Chapter 28.
Classes Ba thru Bz

Topic: Ignore

BinaryFile

Class: SAT Save and Restore

Purpose: Defines the BinaryFile class for doing ACIS save and restore to binary files.

Derivation: BinaryFile : FileInterface : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernutil/fileio/binfile.hxx

Description: This is an abstract base class. It implements most of the virtual methods which are used by all of the binary file formats.

If there is a need to save and restore ACIS ENTITY data in binary form to a target other than a FILE*, then it is advisable to derive a new class from this one rather than directly from FileInterface.

Limitations: None

References: None

Data:

```
protected int read_long_size;
```
Size of the long.

```
protected int write_long_size;
```
Output size.

```
protected logical big_end;
```
Big/little–endian flag.

```
protected logical need_swap;
```
Is byte swapping needed.

Constructor:

```
public: BinaryFile::BinaryFile ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Destructor:

```
public: virtual BinaryFile::~BinaryFile ();
```

C++ destructor for BinaryFile which deallocates memory.

Methods:

```
protected: virtual TaggedData::DataType
    BinaryFile::read_data_type ();
```

Read a data type. The return type is TaggedData::DataType. The read_data_type method is a virtual method of BinaryFile.

```
protected: virtual TaggedData::DataType
    BinaryFile::read_type ();
```

Reads the data type.

```
protected: virtual TaggedData::DataType
    BinaryFile::test_type (
    TaggedData::DataType     // data type
        type_wanted,         // type wanted
    int error_num            // error number
        = 0                  //
    );
```

Reads the next data type tag, and checks to see if it is the required type. If it is not the required type, it signals an error.

```
public: virtual FilePosition BinaryFile::goto_mark (
    FilePosition             // new file position
    ) = 0;
```

This method repositions the file pointer and must be implemented for each class derived from BinaryFile. In a normal save that does not require writing the ENTITY count to the ACIS header, this is only used to reposition the file pointer if there is an error reading the header.

```
protected: virtual size_t BinaryFile::read (
    void* buffer,           // buffer name
    size_t length,          // length
    logical swap            // support byte swapping?
    ) = 0;
```

This method reads the data and must be implemented for all derived classes.

```
protected: logical BinaryFile::read_an_int (
    int long_size,              // long size
    int& retval                 // return value
);
```

Read the integer.

```
protected: virtual size_t BinaryFile::read_and_test (
    void* data,             // data
    size_t num_bytes,       // number of bytes
    logical swap            // support byte swapping?
    );
```

Read in a given number of bytes of data and signal a sys_error if not enough data was read.

```
public: virtual char BinaryFile::read_char ();
```

Read a character.

```
public: virtual TaggedData* BinaryFile::read_data ();
```

Read the data type and the subsequent datum of that type.

```
public: virtual double BinaryFile::read_double ();
```

Read a double.

```
public: virtual int BinaryFile::read_enum (
    enum_table const&       // table
    );
```

Reads in the enumeration table.

```
public: virtual float BinaryFile::read_float ();
```

Reads a float.

```
public: virtual logical BinaryFile::read_header (
    int&,                   // first integer
    int&,                   // second integer
    int&,                   // third integer
    int&                    // fourth integer
    );
```

Reads a header string.

```
public: virtual int BinaryFile::read_id (
    char*,                  // buffer to read from
    int                     // buffer size or –1 for
                            // no limit
        = 0
    );
```

Reads an identifier.

```
public: virtual logical BinaryFile::read_logical (
    const char* f           // character string that
        = ″F″,              // requests FALSE
    const char* t           // character string that
        = ″T″               // requests TRUE
    );
```

Reads the logical.

```
public: virtual long BinaryFile::read_long ();
```

Reads the long.

```
public: virtual void* BinaryFile::read_pointer ();
```

Reads the pointer.

```
public: virtual SPAposition
    BinaryFile::read_position ();
```

Reads the position.

```
public: virtual short BinaryFile::read_short ();
```

Reads the short.

```
public: virtual char* BinaryFile::read_string (
    int&                    // length
    );
```

Reads the string.

```
public: virtual size_t BinaryFile::read_string (
    char*,                  // title
    size_t maxlen           // maximum length
        = 0
    );
```

Reads the string.

```
public: virtual
    size_t BinaryFile::read_string_length (
    TaggedData::DataType    // data type
    );
```

Reads the string length.

```
public: virtual logical
    BinaryFile::read_subtype_end ();
```

Reads the subtype end.

```
public: virtual logical
    BinaryFile::read_subtype_start ();
```

Reads subtype start.

```
protected: virtual TaggedData::DataType
    BinaryFile::read_type ();
```

Reads the data type.

```
public: virtual SPAvector BinaryFile::read_vector ();
```

Reads the vector.

```
protected: virtual long
    BinaryFile::safe_read_long ();
```

The long value.

```
protected: virtual void
    BinaryFile::safe_write_long (
    long                        // long
    );
```

Used to convert between 32 and 64 bit formats.

```
protected: virtual void
BinaryFile::safe_write_long_tagged (
    TaggedData::DataType,       // data type
    long                        // long
    );
```

Used to convert between 32 and 64 bit formats.

```
public: virtual FilePosition
    BinaryFile::set_mark () = 0;
```

This method repositions the file pointer and must be implemented for each class derived from BinaryFile. In a normal save that does not require writing the ENTITY count to the ACIS header, this is only used to reposition the file pointer if there is an error reading the header.

```
protected: virtual void BinaryFile::write (
    const void* data,       // data
    size_t len,             // length
    logical swap            // support byte swapping?
    ) = 0;
```

This method writes the data and must be implemented for all derived classes.

```
public: virtual void BinaryFile::write_char (
    char                    // character to write
    );
```

Writes the character.

```
public: virtual void BinaryFile::write_double (
    double                  // double to be written
    );
```

Writes the double.

```
public: virtual void BinaryFile::write_enum (
    int,                    // value to be written
    enum_table const&       // enumeration table
    );
```

Writes value to enumeration table.

```
public: virtual void BinaryFile::write_float (
    float                   // float to be written
    );
```

Writes the float.

```
public: virtual void BinaryFile::write_header (
    int,                    // first integer
    int,                    // second integer
    int,                    // third integer
    int                     // fourth integer
    );
```

Writes the header.

```
public: virtual void BinaryFile::write_id (
    const char*,            // entity identifier
    int                     // integer
    );
```

Writes an entity identifier.

```
public: virtual void
    BinaryFile::write_literal_string (
    const char*,            // string to be written
    size_t len              // length
        = 0
    );
```

Writes the literal string.

```
public: virtual void BinaryFile::write_logical (
    logical,                // logical to be written
    const char* f           // represents FALSE
        = "F",
    const char* t           // represents TRUE
        = "T"
    );
```

Writes a logical.

```
public: virtual void BinaryFile::write_long (
    long                    // long to be written
    );
```

Writes a long.

```
public: virtual void BinaryFile::write_pointer (
    void*                   // pointer to be written
    );
```

Writes a pointer.

```
public: virtual void BinaryFile::write_position (
    const SPAposition&      // position to be written
    );
```

Writes a position.

```
public: virtual void BinaryFile::write_short (
    short                   // short to be written
    );
```

Writes a short.

```
public: virtual void BinaryFile::write_string (
    const char*,            // string to be written
    size_t len              // length
        = 0
    );
```

Writes a string.

```
public: virtual void
    BinaryFile::write_subtype_end ();
```

Writes a subtype end.

```
public: virtual void
    BinaryFile::write_subtype_start ();
```

Writes a subtype start.

```
protected: virtual void BinaryFile::write_tagged (
    TaggedData::DataType tp,// data type
    const void* data,       // data
    size_t size,            // size
    logical swap            // support byte swapping?
    );
```

Writes tagged data.

```
public: virtual void BinaryFile::write_terminator ();
```

Writes a terminator.

```
public: virtual void BinaryFile::write_vector (
    const SPAvector&        // vector to be written
    );
```

Writes a vector.

Related Fncs:

None

blend_spl_sur

Class: Blending, SAT Save and Restore

Purpose: Provides common functionality and data for all blend surfaces.

Derivation: blend_spl_sur : spl_sur : subtrans_object : subtype_object : ACIS_OBJECT : –

SAT Identifier: blend_spl_sur

Filename: kern/kernel/kerngeom/splsur/blnd_spl.hxx

Description: This is an abstract class that tries to predict some of the fields that derived classes will need; for example, it contains pointers for a left surface, a left curve and a left point although in practice only one of these will be needed in a particular derived class. The reason for doing this is that the base class can (probably) completely handle the administrative functions such as operator=, save and restore, making these trivial for the derived classes.

Limitations: None

References:

KERN	blend_section, blend_support, curve, var_cross_section, var_radius
by KERN	blend_support
BASE	SPAinterval

Data:

`protected BOUNDED_CURVE* _def_bcu;`
Storage for the bounded curve.

`protected CVEC* _def_cvec;`
Storage for the CVEC.

`protected blend_section* _section_data;`
New blend section data.

`protected double initial_fitol;`
The fit tolerance that was requested when the approx sf was first made. When we extend it, we carry on to this same fitol.

`protected int initial_num_u_pts;`
The number of *u* points sampled in fitting the approx sf. When we extend it, we must continue sampling exactly the same points.

`protected SPAinterval _support_u_param_range;`
The *u*-parameters of the supports. The *u*-parameter should just be [0,1], but we'll make it variable, in order to keep things such as shift_u consistent.

`public blend_support *left_support;`
A support entity. May be a surface, a curve or a point. Have to be a pointer, to allow classes derived from blend_support to be used.

`public blend_support *right_support;`
A support entity. May be a surface, a curve or a point. Have to be a pointer, to allow classes derived from blend_support to be used.

`public curve *def_curve;`
Defining curve (reference curve). For a rolling ball blend, this is the blend spine, i.e., the path of the center of the ball. def_cvec is a cvec on the def_curve which is set by each call to evaluate, and may subsequently be used by the application.

`public double left_offset;`
Objects describing the radius. If the radius is constant then the value of the double is used and the var_radius pointer will be zero. Otherwise, the value of the double will be ignored. The variable radius objects are pointers, so that if the left and right radii are not different, then right_rad will point to the same object as left_rad. Also, rad equals left_rad, always, for convenience.

`public double right_offset;`
Objects describing the radius. If the radius is constant then the value of the double is used and the var_radius pointer will be zero. Otherwise, the value of the double will be ignored. The variable radius objects are pointers, so that if the left and right radii are not different, then right_rad will point to the same object as left_rad. Also, rad equals left_rad, always, for convenience.

`public SPAinterval legal_range;`
The “legal” *v* range is that *v* range over which the surface is well–behaved. This is initialized to an infinite interval, but if self-intersecting regions of surface are discovered, this range bounds the surface away from them. When infinite, this means the surface is legal in that direction as far as has been analyzed. Semi–infinite will be quite common.

`public logical approximation_not_reqd;`
Flag to determine whether an approximation is required.

`public logical left_handed;`
Flag to indicate the handedness.

`public logical supports_extended;`
Flag to indicate whether the supports are extended.

`public var_cross_section* section`;
Object describing the cross section. If this is zero then the cross section is assumed to be circular, or elliptical if the radius functions are not equal.

`public var_radius *left_rad;`
Objects describing the radius. If the radius is constant then the value of the double is used and the var_radius pointer will be zero. Otherwise, the value of the double will be ignored. The variable radius objects are pointers, so that if the left and right radii are not different, then right_rad will point to the same object as left_rad. Also, rad equals left_rad, always, for convenience.

`public var_radius *rad;`
Objects describing the radius. If the radius is constant then the value of the double is used and the var_radius pointer will be zero. Otherwise, the value of the double will be ignored. The variable radius objects are pointers, so that if the left and right radii are not different, then right_rad will point to the same object as left_rad. Also, rad equals left_rad, always, for convenience.

`public var_radius *right_rad;`
Objects describing the radius. If the radius is constant then the value of the double is used and the var_radius pointer will be zero. Otherwise, the value of the double will be ignored. The variable radius objects are pointers, so that if the left and right radii are not different, then right_rad will point to the same object as left_rad. Also, rad equals left_rad, always, for convenience.

Constructor:

```
public: blend_spl_sur::blend_spl_sur (
    blend_support* left_support,    // blend support
                                    // for left side
    blend_support* right_support,   // blend support
                                    // for right side
    const curve& def_crv,           // defining curve
    SPAinterval v_range,            // v param range
    double left_offset,             // left offset
    double right_offset,            // right offset
    var_radius* radius1,            // left radius
    var_radius* radius2             // rt rad if diff
        = NULL,                     // from lt
    const var_cross_section* x_sect // cross section,
        = NULL,                     // if not
                                    // circular
    closed_forms u_closure          // u closure
        = OPEN,
    closed_forms v_closure          // v closure
        = CLOSURE_UNSET
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Only certain combinations of input are valid, but this is not enforced by the constructor. The reason for this is that the derived classes are anticipated and have constructors that ensure only valid combinations are passed through to this constructor. For example, the derived classes are expected to make the blend support entities (on the heap) and pass them to this constructor.

The constructor copies the reference curve (which is passed by reference), but assumes ownership of the data that is passed to it by pointer – namely the blend_supports, radius functions and cross sections.

```
public: blend_spl_sur::blend_spl_sur ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: blend_spl_sur::blend_spl_sur (
    const blend_spl_sur&    //blend spl sur
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

Destructor:

```
public: virtual blend_spl_sur::~blend_spl_sur ();
```

C++ destructor, deleting a blend_spl_sur.

Methods:

```
public: virtual int blend_spl_sur::accurate_derivs (
    SPApar_box const&           // parameter box
        = * (SPApar_box* ) NULL_REF
    ) const =0;
```

Returns the number of derivatives which evaluate can find accurately (and fairly directly), rather than by finite differencing, over the given portion of the surface. If there is no limit to the number of accurate derivatives, returns the value ALL_SURFACE_DERIVATIVES, which is large enough to be more than anyone could reasonably want.

```
public: virtual void blend_spl_sur::append_u (
    spl_sur&                // spline surface
    );
```

Concatenate the contents of two surfaces into one. The surfaces are guaranteed to be the same base or derived type, and to have contiguous parameter ranges (“this” is the beginning part of the combined surface (i.e., lower parameter values), the argument gives the end part).

```
public: virtual void blend_spl_sur::append_v (
    spl_sur&                // spline surface
    );
```

Concatenate the contents of two surfaces into one. The surfaces are guaranteed to be the same base or derived type, and to have contiguous parameter ranges (“this” is the beginning part of the combined surface (i.e., lower parameter values), the argument gives the end part).

```
public: virtual double blend_spl_sur::blend_angle (
    SPAunit_vector& Tan,        // tangent direction
    SPAvector const& R0,        // first vectors
    SPAvector const& R1,        // second vectors
    double& rr_sina             // sine of angle
        = * (double* ) NULL_REF,// between vectors
    double& rr_cosa             // cosine of angle
        = * (double* ) NULL_REF // between vectors
    ) const = 0;
```

Find the angle between two vectors, in a plane determined by the tangent vector in the given CVEC.

```
public: virtual double
    blend_spl_sur::blend_total_angle (
    SPAposition& P,             // position
    SPAunit_vector& Tan,        // tangent direction
    SPAvector const& R0,        // first vectors
    SPAvector const& R1,        // second vectors
    double& rr_sina             // sine of angle
        = * (double* ) NULL_REF,// between vectors
    double& rr_cosa             // cosine of angle
        = * (double* ) NULL_REF // between vectors
    ) const = 0;
```

Find the angle between two vectors, in a plane determined by the tangent vector in the given CVEC.

```
public: logical blend_spl_sur::check_cache (
    double v,                // v parameter
    int spine_nder,          // number of required
                             // spine derivs
    int def_nder,            // number of required
                             // spine derivs
    int spring_nder,         // number of required
                             // spring derivs
    logical xcrv_norm,       // whether to fill in
                             // xcurve normal
    blend_section& section,  // all output in here
    int side                 // evaluation side
    ) const;
```

Method for handling cache data.

```
public: void blend_spl_sur::check_safe_range (
    int which_end           // which end to look at
        = 0                 // default both ends
    );
```

Checks for bad singularities at the ends and sets the legal range such that it avoids them. The argument indicates which end to look at. 0 is for both ends (the default). A negative number represents the low end and a positive the high end.

It checks whether or not a safe_range has been applied to the defining curve. If so, it checks for the case in which the angle between the vectors from the spine point to the contact points is 180 degrees, i.e. the contact points and the spine point are collinear, with the spine point in the middle. The related degeneracy, where the angle is zero and the contact points coincide (tangent surfaces), is not a “bad” singularity and doesn’t hurt anything.

If the safe range limits a bad singularity, then the end of the safe range are before the singularity actually happens, so it checks for “close” to a bad singularity at the end.

```
public: virtual void blend_spl_sur::compute_section (
    double v,               // v parameter
    int spine_nder,         // number of required
                            // spine derivs
    int spring_nder,        // number of required
                            // spring derivs
    logical xcrv_norm,      // whether to fill in
                            // xcurve normal
    blend_section& section, // all output in here
    int                     // the evaluation
        = 0                 // location – 1 => above,
                            // –1 => below,
                            // 0 => don’t care
    ) const = 0;
```

A form of evaluation specific to blend_spl_surs (certain numerical algorithms used by blending need this function). Evaluates spine, defining curve, contact points and their derivatives at the given *v*-parameter, according to the blend_section class declaration as above. We may specify exactly how may spine and spring curve derivatives we require. As the two are typically connected you may get more than you asked for, but you are guaranteed to get at least what you ask for. Implementations of this should also ensure it does no more than is necessary. Finally the logical flag indicates whether you require the cross curve normal filled in; again this may (will) have implications on the amount of other stuff you get back, but if passed as TRUE then this is guaranteed to be returned. Note that calling this with for example –1, –1 and TRUE is valid.

```
public: virtual subtrans_object*
    blend_spl_sur::copy () const = 0;
```

Construct a duplicate in free store of this object but with zero use count.

```
public: virtual void blend_spl_sur::debug (
    char const*,            // character
    logical,                // integer
    FILE*                   // file
    ) const;
```

Debug printout. The virtual function debug prints a class-specific identifying line, then calls the ordinary function debug_data to put out the details. It is done this way so that a derived class' debug_data can call its parent's version first, to put out the common data. Indeed, if the derived class has no additional data it need not define its own version of debug_data and use its parent's instead. A string argument provides the introduction to each displayed line after the first, and a logical sets "brief" output (normally removing detailed subsidiary curve and surface definitions).

```
public: void blend_spl_sur::debug_data (
    char const*,            // character
    logical,                // integer
    FILE*                   // file
    ) const;
```

Debug printout. The virtual function debug prints a class-specific identifying line, then calls the ordinary function debug_data to put out the details. It is done this way so that a derived class' debug_data can call its parent's version first, to put out the common data. Indeed, if the derived class has no additional data it need not define its own version of debug_data and use its parent's instead. A string argument provides the introduction to each displayed line after the first, and a logical sets "brief" output (normally removing detailed subsidiary curve and surface definitions).

```
public: virtual spl_sur* blend_spl_sur::deep_copy (
    pointer_map*  pm        // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const = 0;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

In a *deep* copy, all the information about the copied item is self-contained in a new memory block. By comparison, a *shallow* copy stores only the first instance of the item in memory, and increments the reference count for each copy.

The pointer_map keeps a list of all pointers in the original object that have already been deep copied. For example, a deep_copy of a complex model results in self contained data, but identical sub-parts within the model are allowed to share a single set of data.

```
public: CVEC& blend_spl_sur::def_cvec () const;
```

Returns the CVEC on the defining curve of the blend, which is set each time the blend surface is evaluated.

```
public: void blend_spl_sur::determine_singularity ();
```

Determine the singularity of the surface. The results are stored in the u_singularity and v_singularity flags.

```
public: virtual int blend_spl_sur::evaluate (
    SPApar_pos const&,              // parameter
    SPAposition&,                   // pt on surface
                                    // at given param
    SPAvector**                     // array of pts
        = NULL,                     // to vector
                                    // array
    int                             // # derivatives
        = 0,                        // required (nd)
    evaluate_surface_quadrant       // eval location
        = evaluate_surface_unknown
    ) const =0;
```

The evaluate function calculates derivatives, of any order up to the number requested, and stores them in vectors provided by the user. It returns the number it was able to calculate; this will be equal to the number requested in all but the most exceptional circumstances. A certain number will be evaluated directly and (more or less) accurately; higher derivatives will be automatically calculated by finite differencing; the accuracy of these decreases with the order of the derivative, as the cost increases.

Any of the pointers may be NULL, in which case the corresponding derivatives will not be returned. Otherwise they must point to arrays long enough for all the derivatives of that order; i.e., 2 for the first derivatives, 3 for the second, etc.

```
public: virtual SPAunit_vector
    blend_spl_sur::eval_outdir (
    SPApar_pos const&       // outward direction
    ) const;
```

Returns a direction which points outward from the surface. This should be the outward normal if the point is not singular, otherwise a fairly arbitrary outward direction.

```
public: logical blend_spl_sur::extend_approx_sf (
    double start,                   // starting spine
                                    // parameter
    double end,                     // ending spine
                                    // parameter
    double requested_tol,           // requested
                                    // tolerance
    logical stop_if_illegal,        // flag for
                                    // approximating
    SPAbox const& region            // bounding box
        = * (SPAbox const*) NULL_REF// surface
    );
```

This creates the approximating surface and is capable of extending an existing surface. It extends the bs3_surface from spine parameter start to end, or makes it initially if no approximating surface is yet there. The surface is created to the requested tolerance, or better. If the requested tolerance is less than 0, a default is chosen with ”sensible” tolerance.

The start may be bigger than end, and the surface is constructed in decreasing parameter order. Either start or end should match an end of the existing approximating surface (if there is one). The other parameter should clearly continue on from there. If the logical flag is set, the approximating surface is terminated if it is found to self intersect or scrunch up. It still makes what it legally could, however.

The return value is TRUE if all the requested surface was made, or FALSE if it was terminated early, or any other infelicity occurs.

```
public: virtual void blend_spl_sur::extend_surface (
    SPAinterval new_v_range // new v range
    );
```

Extend the blend surface in place, including all the supporting data.

```
public: logical blend_spl_sur::find_stationary_pt (
    double start,               // start parameter
                                // for pt
    logical search_fwd,         // forward search
    logical                     // flag for if start
        ignore_start_root,      // point is root
    double& end                 // limit to search
        = * (double* ) NULL_REF
    );
```

Function to find stationary points. Only does anything if one of both blend_supports is on curve. Searches from the given start parameter for stationary points in the given direction as far as the defining curve allows. If it finds one, it adds that information to the legal range. The first logical flag is for searching up the range. The second logical flag is for whether the start point is to be counted if it turns out to be a root. Optionally, a limit to the search may be given, otherwise a default behavior of searching the rest of the curve will be used.

Returns TRUE if a stationary point is found. The legal_range may be queried to find where it is. If nothing is found returns FALSE. Currently doesn't work correctly for variable radius blends.

```
public: virtual logical
    blend_spl_sur::is_circular () const;
```

Returns TRUE if the given blend_spl_sur is circular; otherwise, it returns FALSE.

```
public: logical
    blend_spl_sur::is_var_rad_type () const;
```

Returns TRUE if the blend_spl_sur is a variable radius type; otherwise, it returns FALSE.

```
public: logical blend_spl_sur::legal_v_param (
    double v_param          // given parameter
    ) const;
```

A query function to check whether a given v–parameter value is within the legal range.

```
public: virtual void blend_spl_sur::make_approx (
    double fit,                 // fit tolerance
    const spline& spl           // pointer to output
        = * (spline*) NULL_REF, // spline approx
    logical force               // flag for forcing
        = FALSE
    ) const;
```

Makes or remakes an approximation of the surface, within the given tolerance.

```
public: logical
    blend_spl_sur::make_approximating_surface (
    double requested_tol        // requested
                                // tolerance
        = –1.0,
    SPAinterval const& range    // interval
        = * (SPAinterval const*)// pntr to interval
        NULL_REF,
    double const& start         // surface start
        = *(double const*)      // pntr to const
        NULL_REF,
    SPAbox const& region        // bounding region
        = * (SPAbox const* ) NULL_REF
    );
```

After a blend_spl_sur has been constructed and all its data is in place, the approximating surface must be made. This function is an interface to extend_approx_sf which does the really hard work. The fit tolerance may be specified or not. If not specified, the default of –1.0 is used. A fit less than zero always means to have the routine choose a ”sensible” value.

If range and start are unspecified, the defining curve’s entire range becomes that of the surface, regardless of whether the surface is legal everywhere. If range is specified and start is not, the range becomes exactly the given range of the surface, regardless of legality or otherwise.

If the range is unspecified but the start is specified, the range becomes as much surface either side of the start as is legal. If the start lies in an illegal region of the surface, the range becomes nothing at all. For periodic defining curves, it is possible for the final surface range to cross the curve’s join parameter.

If the range and start are both given, operation is performed up to the given range, starting where indicated, and watching out for illegal regions of surface. When periodic, operation never allows the bottom of the surface to go beyond the top minus the period.

A region box may be specified to indicate a particular region of interest of the spine – often defining curves may be much longer than we need. If a region is passed, construction of the approximating surface stops when the spine has wandered outside the region. If the spine starts outside the region, construction only stops when it enters the region and then leaves. The region may be omitted or left as a NULL reference to be ignored completely.

Returns TRUE if any surface at all was made, else FALSE.

```
public: virtual logical
    blend_spl_sur::old_make_approximating_surface (
    double requested_tol    // desired tolerance
    );
```

Creates a surface after a blend_spl_sur has been constructed and all its data is in place. This is the old way of creating a surface.

```
public: virtual void blend_spl_sur::operator*= (
    SPAtransf const&        // transformation
    );
```

Transform this blend by the given transform.

```
public: logical blend_spl_sur::operator== (
    subtype_object const&   // subtype object
    ) const;
```

Tests two blends for equality. This does not guarantee that all effectively equal surfaces are determined to be equal, but does guarantee that different surfaces are correctly identified as such.

```
public: virtual SPApar_pos blend_spl_sur::param (
    SPAposition const&,         // given point
    SPApar_pos const&           // guess result
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Find the parameter values of a point on the surface.

```
public: virtual void blend_spl_sur::point_perp (
    SPAposition const&,          // given point
    SPAposition&,                // resulting pt
                                 // on the surface
    SPAunit_vector&,             // surface normal
    surf_princurv&,              // principal
                                 // curvatures
    SPApar_pos const&            // guess uv
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                  // resulting uv
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak               // weak flag
    = FALSE
    ) const;
```

Find the point on the surface nearest to the given point, iterating from the given parameter values (if supplied). Returns the found point, the normal to the surface at that point, the principal curvatures there, and the parameter values at the found point (if requested).

```
public: void blend_spl_sur::restore_data ();
```

Restores the data from a save file. The restore operation switches on a table defined by static instances of the restore_subtype_def class. This invokes a simple friend function which constructs an object of the right derived type. Then it calls the appropriate base class member function to do the actual work.

```
// restoring supports
read_string             // type of left support curve
left_support–>restore_data              // left support
read_string             // type of right support curve
right_support–>restore_data             // restore right support data
restore_curve                           // restore def curve
read_real                               // left offset
read_real                               // right offset
read_enum                               // rad number
if (rad_num == ONE_RADIUS || rad_num == TWO_RADII) {
    restore_radius                      // restore left radius
if ( rad_num ==  TWO_RADII )// if two radii
    restore_radius                      // restore right radius
else                                    // else
    restore_cross_section               // restore cross section

if ( restore_version_number < APPROX_SUMMARY_VERSION )
    read_interval                       // u range
    read_interval                       // support u param range
    read_interval                       // v range
    read_int                            // closed in u
    read_int                            // closed in v
else                                    // else
    read_interval                       // support u param range
if ( restore_version_number >= 201 )
    read_interval                       // legal interval
    read_int                            // approximation required
    read_real                           // initial fit tolerance
    read_real                           // fit tolerance data not needed

read_int                                // left handed if false
if ( restore_version_number >= APPROX_SUMMARY_VERSION )
    restore_common_data();
}
```

```
public: virtual void
    blend_spl_sur::save_data () const;
```

Saves the information associated with this blend_spl_sur to a SAT file.

```
public: blend_section&
    blend_spl_sur::section_data () const;
```

Method for handling section data.

```
public: void blend_spl_sur::set_fitol (
    double tol              // double tolerance
    );
```

Set the approximating fit tolerance.

```
public: void blend_spl_sur::set_initial_fitol (
    double fitol            // fit tolerance
    );
```

Set the initial fit tolerance.

```
public: void blend_spl_sur::set_initial_num_u (
    int num_u               // u value
    );
```

Set the initial u value.

```
public: void blend_spl_sur::set_left_bs2_curve (
    bs2_curve               // bs2 curve
    );
```

Set the bs2_curves into the support data.

```
public: void blend_spl_sur::set_right_bs2_curve (
    bs2_curve               // bs2 curve
    );
```

Set the bs2_curves into the support data.

```
public: void blend_spl_sur::set_sur (
    bs3_surface approx      // approx. bs3 surface
    );
```

Set the approximating surface tolerance.

```
public: void blend_spl_sur::set_u_closure (
    closed_forms cl         // closure
    );
```

Set closure properties. This is a protected member of spl_sur.

```
public: void blend_spl_sur::set_u_range (
    double start,           // start
    double end              // end
    );
```

Set the *u*-parameter range. Don't allow start > end. If so, makes an empty interval.

```
public: void blend_spl_sur::set_v_closure (
    closed_forms cl         // closure
    );
```

Set closure properties. This is a protected member of spl_sur.

```
public: void blend_spl_sur::set_v_range (
    double start,           // start
    double end              // end
    );
```

Set the *v*-parameter range. Don't allow start > end. If so, makes an empty interval.

```
public: virtual void blend_spl_sur::shift_u (
    double                  // double
    );
```

Parameter shift: adjust the spline surface to have a parameter range increased by the argument value (which may be negative). This is only used to move portions of a periodic surface by integral multiples of the period.

```
public: virtual void blend_spl_sur::shift_v (
    double                  // double
    );
```

Parameter shift: adjust the spline surface to have a parameter range increased by the argument value (which may be negative). This is only used to move portions of a periodic surface by integral multiples of the period.

```
public: virtual void blend_spl_sur::split_u (
    double,                  // double
    spl_sur* [ 2 ]           // second spline surface
    );
```

Divide a surface into two pieces at the *u*-parameter value. If the split is at the end of the parameter range, the spl_sur is just returned as the appropriate half (in increasing parameter order), and the other is NULL. Otherwise a new spl_sur is used for one part, and the old one is modified for the other.

```
public: virtual void blend_spl_sur::split_v (
    double,                  // double
    spl_sur* [ 2 ]           // second spline surface
    );
```

Divide a surface into two pieces at the *v*-parameter value. If the split is at the end of the parameter range, the spl_sur is just returned as the appropriate half (in increasing parameter order), and the other is NULL. Otherwise a new spl_sur is used for one part, and the old one is modified for the other.

```
public: CVEC& blend_spl_sur::support_cvec (
    int i                    // left support if 0,
                             // otherwise right
    ) const;
```

Returns the CVEC on the left or right blend support, if that support is or contains a curve. This CVEC is set each time the blend surface is evaluated.

```
public: SVEC& blend_spl_sur::support_svec (
    int i                    // left support if 0,
                             // otherwise right
    ) const;
```

Returns the SVEC on the left or right support, if that support is, or contains a surface. This SVEC is set each time the blend surface is evaluated.

```
public: virtual char const*
    blend_spl_sur::type_name () const =0;
```

Returns the string “blend_spl_sur”.

```
public: void blend_spl_sur::update_legal_range (
    double v_param,          // range value
    logical is_upper_bound   // true if upper bound
    );
```

Update the legal_range of the blend surface, given the parameter at which the surface must stop, and whether the bound is an upper bound or not. Does the correct thing for periodic def_curves.

```
public: curve* blend_spl_sur::u_param_line (
    double v,                // constant u parameter
    spline const& owner      // surface where curve is
    ) const;
```

Constructs an isoparameter line on the surface. A *u* parameter line runs in the direction of increasing *u* parameter, at constant v. A *v* parameter line runs in the direction of increasing *v*, at constant *u*. The parameterization in the non-constant direction matches that of the surface, and has the range obtained by use of param_range_u() or param_range_v() appropriately.

```
public: curve* blend_spl_sur::v_param_line (
    double u,                // constant u parameter
    spline const& owner      // surface where curve is
    ) const;
```

For v_param_line, we can make a blend_int_cur rather than a par_int_cur, but otherwise do the same as the base class. A blend_int_cur is the same as a par_int_cur, but more wary about zero length derivatives at the end of the curve.

```
public: virtual logical
    blend_spl_sur::zero_end_radius (
    logical at_start,        // at start point if true
    double tol               // tolerance
        = SPAresabs
    ) const;
```

Returns TRUE if the blend radius at the start or end point of the blend_spl_sur is zero (i.e., less than SPAresabs).

```
public: virtual logical
    blend_spl_sur::zero_end_rad_slope (
    logical at_start,       // at start point if true
    double tol              // tolerance
        = SPAresabs
    ) const;
```

Returns TRUE if the blend radius slope at the start or end point of the blend_spl_sur is zero (i.e., less than SPAresabs).

Internal Use: deep_copy_elements_blend, full_size

Related Fncs: restore_blend_spl_sur

BODY

Class: Model Topology, SAT Save and Restore

Purpose: Represents a wire, sheet, or solid body.

Derivation: BODY : ENTITY : ACIS_OBJECT : –

SAT Identifier: “body”

Filename: kern/kernel/kerndata/top/body.hxx

Description: A BODY models a wire, sheet, or solid body. A body may be several disjoint bodies treated as a collection of lumps.

Lumps represent solids, sheets, and wires. In a manifold solid, every edge is adjacent to two faces. A nonmanifold solid may have edges that are adjacent to more than two faces. A nonmanifold solid may also have more than one set of faces at a vertex. Edges in a sheet may bound any number of faces. Edges of a wire do not bound any faces.

A pure wire body contains wires, edges, coedges, and vertices, but no faces. Wires can represent isolated points, open or closed profiles, and general wireframe models that are unsurfaced, i.e., have no faces. Wires are attached as a component of a shell and are not directly attached to the body.

A solid body is represented by the boundary of the region of space that is enclosed by a single lump. The lump is composed of one or more disjoint shells that contain no wires.

The geometry of body is given in a local coordinate system. This relates to the universal one by a transformation stored with the body.

Functions for traversing the topology are located in kernel/kerndata/top/query.hxx. These are useful for generating lists of faces, edges, and vertices on other topological entities. Other functions of note include: get_body_box to retrieve or recalculate the bounding box of a body; point_in_body to determine the containment of a point versus a body; and raytest_body to determine the intersections of a ray with a body.

Limitations: None

References:

KERN	LUMP, TRANSFORM, WIRE
by KERN	LUMP, pattern_holder

Data:

None

Constructor:

```
public: BODY::BODY ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: BODY::BODY (
    LUMP*                   // LUMP pointer
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: BODY::BODY (
    WIRE*                   // WIRE pointer
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void BODY::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual BODY::~BODY ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new BODY(...) then later x–>lose.)

Methods:

```
public: SPAbox* BODY::bound () const;
```

Returns the pointer to a geometric bounding region (a box) that includes the complete body with respect to its internal coordinate system. The pointer is NULL if a bound was not calculated since the body was last changed.

```
protected: virtual logical
    BODY::bulletin_no_change_vf (
    ENTITY const* other,        // other pointer
    logical identical_comparator// comparator
    ) const;
```

Compare this object with its change bulletin partner to see if the two entities are really the same.

```
public: virtual void BODY::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int BODY::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier BODY_TYPE. If level is specified, returns BODY_TYPE for that level of derivation from ENTITY. The level of this class is defined as BODY_LEVEL.

```
public: virtual logical BODY::is_deepcopyable (
    ) const;
```

Returns TRUE if this can be deep copied.

```
public: logical BODY::is_pattern_child () const;
```

Returns TRUE if this is a pattern child.

```
public: LUMP* BODY::lump () const;
```

Returns a pointer to the beginning of the list of bounding lumps of a body.

```
public: logical BODY::patternable () const;
```

Returns TRUE.

```
public: logical BODY::remove_from_pattern_list ();
```

Removes this entity from the list of entities maintained by its pattern, if any. Returns FALSE if no pattern is found, otherwise TRUE.

```
public: logical BODY::remove_pattern ();
```

Removes the pattern on this and all associated entities. Returns FALSE if no pattern is found, otherwise TRUE.

```
public: void BODY::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

There is a change to the body record at version 1.6. Previously there was a direct SHELL pointer. Now it indirects through a LUMP list. When reading an old save file, construct the intervening lump.

```
if (restore_version_number >= PATTERN_VERSION)
    read_ptr                          Pointer to record in save file for
                                      APATTERN on loop
if (apat_idx != (APATTERN*)(–1))
    pattern_ptr–>restore_cache();
if (restore_version_number <LUMP_VERSION)
    read_ptr                          Pointer to shell tag
    if ((int)shell_tag >= 0)          if the shell_tag is not NULL, then
                                      create a new LUMP pointer.
    else                              if the shell_tag is NULL, then the
                                      LUMP pointer is also NULL.
else                                  if the lump is not NULL
    read_ptr                          Pointer to record in save file for
                                      first LUMP shell in body
read_ptr                              Pointer to record in save file for
                                      first WIRE in body.
read_ptr                              Pointer to record in save file for
                                      body TRANSFORM.
```

```
public: void BODY::set_bound (
    SPAbox*                 // pointer to new box
    );
```

Sets the body's SPAbox pointer to point to the given SPAbox. This method is generally called internally in conjunction with the get_body_box function. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void BODY::set_lump (
    LUMP*,                  // pointer to new LUMP
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the body's LUMP pointer to point to the given LUMP. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void BODY::set_pattern (
    pattern* in_pat             // pattern
    );
```

Set the pattern.

```
public: void BODY::set_transform (
    TRANSFORM*              // ptr to new TRANSFORM
    );
```

Sets the body's TRANSFORM pointer to point to the given TRANSFORM. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: void BODY::set_wire (
    WIRE*,                  // pointer to new WIRE
    logical reset_pattern   // reset or not
    = TRUE
    );
```

Sets the body's WIRE pointer to point to the given WIRE. Before performing a change, it checks if the data structure is posted on the bulletin board. If not, the method calls backup to put an entry on the bulletin board.

```
public: TRANSFORM* BODY::transform () const;
```

Returns a pointer to the transformation that relates the local coordinate system to the global one in which the body resides.

```
public: void BODY::transform_patterns (
    const SPAtransf& tform      // transform
    );
```

Perform the transform on the pattern.

```
public: virtual const char* BODY::type_name () const;
```

Returns the string "body".

```
public: WIRE* BODY::wire () const;
```

Returns a pointer to the start of list-of-wires of a body.

Related Fncs:

is_BODY

bounded_arc

Class: Model Geometry

Purpose: Defines a bounded_arc as a subtype of a bounded_curve.

Derivation: bounded_arc : bounded_curve : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/geomhusk/bnd_arc.hxx

Description: This class adds no new data to the bounded_curve class from which it is derived, but it provides additional constructors and redefines some virtual functions.

Limitations: None

References: by KERN bounded_curve, bounded_line

Data:

None

Constructor:

```
public: bounded_arc::bounded_arc ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: bounded_arc::bounded_arc (
    const bounded_arc&      // given bounded arc
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: bounded_arc::bounded_arc (
    const SPAposition& center,  // center
    const SPAposition& pt1,     // edge point 1
    const SPAposition& pt2,     // edge point 2
    const SPAunit_vector& normal// normal vector
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

If pt1 equals pt2, then a full circle is created. Use normal only if center, pt1, and pt2 do not determine a plane.

```
public: bounded_arc::bounded_arc (
    const SPAposition& center,  // center
    const SPAunit_vector& normal,// normal vector
    const SPAvector& majax,     // 0-angle/radius
                                // vector
    double t0,                  // major axis start
                                // angle
    double t1,                  // major axis end
                                // angle
    double ratio                // radius ratio
        = 1.0
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

The angles are in radians. The radius is the length of the major_axis vector. The center_pt + major_axis corresponds to the point at the 0-degree angle on the arc.

```
public: bounded_arc::bounded_arc (
    const SPAposition& center,  // center
    double radius,              // radius
    const SPAunit_vector& normal// plane normal
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: bounded_arc::bounded_arc (
    const SPAposition& pt1, // position 1
    const SPAposition& pt2, // position 2
    const SPAposition& pt3, // position 3
    logical full            // positions colinear?
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

The arc passes through three points. If the positions are colinear, this method returns an error.

```
public: bounded_arc::bounded_arc (
    const SPAposition& pt1,     // position 1
    const SPAposition& pt2,     // position 2
    const SPAunit_vector& normal,// normal vector
    logical full                // positions
                                // colinear?
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates an arc given two points on the diagonal. If the positions are colinear, this method returns an error.

```
public: bounded_arc::bounded_arc (
    EDGE*,                  // edge
    const SPAtransf*        // transformation
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

The edge must be an ellipse.

Destructor:

None

Methods:

```
public: virtual logical bounded_arc::change_end_pt (
    const SPAposition&      // end position
    );
```

Changes the end position.

```
public: virtual logical
    bounded_arc::change_start_pt (
    const SPAposition&      // start position
    );
```

Changes the start position.

```
public: virtual bounded_curve* bounded_arc::copy (
    const SPAtransf*        // transformation
        = NULL
    ) const;
```

Creates a transformed copy.

```
public: SPAposition bounded_arc::get_center () const;
```

Returns the center.

```
public: SPAvector bounded_arc::get_major_axis ()
const;
```

Returns the major axis.

```
public: virtual SPAunit_vector
    bounded_arc::get_normal () const;
```

Returns the SPAunit_vector normal.

```
public: double bounded_arc::get_radius () const;
```

Returns the radius.

```
public: double
    bounded_arc::get_radius_ratio () const;
```

Returns the radius ratio of the arc.

```
public: sense_type bounded_arc::get_sense () const;
```

Returns the sense.

```
public: double bounded_arc::get_subtend () const;
```

Returns the subtended angle.

```
public: virtual logical bounded_arc::is_arc () const;
```

Determines if entity is an arc.

Returns TRUE if the given ENTITY is a bounded_arc; otherwise, it returns FALSE.

```
public: void bounded_arc::set_center (
    const SPAposition&      // arc center position
    );
```

Modifies the arc center position.

```
public: void bounded_arc::set_major_axis (
    const SPAvector&        // arc major axis
    );
```

Modifies the major axis of the arc.

```
public: void bounded_arc::set_normal (
    const SPAunit_vector&   // arc normal
    );
```

Modifies the normal to the arc.

```
public: void bounded_arc::set_radius (
    double                  // arc radius
    );
```

Modifies the arc radius.

```
public: void bounded_arc::set_radius_ratio (
    double                  // arc radius ratio
    );
```

Modifies the radius ratio of the arc.

Related Fncs:

None

bounded_curve

Class: Model Geometry

Purpose: Defines a bounded curve.

Derivation: bounded_curve : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/geomhusk/bnd_crv.hxx

Description: This class defines bounded curves. A bounded curve is a curve with a start and end parameters that specify the bounds of the curve. This class makes it easy to extract data from wireframe geometry. This class supports most of the functions, such as evaluation, curve length, etc., that are provided in the curve class.

Limitations: None

References: KERN curve

Data:

```
protected curve* acis_curve;
```
The pointer to an ACIS curve.

```
protected double end_param;
```
The end parameter of the ACIS curve.

```
protected double start_param;
```
The start parameter of the ACIS curve.

Constructor:

```
public: bounded_curve::bounded_curve ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: bounded_curve::bounded_curve (
    const bounded_curve&    // given bounded curve
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: bounded_curve::bounded_curve (
    const curve*,           // curve
    const SPAposition&,     // start position
    const SPAposition&      // end position
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a bounded curve, given a curve and start and end positions. The bounded curve created by this constructor does not own the curve, and it must be deleted explicitly, if needed.

```
public: bounded_curve::bounded_curve (
    const curve*,           // curve
    double,                 // start parameter
    double                  // end parameter
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a bounded curve, given a curve and start and end parameters.

```
public: bounded_curve::bounded_curve (
    EDGE*,                  // given edge
    const SPAtransf*        // transformation
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

```
public: virtual bounded_curve::~bounded_curve ();
```

C++ destructor, deleting a bounded_curve.

Methods:

```
public: int bounded_curve::acis_type () const;
```

Returns the type of underlying curve.

```
public: double bounded_curve::adjust_parameter (
    double t                // param value to adjust
    ) const;
```

Adjusts a parameter value so that it is within the principle range of a periodic curve. If the curve is not periodic, this method returns the input parameter. For a periodic curve, this method returns a parameter value between the start parameter and end parameter.

```
public: virtual double
    bounded_curve::approx_error () const;
```

Returns a distance value, that represents the greatest discrepancy between positions calculated by calls to eval or eval_position with the approx_OK logical set by turns to TRUE and FALSE. This method returns 0 as the default for curves that do not distinguish between these cases.

```
public: SPAbox bounded_curve::bound () const;
```

Computes a bounding box around the curve. There is no guarantee that the bound is minimal.

```
public: virtual logical
    bounded_curve::change_end_pt (
    const SPAposition&      // end point
    );
```

Moves the end point of a curve to a new location. For some types of curves this may not be possible; in which case, this method acts the same as set_end_pt.

```
public: virtual logical
    bounded_curve::change_start_pt (
    const SPAposition&      // start point
    );
```

Moves the start point of a curve to a new location. For some types of curves this may not be possible; in which case, this method acts the same as set_start_pt.

```
public: virtual logical
    bounded_curve::closed () const;
```

Indicates if a curve is closed. This method joins itself (smoothly or not) at the ends of its principal parameter range. If the periodic method returns TRUE, this method also returns TRUE.

```
public: virtual bounded_curve* bounded_curve::copy (
    const SPAtransf* transform  // transformation
        = NULL
    ) const;
```

Copies the bounded curve, and applies the transform, if given, to the copy.

```
public: virtual void bounded_curve::debug (
    char const*,            // indentation
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes the debug output for a bounded curve.

```
public: virtual void bounded_curve::eval (
    double,                 // parameter value
    SPAposition*,           // point on curve
    SPAvector*              // first derivative
        = NULL,
    SPAvector*              // second derivative
        = NULL,
    logical                 // repeated evaluation?
        = FALSE,
    logical                 // approx. results OK?
        = FALSE
    ) const;
```

Evaluates a curve at a given parameter value, returning the position and the first and second derivatives.

For this and the following inquiry methods, there are two optional logical arguments. The first, if TRUE, is a guarantee from the calling code that the most recent call to any curve or surface member method was in fact to one of these six methods for the same curve as the current call. It allows an implementation to cache useful intermediate results to speed up repeated evaluations, but use it with extreme care. The second logical argument may be set TRUE if an approximate return value is acceptable. Here, approximate may be assumed to be sufficient for visual inspection of the curve.

```
public: virtual SPAvector
    bounded_curve::eval_curvature (
    double,                 // parameter value
    logical                 // repeated evaluation?
        = FALSE,
    logical                 // approx. results OK?
        = FALSE
    ) const;
```

Finds the curvature at the given parameter value on the curve.

```
public: virtual SPAvector bounded_curve::eval_deriv (
    double,                 // parameter value
    logical                 // repeated evaluation?
        = FALSE,
    logical                 // approx. results OK?
        = FALSE
    ) const;
```

Finds the derivative (direction and magnitude) at the given parameter value on the curve.

```
public: virtual double
    bounded_curve::eval_deriv_len (
    double,                 // parameter value
    logical                 // repeated evaluation?
        = FALSE,
    logical                 // approx. results OK?
        = FALSE
    ) const;
```

Finds the magnitude of the derivative at the given parameter value on the curve.

```
public: SPAunit_vector bounded_curve::eval_direction
(
    double,                 // parameter value
    logical                 // repeated evaluation?
        = FALSE,
    logical                 // approx. results OK?
        = FALSE
    ) const;
```

Finds the tangent direction at the given parameter value on the curve.

```
public: virtual SPAposition
    bounded_curve::eval_position (
    double,                  // parameter value
    logical                  // repeated evaluation?
        = FALSE,
    logical                  // approx. results OK?
        = FALSE
    ) const;
```

Finds the point on a curve corresponding to a given parameter value.

```
public: virtual double bounded_curve::eval_t (
    const pick_ray&          // pick ray
    ) const;
```

Finds the closest point on a curve to a given pick location and return the curve parameter value.

```
public: virtual curve_extremum*
    bounded_curve::find_extrema (
    SPAunit_vector const&    // unit vector
    ) const;
```

Finds the extrema of a curve in a given direction. curve_extremum is defined in kernel/curve/curdef.hxx.

```
public: curve*
    bounded_curve::get_acis_curve () const;
```

Returns the underlying curve.

```
public: SPAunit_vector
    bounded_curve::get_end_dir () const;
```

Returns the end direction.

```
public: double bounded_curve::get_end_param () const;
```

Returns the end parameter.

```
public: SPAposition bounded_curve::get_end_pt ()
const;
```

Returns the end point.

```
public: bounded_curve*
    bounded_curve::get_full_curve () const;
```

Returns a copy of this curve. If the curve is a subset of a curve as a result of setting the parameter range, this method returns the full curve.

```
public: virtual SPAunit_vector
    bounded_curve::get_normal () const;
```

Returns the vector normal to the curve. This method returns the zero vector if the curve is straight or is nonplanar.

```
public: double
    bounded_curve::get_parameter_tolerance (
    double t,               // tolerance parameter
    double tol              // tolerance for points
    ) const;
```

Returns a tolerance to use for comparing if two parameter values evaluate to the same point.

```
public: double bounded_curve::get_range () const;
```

Returns the parameter range.

```
public: int bounded_curve::get_side (
    const SPAunit_vector&,  // unit vector for plane
    const SPAposition&      // point
    );
```

Determines which side of the curve a given point is on relative to a plane defined by a SPAunit_vector. This method returns +1 for right and –1 for left.

```
public: SPAunit_vector
    bounded_curve::get_start_dir () const;
```

Returns the start direction.

```
public: double
    bounded_curve::get_start_param () const;
```

Returns the start parameter.

```
public: SPAposition
    bounded_curve::get_start_pt () const;
```

Returns the start point.

```
public: virtual logical
    bounded_curve::is_arc () const;
```

Returns TRUE if the given ENTITY is a bounded_curve arc; otherwise, it returns FALSE.

```
public: virtual logical
    bounded_curve::is_in_parallel_plane (
    const SPAunit_vector&   // unit vector
    ) const;
```

Returns TRUE if the given ENTITY if a curve lies in a plane that is perpendicular to the given SPAunit_vector.; otherwise, it returns FALSE.

```
public: virtual logical bounded_curve::is_in_plane (
    const SPAposition&,     // position
    const SPAunit_vector&   // normal to plane
    ) const;
```

Returns TRUE if the given ENTITY if a curve lies in a plane; otherwise, it returns FALSE.

```
public: virtual logical
    bounded_curve::is_line () const;
```

Checks for the line subclass.

Returns TRUE if the given ENTITY is a bounded_curve line; otherwise, it returns FALSE.

```
public: virtual logical bounded_curve::is_point ()
const;
```

Returns TRUE if the given bounded_curve is a bounded_point; otherwise, it returns FALSE. The existence of this method makes the base class aware of some of the derived classes. One often wants to know if a bounded curve is really a line or an arc to do special operations. This is added as a convenience.

```
public: virtual double bounded_curve::length (
    double t0,              // first parameter
    double t1               // second parameter
    ) const;
```

Returns the algebraic distance along the curve between the given parameters. The sign is positive if the parameter values are given in increasing order, and negative if they are in decreasing order. The result is undefined if either parameter value is outside the parameter range of a bounded curve. For a periodic curve, the parameters are not reduced to the principal range, and so the portion of the curve evaluated may include several complete circuits. This method is always a monotonically increasing function of t1 if t0 is held constant, and a decreasing function of t0 if t1 is held constant.

```
public: virtual double bounded_curve::length_param (
    double,                 // datum parameter
    double                  // arc length
    ) const;
```

Returns the parameter value of the point on the curve at the given algebraic arc length from that defined by the datum parameter. This method is the inverse of the length method. The result is not defined for a bounded nonperiodic curve if the datum parameter is outside the parameter range, or if the length is outside the range bounded by the values for the ends of the parameter range.

```
public: bs3_curve
    bounded_curve::make_bs3_curve () const;
```

Creates a bs3_curve from this bounded curve.

```
public: virtual EDGE*
    bounded_curve::make_edge () const;
```

Creates an EDGE from this curve.

```
protected: void bounded_curve::make_valid (
    logical signal_error     // signal an error?
        = FALSE
    );
```

Ensures that the data in a curve is valid. This method helps to avoid checking for a valid curve pointer in acis_curve or the zero parameter range. If logical is TRUE, then this method causes an error to generate.

```
public: virtual bounded_curve&
    bounded_curve::negate ();
```

Reverses the direction of the curve.

```
public: virtual bounded_curve&
    bounded_curve::operator*= (
    SPAtransf const&         // transformation
    );
```

Transforms a curve.

```
public: virtual double bounded_curve::param (
    const SPAposition&,      // point on the curve
    const double*            // approx. param value
        = NULL
    ) const;
```

Finds the parameter value of a point on a curve, corresponding to the given point. The results of this method are only guaranteed to be valid for points on the curve, though particular curve types may give useful curve-dependent results for other points.

```
public: double bounded_curve::param_from_01 (
    double t                 // parameters
    );
```

Converts from parameters ranging from 0 to 1 to the double range.

```
public: virtual double
    bounded_curve::param_period () const;
```

Returns the period of a periodic curve. This method returns 0 if the curve is not periodic.

```
public: double bounded_curve::param_to_01 (
    double                  // parameters
    );
```

Converts to parameters ranging from 0 to 1 to the double range.

```
public: virtual logical
    bounded_curve::periodic () const;
```

Indicates if a curve is periodic. This method joins itself smoothly at the ends of its principal parameter range, so that edges may span the seam.

```
public: virtual void bounded_curve::point_perp (
    const SPAposition&,     // position
    SPAposition*,           // returned point
    SPAunit_vector*,        // returned normal
    double const*           // guess parameter
        = NULL,
    double*                 // actual parameter
        = NULL,
    logical f_weak          // weak flag
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve, and tangent to the curve at that point, and its parameter value.

If an input parameter value is supplied (as the fourth argument), the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one at which the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case it is ignored.

```
public: bounded_curve*
    bounded_curve::project_to_plane (
    const plane&            // plane
    ) const;
```

Returns a curve that is the projection of this curve onto a plane.

```
protected: logical bounded_curve::set_acis_curve (
    curve*                  // curve
    );
```

Sets the ACIS curve for this bounded curve.

```
public: double bounded_curve::set_end_param (
    double                  // end parameter
    );
```

Sets the end parameter.

```
public: double bounded_curve::set_end_t (
    const SPAposition&,     // position
    const double*           // approximate parameter
        = NULL              // position
    );
```

Sets the end points of a curve. This method assumes that the given position lies on the curve, and it modifies the curve so it ends at that position. If the position is not on the curve, the closest position on the curve is used.

```
public: void bounded_curve::set_parameter_range (
    double,                 // start parameter
    double                  // end parameter
    );
```

Sets the parameter range.

```
public: double bounded_curve::set_start_param (
    double                  // start parameter
    );
```

Sets the start parameter.

```
public: double bounded_curve::set_start_t (
    const SPAposition&,     // position
    const double*           // approximate parameter
        = NULL              // position
    );
```

Sets the start points of a curve. This method assumes that the given position lies on the curve, and it modifies the curve so it starts at that position. If the position is not on the curve, the closest position on the curve is used.

For curves, these methods take an object of class SPAparameter as input for an approximation. For consistency, these methods all use doubles for curve parameters.

```
public: virtual bounded_curve* bounded_curve::split (
    double,                 // parameter value
    SPAposition const&      // position curve passes
    );
```

Splits a curve at given parameter value. If the curve is splittable (not closed-in practice one defined or approximated by one or more splines). This method returns a new curve for the low-parameter part, and the old one as the high-parameter part. For a nonsplittable curve, it leaves the old one alone and returns NULL. The default is to make the curve nonsplittable.

```
public: logical bounded_curve::test_point (
    const SPAposition& pos,     // point
    const double* param_guess   // guess value
        = NULL,
    double* param_actual        // actual value
        = NULL
    ) const;
```

Tests point-on-curve, optionally returning the exact parameter value if the point is on the curve. This method tests to standard system precision.

```
public: virtual logical
    bounded_curve::test_point_tol (
    const SPAposition&,     // position
    double                  // tolerance
        = 0,
    const double*           // guess value
        = NULL,
    double*                 // actual value
        = NULL
    ) const;
```

Tests point-on-curve, optionally returning the exact parameter value if the point is on the curve. This method tests to a given precision.

```
public: const char*
    bounded_curve::type_name () const;
```

Returns the string “bounded_curve”.

Related Fncs:

get_bounded_curve, new_ellipse

bounded_line

Class: Model Geometry

Purpose: Defines a bounded_line as a subtype of bounded_curve.

Derivation: bounded_line : bounded_curve : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/geomhusk/bnd_line.hxx

Description: This class adds no new data to bounded_curve, but it provides additional constructors and redefines some virtual functions.

Limitations: None

References: by KERN bounded_curve

Data:

None

Constructor:

```
public: bounded_line::bounded_line ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: bounded_line::bounded_line (
    const bounded_line&     // original constructor
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: bounded_line::bounded_line (
    const SPAposition&,      // first position
    const SPAposition&       // second position
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: bounded_line::bounded_line (
    const SPAposition&,      // position
    const SPAunit_vector&,   // direction
    double                   // distance
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a line from a position, a direction, and a distance.

```
public: bounded_line::bounded_line (
    straight&,               // straight
    double,                  // first parameter
    double                   // second parameter
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a line from a position, a direction, and a distance.

```
public: bounded_line::bounded_line (
    EDGE*,                   // edge
    const SPAtransf*         // transformation
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

The edge must be a straight.

Destructor:

None

Methods:

```
public: virtual logical bounded_line::change_end_pt (
    const SPAposition&      // end position
    );
```

Changes the end position.

```
public: virtual logical
    bounded_line::change_start_pt (
    const SPAposition&      // start position
    );
```

Changes the start position.

```
public: virtual bounded_curve* bounded_line::copy (
    const SPAtransf*        // transformation
        = NULL
    ) const;
```

Makes a transformed copy of the line.

```
public: virtual double bounded_line::eval_t (
    const pick_ray&         // pick location
    ) const;
```

Finds the closest point on a curve to a given pick location and returns the curve parameter value.

```
public: virtual logical
    bounded_line::is_line () const;
```

Finds if entity is a line.

Returns TRUE if the given ENTITY is a bounded_line; otherwise, it returns FALSE.

```
public: EDGE* bounded_line::make_edge () const;
```

Makes an edge from the line.

Related Fncs:

create_line_offset, new_line

BULLETIN

Class: History and Roll, SAT Save and Restore

Purpose: Describes the records that are chained into bulletin-boards.

Derivation: BULLETIN : ACIS_OBJECT : –

SAT Identifier: “bulletin”

Filename: kern/kernel/kerndata/bulletin/bulletin.hxx

Description: A bulletin has a type signifying the creation, change, or deletion of a model entity. The type is not stored, but deduced from the presence or absence of new and old entity pointers. Bulletins are chained into bulletin-boards, in a doubly-linked list.

Limitations: None

References:

KERN	BULLETIN_BOARD, ENTITY
by KERN	BULLETIN_BOARD, ENTITY

Data:

```
public BULLETIN *next_ptr;
```
list pointer

```
public BULLETIN *previous_ptr;
```
list pointer

```
public BULLETIN_BOARD* owner_ptr;
```
pointer to the owner of this bulletin

```
public BULLETIN *next_bb_b_ptr;
```
next pointer

Constructor:

```
public: BULLETIN::BULLETIN (
    ENTITY*,                // old entity
    ENTITY*                 // new entity
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Creates a bulletin for the given old and new entities, and adds it to the current bulletin-board (which must already exist).

```
public: BULLETIN::BULLETIN ();
```

C++ constructor.

Destructor:

```
public: BULLETIN::~BULLETIN ();
```

C++ destructor, deleting a BULLETIN.

Methods:

```
public: logical
    BULLETIN::attrib_only_change () const;
```

Returns whether or not there has been a change to only the attribute.

```
public: void BULLETIN::clear_history ();
```

Clear the history stream.

```
public: void BULLETIN::debug (
    FILE*                   // file pointer
        = debug_file_ptr
    ) const;
```

Outputs debug information about BULLETIN to standard output or to the specified file.

```
public: void BULLETIN::debug (
    int id,                 // id
    int level,              // level
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes information about the BULLETIN to the debug file or to the specified file.

```
public: ENTITY* BULLETIN::entity_ptr () const;
```

Returns a pointer to the current entity.

```
public: logical BULLETIN::fix_pointers (
    ENTITY* elist[],        // pointers to fix
    BULLETIN_BOARD* owner   // owner
    );
```

The fix_pointers method for each entity in the restore array is called, with the array as argument. This calls fix_common, which calls its parent's fix_common, and then corrects any pointers in the derived class. In practice there is never anything special for fix_pointers to do, but it is retained for consistency and compatibility. (Supplied by the ENTITY_FUNCTIONS and UTILITY_DEF macros.)

```
public: logical BULLETIN::fix_pointers (
    ENTITY_ARRAY&  elist,    // pointers to fix
    BULLETIN_BOARD* owner    // owner
    );
```

The fix_pointers method for each entity in the restore array is called, with the array as argument. This calls fix_common, which calls its parent's fix_common, and then corrects any pointers in the derived class. In practice there is never anything special for fix_pointers to do, but it is retained for consistency and compatibility. (Supplied by the ENTITY_FUNCTIONS and UTILITY_DEF macros.)

```
public: HISTORY_STREAM* BULLETIN::history_stream (
    logical from_ents           // from entities
        = FALSE                 // or not
    ) const;
```

Gets history from either bulletin board or entities

```
public: void BULLETIN::make_delete ();
```

Concatenates a change (or create) operation and a delete bulletin on the same ENTITY on the same bulletin board.

```
public: logical BULLETIN::mixed_streams (
    HISTORY_STREAM*& ent_hs,    // entity history
    logical&  can_be_fixed,     // fixable or not
    logical&  stream_corrupt,   // corrupt or not
    HISTORY_STREAM* bb_hs       // bulletin board
        = NULL                  // history
) const;
```

Returns TRUE when the entity's history, ent_hs, does not match the bulletin board's history, bb_hs. The entity's history is returned. The bulletin board's history can either be supplied (for performance) or figured out.

```
public: ENTITY* BULLETIN::new_entity_ptr () const;
```

Returns a pointer to the new entity created after an operation on the model.

```
public: BULLETIN* BULLETIN::next () const;
```

Returns the pointer to the next bulletin on the bulletin board.

```
public: BULLETIN* BULLETIN::next_bb_b () const;
```

Bulletin for an entity on the next bulletin board.

```
public: logical BULLETIN::no_change () const;
```

Returns whether or not there has been a change.

```
public: void BULLETIN::null_new_entity_ptr ();
```

Null the old entity pointer.

```
public: void BULLETIN::null_old_entity_ptr ();
```

Null the new entity pointer.

```
public: ENTITY* BULLETIN::old_entity_ptr () const;
```

Returns the pointer to the old entity.

```
public: BULLETIN_BOARD* BULLETIN::owner () const;
```

Returns the owner of the entity corresponding to the bulletin.

```
public: BULLETIN* BULLETIN::previous () const;
```

Returns the pointer to the previous bulletin on the bulletin board.

```
public: logical BULLETIN::restore (
    BULLETIN* previous_b,    // previous bull. board
    logical ignore_string_version   // ingore version
        = FALSE                 // or not
    );
```

Modifies the bulletin such that the new and old ENTITY pointers reflect the change of state.

#if SAVE_NAMES	
read_id	Read in the identification for the bulletin
#endif	
read_pointer	Pointer to a record in the SAT file for the old ENTITY.
read_pointer	Pointer to a record in the SAT file for the new ENTITY.

```
public: void BULLETIN::roll ();
```

Modifies the bulletin such that the new and old ENTITY pointers reflect the change of state.

```
public: logical BULLETIN::save (
    ENTITY_LIST& elist      // entity list
    logical ignore_string_version   // ingore version
        = FALSE                 // or not
    );
```

Saves the entities corresponding to the bulletin.

```
public: void BULLETIN::set_entity_ptrs (
    ENTITY* old_ent,            // old entity
    ENTITY* new_ent             // new entity
    );
```

Set the entity pointers.

```
public: void BULLETIN::set_history (
    HISTORY_STREAM* hist        // history stream
    ) const;
```

Set the current history stream.

```
public: void BULLETIN::set_next_bb_b (
    BULLETIN*                   // bulletin
    );
```

Set the corresponding bulletin in the next bulletin board.

```
public: int BULLETIN::size (
    logical include_backups // include backups
        = TRUE              // as part of size
    ) const;
```

Returns the size of the BULLETIN.

```
public: void BULLETIN::swap (
    ENTITY* this_ent,       // this entity
    ENTITY* that_ent        // that entity
    );
```

Swap one entity for another.

```
public: BULLETIN_TYPE BULLETIN::type () const;
```

Returns the type of BULLETIN. Four types of bulletins are defined: NO_BULLETIN, CREATE_BULLETIN, CHANGE_BULLETIN, and DELETE_BULLETIN.

Related Fncs:

abort_bb, change_state, clear_rollback_ptrs, close_bulletin_board, current_bb, current_delta_state, debug_delta_state, delete_all_delta_states, delete_ds_branch, get_default_stream, initialize_delta_states, open_bulletin_board, release_bb, set_default_stream

BULLETIN_BOARD

Class: History and Roll, SAT Save and Restore

Purpose: Creates a record of the changes to a single ENTITY during the current operation on the model.

Derivation: BULLETIN_BOARD : ACIS_OBJECT : –

SAT Identifier: “bulletin_board”

Filename: kern/kernel/kerndata/bulletin/bulletin.hxx

Description: A BULLETIN_BOARD contains a list of BULLETINs, each of which records the changes to a single ENTITY during the current operation on the model. There are two types of current bulletin-board, mainline and stacked, and completed ones may be successful or failed, depending on the reported success of the completed operation.

Limitations: None

References:
KERN BULLETIN, DELTA_STATE, HISTORY_STREAM
by KERN BULLETIN, DELTA_STATE, outcome

Data:

```
public BULLETIN *end_b;
```
Pointer to last bulletin.

```
public BULLETIN *start_b;
```
Pointer to first bulletin.

```
public BULLETIN_BOARD *next_ptr;
```
Chains bulletin boards from a delta state.

```
public DELTA_STATE *owner_ptr;
```
The delta state from which this is chained.

```
public bb_status status;
```
Status of the bulletin board. Possible values are
- bb_open_mainline,
- bb_open_stacked,
- bb_closed_succeeded,
- bb_closed_failed

```
public int logging_level_when_stacked;
```
The number of api_begin's minus the number of api_end's made so far. In effect, this is the current API nesting level.

Constructor:

```
public: BULLETIN_BOARD::BULLETIN_BOARD (
    DELTA_STATE* ds          // change state
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Creates a new bulletin board at start of list given by bb_ptr in the current delta_state.

```
public:BULLETIN_BOARD::BULLETIN_BOARD(
    logical in_current_ds    // delta state
    );
```

C++ constructor.

Destructor:

```
public: BULLETIN_BOARD::~BULLETIN_BOARD ();
```

C++ destructor, deleting a BULLETIN_BOARD (usually at head of list of bulletin-boards in the delta_state) and deletes its bulletin entries.

```
public: void BULLETIN_BOARD::reset_history_on_delete
();
```

C++ destructor, resets the history stream on deletion.

Methods:

```
public: void BULLETIN_BOARD::add (
    BULLETIN*               // bulletin board
    );
```

Adds a new BULLETIN_BOARD to this delta state.

```
public: int BULLETIN_BOARD::add_dead_entity (
    ENTITY* ent                 // entity
    );
```

Add this to the dead entities list.

```
public: logical BULLETIN_BOARD::can_be_moved ()
const;
```

Returns whether or not the bulletin board can be moved.

```
public: logical BULLETIN_BOARD::checked () const;
```

Returns whether or not the bulletin board has been checked.

```
public: void
    BULLETIN_BOARD::clear_dead_entity_list ();
```

Clear the dead entity list.

```
public: void BULLETIN_BOARD::clear_history_ptrs ();
```

Clear history pointers.

```
public: logical BULLETIN_BOARD::closed () const;
```

Returns TRUE if the bulletin board closed successfully; otherwise, it returns FALSE.

```
public: logical BULLETIN_BOARD::corrupt () const;
```

Returns the check status, whether or not any history streams are corrupt.

```
public: void BULLETIN_BOARD::debug (
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes information about the bulletin board to the debug file or to the specified file.

```
public: void BULLETIN_BOARD::debug (
    int id,                 // entity id
    int level,              // entity level
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes information about the bulletin board to the debug file or to the specified file. The first two arguments specify a branch of the entity derivation hierarchy to call debug_ent on, in addition to the normal bulletin board debugging stuff.

```
public: DELTA_STATE* BULLETIN_BOARD::delta_state (
    ) const;
```

Returns a pointer to the owner of the delta state.

```
public: BULLETIN*
    BULLETIN_BOARD::end_bulletin () const;
```

Returns the last bulletin in the bulletin board.

```
public: logical BULLETIN_BOARD::failure () const;
```

Returns TRUE if the bulletin board failed to close successfully; otherwise, it returns FALSE.

```
public: void BULLETIN_BOARD::find_bulletins (
    int type,               // entity type
    int level,              // entity level
    BULLETIN_LIST& blist    // bulletin list
    ) const;
```

Function for finding annotations. The first two arguments specify a branch of the entity derivation hierarchy to return bulletins for. For annotation use, we can use ANNOTATION_TYPE and ANNOTATION_LEVEL. It may also be useful to be more specific, such as SWEEP_ANNOTATION_TYPE and SWEEP_ANNOTATION_LEVEL. The is_XXXX functions generated by the ENTITY_DEF macro work well.

```
public: void BULLETIN_BOARD::find_bulletins (
    is_function tester,     // testing function
    BULLETIN_LIST& blist    // bulletin list
    ) const;
```

Function for finding annotations. The first two arguments specify a branch of the entity derivation hierarchy to return bulletins for. In this form the tester identifies the type of entity to look for. For annotation use, we can use ANNOTATION_TYPE and ANNOTATION_LEVEL. It may also be useful to be more specific, such as SWEEP_ANNOTATION_TYPE and SWEEP_ANNOTATION_LEVEL. The is_XXXX functions generated by the ENTITY_DEF macro work well.

```
public: logical BULLETIN_BOARD::fix_pointers (
    ENTITY_ARRAY& elist,    // pointers to fix
    DELTA_STATE_LIST& dslist// delta state list
    );
```

The fix_pointers method for each entity in the restore array is called, with the array as argument. This calls fix_common, which calls its parent's fix_common, and then corrects any pointers in the derived class. In practice there is never anything special for fix_pointers to do, but it is retained for consistency and compatibility. (Supplied by the ENTITY_FUNCTIONS and UTILITY_DEF macros.)

```
public: HISTORY_STREAM*
     BULLETIN_BOARD::get_alternate_stream (
    ) const;
```

Get the history stream that the bulletin board needs to be in.

```
public: bb_check_status
     BULLETIN_BOARD::get_check_status () const;
```

Returns the bulleting board check status, indicating whether the bulletin board has been checked or not, and the result of the checking.

```
public: HISTORY_STREAM*
    BULLETIN_BOARD::history_stream (
    ) const;
```

Returns the history stream associated with the owner pointer.

```
public: logical BULLETIN_BOARD::is_dead_entity (
    ENTITY* ent                 // entity
    );
```

Returns TRUE if this is a dead entity.

```
public: logical BULLETIN_BOARD::merge_next (
    logical rollback_set    // on/off indicator
    );
```

Merges next bulletin into roll back history.

```
public: logical BULLETIN_BOARD::mixed () const;
```

Returns TRUE if this is a mixed stream.

```
public: logical BULLETIN_BOARD::mixed_streams (
    HISTORY_STREAM*& alternative_hs,// alternate
                                    // stream
    logical&  move_fixes            // move fixes
        = * (logical* )NULL_REF,    // or not
    logical remove_bulls            // remove bulletins
        = FALSE                     // or not
    );
```

Returns TRUE if the bulletin board's history is not the same as the history in entities on the bulletin board.

```
public: BULLETIN_BOARD*
    BULLETIN_BOARD::next () const;
```

Returns the next bulletin in the bulletin board.

```
public: logical BULLETIN_BOARD::open () const;
```

Returns TRUE if the bulletin board opened successfully; otherwise, it returns FALSE.

```
public: logical BULLETIN_BOARD::pending () const;
```

Returns whether or not a bulleting board merge is pending.

```
public: void BULLETIN_BOARD::remove (
    BULLETIN*               // bulletin board
    );
```

Removes a bulletin board from this delta state.

```
public: int BULLETIN_BOARD::remove_dead_entity (
    ENTITY* ent                 // entity
    );
```

Remove this dead entity.

```
public: logical BULLETIN_BOARD::restore (
    BULLETIN_BOARD* previous_bb // previous state
    logical ignore_string_version   // ingore version
        = FALSE                 // or not
    );
```

Restores roll back to previous state.

```
if (!ignore_string_version &&  restore_version_number
                                        STRINGLESS_HISTORY_VERSI
                                        ON)
     read_id                            // id for bulletin board
read_pointer          // owning DELTA_STATE pointer
read_int              // status
if(read_int)          // if there is at least one bulletin
       BULLETIN::restore                // Restore an individual bulletin
          while(read_int)               // if there are more bulletins
               BULLETIN::restore        // Restore an individual bulletin
```

```
public: void BULLETIN_BOARD::roll ();
```

Rolls back over a complete delta state, inverting it so as to allow roll forward the next time.

```
public: logical BULLETIN_BOARD::rollbacks_cleared (
    ) const;
```

Returns whether or not rollbacks have been cleared.

```
public: logical BULLETIN_BOARD::save (
    ENTITY_LIST& elist,      // entities
    DELTA_STATE_LIST& dslist// delta states
    logical ignore_string_version   // ingore version
        = FALSE                  // or not
    );
```

Saves the delta states and entities corresponding to this bulletin board.

```
public: void BULLETIN_BOARD::set_alternate_stream (
    HISTORY_STREAM* ahs          // alternate stream
    );
```

Set the history stream that the bulletin board needs to be in.

```
public: void BULLETIN_BOARD::set_check_status (
    bb_check_status s            // status
    );
```

Sets the check status.

```
public: void BULLETIN_BOARD::set_pending (
    logical pending_value       // value
    );
```

Set the pending value.

```
public: void BULLETIN_BOARD::set_rollbacks_cleared (
    logical severed             // cleared
    );
```

Merge method, set whether or not rollbacks are cleared on merge.

```
public: int BULLETIN_BOARD::size (
    logical include_backups // include backups
        = TRUE              // as part of the size
    ) const;
```

Returns the size of the bulletin board.

```
public: BULLETIN*
    BULLETIN_BOARD::start_bulletin () const;
```

Returns the last bulletin in the bulletin board.

```
public: logical BULLETIN_BOARD::successful () const;
```

Returns TRUE if the bulletin-board closed successfully; otherwise, it returns FALSE.

Internal Use: full_size

Related Fncs: abort_bb, change_state, clear_rollback_ptrs, close_bulletin_board, current_bb, current_delta_state, debug_delta_state, delete_all_delta_states, delete_ds_branch, get_default_stream, initialize_delta_states, open_bulletin_board, release_bb, set_default_stream