U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 4353

National PDES Testbed
Report Series

NIST STEP Working Form Programmer's Reference

Revised November, 1990

NATIONAL PDES TESTBED

NIST

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 4353

National PDES Testbed

NATIONAL PDES TESTBED

NIST STEP Working Form Programmer's Reference

Stephen Nowland Clark

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary of Commerce

National Institute of Standards and Technology
John W. Lyons, Director

November 29, 1990

DEPARTMENT OF COMMERCE
UNITED STATES OF AMERICA

Table Of Contents

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied

Unix is a trademark of AT&T Technologies, Inc.

NIST STEP Working Form Programmer's Reference

Stephen Nowland Clark

1 Introduction

The NIST STEP physical file parser [Clark90c], and its associated STEP parser, STEPparse, are Public Domain tools for manipulating product models stored in the STEP physical file format [Altemueller88]. These tools are a part of the NIST PDES Toolkit [Clark90a], and are geared particularly toward building STEP translators. This reference manual discusses the internals of the STEP Working Form, including STEPparse. The reader is assumed to be familiar with the design of the Toolkit ([Clark90a], [Clark90b], [Clark90c]). In some cases, technical knowledge of the Express Working Form [Clark90e] is also required.

The STEP Working Form relies on the NIST Express Working Form [Clark90b] as an in-core data dictionary, which provides a context in which STEP models can be interpreted. The tight dependency of the STEP Working Form abstractions on those of the Express Working Form is due to the schema-independent nature of the former. The STEP Working Form, and, in particular, STEPparse, contain no knowledge of any particular information model. Applications built on these tools can thus manipulate STEP product models in the context of any number of Express information models without requiring recompilation.

1.1 Context

The PDES (Product Data Exchange using STEP) activity is the United States' effort in support of the Standard for the Exchange of Product Model Data (STEP), an emerging international standard for the interchange of product data between various vendors' CAD/CAM systems and other manufacturing-related software [Smith88]. A National PDES Testbed has been established at the National Institute of Standards and Technology to provide testing and validation facilities for the emerging standard. The Testbed is funded by the CALS (Computer-aided Acquisition and Logistic Support) program of the Office of the Secretary of Defense. As part of the testing effort, NIST is charged with providing a software toolkit for manipulating PDES data. This NIST PDES Toolkit is an evolving, research-oriented set of software tools. This document is one of a set of reports which describe various aspects of the Toolkit. An overview of the Toolkit is provided in [Clark90a], along with references to the other documents in the set.

For further information on the STEP Working Form or other components of the Toolkit, or to obtain a copy of the software, use the attached order form.

2 STEPparse Control Flow

A STEPparse translator consists of two separate passes: parsing and output generation. The first pass builds an instantiated `Product` representing the product model specified in the STEP input file. This `Product` can then be traversed by an output module in the second pass, producing whatever report is desired. It is anticipated that users will need output formats other than those provided with the NIST Toolkit. The process of writing a report generator for a new output format is discussed in detail in section 4.

2.1 First Pass: Parsing

The first pass of a STEPparse translator is a very simple parser. The STEPparse grammar itself is independent of any conceptual schema. The lexical analyzer recognizes any entity class name simply as an identifier; the actions associated with rules in the grammar then interpret this name as referring to a particular Express entity, and construct appropriate objects. As each construct is parsed, it is added to the Working Form. Because the STEP physical file format does not allow forward references to as-yet-undefined entity instances, all symbol references can be (and are) resolved during this parsing pass, so that no symbol resolution pass is required.

The STEPparse parser is written using the standard Unix™ parser generation languages, Yacc and Lex. The grammar is processed by Bison, the Free Software Foundation's[1] implementation of Yacc. The lexical analyzer is produced by Flex[2], a fast, Public Domain implementation of Lex.

2.2 Second Pass: Output Generation

The report or output generation pass manages the production of the various output files. In the dynamically linked version of STEPparse, this pass loads successive output modules dynamically, calling each to traverse the Working Form. The dynamic linking mechanism is discussed briefly in [Clark90d]. It is also possible to build a statically linked translator, with a particular output module loaded in at build time; this is, in fact, the only mechanism available in an environment which is not derived from BSD 4.2 Unix.

A report generator is an object module, most likely written in C, which has been compiled as a component module for a larger program (i.e., with the `-c` option to a Unix C compiler). In the dynamically linked version, the object module is linked into the running parser, and its entry point (by convention a function called `print_file()`) is

---

1. The Free Software Foundation (FSF) of Cambridge, Massachusetts is responsible for the GNU Project, whose ultimate goal is to provide a free implementation of the Unix operating system and environment. These tools are not in the Public Domain: FSF retains ownership and copyright priviledges, but grants free distribution rights under certain terms. At this writing, further information is available by electronic mail on the Internet from gnu@prep.ai.mit.edu.
2. Vern Paxson's Fast Lex is usually distributed with GNU software, although, being in the Public Domain, it is not an FSF product and does not come under the FSF licensing restrictions.

called. The code of this module consists of calls to STEP Working Form access functions and to standard output routines. Chapter 4 provides a detailed description of the creation of a new output module.

3 Working Form Implementation

As in the Express Working Form [Clark90e], the Instance abstraction is implemented as an `Object` header block which ultimately points to a private `struct Instance`. This C structure contains the real definition of the abstraction, but is never manipulated directly outside of the Instance module. Product is implemented as a pointer to a private structure, `struct Product`.

Most stylistic and other conventions from the Express Working Form are equally valid for STEP; they are reiterated here for emphasis.

3.1 Primitive Types

The STEP Working Form makes use of several modules from the Toolkit general libraries, including the Error and Linked_List modules. These are described in [Clark90d].

3.2 STEP Working Form Manager Module

In addition to the abstractions discussed in [Clark90c], `libstep.a` contains one more (conceptual) module, the package manager. Defined in `step.c` and `step.h`, this module includes calls to intialize the entire STEP (and Express) Working Form package, and to run each of the passes of a STEPparse translator.

3.3 Code Organization and Conventions

Each abstraction is implemented as a separate module. Modules share only their interface specifications with other modules. A module Foo is composed of two C source files, `foo.c` and `foo.h`. The former contains the body of the module, including all non-inlined functions. The latter contains function prototypes for the module, as well as all type and macro definitions. In addition, global variables are defined here, using a mechanism which allows the same declarations to be used both for `extern` declarations in other modules and the actual storage definition in the declaring module. These globals can also be given constant initializers. Finally, `foo.h` contains inline function definitions. In a compiler which supports inline functions, these are declared `static inline` in every module which includes `foo.h`, including `foo.c` itself. In other compilers, they are undefined except when included in `foo.c`, when they are compiled as ordinary functions. `foo.c` resides in `~pdes/src/step/`; `foo.h` in `~pdes/include/`.

The type defined by module Foo is named `Foo`, and its private structure is `struct Foo`. Access functions are named as `FOOfunction()`; this function prefix is abbreviated for longer abstraction names, so that access functions for type `Foolhardy_Bartender` might be of the form `FOO_BARfunction()`. Some functions may be implemented as macros; these macros are not distinguished typographically from other functions, and are guaranteed not to have unpleasant side effects like evaluating arguments more than once. These macros are thus virtually indistinguishable from functions. Functions which are intended for internal use only are named `FOO_function()`, and are usually `static` as well, unless this is not possible. Global variables are often named `FOO_variable`; most enumeration identifiers and constants are named `FOO_CONSTANT` (although these latter two rules are by no means universal). For example, every abstraction defines a constant `FOO_NULL`, which represents an empty or missing value of the type.

3.4 Memory Management and Garbage Collection

In reading various portions of the STEP Working Form documentation, one may get the impression that the Working Form does some reasonably intelligent memory management. This is not entirely true. The NIST PDES Toolkit is primarily a research tool. This is especially true of the Express and STEP Working Forms. The Working forms allocate huge chunks of memory without batting an eye, and this memory often is not released until an application exits. Hooks for doing memory management do exist (e.g., OBJfree`()` and reference counts), and some attempt is made to observe them, but this is not given high priority in the current implementation.

3.5 Instance

The Instance abstraction is the basic building block of the STEP Working Form. An `Instance` is created for each unit of value in a PDES/STEP product model: each entity instance, aggregate, integer, string, etc. On the surface, this would seem to be a reasonably straightforward module to implement: each `Instance` has an optional name, a `Type`, and a value. The value may be simple or structured; in either case, it basically comes down to a pointer - either to an array of `Instances`, or to an integer, real, string, etc.

The definition of an instance is encapsulated in a private `struct Instance`, which is defined thus:

```
struct Instance {
     Type      type;
     Generic   user_data;
     union {
          Constant  enumeration;
          Integer   integer;
          Logical   logical;
          Real      real;
          String    string;
          Instance* entity;
          Aggregate aggregate;
    }         value;
};
```

The first two fields are pretty straightforward. Note that user_data is a generic pointer field. In strict ANSI C, only a pointer can be safely stored into this field and later retrieved; it is safest to only store pointers in this field. In particular, the age-old trick of casting pointers and integers back and forth, never completely portable, is now officially frowned upon.

The value union is where things get tricky. This field contains the actual value of the object represented. Unstructured types (numbers, logicals, and strings) are represented directly; e.g., instance.value.integer contains an integer, and instance.value.string, a character pointer. The value of an enumeration instance is represented as a Constant, which will be an element of the appropriate enumeration. The integer representation of this enumeration element can be retrieved by calling (int)CSTget_value(instance.value.enumeration).

An entity instance's value field, value.entity, is a pointer to the base of an array of instances. Each element of this array corresponds to an attribute of the entity; attributes appear in the same order as in a PDES/STEP physical file, with empty attributes explicitly represented by INSTANCE_NULL. The offset to a particular attribute value is retrieved from the Express data dictionary by calling ENTITYget_attribute_offset(entity, attribute), where entity is the entity class of the instance in question and attribute is the Variable representing the attribute to be located.

The most convoluted instance value representation is that for aggregates. An aggregate value is represented as a pointer to a struct Aggregate, defined as

```
struct Aggregate {
     int            low;
     int            high;
     Expression     max;
     Instance*      contents;
};
```

The last field, contents, holds the actual contents of the aggregate, as an array of Instances. The low field provides a lower bound on allowable indices into this array, and doubles as a logical offset to the first element of the array. This value is 1 for any non-array aggregate. Thus, when low is 1, some_aggregate[1] is found at contents[0]. Similarly, in an array whose low is 10, the some_array[12] is found at contents[12-10 = 2]. low remains constant in any particular aggregate instance. The high field gives an upper bound on the indices of currently filled slots in an aggregate instance. Every index into the aggregate beyond high which is in bounds is guaranteed to return INSTANCE_NULL. The end result is that a loop of the form for (i = low; i <= high; ++i) <use contents[i-low]> will always hit all of the elements of an aggregate. This function of offsetting by the lower bound is bundled into the various aggregate indexing functions of the working form (INSTaggr_at(), INSTlist_insert(), etc.), so that the indices which a user sees will be the ones which would be expected based on the Express model. In the current implementation, high in an aggregate whose type (from Express) gives a finite upper bound always remains constant at this bound. In the case of an aggregate with no specified upper bound, however, `high` may vary with the number of elements actually in the aggregate. The expression (from Express) giving the absolute upper bound on an aggregate is cached in `aggregate->max`. `high` is never allowed to be greater than the value of this expression.

The two calls `INSTaggr_at()` and `INSTaggr_at_put()` can be used with any kind of aggregate, although they are intended to be used primarily for building general aggregates which will later be `INSTtype_cast()` into specific types of aggregates. This is how STEPparse builds aggregates, since it is considerably easier than figuring out at parse time what type of aggregate should be built. The various class-specific manipulations (list concatenation, set intersection, etc.) are provided by calls requiring aggregates of a particular class: `INSTlist_concat()`, `INSTset_intersect()`, etc. It should be noted that the calls for combning aggregates are destructive: each modifies its first argument to hold its computed result. In general, the two arguments may safely be set equal. Exceptions are noted in the individual function specifications.

Finally, a word about type conversion (also known as casting, as in C). Type conversions of existing `Instances` are handled by `INSTtype_cast(Instance, Type, Error*)`. Only certain conversions are allowed; other attempted casts leave the `Instance` unchanged and return an error code. Clearly, any `Instance` can trivially be cast into its own type. The different numeric types can be cast about at will. A general aggregate can be cast into any specific aggregate class; otherwise, an aggregate can only be cast into another aggregate type of the same class: an array cannot be cast into a set, etc. Each element of the aggregate being cast must, of course, be recursively cast into the appropriate base type; each of these conversions is subject to the same rules as any other cast. Finally, an entity `Instance` can be converted into an instance of a supertype of its class, or into an instance of a `SELECT` type containing some type to which it can be cast. These casts of entity instances in fact do not modify the `Instance` being cast.

3.6 Product

A product in STEP contains a large number of interrelated entity instances, and is represented by the `Product` abstraction. Each `Product` is named, and includes a pointer to the Express model which provides the scope in which its component `Instances` are defined. These component instances can be retrieved from the `Product` in several ways: a specific (external) entity instance can be retrieved by name; a `Linked_List` of all of the (external) entity instances in the `Product` can be requested; or a particular entity class in the `Product`'s conceptual schema can be queried for all of its instances (note that this last method retrieves both internal and external entity instances). Internal (embedded) entity instances and non-entity `Instances` must appear as attribute values or aggregate elements somewhere in the `Product`, and are only accessible via `ENTITYget_instances()` and component retrieval from the containing `Instances`.

The above three access methods are supported by storing three references to each `Instance` in a `Product`. When an `Instance` is added to a `Product`, it is added to the end of the list of external instances. This list preserves the order in which the `In-`

`stances` were added to the `Product`, and so is appropriate for applications, such as writing a STEP physical file, which require that there be no forward references to as-yet-undefined `Instances`. Each external `Instance` is also added to a dictionary which the `Product` maintains, to allow retrieval by name. And when an entity instance is first created, it is added to the instance list of its class.

4 Writing An Output Module

We now turn to the topic of actually writing a report generator. The end result of this process will be an object module (under Unix, a `.o` file) which can be loaded into STEPparse. This module contains a single entry point which traverses a given `Product` and writes its output to a particular file. The conceptual entry point is conventionally called `print_file()`, while the physical entry point, which simply dispatches to `print_file()`, is called `entry_point()`.

In most cases, there will be a one-to-one correspondence between Instances in the instantiated Working Form and records to be written on the output. When this is the case, the meat of the report generator can be made fairly simple. Since a list of all of the Instances in the Working Form is available, it is easy to iterate over this list and output each Instance in sequence:

```
STEPprint(Product product, FILE* file)
{
     Linked_List list;
     list = PRODget_contents(product);
     LISTdo(list, inst, Instance)
          INSTprint(inst, file);
     LISTod;
}
```

The only remaining problem is to write a function `INSTprint()` which emits the output record for a single Instance. Given the variety of types of Instances, this function will probably be controlled by a large `switch` statement, selecting on the Instance's type class (numbers, strings, and aggregates all have to be printed differently). Code to deal with multi-dimensional arrays an internal/external entity references can get tricky, and should be written carefully. An example of a fairly simple report generator is that used by STEPparse-QDES. The source code for this module is in `~pdes/src/stepparse_qdes/step_output_smalltalk.c`.

4.1 Layout of the C Source

The layout of the C source file for a report generator which will be dynamically loaded is of critical importance, due to the primitive level at which the load is carried out. The very first piece of C source in the file must be the `entry_point()` function, or the loader may find the wrong entry point to the file, resulting in mayhem. Only comments may precede this function; even an `#include` directive may throw off the loader. An output module is normally layed out as shown:

```
void
entry_point(void* product, void* file)
{
     extern void print_file();
     print_file(product, file);
}

#include "step.h"

... actual output routines . ..

void
print_file(void* product, void* file)
{
     print_file_header((Product)product,
          (FILE*)file);
     STEPprint(product, file);
     print_file_trailer((Product)product,
          (FILE*)file);
}
```

The `print_file()` function will probably always be quite similar to the one shown, although in many cases, the file header and/or trailer may well be empty, eliminating the need for these calls. In this case, `STEPprint()` and `print_file()` will probably become interchangeable.

Having said all of the above about templates, code layout, and so forth, we add the following note: In the final analysis, the output module really is a free-form piece of C code. There is one and only one rule which must be followed: The entry point (according to the `a.out` format) to the `.o` file which is produced when the report generator is compiled must be appropriate to be called with a `Product` and a `FILE*`. The simplest (and safest) way of doing this is to adhere strictly to the layout given, and write an `entry_point()` routine which jumps to the real (conceptual) entry point. But any other convention which guarantees this property may be used.

4.2 Output Module Linkage Mechanisms

One of the powers of STEPparse is the flexibility which it gives a user with regard to generating output. An important component of this flexibility on BSD Unix systems is the dynamic loading of output modules. Both static and dynamic binding of output modules are supported by STEPparse. This is implemented by providing two distinct versions of the Working Form manager. Code common to both versions (including initialization code and the STEPparse parser itself) is found in `step.c`, which is included by each of the distinct manager modules. The static linking version of the output pass, without any output module, is in `step_static.c`, and the corresponding `step_static.o` is included in `libstep.a`, making it the default; the dynamic loading version is in `step_dynamic.c`.

Since `step_static.o` and `step_dynamic.o` both define the function `STEPreport()`, only one is linked into any given executable. This selection is what determines whether a STEPparse translator links in output modules statically or dynamically. By default, the linkage mechanism will be `step_static.o`, which actually appears in the Working Form library. This choice can be overridden by placing `step_dynamic.o` before `libstep.a` in the link command. Note that a suitable output module (`.o` file) must appear *after* `step_static.o` in the linker's argument list when a statically linked translator is being built. For more information on how to build a report generator into a STEPparse translator, see [Clark90d].

5 Working Form Routines

The remainder of this manual consists of specifications and brief descriptions of the access routines and associated error codes for the STEP Working Form. The error codes are manipulated by the Error module [Clark90d]. Each subsection below corresponds to a module in the Working Form library. The Working Form Manager module is listed first, followed by the remaining data abstractions in alphabetical order.

5.1 Working Form Manager

Procedure: STEPinitialize
Parameters: Error* errc - buffer for error code
Returns: void
Description: Initialize the STEP Working From package. In a typical STEP translator, this is called by the default `main()` provided in the Working Form library. Other applications should call this function at initialization time.
Errors: none

Procedure: STEPparse
Parameters: String filename - the name of the file to be parsed
Express data_model - conceptual schema (as produced by `EXPRESSpass_2()`)
Returns: Product - the product model parsed
Description: Parse a STEP physical file into the Working Form

Procedure: STEPreport
Parameters: Product product - the product to output
Returns: void
Description: Invoke one or more report generators for a STEP Working Form model.
Description: Invoke one (or more) report generator(s), according to the selected linkage mechanism.

5.2 Instance

Procedure: INSTaggr_at
Parameters: Instance instance - instance to examine
int index - index of requested element
Error* errc - buffer for error code
Returns: Instance - value at requested position
Description: Retrieves the value at some position in an aggregate. Note that the calls which are specific to a particular aggregate class are much to be preferred.
Errors: `ERROR_index_out_of_range` - the index is outside of the bounds of the aggregate

Procedure: INSTaggr_at_put
Parameters: Instance instance - instance to modify
int index - index at which to put element
Instance value - value to insert
Error* errc - buffer for error code
Returns: void
Description: Store a value into an aggregate instance. Note that the calls which are specific to a particular aggregate class are much to be preferred.
Errors: `ERROR_index_out_of_range` - the index is outside of the bounds of the aggregate

Procedure: INSTaggr_lower_bound
Parameters: Instance instance - instance to examine
Error* errc - buffer for error code
Returns: int - the lower bound of the instance
Description: Retrieves the lower bound of an aggregate instance. For an array, this is the index of the first element of the array. For other aggregates, it is 1.
Errors: none

Procedure: INSTaggr_upper_bound
Parameters: Instance instance - instance to examine
Error* errc - buffer for error code
Returns: int - the upper bound of the instance
Description: Retrieves the upper bound of an aggregate instance. For an aggregate with a constrained size, this is the value of the upper limit or index. For an aggregate with an infinite upper bound, the value returned is guaranteed to be larger than the highest index of a filled slot in the aggregate.
Errors: none

Procedure: INSTarray_at
Parameters: Instance array - array to examine
int index - index of requested element
Error* errc - buffer for error code
Returns: Instance - value at requested position
Description: Retrieves the value at some position in an array.
Errors: `ERROR_index_out_of_range` - the index is outside of the bounds of the aggregate

Procedure: INSTarray_at_put
Parameters: Instance array - array to modify
int index - index at which to put element
Instance value - value to insert
Error* errc - buffer for error code
Returns: void
Description: Store a value into an array instance.
Errors: `ERROR_index_out_of_range` - the index is outside of the bounds of the aggregate

Procedure: INSTbag_add
Parameters: Instance bag - bag to modify
Instance item - item to add
Error* errc - buffer for error code
Returns: void
Description: Inserts an instance into a bag.
Errors: `ERROR_bag_full` - there is no more room in the bag

Procedure: INSTbag_includes
Parameters: Instance bag - bag to test
Instance item - item to test for
Error* errc - buffer for error code
Returns: Boolean - does this bag contain this item?
Errors: none

Procedure: INSTbag_intersect
Parameters: Instance bag - bag to intersect into
Instance unitee - bag to intersect with
Error* errc - buffer for error code
Returns: void
Description: Intersects two bags. This operation is destructive: the first bag holds the resulting intersection on return.
Errors: none

Procedure: INSTbag_remove
Parameters: Instance bag - bag to remove from
Instance item - item to remove
Error* errc - buffer for error code
Returns: void
Description: Remove a single occurence of some item from a bag, if it appears.
Errors: none

Procedure: INSTbag_remove_all
Parameters: Instance bag - bag to remove from
Instance remove - bag of items to remove
Error* errc - buffer for error code
Returns: void
Description: Removes all items in a bag from some other bag. This is bag subtraction. This operation is destructive: the first bag holds the result on return.
Errors: none

Procedure: INSTbag_subset
Parameters: Instance bag - bag to test as superset
Instance subset - bag to test as subset
Error* errc - buffer for error code
Returns: Boolean - does the first bag contain the second as a subset?
Description: This implementation is not completely correct. In particular, the following returns `true`: `INSTbag_subset({a, b, c}, {a, a})`.
Errors: none

Procedure: INSTbag_unite
Parameters: Instance bag - bag to unite onto
Instance unitee - bag to unite with
Error* errc - buffer for error code
Returns: void
Description: Adds the contents of a bag to another bag. This operation is destructive: the first bag holds the resulting union on return. It is not safe to unite a bag with itself.
Errors: none

Procedure: INSTcreate
Parameters: Type type - type to instantiate
Error* errc - buffer for error code
Returns: Instance - a new, empty instance of the given type
Errors: `ERROR_cannot_instantiate` - the type given cannot be instantiated (e.g., `Generic`)

Procedure: INSTcreate_entity
Parameters: Entity entity - entity class to instantiate
Linked_List attributes - list of attribute values
int line - source line number of the instance to be created
Error* errc - buffer for error code
Returns: Instance - a new entity instance, as described
Description: A new instance of the specified entity type is created. There should be a one-to-one correspondence between the values on the attribute value list and the list of attributes for the entity being instantiated.
Errors: `ERROR_insufficient_attributes` - not enough attribute values in the list provided
`ERROR_too_many_attributes` - too many attribute values in the list provided

Procedure: INSTcreate_ud_entity
Description: Create a user-defined entity. This procedure is not yet implemented.

Procedure: INSTfast_get_attribute
Parameters: Instance instance - instance to examine
Variable attribute - attribute to retrieve
Error* errc - buffer for error code
Returns: Instance - value of attribute
Description: Retrieves the value of an attribute from an entity instance. This call is faster than `INSTget_attribute()` when the caller already has the actual attribute record for the desired attribute, rather than simply having its name (as expected by `INSTget_attribute()`).
Errors: none

Procedure: INSTfast_put_attribute
Parameters: Instance instance - instance to modify
Variable attribute - attribute to store into
Instance value - value to store into attribute
Error* errc - buffer for error code
Returns: void
Requires: TYPEget_class(INSTget_type(instance)) == TYPE_ENTITY
Description: Store a value into an attribute of an entity instance. This call is faster than `INSTput_attribute()` when the caller already has the actual attribute record for the desired attribute, rather than simply having its name (as expected by `INSTput_attribute()`).
Errors: Same as for `INSTput_attribute()`.

Procedure: INSTget_attribute
Parameters: Instance instance - instance to examine
String attributeName - name of attribute to retrieve
Error* errc - buffer for error code
Returns: Instance - value of the named attribute
Description: Retrieves the value of a named attribute from an entity instance. This call is the slower method for retrieving an attribute value. If the actual attribute recored is already available, use `INSTfast_get_attribute()` instead.
Errors: none

Procedure: INSTget_name
Parameters: Instance instance - instance to examine
Returns: String - the instance's name
Description: Retrieves the name of an instance. Unnamed instances, which would normally be embedded entities and non-entities, yield `STRING_NULL`.
Errors: none

Procedure: INSTget_type
Parameters: Instance instance - instance to examine
Returns: Type - the type of the instance
Errors: none

Procedure: INSTget_user_data
Parameters: Instance instance - instance to examine
Error* errc - buffer for error code
Returns: Generic - value of user data field for this instance
Errors: none

Procedure: INSTget_value
Parameters: Instance instance - instance to examine
Error* errc - buffer for error code
Returns: Generic - the instance's value
Description: Retrieves the value of a single-valued instance. The value returned will be a `char*` for a string object, a `Constant` for an enumeration object, and a pointer to an `int`, `double`, or `Boolean` for an integer, real, or logical object, respectively. See `INSTarray_at()`, `INSTbag_includes()`, `INSTlist_at()`, and `INSTset_at()` to read from an aggregate. See `INSTget_attribute()` to read from an entity instance.
Errors: none

Procedure: INSTinitialize
Parameters: Error* errc - buffer for error code
Returns: void
Description: Initialize the Instance module. This is called by STEPinitialize().
Errors: none

Procedure: INSTis_external
Parameters: Instance instance - instance to examine
Returns: Boolean - is this an external instance (non-embedded entity)?
Errors: none

Procedure: INSTis_internal
Parameters: Instance instance - instance to examine
Returns: Boolean - is this an internal instance (embedded entity)?
Errors: none

Procedure: INSTlist_add_first
Parameters: Instance list - list to modify
Instance item - item to insert
Error* errc - buffer for error code
Returns: void
Description: Adds an item to the beginning of a list. This function is not yet implemented.
Errors: none

Procedure: INSTlist_add_last
Parameters: Instance list - list to modify
Instance item - item to insert
Error* errc - buffer for error code
Returns: void
Description: Adds an item to the end of a list. This function is not yet implemented.
Errors: none

Procedure: INSTlist_concat
Parameters: Instance list - list to concatenate onto
Instance tail - list to concatenate
Error* errc - buffer for error code
Returns: void
Description: Concatenate a list onto the end of another. This operation is destructive: the first list is modified so that it includes a copy of the second. Changes to the second will not appear in the first. This function is not yet implemented.
Errors: none

Procedure: INSTput_attribute
Parameters: Instance instance - instance to modify
String attributeName - name of attribute to store into
Instance value - value to store into attribute
Error* errc - buffer for error code
Returns: void
Requires: TYPEget_class(INSTget_type(instance)) == TYPE_ENTITY
Description: Stores a value into a named attribute of an entity instance. This call is the slower method for storing into an attribute. If the actual attribute record is available, for example from traversing the `Entity`'s attribute list, use `INSTfast_put_attribute()` instead.
Errors: `ERROR_aggregate_expected` - value given for an aggregate was not an aggregate
`ERROR_array_expected` - value given for an array was not an array
`ERROR_bag_expected` - value given for a bag was not a bag
`ERROR_entity_expected` - value given for an entity was not an entity
`ERROR_external_expected` - an `external` attribute was given an internal (embedded) entity as a value
`ERROR_inappropriate_entity` - the entity given as a value was not of an expected class
`ERROR_integer_expected` - value given for an integer was not an integer
`ERROR_internal_expected` - an `internal` attribute was given an external entity reference as a value
`ERROR_list_expected` - value given for a list was not a list
`ERROR_logical_expected` - value given for a logical was not a logical
`ERROR_number_expected` - value given for a number was not a number
`ERROR_set_expected` - value given for a set was not a set
`ERROR_string_expected` - value given for a string was not a string
`ERROR_incompatible_types` - the value given is not of the expected type, in some way not covered by any of the above messages

Procedure: INSTput_name
Parameters: Instance instance - instance to modify
String name - name for instance
Returns: void
Description: Sets the name (identifier) of an instance; normally, only entity instances which are not embedded are named.
Errors: none

Procedure: INSTput_user_data
Parameters: Instance instance - instance to modify
Generic value - user data value for instance
Error* errc - buffer for error code
Returns: Generic - old value of user data field for this instance
Description: Stores a value into an instance's user data field
Errors: none

Procedure: INSTput_value
Parameters: Instance instance - instance to modify
Generic value - value for instance
Error* errc - buffer for error code
Returns: void
Description: Sets the value of a single-valued instance. The value given should be a `char*` for a string object. For an integer, real, or logical object, it should be an `int*`, `double*`, and `Boolean*`, respectively. For an enumeration object, the value given should be of type `Constant`. See `INSTaggr_at_put()`, `INSTarray_at_put()`, `INSTbag_add()`, `INSTlist_add_first()`, `INSTlist_add_last()`, and `INSTset_add()` to store into an aggregate. See `INSTput_attribute()` to store into an entity instance.
Errors: none

Procedure: INSTset_add
Parameters: Instance set - set to modify
Instance item - item to add
Error* errc - buffer for error code
Returns: void
Description: Inserts an instance into a set, if it is not already present.
Errors: `ERROR_set_full` - there is no more room in the set

Procedure: INSTset_includes
Parameters: Instance set - set to test
Instance item - item to test for
Error* errc - buffer for error code
Returns: Boolean - does this set contain this item?
Errors: none

Procedure: INSTset_intersect
Parameters: Instance set - set to intersect into
Instance with - set to intersect with
Error* errc - buffer for error code
Returns: void
Description: Intersects two sets. This operation is destructive: the first set holds the resulting intersection on return.
Errors: none

Procedure: INSTset_remove
Parameters: Instance set - set to remove from
Instance item - item to remove
Error* errc - buffer for error code
Returns: void
Description: Remove an item from a set, if it appears.
Errors: none

Procedure: INSTset_remove_all
Parameters: Instance set - set to remove from
Instance remove - set of items to remove
Error* errc - buffer for error code
Returns: void
Description: Removes all items in a set from some other set. This is set subtraction. This operation is destructive: the first set holds the result on return.
Errors: none

Procedure: INSTset_subset
Parameters: Instance set - set to test as superset
Instance subset - set to test as subset
Error* errc - buffer for error code
Returns: Boolean - does the first set contain the second as a subset?
Errors: none

Procedure: INSTset_unite
Parameters: Instance set - set to unite onto
Instance unitee - set to unite with
Error* errc - buffer for error code
Returns: void
Description: Forms the union of two sets. This operation is destructive: the first set holds the resulting union on return.
Errors: none

Procedure: INSTtype_cast
Parameters: Instance instance - instance to be cast
Type type - type to cast to
Error* errc - buffer for error code
Returns: Instance - the instance, cast to the requested type
Description: Converts an instance to a new type, if possible. If the cast is successful (`*errc == ERROR_none`), the original instance should no longer be used. It is guaranteed to be valid only when an error is reported. This call does not report errors to `stderr`; it is the callers responsibility to check `*errc` and to call `ERRORreport(*errc, (String)context)` if it is not `ERROR_none`.
Errors: `ERROR_aggregate_expected` - value given for an aggregate was not an aggregate
`ERROR_array_expected` - value given for an array was not an array
`ERROR_bag_expected` - value given for a bag was not a bag
`ERROR_entity_expected` - value given for an entity was not an entity
`ERROR_inappropriate_entity` - the entity given as a value was not of an expected class
`ERROR_integer_expected` - value given for an integer was not an integer
`ERROR_list_expected` - value given for a list was not a list
`ERROR_logical_expected` - value given for a logical was not a logical
`ERROR_number_expected` - value given for a number was not a number
`ERROR_set_expected` - value given for a set was not a set
`ERROR_string_expected` - value given for a string was not a string
`ERROR_incompatible_types` - the value given is not of the expected type, in some way not covered by any of the above messages

5.3 Product

Procedure: PRODadd_instance
Parameters: Product product - product to modify
Instance instance - entity instance to add
Returns: void
Requires: TYPEget_class(INSTget_type(instance)) == TYPE_ENTITY
Description: Adds an entity instance to a product model. The instance is assumed already to have been added to the instance list of its class, since `INSTcreate_entity()` does this.
Errors: none

Procedure: PRODcreate
Parameters: String name - name for new product
Express model - conceptual schema in which to create product
Returns: Product - a new, empty product
Description: Creates an empty product within a particular conceptual schema.
Errors: none

Procedure: PRODget_conceptual_schema
Parameters: Product product - product to examine
Returns: Express - conceptual schema in which the product exists
Errors: none

Procedure: PRODget_contents
Parameters: Product product - product to examine
Returns: Linked_List - entity instances which make up the product
Description: Retrieves a list of the instances in a product model, in the order in which they were created.
Errors: none

Procedure: PRODget_name
Parameters: Product product - product to examine
Returns: String - the name of the product
Errors: none

Procedure: PRODget_named_instance
Parameters: Product product - product to examine
String name - name of instance to retrieve
Returns: Instance - the named instance
LDescription: Retrieves a named instance from a STEP product model, if it is defined.
Errors: none

Procedure: PRODintiialize
Parameters: -- none --
Returns: void
Description: Initializes the Product module. This is called by STEPinitialize().
Errors: none

6 STEP Working Form Error Codes

The Error module, which is used to manipulate these error codes, is described in [Clark90d]. All STEP Working Form error codes are defined in the Instance module.

Error: ERROR_aggregate_expected
Severity: SEVERITY_ERROR
Meaning: A non-aggregate value was provided for an aggregate attribute
Format: %s - attribute name

Error: ERROR_array_expected
Severity: SEVERITY_ERROR
Meaning: An aggregate of a specific non-array class was provided for an array attribute
Format: %s - attribute name

Error: ERROR_bag_expected
Severity: SEVERITY_ERROR
Meaning: An aggregate of a specific non-bag class was provided for a bag attribute
Format: %s - attribute name

Error: ERROR_bag_full
Severity: SEVERITY_WARNING
Meaning: An item was inserted into an already full bag
Format: -- none --

Error: ERROR_cannot_instantiate
Severity: SEVERITY_ERROR
Meaning: An attempt was made to instantiate an uninstantiable type
Format: %s - type name

Error: ERROR_entity_expected
Severity: SEVERITY_ERROR
Meaning: A non-entity Instance was provided for an attribute having an entity type
Format: %s - attribute name

Error: ERROR_external_expected
Severity: SEVERITY_WARNING
Meaning: An embedded (internal) entity was provided for an attribute with "external" reference class
Format: %s - attribute name

Error: ERROR_inappropriate_entity
Severity: SEVERITY_ERROR
Meaning: An entity of the wrong type was provided for an attribute having an entity type
Format: %s - attribute name

Error: ERROR_incompatible_types
Severity: SEVERITY_ERROR
Meaning: Some other type problem was encountered in specifying an attribute of some instance.
Format: %s - attribute name

Error: ERROR_index_out_of_range
Severity: SEVERITY_WARNING
Meaning: An attempt was made to index an aggregate instance outside of the legal bounds
Format: %d - index value

Error: ERROR_insufficient_attributes
Severity: SEVERITY_WARNING
Meaning: Too few attribute values were provided for a particular entity instantiation
Format: %s - entity instance identifier

Error: ERROR_integer_expected
Severity: SEVERITY_ERROR
Meaning: A non-integer value was provided for an integer attribute
Format: %s - attribute name

Error: ERROR_internal_expected
Severity: SEVERITY_WARNING
Meaning: An non-embedded (external) entity was provided for an attribute with "internal" reference class
Format: %s - attribute name

Error: ERROR_list_expected
Severity: SEVERITY_ERROR
Meaning: An aggregate of a specific non-list class was provided for a list attribute
Format: %s - attribute name

Error: ERROR_logical_expected
Severity: SEVERITY_ERROR
Meaning: A non-logical value was provided for a logical attribute
Format: %s - attribute name

Error: ERROR_number_expected
Severity: SEVERITY_ERROR
Meaning: A non-numeric value was provided for a numeric attribute
Format: %s - attribute name

Error: ERROR_set_duplicate_entry
Severity: SEVERITY_ERROR
Meaning: A duplicate entry was added to a set
Format: -- none --

Error: ERROR_set_expected
Severity: SEVERITY_ERROR
Meaning: An aggregate of a specific non-set class was provided for a set attribute
Format: %s - attribute name

Error: ERROR_set_full
Severity: SEVERITY_WARNING
Meaning: An item was inserted into an already full set
Format: -- none --

Error: ERROR_string_expected
Severity: SEVERITY_ERROR
Meaning: A non-string Instance was provided for a string attribute
Format: %s - attribute name

Error: ERROR_too_many_attributes
Severity: SEVERITY_WARNING
Meaning: Too many attribute values were provided for a particular entity instantiation
Format: %s - entity instance identifier

Error: ERROR_undefined_reference
Severity: SEVERITY_ERROR
Meaning: A reference was made to an unknown entity instance identifier
Format: %s - entity instance identifier

Error: ERROR_unknown_entity
Severity: SEVERITY_ERROR
Meaning: A reference was made to an unknown entity class (type)
Format: %s - entity class name

A References

[Altemueller88] Altemueller, J., The STEP File Structure, ISO TC184/SC4/WG1 Document N279, September, 1988

[ANSI89] American National Standards Institute, Programming Language C, Document ANSI X3.159-1989

[Clark90a] Clark, S. N., An Introduction to The NIST PDES Toolkit, NISTIR 4336, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[Clark90b] Clark, S.N., Fed-X: The NIST Express Translator, NISTIR 4371, National Institute of Standards and Technology, Gaithersburg, MD, August 1990

[Clark90c] Clark, S.N., The NIST Working Form for STEP, NISTIR 4351, National Institute of Standards and Technology, Gaithersburg, MD, June 1990

[Clark90d] Clark, S.N., The NIST PDES Toolkit: Technical Fundamentals, NISTIR 4335, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[Clark90e] Clark, S.N., NIST Express Working Form Programmer's Reference, NISTIR 4407, National Institute of Standards and Technology, Gaithersburg, MD, September 1990

[Smith88] Smith, B., and G. Rinaudot, eds., Product Data Exchange Specification First Working Draft, NISTIR 88-4004, National Institute of Standards and Technology, Gaithersburg, MD, December 1988

ORDER and INFORMATION FORM

NATIONAL PDES TESTBED

MAIL TO:

National Institute of Standards and Technology
Gaithersburg MD., 20899
Metrology Building, Rm-A127
Attn: Secretary National PDES Testbed
(301) 975-3508

Please send the following documents and/or software:

- ☐ Clark, S.N., An Introduction to The NIST PDES Toolkit
- ☐ Clark, S.N., The NIST PDES Toolkit: Technical Fundamentals
- ☐ Clark, S.N., Fed-X: The NIST Express Translator
- ☐ Clark, S.N., The NIST Working Form for STEP
- ☐ Clark, S.N., NIST Express Working Form Programmer's Reference
- ☐ Clark, S.N., NIST STEP Working Form Programmer's Reference,
- ☐ Clark, S.N., QDES User's Guide
- ☐ Clark, S.N., QDES Administrative Guide
- ☐ Morris, K.C., Translating Express to SQL: A User's Guide
- ☐ Nickerson, D., The NIST SQL Database Loader: STEP Working Form to SQL
- ☐ Strouse, K., McLay, M., The PDES Testbed User Guide

OTHER (PLEASE SPECIFY)

____

____

____

____

These documents and corresponding software will be available from NTIS in the future. When available, the NTIS ordering information will be forthcoming.

NIST