Chapter 30.
Classes Da thru Dz

Topic: Ignore

DEBUG_LIST

Class: Debugging

Purpose: Defines a simple list pointer, which allows all active lists to be scanned.

Derivation: DEBUG_LIST : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerndata/data/debug.hxx

Description: This class defines a simple list pointer, which allows all active lists to be scanned.

Limitations: None

References: KERN ENTITY_LIST

Data:

None

Constructor:

```
public: DEBUG_LIST::DEBUG_LIST (
    char const* type        // type name
    );
```

C++ initialize constructor requests memory for this object, initialize its members, and then link it at the end of the chain of headers. The type argument is in three classes:

NULL pointer – The type name is to be obtained from the first entity in the list

null string – Same as NULL pointer, except that when performing debug printout, the list is not printed

non–null string – The type name to be used when identifying the list

```
public: logical DEBUG_LIST::debug_list (
    FILE*                   // file name
    );
```

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

Destructor:

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

C++ destructor, deleting a DEBUG_LIST.

Methods:

```
public: int DEBUG_LIST::count () const;
```

Counts the entities in the list.

```
public: virtual void DEBUG_LIST::debug (
    ENTITY const*,          // entity
    FILE*                   // file name
    ) const;
```

Displays one entity from the DEBUG_LIST.

```
public: logical DEBUG_LIST::debug_list (
    FILE*                   // file name
    );
```

Displays each entity in the list, starting at the number_printed.

```
public: char const* DEBUG_LIST::entity_name ();
```

Allows unprivileged utilities to follow the list pointer to return the entity type.

```
public: ENTITY const* DEBUG_LIST::fetch (
    int                     // index number of entity
    ) const;
```

Obtains the indexed entity from the list.

```
public: int DEBUG_LIST::lookup (
    ENTITY const*,          // entity to lookup
    logical                 // add to list option
    );
```

Search for the given entity in the list, optionally adding it, and returning the index number.

```
public: DEBUG_LIST* DEBUG_LIST::next ();
```

Allows unprivileged utilities to follow the list pointer to return the next entity type.

```
public: virtual unsigned
    DEBUG_LIST::size_list () const;
```

Determines the total space occupied by all of the entities in the list. It does not include subsidiary structures.

Related Fncs:

clear_debug_lists, debug_add, debug_all, debug_box, debug_entity, debug_header, debug_int, debug_leader, debug_lists, debug_new_pointer, debug_old_pointer, debug_real, debug_sib_pointer, debug_size, debug_string, debug_title, debug_transform, format_pointer, size_all

DELTA_STATE

Class: History and Roll, SAT Save and Restore

Purpose: Retrieves a sequence of bulletin boards.

Derivation: DELTA_STATE : ACIS_OBJECT : –

SAT Identifier: “delta_state”

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

Description: This class returns a sequence of bulletin boards that change the modeler from the from_state to the to_state. The bulletin boards are created between successive calls to note_state. They are chained together in a singly-linked list beginning at bb_ptr.

Limitations: None

References:

KERN	BULLETIN_BOARD, DELTA_STATE_LIST, DELTA_STATE_user_data, HISTORY_STREAM
by KERN	BULLETIN_BOARD, HISTORY_MANAGER, HISTORY_STREAM

Data:

```
public BULLETIN_BOARD *bb_ptr;
```
Pointer to bulletin board.

```
public DELTA_STATE *next_ds;
```
A delta state whose from_state equals this to_state.

```
public DELTA_STATE *partner_ds;
```
Circular list of delta states with same from_state.

```
public DELTA_STATE *prev_ds;
```
The delta state whose to_state equals this from_state.

```
public DELTA_STATE* merged_with_ds;
```
The delta state this one merges into.

```
public DELTA_STATE_LIST* merged_states;
```
The delta states merged into this one.

```
public DELTA_STATE_user_data *user_data;
```
Pointer to optional application data attached to the DELTA_STATE.

```
public HISTORY_STREAM *owner_stream;
```
Allows history stream to be found from delta state.

```
public STATE_ID from_state;
```
Previous modeler state.

```
public STATE_ID this_state;
```
Set when state is noted.

```
public STATE_ID to_state;
```
Next modeler state to change to.

```
public char *name_str;
```
Name string for the DELTA_STATE.

```
public logical hidden;
```
Not counted for roll_n_states or max_states.

```
public logical rolls_back;
```
Delta records a backward change.

Constructor:

```
public: DELTA_STATE::DELTA_STATE (
    HISTORY_STREAM*          // history
        = NULL
    );
```

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

Creates a new state, with the to_state set to the current state, the from_state set to 0, and the bb_ptr set to NULL.

Destructor:

```
public: DELTA_STATE::~DELTA_STATE ();
```

C++ destructor, deleting a DELTA_STATE.(and their bulletins) that constitute the DELTA_STATE.

Methods:

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

Adds a new bulletin board to this delta state.

```
public: logical DELTA_STATE::backward () const;
```

Rolls the current state to the previous one in the DELTA_STATE.

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

Returns the bb_ptr.

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

Clear reference to this history from entities in the delta state.

```
public: void DELTA_STATE::compress ();
```

Performs compression on a given DELTA_STATE.

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

Outputs information about the DELTA_STATE to the debug file or to the specified file.

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

Outputs debug information about DELTA_STATE to the debug file or to the specified file.

```
public: void DELTA_STATE::debug_list (
    DELTA_STATE_LIST& dslist,   // delta state
    int id,                     // id
    int level,                  // level in state
    int ent_level               // entity level
        = 0,
    FILE*                       // file name
        = debug_file_ptr
    );
```

Prints debugging information with annotation support. The second and third arguments specify a branch of the entity derivation hierarchy to call debug_ent on, in addition to the normal bulletin board debugging information. For annotations we use ANNOTATION_TYPE and ANNOTATION_LEVEL.

```
public: void DELTA_STATE::debug_list (
    DELTA_STATE_LIST& dslist,   // delta state
    int level                   // level in state
        = 0,
    FILE*                       // file name
        = debug_file_ptr
    );
```

Aids in debugging the DELTA_STATE.

```
public: void DELTA_STATE::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 DELTA_STATE::find_bulletins (
    is_function tester,     // test 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: void DELTA_STATE::find_entities (
    enum ENTITY_TYPE,       // type of entity
    ENTITY_LIST&            // entity list
    );
```

Searches in entity list for a type of entity recorded in the bulletin.

```
public: logical DELTA_STATE::fix_pointers (
    ENTITY_ARRAY& elist,        // pointers to fix
    HISTORY_STREAM_LIST& hslist,// hist stream list
    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: logical DELTA_STATE::forward () const;
```

Rolls the current state to the next one in the DELTA_STATE.

```
public: STATE_ID DELTA_STATE::from () const;
```

Read only access to the originating DELTA_STATE.

```
public: const char* DELTA_STATE::get_name ();
```

Returns a name string of the DELTA_STATE.

```
public: DELTA_STATE_user_data*
    DELTA_STATE::get_user_data ();
```

Returns the user data that was attached to the DELTA_STATE.

```
public: logical DELTA_STATE::hide (
    logical h               // hidden or not
    );
```

Hides the given DELTA_STATE.

```
public: HISTORY_STREAM*
    DELTA_STATE::history_stream ();
```

Returns the owner of the stream.

```
public: STATE_ID DELTA_STATE::id () const;
```

Returns the STATE_ID.

```
public: logical DELTA_STATE::is_empty () const;
```

Returns true if the DELTA_STATE contains no BULLETINs.

```
public: logical DELTA_STATE::is_named (
    const char* n           // name of delta state
    );
```

Returns the name of the delta state.

```
public: void DELTA_STATE::merge_next ();;
```

Merge with the next DELTA_STATE, keeping all the BULLETINs and BULLETIN_BOARDs from both states in the correct order in this state, and then deleting next. If the next state had partners, indicating a branch, the branch is pruned as there would no longer be a sensible way to roll the model to states on the branch. Repeated calls can be used to compress any linear range of delta states with the same roll direction, into one state.

```
public: logical DELTA_STATE::mixed_streams (
    HISTORY_STREAM*& alternate_hs   // alternate
                                    // stream
    );
```

Checks for mixed history streams.

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

Returns the next DELTA_STATE.

```
public: DELTA_STATE* DELTA_STATE::partner () const;
```

Returns the partner DELTA_STATE.

```
public: DELTA_STATE* DELTA_STATE::prev () const;
```

Returns the previous DELTA_STATE.

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

Removes a new bulletin board from this delta state.

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

Reset the history stream on deletion.

```
public: logical DELTA_STATE::restore ();
```

Restores DELTA_STATE to the state provided by a previous bulletin board.

read_int	This state
read_int	Rolls back to
read_int	hidden
read_pointer	Pointer to record in SAT file for Previous DELTA_STATE
read_pointer	Pointer to record in SAT file for Next DELTA_STATE
read_pointer	Pointer to record in SAT file for Partner DELTA_STATE
read_pointer	Pointer to record in SAT file for Merged with DELTA_STATE
read_pointer	Pointer to record in SAT file for Owner HISTORY_STREAM
read_string_or_null	Name string
if (read_int)	If there is at least one bulletin board, represented by a number 1
BULLETIN_BOARD::restore	Restore an individual bulletin board
while (read_int)	While there are more bulletin boards, represented by a number 1
BULLETIN_BOARD::restore	Restore an individual bulletin board
read_int	Number of merged states
if(num_merged_states != 0)	
while(num_merged_states—)	
read_pointer	Pointer to record in SAT file to the merged DELTA_STATE.
read_data	Read until terminator

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

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

```
public: void DELTA_STATE::scan (
    DELTA_STATE_LIST& dslist// change state list
    ) const;
```

Adds connectees to the delta state list.

```
public: void DELTA_STATE::set_from (
    STATE_ID from_id        // state ID number
    );
```

Sets the identification of the from STATE_ID.

```
public: void DELTA_STATE::set_history_ptrs ();
```

Set history pointers.

```
public: void DELTA_STATE::set_name (
    const char* n           // name
    );
```

Changes name of DELTA_STATE.

```
public: void DELTA_STATE::set_to (
    STATE_ID to_id          // state ID number
    );
```

Sets the identification of the to_state to STATE_ID.

```
public: void DELTA_STATE::set_user_data (
    DELTA_STATE_user_data* d// pointer to data
    );
```

Permits users to change user data in DELTA_STATE.

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

Returns the size of the DELTA_STATE.

```
public: STATE_ID DELTA_STATE::to () const;
```

Read only access to the destination DELTA_STATE.

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

DELTA_STATE_LIST

Class: History and Roll

Purpose: Implements a variable length list of delta states.

Derivation: DELTA_STATE_LIST : –

SAT Identifier: None

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

Description: This class provides a constructor (which creates an empty list), a destructor, a function to add an delta state (only if not already there), a function to look up a delta state by pointer value, and a function to count the number of delta states listed. Also provides an overloaded "[ ]" operator for access by position. This was created using the LIST macro.

The functions are all essentially dummy; just indirecting through the header pointer. This is done in order to insulate the application programs completely from the implementation of lists.

The current implementation uses hashing so that look up is fast provided lists are not very long; it is also efficient for very short lists and for repeated lookups of the same delta state.

When a group of similar arguments must be returned, and the number of arguments is not known in advance, the system returns the arguments as an DELTA_STATE_LIST. The number of members of an DELTA_STATE_LIST can be found using the member function count, and individual members can be accessed with the subscript operator [ ].

The DELTA_STATE_LIST class is a variable length associative array of DELTA_STATE pointers. When using the subscript operator, a cast is required to change the DELTA_STATE pointer into the correct type. Many ACIS internal algorithms use DELTA_STATE_LIST including the part copy, save, and restore algorithms. DELTA_STATE_LIST is also useful in ACIS components and applications.

Limitations: NT, UNIX platforms only.

References: by KERN DELTA_STATE, HISTORY_STREAM

Data:

None

Constructor:

```
public: DELTA_STATE_LIST::DELTA_STATE_LIST ();
```

C++ constructor, creating a DELTA_STATE_LIST.

Destructor:

```
public: DELTA_STATE_LIST::~DELTA_STATE_LIST ();
```

C++ destructor, deleting a DELTA_STATE_LIST.

Methods:

```
public: int DELTA_STATE_LIST::add (
    DELTA_STATE* e          // state to add
    );
```

Add a delta state to the list and returns its index number.

```
public: void DELTA_STATE_LIST::clear ();
```

Empties a list for construction of a new one.

```
public: int DELTA_STATE_LIST::count () const;
```

Returns the number of delta states in a given list.

```
public: void DELTA_STATE_LIST::init () const;
```

Initializes the correct position in the list for next.

```
public: int
    DELTA_STATE_LIST::iteration_count () const;
```

Counts how many delta states there are in the list not including deleted entries. Uses the iterator.

```
public: int DELTA_STATE_LIST::lookup (
    DELTA_STATE const* ce   // delta state
    ) const;
```

Looks up a delta state in the debug list.

```
public: DELTA_STATE* DELTA_STATE_LIST::next () const;
```

Returns the next undeleted entry.

```
public: DELTA_STATE* DELTA_STATE_LIST::operator[] (
    int i                   // index number of item
    ) const;
```

Returns a given item from a list.

```
public: int DELTA_STATE_LIST::remove (
    DELTA_STATE const* ce   // state to remove
    );
```

Deletes a delta state from the list; however, it does not free space.

Related Fncs:

None

discontinuity_info

Class: Construction Geometry, SAT Save and Restore

Purpose: Stores discontinuity information for a curve or surface.

Derivation: discontinuity_info : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerngeom/curve/discinfo.hxx

Description: Used to store parameter values at which a curve has a discontinuity in some derivative, or at which a surface has a line of discontinuity in some derivative. This class stores discontinuity information for a curve or surface. Only C1, C2, and C3 discontinuities are stored since we are not interested in C4 discontinuities and above.

Limitations: None

References: by KERN int_cur, intcurve, spl_sur, spline

Data:

None

Constructor:

```
public: discontinuity_info::discontinuity_info ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore.

```
public: discontinuity_info::discontinuity_info (
    const discontinuity_info& old   // instance to
                                    // copy
    );
```

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

Destructor:

```
public: discontinuity_info::~discontinuity_info ();
```

C++ destructor for discontinuity_info which deallocates memory.

Methods:

```
public: void discontinuity_info::add_discontinuity (
    double value,           // value to add
    int order               // where
    );
```

Adds a discontinuity value to the list. In periodic cases, it's up to the application to ensure that the values are in the same parameter period. but we check this here.

```
public: const double*
    discontinuity_info::all_discontinuities (
    int& n_discont,         // number of
                            // discontinuities
    int order               // order of disc
    );
```

Accesses the all discontinuities list, returning a read–only array.

```
public: void discontinuity_info::debug (
    char const*,            // title line
    FILE*                   // file pointer
    ) const;;
```

Outputs a title line and the details of the <class_name> for inspection to standard output or to the specified file.

```
public: const double*
    discontinuity_info::discontinuities (
    int& n_discont,          // number of
                             // discontinuities
    int order                // order of disc.
    ) const;
```

This is an access function that returns a read-only array.

```
public: int discontinuity_info::discontinuous_at (
    double t                 // where to test
    ) const;
```

States whether a particular parameter value is a discontinuity.

```
public: void discontinuity_info::merge (
    discontinuity_info& old // instance to merge
    );
```

Merges two discontinuity_info entities, keeping the supplied one unchanged. The entries from the second are added into the first one at a time. Not very efficient, but we don't expect these arrays to contain much data.

```
public: void discontinuity_info::negate ();
```

Negates the data for the discontinuity_info.

```
public: discontinuity_info&
    discontinuity_info::operator= (
    const discontinuity_info& old   // list to use
    );
```

Sets the pointer to the current discontinuity_info object to the input object pointer.

```
public: double discontinuity_info::period () const;
```

Periodicity. This class handles periodicity, although it does not know the "base range" of the parameters. When building up the list using add_discontinuity, the application must ensure that the parameters are in the correct range (e.g. the param_range of a curve). Once the list is built, this class will accept parameters out of the base range and interpret them as though they were in range.

```
public: void
    discontinuity_info::remove_discontinuity (
    double value            // value to remove
    );
```

Removes a discontinuity value from the list.

```
public: void discontinuity_info::reparam (
    double a,               // slope
    double b                // offset
    );
```

Makes a linear change of parameter to all the discontinuity values (new value = a * old_value + b).

```
public: void discontinuity_info::reset ();
```

Deletes discontinuity information and reinitializes the data fields to zero.

```
public: void discontinuity_info::restore ();
```

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.

```
read_int                        // number of C1 discontinuities
if (n_C1 > 0)                   // if any C1 discontinuities
     foreach ( n_C1 )           // for each one
          read_real             // read the discontinuity
read_int                        // number of C2 discontinuities
if ( n_C2 > 0 )                 // if any C2 discontinuities
     foreach ( n_C2 )           // for each one
          read_real             // read the discontinuity
read_int                        // number of C3 discontinuities
if ( n_C3 > 0 )                 // if any C3 discontinuities
     foreach ( n_C3 )           // for each one
     read_real                  // read the discontinuity
read_real                       // total discontinuities
```

```
public: void discontinuity_info::save () const;
```

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 while storing information to a save file.

```
public: void discontinuity_info::set_periodic (
    double per              // period
    );
```

Establishes the periodicity.

```
public: void discontinuity_info::shift (
    double incr             // amount to shift
    );
```

Shifts all of the discontinuity values by a constant amount.

```
public: discontinuity_info
    discontinuity_info::split (
    double param            // parameter value
    );
```

Split the discontinuity lists into two at a given parameter value. Like curve::split, the return value contains the initial values (before the split parameter), and the original discontinuity_info contains the others (after the split parameter). If the split parameter is itself a discontinuity, it is removed from the list.

Internal Use: full_size

Related Fncs: test_discontinuity