```

If no argument is given, the current values of the facet size tolerances are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
list of two values	option RROPFS is set with the given values (in the order below)
anything else	option RROPFS is unset (default)

The facet size tolerances are:

- Maximum number of sides per facet
- Maximum width of facet in model units

facet_strips (RROPTS)

This option controls the output of faceted data of entities passed to the facet function to be in “triangle strips” which form triangle facets.

```
> (graphics facet_strips [ n | nil ])
```

If no argument is given, the current value of the `facet_strips` option is returned. If an argument is given, it is interpreted as follows:

Value	Description
n	option RROPTS is set with the given maximum number of facets in a strip
nil	option RROPTS is unset (default)

first_derivatives (RROPD1)

This option controls the output of first derivatives data for entities passed to the facet function.

```
> (graphics first_derivatives [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPD1 is set
nil	option RROPD1 is not set (default)

hierarchical (RROPHR)

This option controls the output of hierarchical data for entities passed to the hidden function by outputting the data for the invisible part of partial visible lines.

```
> (graphics hierarchical [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPHR is set
nil	option RROPHR is not set (default)

hierarchical_no_geom (RROPHN)

As for the hierarchical option, this option controls the output of hierarchical data for entities passed to the hidden function by outputting the data for the invisible part of partial visible lines, but omitting any geometry segments.

```
> (graphics hierarchical_no_geom [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPHN is set
nil	option RROPHN is not set (default)

hierarchical_parametrised (RROPHP)

As for the hierarchical option, this option controls the output of hierarchical data for entities passed to the hidden function by outputting curve parameters with visibility segments.

```
> (graphics hierarchical_parametrised [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPHP is set
nil	option RROPHP is not set (default)

holes_permitted (RROPHO)

This option controls whether or not facets passed to the facet function are represented with holes in their interiors.

```
> (graphics holes_permitted [ t | nil ])
```

If no argument is given, the current values of the holes permitted are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPHO is set
nil	option RROPHO is not set (default)

ignore_loops (RROPIL)

This option specifies those loops that are to be ignored when faceting a body.

```
> (graphics ignore_loops 56)          -- ignore a specified loop
> (graphics ignore_loops '(56 70 92)  -- or a list of loops
> (graphics ignore_loops 'loo_object) -- or a loo (loop) object
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
tag/list of tags	faceting ignores the specified loop/s
tag of loo (loop) object	faceting ignores the loops in the loo object – if the loo object is updated, the next faceting call takes note of this
nil	option RROPIL is not set (default)

Note: It may be difficult to select the loops as there are few functions available for loops in KID. The following example may help.

Selecting a loop to ignore:

```
(defun intersection (a b)
      (cond
        ((null a) nil)
        ((member (car a) b) (cons (car a)
                            (intersection (cdr a) b)))
        (t (intersection (cdr a) b))))

(modeller start)
(graphics open_device ’x)
((define b0 p_block) create)
((define c0 p_cylinder) height 20; point ’(0 0 5);
 direction ’(0 1 1 ); radius 2; create)
((define t0 p_torus) point ’(0 5 10);
 direction ’(1 0 0 ); minrad 1; majrad 6; create)

(b0 subtract ’c0)
(b0 subtract ’t0)
(graphics silhouette t; sketch ’b0;ar)
((define f0 face) pick_from b0;
     pick_using ’(eq (f0 enquire ’type) ’cylindrical))

((define e0 edge) pick
    ’(( 222.09889296390998 128.22734466248986
        155.80355543323157 )
    ( -0.75000326975537235 -0.43300948728521493
      -0.49999787927274819 )))
((define loop_0 loo) tag (intersection
        ((define loop_1 loo) pick_from f0)
        ((define loop_2 loo) pick_from e0)))

(graphics ignore_loops ’loop_0; clear; facet ’f0; ar)
```

image_smoothness (RROPIS)

This option controls whether the hidden function calculates the smoothness of edges in the image, i.e. whether the faces either side of the edge are tangent. If an edge is smooth, the hidden function also calculates whether or not it is coincident with a silhouette. The example KID Frustrum does not draw smooth edges which are not coincident with silhouettes if `(graphics smooth t)` is selected, so for example mergeable edges are omitted from hidden line pictures.

```
> (graphics smooth [ t|nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	options RROPIS and RROPDS are set
nil	options RROPIS and RROPDS are not set (default)

internal_edges (RROPIE/N)

This option controls the output data for an edge that is passed to the hidden and sketch functions by specifying whether or not it is an internal edge.

```
> (graphics internal_edges [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPIE/N is set
nil	option RROPIE/N is not set (default)

Note: This turns both RROPIN and RROPIE on for the hidden and sketch relevant functions. To switch the options individually use (graphics rropie) and (graphics rropin).

invisible (RROPIV)

This option controls the output for hidden lines passed to the hidden function, so that hidden lines can be rendered in a dotted line-style.

```
> (graphics invisible [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPIV is set
nil	option RROPIV is not set (default)

lights function

This function controls the light sources that are passed to the shade function.

```
> (graphics lights [<list of lists each containing seven
values>])
```

The argument given is a list containing a number of lists, each defining a light source. If no argument is given, the current list of light sources is returned. If an argument is given, the current list of light sources is set to the value of the argument.

A number of pre-defined light sources are available as follows:

Value	Description
ambient_high	(3 0.75 0.75 0.75 1 1 1)
ambient_med	(3 0.50 0.50 0.50 1 1 1)

Value	Description
ambient_low	(3 0.25 0.25 0.25 1 1 1)
lightx_high	(1 0.75 0.75 0.75 1 0 0)
lightx_med	(1 0.50 0.50 0.50 1 0 0)
lightx_low	(1 0.25 0.25 0.25 1 0 0)
lighty_high	(1 0.75 0.75 0.75 0 1 0)
lighty_med	(1 0.50 0.50 0.50 0 1 0)
lighty_low	(1 0.25 0.25 0.25 0 1 0)
lightz_high	(1 0.75 0.75 0.75 0 0 1)
lightz_med	(1 0.50 0.50 0.50 0 0 1)
lightz_low	(1 0.25 0.25 0.25 0 0 1)
lightxy_high	(1 0.75 0.75 0.75 1 1 0)
lightxy_med	(1 0.50 0.50 0.50 1 1 0)
lightxy_low	(1 0.25 0.25 0.25 1 1 0)
lightxz_high	(1 0.75 0.75 0.75 1 0 1)
lightxz_med	(1 0.50 0.50 0.50 1 0 1)
lightxz_low	(1 0.25 0.25 0.25 1 0 1)
lightyz_high	(1 0.75 0.75 0.75 0 1 1)
lightyz_med	(1 0.50 0.50 0.50 0 1 1)
lightyz_low	(1 0.25 0.25 0.25 0 1 1)
lightxyz_high	(1 0.75 0.75 0.75 1 1 1)
lightxyz_med	(1 0.50 0.50 0.50 1 1 1)
lightxyz_low	(1 0.25 0.25 0.25 1 1 1)

The default lights are ambient_high, lightxz_med and lightyz_low

no_fitting (RROPNF)

This option controls the way facets fit together at the edges of adjacent faces when they are passed to the facet function.

```
> (graphics no_fitting [ t | nil ])
```

If no argument is given, the current values of no fitting are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPNF is set
nil	option RROPNF is unset (default)

nurbs_curves (RROPNC)

This option controls the output of NURBs curves passed to the sketch and hidden functions by allowing the NURB curve data to be presented to the Graphical Output in B-spline form.

```
> (graphics nurbs_curves [argument])
```

If no argument is given, the current setting is returned. If an argument is given, it is interpreted as follows:

Value	Description
anything but nil	option RROPNC is set
nil	option RROPNC is unset (default)

Note: The “nurbs_curves (RROPNC)” and “parametric_curves (RROPPC)” options are mutually exclusive and turn each other off.

para_hatch (RROPPA)

This option controls the hatching of composite B-surfaces passed to the sketch and hidden functions. Two parameters are given to the option that specify the spacing between hatchlines in both the u and v directions.

```
> ( f0 para_hatch )             -- enquire face hatching
> ( f0 para_hatch <space> )     -- set both u and v para spacing
> ( f0 para_hatch <u> <v> )     -- set both independently
```

The faces are not actually hatched by this command. They appear hatched in the next graphics command providing the global graphics rendering switches are set:

```
> ( graphics para_hatch t )
```

Hatching is disabled by using the argument nil.

```
> (graphics para_hatch nil )
```

parameter_information (RROPPI)

This option controls the output of parameter information for entities passed to the facet function.

```
> (graphics parameter_information [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPPI is set
nil	option RROPPI is not set (default)

parametric_curves (RROPPC)

This option controls the output of B-curves (Bezier curves) passed to the sketch and hidden functions by allowing the B-curve data to be presented to the Graphical Output in Bezier form or as poly lines.

```
> (graphics parametric_curves [argument])
```

If an argument is given, it is interpreted as follows:

Value	Description
anything but nil	option RROPPC is set
nil	option RROPPC is unset (default)

Note: The “nurbs_curves (RROPNC)” and “parametric_curves (RROPPC)” options are mutually exclusive and turn each other off.

perspective (RROPPS)

This option controls the whether or not the entities passed to the sketch, hidden, sketch and shade functions are created in a perspective view.

```
> (graphics perspective [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is supplied, it is interpreted as follows:

Value	Description
t	option RROPPS is set
nil	option RROPPS is not set (default)

Note: The “drafting (RROPDR)” and “perspective (RROPPS)” options are mutually exclusive and turn each other off.

planar_hatch (RROPPH)

This option controls the hatching of planar surfaces passed to the sketch and hidden functions. It takes an argument, which if set to nil, disables planar hatching, otherwise a list of 4 parameters is required:

```
> ( f0 planar_hatch )        -- enquire face hatching
> ( f0 planar_hatch <gap>)-- set hatching space (default Z dir)
> ( f0 planar_hatch <gap> <direction> )
```

The faces are not actually hatched by this command. They appear hatched in the next graphics command providing the global graphics rendering switches are set:

```
> ( graphics planar_hatch t )
```

Planar hatching is disabled by using the argument nil:

```
> ( graphics planar_hatch nil )
```

planarity_tolerance (RROPPT)

This option controls the planarity tolerance of facets passed to the facet function.

```
> (graphics planarity_tolerance [argument])
```

If no argument is given, the current values of the surface tolerances are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
list of two values	option RROPPT is set with the given values (in the order below)
anything else	option RROPPT is unset (default)

The planarity tolerances are:

- distance tolerance in model units
- angular tolerance in radians

radial_hatch (RROPRH)

This option controls the radial hatching of entities passed to the sketch and hidden functions.

The argument set to nil disables radial hatching. For hatching on, a list of three parameters is required:

```
> ( f0 radial_hatch )              -- enquire face hatching
> ( f0 radial_hatch <gap>            -- and angle around spine
> ( f0 radial_hatch <gap> <ang1> <ang2> )
                                     -- and angle about spine
```

The faces are not actually hatched by this command. They appear hatched in the next graphics command providing the global graphics rendering switches are set:

```
> ( graphics radial_hatch t )
```

Radial hatching is disabled by using the argument nil:

```
> ( graphics radial_hatch nil )
```

regional (RROPRG)

This option controls the production of regional data for all visible edges and silhouettes passed to the hidden function.

```
> (graphics regional [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is supplied, it is interpreted as follows:

Value	**Description**
t	option RROPRG is set
nil	option RROPRG is not set (default)

regional_attribute (RROPRA)

This option controls the controls the creation of regional data for all visible edges and silhouettes adjacent to any face with regional-data that are passed to the hidden function.

```
> (graphics regional_attribute [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is supplied, it is interpreted as follows:

Value	**Description**
t	option RROPRA is set
nil	option RROPRA is not set (default)

resolution

This function controls the pixel map that is passed to the shade function.

```
> (graphics resolution [argument])
```

If no argument is given, the current resolution value is returned. If an argument is given, the pixel map is set with values according to the argument as follows:

Value	Description
very high	512 512 pixel_size_x pixel_size_y 256 256
high	256 256 pixel_size_x pixel_size_y 128 128 (default)
medium	128 128 pixel_size_x pixel_size_y 64 64
low	64 64 pixel_size_x pixel_size_y 32 32
list of six values	the pixel map is set with the given values
anything else	64 64 pixel_size_x pixel_size_y 32 32

The pixel sizes are calculated automatically according to the resolution (e.g. 256 x 256) and the current window size. It is therefore best to sketch the object(s), autowindow, redraw and then shade.

second_derivatives (RROPD2)

This option controls the output of first derivatives data for entities passed to the facet function.

```
> (graphics second_derivatives [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPD2 is set
nil	option RROPD2 is not set (default)

shade_file

This function controls the name of shade file output by the shade function.

```
> (graphics shade_file [ <'file_name> ])
```

If no argument is given, the name of the current shade file is returned. If an argument is given, the shade file is set to the value of the argument.

The default shade file “graphics.sig” is a binary file which consists of header information, of which the first four arguments detail the offset of the pixel image, followed by the run-length encoded RGB pixel intensities.

When required the following C program can be used to decode .sig files:

```
#include <stdio.h>

typedef struct    {  int a;
                     int b;
                     int x;
                     int y;
                  }  header_t;

typedef struct    {  double red;
                     double green;
                     double blue;
                     int count;
                   }  run_t;
   main ( argc, argv )
   int    argc;
   char * argv[];
   {
   run_t    run;
   header_t header;

   /*Read header*/

   fread( &a, sizeof( int ), 1, stdin );
   fread( &b, sizeof( int ), 1, stdin );
   fread( &x, sizeof( int ), 1, stdin );
   fread( &y, sizeof( int ), 1, stdin );
   printf( "HEADER %d %d x=%d, y=%d\n",
            header.a,header.b,header.x,header.y);
   printf("COUNT RED GREEN BLUE\n" );
   while (fread( &run, sizeof( run_t ), 1, stdin) > 0)
         {
             printf( "%-6d %-8g %-8g %-8g\n",
             run.count, run.red, run.green, run.blue );
         }
   }
```

Assume b0 to be a sphere with a transparency attribute attached to its face, and b1 to be a large cube. b0 lies in front of b1:

```
> (graphics select)
> (graphics anti_aliasing t)
> (graphics surface_reflection '(0.99 0.00 0.01 0.10 20))
> (graphics depth_modulation '(0.75))
> (graphics face_colour '(1 0 0))
> (graphics background_colour '(0 0 1))
> (graphics translucence t)
> (graphics resolution 'very_high)
> (setq lightxz_red '(1 0.75 0.00 0.00 1 0 1))
> (setq lightyz_blue '(1 0.00 0.00 0.50 0 1 1))
> (graphics lights
     (list ambient_high lightxz_red lightyz_blue))
> (graphics shade_file "shade_file.pix")
> (graphics shade '(b0 b1))
```

silhouette (RROPSI)

This option controls the creation and labelling of silhouette lines for entities passed to the sketch function.

If the optional argument is not given, silhouette curves are drawn in the current drawing list only. If the optional argument is set to `t`, silhouettes are drawn in every rendering operation.

```
> (graphics silhouette [ t | nil ])
```

The blend option controls unfixed blend surfaces rendering. If set to t, they are included in the next draw. Integers given as arguments can be used in several ways:

```
> (graphics blend 1)        -- draw as attributes specify
   OR
> (graphics blend t)       -- exactly equivalent
> (graphics blend 2)       -- only draw blend boundaries
> (graphics blend '(3 n ))
  -- where n is set to rib spacing draw boundaries and
     overwrite rib attribute if different
```

silhouette_density (RROPSD)

This option controls the silhouette density output for entities passed to the facet function.

```
> (graphics silhouette_density [argument])
```

If no argument is given, the current value of the silhouette density option is returned. If an argument is given, it is interpreted as follows:

Value	Description
list of five values	option RROPSD is set with the given values
nil	option RROPSD is unset (default)

The five values represent the following properties:

- the first three values define the view direction
- the fourth defines the angular tolerance
- the fifth defines the chordal tolerance

smooth_edges (RROPSM)

This option controls whether the sketch function calculates the smoothness of edges in the image, i.e. whether the faces either side of the edge are tangent.

```
> (graphics smooth_edges [ t | nil ])
```

If no argument is given, the current state of this option is returned. If an argument is given it is interpreted as follows:

Value	Description
t	option RROPSM is set
nil	option RROPSM is not set (default)

smooth_edges_do_not_block (RROPDS)

This option controls whether or not the hidden function allows smooth edges which are not coincident with silhouettes to occlude other lines in drafting mode. As the KID Example Frustrum does not draw smooth edges if (graphics smooth) is selected, then this option should also be selected with it. The option ensures that the situation where a smooth edge occludes another but then does not itself get drawn cannot occur.

```
> (graphics smooth_edges_do_not_block [ t | nil ])
```

The option is automatically selected when `(graphics smooth)` is selected. You probably only need to control it independently of `(graphics smooth)` when the frustrum is designed to draw smooth edges in a different line style rather than omitting them completely, for example.

Value	Description
t	option RROPDS is set
nil	option RROPDS is not set (default)

surface_reflection (RROPSF)

This option controls the surface reflectivity of entities passed to the shade function.

```
> (graphics surface_reflection [argument])
```

If no argument is given, the current value of the surface reflectivity option is returned. If an argument is given, it is interpreted as follows:

Value	Description
list of five values	option RROPSF is set with the given values (in the order below)
anything else	option RROPSF is unset (default)

The five values represent the following properties, if the option is not set, the defaults shown are used.

Property	Default
Coefficient of specular reflection	0.90
Proportion of colour in highlights	0.00

Property	Default
Coefficient of diffuse reflection	0.60
Coefficient of ambient reflection	0.25
Reflection power	20

surface_tolerance (RROPST)

This option controls the surface tolerance of entities passed to the facet function by considering the approximation to the surface.

```
> (graphics surface_tolerance [argument])
```

If no argument is given, the current values of the surface tolerances are used if they have been set; otherwise defaults are used. If an argument is given, it is interpreted as follows:

Value	Description
list of two values	option RROPST is set with the given values (in the order below)
anything else	option RROPST is unset (default)

The surface tolerances are:

- distance tolerance in model units
- angular tolerance in radians

transform (RROPTR)

This option controls the transformed positions of all entities passed to the sketch, hidden, shade and facet functions.

```
> (graphics transform [argument])
```

An example of applying a transform when rendering:

```
> ( (define t0 p_translation) direction '(0 0 1);
    distance 2; create )
> ( (define b0 p_block) create )
> ( graphics ske 'b0; ar; transform 't0; sketch; ar )
```

When using the above sequence of commands, if there are many entities in the drawing list, but only one transform is supplied, then the transform list is replicated to ensure that there are an equal number of transforms as entities. At all times the list of transforms must match the list of entities.

translucence (RROPTL)

This option controls the translucent rendering of those faces passed to the shade function that have a specifying attribute attached.

```
> (graphics translucence [ t | nil ])
```

If no argument is given, the current value of the translucence option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPTL is set
nil	option RROPTL is not set (default)

vertex_matching (RROPVM)

This option ensures that there are no gaps along the model edges and that along these edges there are no facet vertices which are interior to an adjacent facet edge in the entity that is passed to the facet function.

```
> (graphics vertex_matching [ t | nil ])
```

If no argument is given, the current value of the vertex_matching option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPVM is set
nil	option RROPVM is unset (default)

Note: The “facet_infinite (RROPFI)”,“facet_perspective (RROPFP)” and “vertex_matching (RROPVM)” options are exclusive and turning one on turns the others off.

vertex_normals (RROPVN)

This option allows the surface normal to be output at every facet vertex in the entity passed to the facet function.

```
> (graphics vertex_normals [ t | nil ])
```

If no argument is given, the current value of the vertex_normals option is returned. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPVN is set
nil	option RROPVN is unset (default)

viewport (RROPVP)

This option attempts to render those bodies/faces which are inside or partly inside the supplied viewport.

```
> (graphics viewport [argument])
> (graphics viewport)          --- returns the current viewport
```

If no argument is given, the default viewport is used in which the full image is rendered. If an argument is given, it is interpreted as follows:

Value	Description
t	option RROPVP is set
nil	option RROPVP is unset (default)
list of 15 doubles	sets the viewport
’box	create a viewport for the boxes of all the items in the current drawing list

Whenever the viewport is turned ON (by either of the three relevant options), the view is recalculated after each “zoom” or “autowindow”.

Picking 14

14.1 Introduction

Assemblies, bodies, points, edges, faces, vertices, curves and surfaces can be picked from the screen or directly from other entities:

- an assembly can be picked from an assembly
- a body can be picked from an assembly or body
- points, edges, faces, vertices, curves and surfaces can be picked from themselves or entities higher in the structure

Picking makes a connection between the KID object and the tag of the kernel item.

14.2 Picking from the screen

pick

To pick an item from the screen, the command `pick` is used. Initially the user has to define the type of entity which is to be picked from the items displayed on the screen. The command:

```
> ( <entity> pick )
```

produces a cursor on the screen, which can then be positioned over the entity (face, edge, body, etc.) which is to be picked. Pressing an appropriate key records the tag or coordinates for the pick. In the case of topological items further calls to `( <entity> pick )` allow further entities of the same type to be picked and added to the tag list of `<entity>`.

Object	Function
entity	pick

```
> (b0 sketch)         -- sketch the body
> (define f1 face)    -- define the object
> (f1 pick)           -- puts up the moveable crosswires -
                         press any key to pick the item
> (f1 sketch)         -- now the object can be used
```

`pick` is also a function of `p_points`, and can be used to pick coordinate sets from the screen in image space, or directly by giving an optional argument of a set of coordinates. This is described in the section “Using p_points to create a p_profile”. The function pick can pick indirectly from the screen when used with two optional arguments.

The pick can only be made from a graphics view, so that if a model is not drawn the pick cannot be made. The entity to be picked must first be defined.

To assist the above operations the commands pick2, and pick3 are used, these commands expect multiple picks from the screen to be made.

Using p_points to create a p_profile

The primitive `p_points` can be used with the function “pick” to build a set of coordinates in the image plane. The `p_points` object must first be defined, and the cursor enabled by the “pick” function. A series of points can be picked in the image plane, by pressing an appropriate key, and the sequence terminated by picking a previously picked point. The item of class `p_points` can be used to create a body from a profile using `p_profile`, which in turn can be swept or swung to produce a new item.

```
> (define b0 p_block)
> (b0 x 10; y 10; z 10; create)
> (graphics sketch ’b0)
> (define p1 p_points) -- define p1 as p_points
> (p1 pick)            -- enable cursor for picking coordinate
                       -- set in the image plane
> (define b1 p_profile)
```

The coordinates for b0 can also be given in either of the ways shown:

```
> (b1 coordinate (p1 coordinate))  -- implicit coordinate list
> (b1 coordinate ‘( (1 2 3 ) (...)))
    -- explicit coordinate list
> (b1 create)
    -- create kernel item (acorn, wire  or sheet body)
```

14.2.1 Pick with one argument

When “pick” is invoked with just one argument (the position vector of the eye point in model space), the nearest entity to that point in the current view direction is picked. Only a face facing toward the `eye_point` would be picked, assuming a face had been defined.

14.2.2 Pick with two arguments

When both arguments are given, the implicit current view direction is overwritten with the second argument, and entities are picked in an identical way. This mode of use could be applied when an eye point and view direction are already known, for example when taken directly from a journal file.

```
> (f1 pick ‘(200 200 200) ‘(1 1 1) )
```

Note: The “pick” command actually uses the graphics “drawing_list” to determine which entities have been selected.

The “drawing_list” is not reset by the (graphics clear) command but needs to be reset independently:

```
> (graphics drawing_list nil)
```

If the “drawing_list” is not reset then it is possible to pick from objects which no longer appear in the display. Conversely, if “pick” is being used with arguments then it is only necessary to add the entities to the drawing_list in order to pick from them – there is no actual need to display them:

```
> (define b0 p_block)              -- define a p_block
> (b0 create)                      -- create the block
> (graphics drawing_list nil)      -- empty the drawing_list
> (graphics add ’b0)               -- add b0 to the drawing_list

               (graphics drawing_list ’(b0)) would be
               equivalent to the previous two commands

> (define f0 face)         -- define a face
> (f0 pick ’(0 0 11))      -- pick the face nearest to ’(0 0 11)
                              in the current view_direction
```

14.3 Picking directly from other objects

Picking can be done without an image on the screen. This type of logical picking is achieved by successive qualification, for example by picking all the faces in a body, then selecting only those with specified geometric properties. Three functions can be used, “pick_from”, “pick_using” and “pick_node”.

pick_from

“pick_from” collects all items of a particular type from an object below the entity class. The equivalent statement might be “all edges in the body”. “pick_from” purges any duplicates from the resulting tag list. Unlike “pick”, “pick_from” is not cumulative, and a further call to it replaces the tags collected previously with a new set.

pick_using

“pick_using” filters a set of items using a function as a qualifying clause. This is equivalent to a statement such as “only those faces with spherical surfaces”. If “pick_using” finds no items, an empty sublist is returned. A few examples of this are shown next.

pick_node

“pick_node” uses the node identifier of an object to pick it from the body or KI assembly to which it belongs.

Object	Function
entity	pick_from, pick_using, pick_node

```
> (define b1 p_block)
> (b1 x 20; y 20; z 10; create)   -- create block b1
> (define f1 face)
> (define e1 edge)
> (define e2 edge)
> (define v1 vertex)
> (f1 pick_from 'b1)       -- f1 has a list of all 6 face tags
> (e1 pick_from 'b1)       -- e1 has a list of all 12 edge tags
> (v1 pick_from 'b1)       -- v1 has a list of all 8 vertex tags

> (f1 pick_using '(f1 clash '(0 0 0)))
      -- f1 will now only possess tags of faces which
         contain the point '(0 0 0) i.e. 1 tag only

> (e1 pick_using '(e1 clash '(10 10 0)))
      -- e1 will now only possess tags of edges which
         contain the point '(10 10 0) i.e. 3 tags

> (v1 pick_using t)        -- v1 tag list will remain unchanged
                              as condition is always true
```

Assume a body b1 has been created with some toroidal faces. The example which follows shows how to find those faces which are toroidal and have a specific major radius.

```
> (define f1 face)
> (f1 pick_from b1)        -- f1 will contain all faces in b1
> (f1 pick_using '(eq (f1 enquire 'type) 'toroidal))
> (f1 pick_using '(eq (f1 enquire 'majrad ) 10.0 ))

> (f1 pick_using '(equal (f1 enquire 'point) '(0.0 0.0 0.0)))
    -- f1  will now only contain toroidal faces with a
       major radius of 10.0, and axis point at the origin

   NOTE:  >eq< only works for atoms >equal< for lists and atoms
```

The next example shows how the top face of a block can be picked without using the cursor, so that a new surface can be exchanged for this one using `tweak`.

```
> (define b1 p_block)
> (b1 x 10;y 10;z 10;create)
> (face help tweak)    -- for information on how to tweak a face
> (define f1 face)
> (f1 pick_from 'b1)   -- f1 now contains all 6 faces of b1
> (f1 pick_using '(f1 clash '(0 0 10)))
                       -- f1 is now the top face of b1

> (p_planar help create)     -- information
> (define s1 p_planar)
> (s1 point '(0 0 50); direction '(0 0 1); create)
                  -- s1 is plane at Z = 50 parallel to XY plane

> (f1 tweak 's1) -- this will raise the face up to the surface
                     s1, and now body b1 has the
                      dimensions x 10, y 10, z 50
```

If pick_using finds no items an error message is returned, e.g.

```
( error "f2; no match entity" )
```

and the existing list is left unaltered.

```
> ((define f0 face) pick_node 34 'b0)
> ((define f0 face) pick_node '(34 45 56) 'b0)
> ((define c0 curve) pick_node '(21 23 25) 'b0)
```

Using a list of tags to manipulate an object

Some KID functions result in the tag property of an entity having a LISP list of tags. An example of this is the function `pick_from`. It is often possible to manipulate an object with a list of tags using the same functions as if it had a single tag. This facility is not provided by all KID functions.

```
> (define b1 p_block)
> (b1 x 10; y 20; z 10; create)   -- create block b1
> (define f1 face)
> (f1 pick_from 'b1)           -- f1 is a list of b1's 6 faces
> (f1 pick_using '(f1 clash '(5 5 10)))
     -- leave only those faces which clash with this point,
        f1 now refers to a set of two faces from body b1

> (f1 enquire )                  --> information
> (f1 direction '(10 9 8); move)  -- this will move both faces
```

14.4 Picking vector points

A class `p_points` exists as a subclass of primitive. The class has a function "pick" which allows the user to give a list of vector points as its argument. These are held in the object's coordinate property. The user can either supply the list as an argument to pick, or can supply no argument and thus employ the cursor to select the vector points. Consider the following examples:

Object	Function
p_points	pick

```
> (define b0 p_points)
> (b0 help pick)          --> information
> (b0 pick '(0 0 0))      -- b0 is set to the origin
> (b0 coordinate)         --> (0 0 0)
> (define b0 p_points)
> (b0 pick '( (0 0 0) (1 1 1) (2 2 2) ) )  -- b0 is set to
> (b0 coordinate) --> ((0 0 0) (1 1 1) (2 2 2))
                     coordinate list
> (define b0 p_points)
> (b0 pick)
```

This then prompts the user to select the list of points by moving the cursor in the graphics frame with the arrow keys. Points are selected by moving to the desired location and striking any key, the list is terminated when a key is struck without the user having moved the cursor. The selected vector points are located on the graphics viewing plane. (The viewing plane is a plane perpendicular to the current view_direction passing through the current view_to point.)

No other functions are provided; `p_points` is intended only as a convenient location for storing a list of coordinates.

14.5 Picking an entity from an assembly

To pick from an assembly, first the assembly must be converted from a single layer assembly to a list of bodies (changes the type of entity from an assembly to a body).

```
> (a0 disassemble)    -- unpacks bodies from instances and
                         applies transforms; a0 is now a body
> (graphics ske a0)
> (define e0 edge)
> (e0 pick)  -- pick the edge from the body a0 using the cursor
```

assemble function

The inverse operation, to create the assembly, takes the part with one or more tags and makes each an instance in an assembly, which can then be transmitted as a single entity.

```
> (a0 tag)
> (100 174 279 1000)
> (a0 assemble)            -- a0 is now an assembly
> (a0 transmit "part_name")
```

Fault Reporting in KID 15

15.1 Introduction

One of KID's important functions is as a medium in which to report back any suspected faults in PARASOLID in a consistent and system (both hardware and software) independent manner and in a form which makes it easy for the fault to be investigated. If it is possible to demonstrate the absence or presence of the fault using a KID journal file, there is the additional benefit that the KID journal file can be incorporated into regression tests.

15.2 Fault types

In general, faults fall into two categories:

- Faults which generate an error message. These can normally be reproduced with a journal or KID file.
- Faults which generate a visual error, or no system error message. These require careful description and an accompanying picture, if appropriate, for diagnosis.

15.3 Fault isolation and simplification

It is of considerable help if the user can isolate the fault occurrence to the shortest possible KID journal file. In many cases it is possible to trim out irrelevant operations that are not implicated in the fault and this should be done. For example, if a fault is found in the process of building a large model, it may be possible to reproduce the fault with a much simplified version of the model.

- The journal file should not contain all the operations necessary to create the bodies involved in demonstrating the fault. These should be transmitted and the transmit files sent along with the KID journal file, which should contain appropriate receive statements:

```
    > (define b0 body)
    > (b0 receive ”fault_body.xmt_txt”)
```

- Frequent use of `(modeller mark)` aids debugging.
- It helps to understand a fault, and therefore enables a quicker fix to be produced, if the following is completed:
  - whenever possible simplify the fault
  - keep the KID journal file, that is required to reproduce the fault, as short as possible

KID Class Structure A

A.1 Introduction

This appendix gives the class tree of KID, and the functions which are available for each class. The functions are described in detail elsewhere. The names of all KID classes are reserved words within KID and should not be overwritten.

A.1.1 Modeller substructure

The class structure for KID, along with the class functions, appears on the following pages. The notation to be adopted is highest order class to the extreme left of the page, class property functions to the extreme right. Modeller is the root class of the tree shown.

Class	Function	Description
modeller	start	start modeller
	stop	stop modeller
	replicate	copy properties and item
	rename	rename object
	delete	undefine object, delete item if one
	enquire	user help on objects
	resabs	returns linear model resolution
	resang	returns angular model resolution
	mark	sets rollback mark
	roll	rolls back kernel to last mark set
	roll_class	rolling back live KID objects
option	bb	bulletin board use
	bb_user	user fields bulletin option
	bspline_io	switches output parametrisations from bezier to bspline
	bspline_geometry	sets B-curve/surface modeling to composite geometry
	check	local checking switch
	continuity_checking	continuity checking
	data_checking	consistency checks for ATTGEO
	enquire	information on option settings
	journal	file for kernel output
	logging	enables rollback
	parameter_checking	parameter checking
	self_checking	self intersection checking
	get_snapshot	binary/text reception of a snapshot
	save_snapshot	binary/text saving of a snapshot
	reset	sets all options to STAMOD defaults
	receive	binary/text reception
	transmit	binary/text transmission
	timing	controls timing information
	user_field	set user field length
	logging	set logging type

Class	Function	Description
	logging_number	set rollmark limit
	logging_forward	set forward logging
	raise errors	raise LISP error for Parasolid failures reported through tokens
entity	(see section entity sub-structure)	
primitive	(see section primitive sub-structure)	
xgraphics	(see section xgraphics sub-structure)	

A.1.2 Entity substructure

The entity sub-structure contains all the topological and geometric attributes which are used in the solid modeller. It contains everything from a single point to an assembly. Functions to modify or remove many of the features are also included in the entity sub-structure.

Class				Function	Description
entity				pick	picks items from screen
				pick_from	picks connected kernel entities
				pick_using	logical pick
				pick_node	uses node id to pick entity
				include	add entities to object
				remove	remove entities from object
	transformable			move	translate transformable items
				rotate	rotate transformable items
		transformation		apply	applies transformation to entity
		topology		sweep	sweep item to create new item
				swing	swing item to create new item
				clash	coordinate/topology clash test
				sketch	add sketch of items to selected view
				box	box (return box vector) item
				faces	faces of entity
				edges	edges of entity
				vertices	vertices of entity
				hidden	hidden line entity
				facet	facet entity

Class				Function	Description
				shade	shade entity
				silhouette	silhouette entity
				fillet	fillet all edges with given radius
				chamfer	chamfer all edges with given radius
				imprint	imprint tool on entity
				min_distance	closest approach to point or entity
		instance			
		part		transmit	transmit part to file
				receive	receive part from file
				state	part state
				key	key of loaded part
				remove_key	clear key from loaded part
				mass	compute mass of part
				identify	look up tag of named part
				assemble	creates assembly from parts
			assembly	disassemble	break down into bodies
				bodies	list first level bodies
				instances	list first level instances
				transforms	list first level transforms
				level	flattens assembly
			body	check	consistency check
				intersect	intersection of target/tool bodies
				merge	remove redundant edges vertices faces
				section	section body with surface
				subtract	subtract tool from target body
				unite	unite tool with target body
				blend_fix	fix unfixed blends in body
				regions	regions of body
				volume	volume of body
				cofg	centre of gravity of body
				area	surface area of body
				halve	section through cofg along axis

Class				Function	Description
				quarter	two sections
				reflect	reflect body in planar surface
				mirror	unite with mirror image
				magnify	scale body by factor
				offset	offset body distance
				hollow	hollow to given thickness
				sew	sew a collection of sheet bodies
		multiply	shell		
			feature		
			loo		loop (loop is LISP reserved word)
		single		merge	remove redundant faces,edges and vertices
				unfix	detach geometry from faces, edges or vertices
			region		
			vertex	fillet	blend at vertices with given radii
			face	check	consistency check
				delete_faces	delete face from body
				create_solid	create new body with copied faces
				create_sheet	create new sheet body from face
				remove_faces	remove face(s) and create new body
				move	move face(s)
				rotate	rotate face(s)
				taper	draft planar, cylindrical, conical face(s)
				tweak	modify face surface to given surface
				ntweak	modify face surface to reversed surface
				twefac	modify faces by given transforms
				fix	fit a surface to a face
				hatch_enq	enquire hatching attribute
				planar_hatch	get/set planar hatching
				radial_hatch	get/set radial hatching
				para_hatch	get/set parametric hatch
				cofg	centre of gravity of face
				area	surface area of face

Class				Function	Description
			edge	blend_check	check blends
				blend_enquire	blend information
				blend_remove	remove unfixed blends
				pick_blends	picks edges with unfixed blends
				length	arc length of edge
		geometry		check	consistency check
				make_body	create body from geometry
				part	part to which geometry is attached
			surface	intersect	intersects with supplied surface
				parameterise	parameters at position
				deparameterise	position at parameters
				faces	faces to which surface is attached
				uvbox	surface parameter uvbox
				nabx	non-aligned box
			curve	parameterise	parameter at position
				deparameterise	position at parameter
				march	list of points on the curve
				edges	edges to which curve is attached
				fin	fin to which curve is attached
				interval	curve parameter interval
				nabx	non-aligned box
			point	vertex	vertex to which point is attached
		associated	attribute	system_attribute	name
					blend_v5
					translucency
					reflectivity
					phull
					plines
					density
					hatching
					blend
					colour

A.1.3 Primitive substructure

The primitive sub-structure contains all the functions which are required to create anything in KID, from single points, surfaces, solid objects, through to assemblies.

The create functions create primitive solids, curves, surfaces, assemblies and instances.

Class						Function	Description
primitive							
	p_transformable						
	p_geometry		p_vector				
		p_surface	p_offset			create	
			p_spun			create	
			p_swept			create	
			p_toroidal			create	
			p_spherical			create	
			p_planar			create	
			p_cylindrical			create	
			p_conical			create	
		p_unbounded_curve	p_line			create	
			p_ellipse			create	
			p_circle			create	
			p_intersection			create	
			p_points			pick	picks point sequence in image plane
			p_point				
			p_bounded_curve			scribe	inscribes bounded curve on face
		p_transformation	p_general_transformation			create	
			p_translation			create	
			p_rotation			create	
			p_reflection			create	
			p_equal_scaling			create	
		p_topology	p_body	p_sheet		create	
				p_wire		create	
				p_profile		create	
				p_parasurf		create	

Class						Function	Description
				p_paracurve		create	
				p_pyramid		create	
				p_cylinder		create	
				p_sphere		create	
				p_torus		create	
				p_prism		create	
				p_cone		create	
				p_block		create	
				p_acorn		create	
			p_instance			create	
			p_assembly			create	
			p_feature				
	p_associated	p_attribute	p_system_attribute	p_blend		apply	apply blend to edge
						extract	extracts blend info from edge
					p_fillet		
					p_chamfer		

A.1.4 Graphics substructure

The class **graphics** is a subclass of **xgraphics**, so that if default values are changed in graphics they can be retrieved from xgraphics.

Class	Function	Description
xgraphics	autowindow	set window to minimum
	clear	clear current view
	invisible	switch for hidden lines
	redraw	redraw current view
	ar	autowindow and redraw
	axes	draw axes
	frame	draw frame
	enquire	graphics properties information
	select	select a view
	sketch	sketch all items in drawing list

Class	Function	Description
	blend	draw unfixed blends
	hidden	hidden line view
	silhouette	draw silhouette curves
	smooth	blanks smooth edges
	planar_hatch	planar hatching
	radial_hatch	radial hatching
	para_hatch	parametric surface hatching
	perspective	perspective view
	zoom	magnify by factor
	pan_down	view manipulation functions
	pan_up	
	pan_left	
	pan_right	
	view_direction	view direction
	view_from	eye point
	view_to	view point
	view_vertical	define vector as vertical in view
	view_window_xmax	functions to manipulate image extremes
	view_window_ymax	
	view_window_xmin	
	view_window_ymin	
	pick_window	
	pick_centre	
graphics		

Parasolid LISP Functions B

B.1 Introduction

This appendix gives a quick reference table summary of the Parasolid LISP functions available in KID.

B.2 Arithmetic operators

	integer	real	string	list	address	function	nil	n-ary
plus	*	*	*	*				*
difference	*	*	*	*				*
times	*	*	*	*				*
quotient	*	*	*	*				*
and				*			*	*
or				*			*	*
band	*							*
bor	*							*
eq	*	*	*	*	*	*	*	*
equal	*	*	*	*	*	*	*	*
greaterp	*	*	*		*	*	*	*
lessp	*	*	*		*	*	*	*

Also: add1, sub1

B.3 Environment

	new	overwrite
set	global	local/global
setq	global	local/global
defun	global	local/global
defproc	c	c
trace	trace	
untrace	trace	
oblist	global	

Association & property lists: assoc, iassoc, plist, get, put, remprop

B.4 Monadic operators

	integer	real	string	list	address	function	nil
atom	*	*	*	*	*	*	
null	*	*	*	*	*	*	*
not	*	*	*	*	*	*	*
bnot	*						
abs	*	*	*	*			*
minus	*	*	*	*			
truncate	*	*	*				
character	*	*					
chars	*	*	*		*	*	
ordinal	*	*	*		*	*	
explode	*	*	*		*	*	
implode				*			

Also: charp, subrp, fsubrp, listp, numberp, csubrp, cvarp, onep, zerop, minusp

B.5 List operators

	atom	list	nil	destructive	index
car/cdr		*			
cons	*		*		
list	*	*	*		
rplaca/d		*		*	
last/-cdr		*			
reverse		*			
append		*	*		
flatten		*			
element		*	*		*
member		*	*		*
delete		*	*		
subst		*	*		
replicate	*	*	*		

B.6 Evaluation

Function	Notes
errorset	
eval/apply	

Function	Notes
quote/’	
progn	n-ary body
cond	n-ary body: condition-(n-ary)expr. pairs
map/mapc	
loop	n-ary body
until/while	n-ary body executed when condition true/false
catch/throw	label, expr.

B.7 I/O operators

	filehandle	terminal	prlist	carr. return	punctuation
open	*				
close	*				
eof	*				
getchar	(*)	def			
readline	(*)	def			
read	(*)	def			
write0	*		*		*
write	*		*	*	
prin		*	*		*
princ		*	*		
print		*	*	*	*
printc		*	*	*	
vdu		*	*		
sprint	(*)	def	*	*	
error		*	*	*	

B.8 Time operators

	time&date	cpu-time	resettable
time/ctime/gctime		*	*
clock	*		
reset			

B.9 System functions

	info	set	exit	re-run	operating-system
help	*				
quit			*		
load				*	
messon/off	*	*			
edit/sed					
journal	*	*			
*****					*
!	escape character				
--	end-of-line comment, skip fold				
“..”	(quote ..)				

B.10 System variables

Special symbols: nil, undefined, t, lambda

	value	use
f	nil	
lpar/rpar	“(“ / “)”	
cr	newline	

Also: blank, period, dollar

	value	use
special		sprint: inhibit line breaks
decimalwidth		io, ..: significant digits for real numbers
linewidth		sprint: length of output line
whitespace		times: token-ise string

Error Codes in Parasolid LISP

C

Code	Description
1	too few arguments for system function
2	too many arguments for system function
3	identifier unsuitable for association list or environment
4	unable to open journal file
5	unsuitable value in linewidth or decimalwidth
6	dotted list terminated incorrectly
7	too few brackets in input file
8	identifier exceeds token length
9	compulsory formal parameter follows optional
10	too few arguments for expression
11	too many arguments for expression
12	argument not a list
13	argument not an atom
14	argument is null list
15	unsuitable argument type(s)
16	invalid argument value
17	divide by zero
18	missing or wrong file handle
19	no file open for handle
20	failed to open file
21	failed to close file
22	too many brackets in input file
23	unknown format
24	bad format in decode, encode or rplacv
25	bad pointer in decode
26	too few values to match format in encode or rplacv
27	value can not be coerced to format in encode or rplacv
28	pointer to bad or zero length explicit format
29	too many values to match format in encode or rplacv
30	badly encoded argument to c function
31	argument(s) to c function too long for system
32	c function return format unknown
33	while or until used outside loop
34	no suitable label for long jump

Code	Description
35	no more ‘drone’-functions of suitable return format
36	condition not a pair
37	no true condition in conditional
38	c variable not found
39	c function not found
40	user function not found
41	system function not implemented
42	user error function called
43	system error: please report

List of Parasolid LISP Functions

D

D.1 PARASOLID LISP functions

This is the current list of PARASOLID LISP functions; they are reserved words and must not be overwritten. Only a subset is of use to the KID user, and these are given with a full description or reference in the following section.

For each identifier listed here there is a header indicating whether it is the name of a function or a variable, and giving an indication of the arguments expected by the function. The words Subr, Fsubr and Expr are used to mean:

Subr:	A built-in function which processes its arguments normally
Fsubr:	A built-in function with special argument processing, e.g. it guarantees to process arguments from left to right, or it sometimes does not evaluate all of its arguments
Expr:	A function defined in LISP, not in C

The functions marked with an (*) are described (or additional explanation is supplied) in the following section, all others are described in the ACORNSOFT book “**LISP on the BBC Microcomputer**”.

In use, all of the functions described are used in the **lower case form**. The upper case form has been used to visually clarify the function names within the text.

Note: (help <f/subr>) gives more information on any function.

Function	Type
*ADD1	Expr
ADDRESS	Variable
AND	Fsubr
*ABS	Subr
APPEND	Subr
APPLY	Fsubr
ASSOC	Subr
ATOM	Subr
BACK	Variable
BAND	Fsubr
BLANK	Variable
BNOT	Subr

Function	Type
BOR	Fsubr
BYTE	Variable
CALL	Fsubr
CAR	Subr
CATCH	Fsubr
CDR	Subr
CHAR	Variable
CHARACTER	Subr
CHARP	Subr
CHARS	Subr
CLOCK	Subr
CLOSE	Subr
COND	Fsubr
CONS	Subr
CFSUBRP	Subr
CSUBRP	Subr
CR	Variable
CTIME	Subr
CVARP	Subr
DECIMALWIDTH	Variable
DECODE	Fsubr
*DEFUN	Fsubr
DEFPROC	Fsubr
DELETE	Subr
DIFFERENCE	Fsubr
*DIVIDE	
DOLLAR	Variable
DOUBLE	Variable
EDIT	Expr
ELEMENT	Subr
ENCODE	Fsubr
*ENTWINE	
EOF	Subr
EQ	Fsubr
*EQUAL	Fsubr
ERROR	Subr
ERRORSET	Fsubr
EVAL	Subr
EXPLODE	Subr
F	Variable

Function	Type
*FILTER	
FLATTEN	Subr
FLOAT	Variable
FSUBRP	Subr
FUNCTION	Variable
GCTIME	Subr
GET	Subr
GETCHAR	Subr
GREATERP	Fsubr
*HELP	Fsubr
IMPLODE	Subr
*INSERT	
INT	Variable
JOURNAL	Subr
LAMBDA	Special identifier
LAST	Subr
LASTCDR	Subr
LESSP	Fsubr
*LET	
LINEWIDTH	Variable
LIST	Subr
LISTP	Subr
*LOAD	Subr
LOGICAL	Variable
LOOP	Fsubr
LPAR	Variable
MAP	Expr
MAPC	Expr
MEMBER	Subr
MESSOFF	Subr
MESSON	Subr
MINUS	Subr
MINUSP	Expr
NIL	Special identifier
NOT	Subr
NULL	Subr
NUMBERP	Subr
OBLIST	Subr
ONEP	Expr
OPEN	Subr

Function	Type
OR	Fsubr
ORDINAL	Subr
PERIOD	Variable
PLIST	Subr
*PLUS	Fsubr
POINTER	Subr
PRIN	Subr
PRINC	Subr
PRINT	Subr
PRINTC	Subr
PROGN	Fsubr
PROMPT	Variable
PUT	Subr
*QUIT	Subr
QUOTE	Fsubr
QUOTIENT	Fsubr
READ	Subr
READLINE	Subr
RECLAIM	Subr
*REMAINDER	Expr
REMPROP	Subr
*REPLACE	
REPLICATE	Subr
RESET	Subr
REVERSE	Subr
RPAR	Variable
RPLACA	Subr
RPLACD	Subr
RPLACV	Fsubr
SED	Expr
*SELECT	
SET	Fsubr
SETQ	Fsubr
SHORT	Variable
SPECIAL	Variable
SPRINT	Subr
STRING	Variable
STRUCT	Subr/Variable
*SUB1	Expr
SUBRP	Subr

Function	Type
SUBST	Subr
T	Special identifier
THROW	Fsubr
TIME	Subr
TIMES	Fsubr
TRACE	Fsubr
TRUNC	Subr
UNDEFINED	Special identifier
UNION	Subr
UNTIL	Fsubr
UNTRACE	Fsubr
VDU	Subr
VOID	Variable
WHILE	Fsubr
WHITESPACE	Variable
WRITE	Subr
WRITEO	Subr
ZEROP	Expr
*	Subr
!	Special character
“ <string> ”	Special characters
--	Special characters
@	Special character

D.2 PARASOLID LISP function descriptions

ABS – Subr

```
(ABS x)
```

If x is numeric then the absolute value, |x|, of x is returned. If x is a string then the lower-case string is returned. If x is a list then ABS returns its length:

```
(ABS -4.5) = 4.5
(ABS ”Guten Morgen”) = ”guten morgen”
(ABS ’(a b c)) = 3
```

ADD1

This function adds 1 and is equivalent to:

```
( plus x 1 )
```

DEFUN – Fsubr

```
(DEFUN function-name parameters body ...)
```

DEFUN is a convenient way of defining functions. None of the arguments are evaluated. The use of DEFUN is exactly equivalent to

```
(SETQ function-name)
  '(LAMBDA parameters body ...))
```

The value returned by DEFUN is the name of the function that has been defined. The second argument (parameters) is a list of arguments and local variables that the function uses. Any number of actions can be given for the function to carry out.

Examples:

```
(DEFUN ADD2 (X) (PLUS X 2))
```

defines a function ADD2 by setting ADD2 to the value

```
(LAMBDA (X) (PLUS X 2)).
(DEFUN PR_ADD2 (X (Y)) (SETQ Y (PLUS X 2)) (PRINT Y) Y)
```

defines a function PR_ADD2 with local variable Y (initialized to NIL).

```
(DEFUN INCR (X (Y . 1)) (PLUS X Y))
```

defines a function INCR with optional parameter Y (default 1). Note: only constant values may be used as defaults.

```
(DEFUN ERR (MESS (SEV))
  (PRINTC ”error encountered”)
  (PRINTC MESS)
  (COND ((NULL SEV) NIL) (T (PRINTC ”severity: ” SEV))) )
```

defines a function ERR with an optional parameter.

```
(DEFUN MY_PRINT X
  (LOOP
  (WHILE X)
  (PRIN1 (EVAL (CAR X)))
  (SETQ X (CDR X)) ))
```

defines a function MY_PRINT which receives its arguments unevaluated and in a list.

DIVIDE

This function forces real division, for example:

```
( divide x y )
```

and is equivalent to:

```
( quotient ( plus x 0.0 ) y )
```

ENTWINE

This is a generalized function to join two lists assumed to be the same length together pairwise with any binary function. The binary function is optional and defaults to the list operator with two arguments.

General form:

```
(entwine LIST LIST <binary op> )
```

Example:

```
(entwine '(a b c) '(1 2 3))        =>  ((a 1) (b 2) (c 3))
(entwine '(a b c) '(1 2 3) 'cons)
                                =>  ((a . 1) (b  . 2) (c . 3))
(entwine '(a b c) '(1 2 3) 'plus)  =>  (a1 b2 c3)
```

EQUAL – Fsubr

```
(EQUAL exp exp ...)
```

The basic LISP function EQ compares two or more atoms for equality. When applied to list structures it checks if the pointers to them are identical. EQUAL compares list structures to see whether they have the same shape and the same atoms as leaves. EQUAL may have been defined as:

```
(DEFUN EQUAL (A B) (COND
  ((EQ A B) T)
  ((OR (ATOM A) (ATOM B)) NIL)
  ((EQUAL (CAR A) (CAR B)) (EQUAL (CDR A) (CDR B)))
  (T NIL)))
```

FILTER

This function applies a selective filter to a list identifying those to keep by element number. Positive indices count from the front, negative indices from the end.

Example:

```
( filter '( 1 3 5 7 ) '( a b c d e f ) ) => ( a c e nil )
( filter '( -1 -3 -5 -7 ) '( a b c d e f ) ) => ( f d b nil )
( filter 5 '( a b c d e f ) ) => e
( filter -5 '( a b c d e f ) ) => b
( filter 99 '( a b c d e f ) ) => nil
( filter -99 '( a b c d e f ) ) => nil
```

HELP – Fsubr

```
(HELP), (HELP item), (HELP exp),
(HELP item/expression item/expression)
```

HELP provides information on the state of the system, commands,the various environments maintained (global, local, c, io and trace) and the values of functions and variables. Without argument HELP catalogues all items for which it can provide information. A second item or expression specifies the property in a property list. Items may contain wildcard characters.

Examples:

```
(HELP cons)          --- get information on system function cons
(HELP con*)           --- list identifiers, functions
                          beginning with 'con'
(HELP fred)           --- get environment and value of fred
(HELP fred *)         ---  list all properties of fred
(HELP *create)        --- list all identifiers containing
                          create property
(HELP (eval 'handle)) --- get information on file handle
```

INSERT

This a simple function which inserts the specified value into the list m as the nth element. The abs of m is increased by one. It is assumed that:

```
1 <= n <= ( ( abs m ) + 1 )
```

Example:

```
( insert 1 '( a b c d ) 'h ) => ( h a b c d )
( insert 5 '( a b c d ) 'h ) => ( a b c d h )
( insert 3 '( a b c d ) '( h i ) ) => ( a b  ( h i ) c d )
```

LET

LET takes any number of arguments. The first is interpreted as a list of variables and initializations, the rest as forms to be evaluated. The variables list contains atoms or lists of two items. Atoms are variable names to be initialized to nil. The first element of a list is a variable name, the second its initialization value which is evaluated.

Example:

```
( let (a (b 5) (c (plus 3 5) ) )
  ( printc "a "a "b "b "c "c ))
  "a nil b 5 c 8"
```

Note: Initializations in LET are in “in parallel”, so later initializations cannot make use of earlier ones.

If this functionality is required, use the function LET*. LET* is identical to LET except for allowing later initializations to make use of earlier ones.

LOAD – Subr

```
(LOAD filename [mode] [handler])
```

The argument to LOAD should be the name of a file. LOAD reads the file and evaluates the LISP expressions in it. The value of LOAD is the value of the last expression in the file or UNDEFINED if an unexpected end of file was encountered. If the file is not found an attempt is made to open a system file of the same name. The second argument is optional and when provided may be one of:

Argument	Description
REFLECT:	print out all S expressions read
VERIFY:	print out expressions read and evaluated
REPLAY:	reroute standard input to load file for EOF, GETCHAR, READ, READLINE

A third, optional argument is an error handler to be executed if an error occurs during loading. if the handler returns NIL then loading of the file is abandoned, which is also the default for LOAD.

Examples:

```
(DEFUN ignore () (PRINTC ”ignoring loading error”) T)
(LOAD ’fred ignore)
```

```
(LOAD ’fred T)
```

If no file extension is given, then an extension of the type .lsp is assumed.

PACK

Pack groups elements of a list:

Examples:

```
(pack 3 ’(0 0 0 0 0 1 0 0 2 0 0 3 0 0 4))
    --> ( ( 0 0 0 ) ( 0 0 1 ) ( 0 0 2 ) ( 0 0 3 ) ( 0 0 4 ) )
```

If there are not enough elements available, the last element of the packed list is shorter:

```
(pack 3 ’(1 2 3 4)) --> ( ( 1 2 3 ) ( 4 ) )
```

PLUS – Fsubr

```
(PLUS number number ...)
```

PLUS returns the sum of all its arguments. PLUS can have any number of arguments. If one of the arguments is a string, then the result is the string concatenation of all arguments. The operator handles lists in a manner similar to DIFFERENCE.

Examples:

```
(PLUS 6 2 -3.1) = 4.9
(PLUS ’hello blank ’dolly) = ”hello dolly”
(PLUS ’(1 2 3) ’5) = ’(6 7 8)
(PLUS ’((1 2 3) (4 5 6)) ’(2 4)) = ’((3 4 5) (8 9 10))
```

See DIFFERENCE, MINUS and TIMES.

QUIT – Subr

```
(QUIT)
```

Leaves the Lisp interpreter and closes the journal file for the session. Note that an end-of-file encountered while reading from the standard input device has the same effect as QUIT.

REMAINDER

Remainder has been overloaded to work on lists.

If two lists are passed as arguments then those elements which are common to both p and q are removed from p. The resultant p is returned.

REPLACE

This is a simple function which replaces the nth element of the list m with the specified value. The abs of m is unchanged. It is assumed that:

```
1 <= n <= ( abs m )
```

Example:

```
( replace 1 ’( a b c d ) ’h ) => ( h b c d )
( replace 4 ’( a b c d ) ’h ) => ( a b c h )
( replace 3 ’( a b c d ) ’( hi ) ) => ( a b ( h i ) d )
( replace 2 ’(( a b c ) ( d e f ) ( g h i )) ’( j k ))
                       => (( a b c ) ( j k ) ( g h i ))
```

SELECT

This is a general function to select the first or last n elements from a list. If n is negative the last n elements are selected. If the first argument is a list of integers then the selection proceeds by grouping subsequent elements of the data list in order.

Example:

```
( select 3 '( a b c d e f ) ) => ( a b c )
( select -3 '( a b c d e f ) ) => ( d e f )
( select '( 3 3 ) '( a b c d e f ) ) => ( ( a b c ) ( d e f ) )
( select '( 1 2 3 ) '( a b c d e f ) )
                                => ( ( a ) ( b c ) ( d e f ) )
( select '( -1 -2 -3 ) '( a b c d e f ) )
                                 => ( ( f ) ( d e ) ( a b c ) )
```

SUB1

This function subtracts 1 and is equivalent to:

```
( difference x 1 )
```

"<string>" special character

Any string contained in double quotes is turned into a quoted single identifier. Within double quotes, spaces and punctuation characters (with the exception of double quotes) do not have to be preceded by the escape character, `!`.

Example:

```
"123" = '123
"temp.dat" = 'temp!.dat
"zum Beispiel: " = 'zum! Beispiel!:!
```

-- special character

The double hyphen, `--`, introduces a comment which is terminated by the newline character.

KID Examples E

E.1 Introduction

These examples are intended to demonstrate a range of modeling activities using KID.

E.2 Example 1

Create a simple parametric curve wire.

```
(modeller start)
(undefine pcurve1)
(define pcurve1 p_paracurve)
(pcurve1 help create)
(pcurve1 dim 4;
         ord 4;
         nseg 1;
         verts'(0   0 0 1
                1   1 0 2
                3  -1 0 2
                4   0 0 1);
         create)
(modeller stop)
```

E.3 Example 2

A cube and a parametric sheet body are created. The top planar face of the cube is selected and tweaked to the parametric surface of the sheet using `ntweak` as the surface normal must be reversed.

```
(modeller start)
(undefine blatt_4 f1 blo s1 tf3)
-- create a parametric sheet body

(define blatt_4 p_parasurf)
(blatt_4 dim 3;
         uord 3;
         vord 4;
         nuseg 1;
         nvseg 1;
         verts '(0.0 20.0  0.0
                  0.0 30.0 10.0
                  0.0 40.0  0.0
                 10.0 20.0  0.0
                 10.0 30.0 10.0
                 10.0 40.0  0.0

                 20.0 20.0  0.0
                 20.0 30.0 10.0
                 20.0 40.0  0.0

                 30.0 20.0  0.0
                 30.0 30.0 10.0
                 30.0 40.0  0.0);
         create)
-- select the parametric face from the sheet body

(define f1 face)
(define s1 surface)
(f1 pick_from 'blatt_4)
(f1 pick_using '(eq (f1 enquire 'type) "B-surface"))
-- create a B-surface from the parametric face of the sheet

(s1 pick_from 'f1)
(s1 enquire)
-- create a copy of the surface to avoid trying to share
   across two bodies

( (define s2 surface) replicate 's1)
-- create a cube, centred under the parametric sheet

(define blo p_block)
(blo x 5; y 5 ; z 5; point '(15 30 -15); create)
(define tf3 face)
-- select the top face of blo

(tf3 pick_from 'blo)
(tf3 pick_using '(tf3 clash '(15 30 -10)))
(modeller mark)
-- tweak top face of cube to the reverse of s1

(tf3 ntweak 's2)
(modeller stop)
```

E.4 Example 3

Two identical cylinders are united and the intersection edges are blended using a rolling ball blend.

```
(modeller start)
(undefine c1 c2 e1 b1)
(define c1 p_cylinder)
(define c2 p_cylinder)
(c1 help create)
(c1 radius 10; height 80; point '(-40 0 0); direction '(1 0 0);
create)
(c2 radius 10; height 80; point '(0 -40 0); direction '(0 1 0);
create)
(c1 unite 'c2)
(define e1 edge)

-- collect all edges from the resulting body

(e1 pick_from 'c1)

-- only those which are elliptical

(e1 pick_using '(eq (e1 enquire 'type) 'ellipse))
-- make rolling ball blend

(define b1 p_fillet)
(b1 help apply)
(b1 r1 2)
-- apply blend to edges and fix

(b1 apply 'e1)
-- make sure it's ok

(e1 blend_check)
(c1 blend_fix)
(graphics open_device 'xwindow)
(graphics sketch 'c1)
(graphics autowindow; clear; hidden)
(modeller stop)
```

E.5 Example 4

This next example can be used to demonstrate the differing results of varying combinations of arguments for the local operations `crsofa` and `rmfaso`.

```
(modeller start)

-- create cube and two spheres

(undefine s1 s2 b0 fred)
(define b0 p_block)
(b0 x 10; y 10; z 10; create)
(define s1 p_sphere)
(define s2 p_sphere)
(s1 radius 2; point '(5 5 10); create)
(s2 radius 2; point '(0 0 10); create)
-- unite cube and two spheres, renaming the resulting body

(b0 unite 's1)
(b0 unite 's2)
(define fred body)
(fred tag (unite_temp tag))
(graphics open_device 'xwindow)
(graphics hidden 'fred)
-- pick the two spherical faces into f1

(undefine f1)
(define f1 face)
(f1 pick_from 'fred)
(f1 pick_using '(eq( f1 enquire 'type) 'spherical))
-- set modeller roll point

(modeller mark)
-- copy the two spherical surfaces and create two new bodies
   using the argument 'grow', leaving the parent body
   unaltered.

(f1 crsofa 'grow)
(graphics clear)
(graphics sketch 'fred)
(graphics clear)
(graphics hidden '( crsofa_c1 crsofa_c2))
-- rollback to last set mark and use rmfaso to remove the two
   spherical surfaces and create new bodies.

(modeller roll)
(modeller mark)
(f1 rmfaso 'grow 'growp)
(graphics clear)
(graphics hidden 'rmfaso_p1 )
(graphics clear)
(graphics hidden '(rmfaso_c1 rmfaso_c2))
-- rollback for another combination

(modeller roll)
(modeller mark)
(f1 rmfaso 'cap)
(modeller stop)
```

E.6 Example 5

A cube is created, and two lines scribed onto the top face creating 3 new faces in place of the original. Two of these new faces are swept in opposite directions, changing the geometry and topology of the cube.

Note: This example produces an invalid body.

```
(modeller start)
(undefine b0 plin1 plin2 f1 f2 pbc1 pbc2)
(define b0 p_block)
(b0 x 20; y 20; z 20; create)
(modeller mark)
-- the next sections scribe two bounded curves onto the top
   face of the cube, then sweep two of the newly created faces.

(define plin1 p_line)
(plin1 point '(0 0 20); direction '(1 0 0); create)
(define pbc1 p_bounded_curve)
(define f1 face)
(f1 pick_from 'b0)
(f1 pick_using '(f1 clash '(0 0 20)))
(pbc1 startp '(-10 0 20); endp '(10 0 20))
(pbc1 face 'f1; curve 'plin1; scribe )
(define plin2 p_line )
(plin2 point '(0 0 20); direction '(0 -1 0); create)
(define pbc2 p_bounded_curve)
(define f2 face)
(f2 pick_from 'b0)
(f2 pick_using '(f2 clash '(0 -5 20)))
(pbc2 startp '(0 0 20); endp '(0 -10 20))
(pbc2 face 'f2; curve 'plin2; scribe )
-- use f1 and f2 to define the faces to be swept

(f1 pick_from 'b0)
(f2 replicate 'f1)
(f1 pick_using '(f1 clash '(0 5 20)))
(f2 pick_using '(f2 clash '(5 -5 20)))
(f1 sweep '(0 0 10))                    --> valid
(f2 sweep '(0 0 -10))                   --> self_intersecting
(graphics open_device 'xwindow)
(graphics sketch 'b0)
(b0 check)                              --> invalid body
(modeller roll)
(modeller stop)
```

Machine Dependency in KID F

F.1 Introduction

Very few KID commands are device dependent. The ones described are functions of the graphics class.

F.2 open_device and close_device

In general it is not necessary to change the default settings for a particular graphics device. If these defaults are to be changed, the function `enquire` gives information on these settings. The graphics functions `open_device` and `close_device` are used with an argument, to initialise or change the device type.

Object	Function
graphics	open_device, close_device

```
> ( graphics enquire )           --> information
> ( graphics open_device ’x )
                        -- initialise device setting to Xwindow
> ( graphics close_device ’x ) --> unset current device setting
```

The available devices are: cifer, new_cifer, plotter, laser, vt240_regis, vt240_tek, postscript, Xwindow, Xcolor, Xcolour, nt, ntcolor, ntcolour, interleaf, framemaker and null.

A null device is sometimes useful to ensure that a device is open even if the output is not to be viewed.

The shade function is only available with X devices, not on NT.

F.3 Which key for pick?

The `pick` command belonging to the `entity` and `p_points` classes enables the cursor when used without argument. To pick an object or screen coordinate pressing any key achieves the pick.

```
> ( define p1 p_points )
> ( define b0 p_cone )
> ( define f1 face )
> ( b0 lrad 10; urad 0; height 30; create )
> ( graphics sketch ’b0 )
> ( f1 pick )      -- cursor enabled for face pick
> ( p1 pick )
           -- cursor enabled for image plane coordinate pick(s)
```

F.4 KID interrupts

KID contains an error handler which allows the user to interrupt the execution of a KID or FLICK command without exiting from the KID session. On most machines this responds to ctrl-C.

	user-interrupt	exit KID	kill KID
NT	ctrl-C	ctrl-Z	-
UNIX	ctrl-C	ctrl-D	-

Index