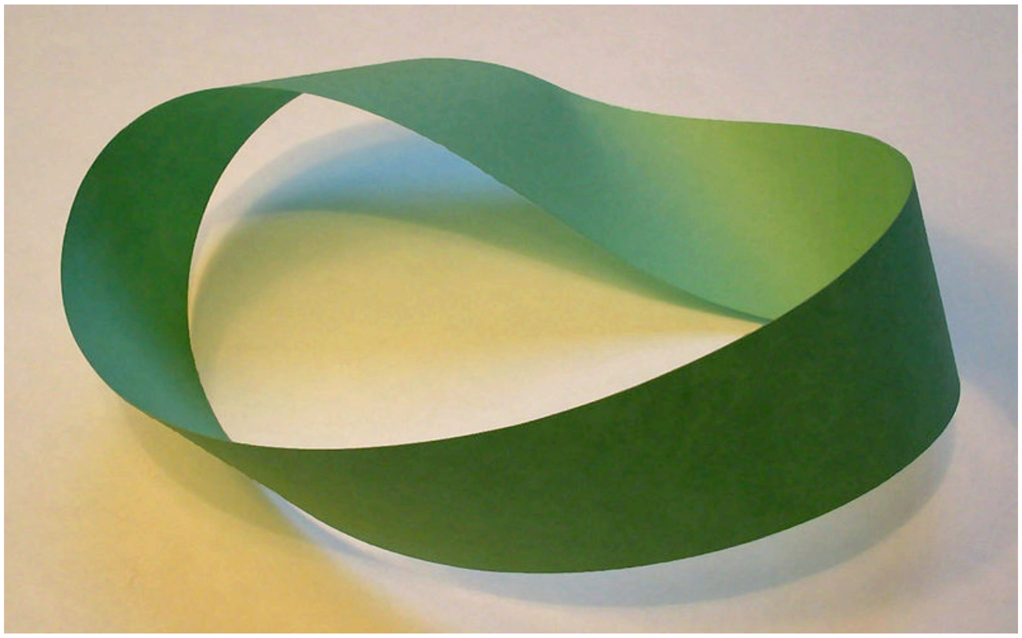
THE CTHULHU BUILD SYSTEM

Using Tcl to build C To Build Tcl Applications



Sean Woods
Test and Evaluation Solutions, LLC
for the
17th Annual Tcl/Tk Developer's Conference

THE CTHULHU BUILD SYSTEM

Using Tcl to build C To Build Tcl Applications

Sean Woods
Test and Evaluation Solutions, LLC
for the
17th Annual Tcl/Tk Developer's Conference

Introduction

This paper describes the *C and Tcl Hashtable Underpinnings for Language High-level Understanding*, or CTHULHU. CTHULHU is a series of tools to build C extensions for Tcl, from Tcl. It's focus is on computing problems that require Tcl to have access to complex data structures that are themselves manipulated by existing C code. These tools were originally developed to support the Test and Evaluation Solution's Integrated Recovery Model, but would be applicable to any problem in which Tcl needs to interact with C data structures in real-time.

CTHULHU Concepts

Overall Structure

Lets assume for a moment that we are developing a Tcl interface to a C application. This application has several different species of objects, each with their own data structure. Each object type is stored in a separate hashtable. We also provide an ensemble to manage both the hashtable and the data structures within.

CTHULHU builds on the venerable Tcl Extension Architecture (TEA). It uses a series of flat-file databases and scripts to generate C source files and headers. The scripting language used, of course, is Tcl.

The process begins with the developer writing up what he or she would like the data structures to look like. This description will serve double-duty as the meta-information that Tcl applications will later use to generate graphical interfaces for.

Containers

Let's call this master index and ensemble of methods a "container" for lack of a better term. In our hypothetical example, let's have a container that handles doorways. Well, since I'm out of creative ideas, let's just call the doors portals, and lift the example right from the kinds of problems that CTHULU was designed to solve.

```
% portal setting p1 open
1
% portal setting p1 jammed 2
```

Example: Using a container

When our program is running, we can use the **portal** command to generate our list of portals, as well as change their state throughout the simulation.

Methods

Containers all require a common set of core functions. Create a structure instance. Link it to another structure. Reset the table. Return a list of the id's in the table. Read and write to individual fields within a single structure.

In a perfect world, these core functions behave consistently across all of your containers, and your Tcl/Tk app can play on the similarities. Perhaps even glean meta information enough to build screen elements.

Some of these functions are pretty copy and paste-able. For instance, a function to spit out the IDs usually differs only in the name of the variables used. There are other functions that require a custom parser for every structure. Reading and writing individual fields of a record, for instance.

A great example of that later is the **setting** method. Setting takes in the id of a record, a field, and optionally a value. If the value is not given, the current value is returned. If the value is given, the data structure is updated with the new value for that field.

Wouldn't you know that "simple" function **setting** is one of the hairiest and most complex I have to implement? Every time I add a field, I have to drop it into 5 or 6 places in C. (And that is just for one method, mind you.)

There is also the headache of ensuring the user doesn't try to put a floating point value into a boolean field. Not to mention handling when a user tries to stuff in a value for a field that doesn't exist. Over time I was seeing a pattern emerge, a template that was being stamped out one module at a time. Being the lazy programmer I decided to stop implementing it piecemeal and write a general case automated procedure.

State Structures

Portals are very simple creatures, of course. They have a finite number of states: open, closed, opening or closing. Doors can also be damaged in two different ways:

1. Jammed open, meaning they are open and cannot be closed
2. Jammed closed: meaning they are closed and cannot be opened

So if I want to inquire about the state of a door we can use the setting method of the container.

The process begins with the user describing the structure. CTHULHU use a simple dict, with all of the information that both C and the resulting Tcl application will need captured in one place. The description for portal is to the right.

```
cstructDefine Portal {
  key id
  delta 1
  prefix p
  comment {Each portal is an ,,,
  static {
    Roid id;               /* Num...
    Entity *pType;
    Portal *delta;
    Compartment *pFrom;  /* Com...
    Compartment *pTo;    /* Com...
    Crew *holding;       /* Cre...
    Portal *airlock;     /* Poi...
  }
  fields {
    flooding {
      storage real
      desc {% of the door cover...
    }
    toid {
      storage Roid
      desc {Compartment To}
    }
    fromid {
      storage Roid
      desc {Compartment From}
    }
    jammed {
      storage u4
      values {
        0 normal {Door is normal}
        1 jam-close {Door is jamm...
        2 jam-open  {Door is jamm...
      }
    }
    reserved {
      storage bool
      desc {Door is being held clo...
    }
    open {
      storage u4
      values {
        0 normal {Door is closed}
        1 open   {Door is open}
        2 opening {Crew is opening...
        3 closing {Crew is closing...
      }
      desc {Current open status of...
    }
  }
}
```

Example: a dict describing the portal structure

The *static* field stores a block of C code. This is a great place to stuff pointers, variables you'd like hidden from the user, etc. The *fields* field is itself a dict, with an entry for each public variable that the developer wishes Tcl to be able to directly interact with. This dict also captures comments, enumerated values, and provides hints about storage requirements to C.

The data structure our cstructDefine statement produces is on the right. Major features:

1. #define statements for each enumerated field value
2. Sub-byte fields are stored in bitfields
3. The fields that are accessible to Tcl are marked public_XXXX. This makes them easier to spot while coding.
4. The flags in the toplevel of the definition trigger other fields to be automatically added. For instance set "location 1" flag added the *deckid*, *x, y,* and *zoff* fields.

```
/* Define the Portal Structure */
#define PORTAL_JAMMED_NORMAL 0 /* Door...
#define PORTAL_JAMMED_JAM_OPEN 2 /* Do...
#define PORTAL_OPEN_NORMAL 0 /* Door i...
#define PORTAL_OPEN_OPEN 1 /* Door is ...
#define PORTAL_OPEN_OPENING 2 /* Crew ...
#define PORTAL_OPEN_CLOSING 3 /* Crew ...
struct Portal {
   /* Place holder for ID */
   Tcl_Obj *idString;
   /* Begin hard-coded structure */

   Roid id;             /* Numeric ID ...
   Entity *pType;
   Portal *delta;

   Compartment *pFrom;  /* Compartment ...
   Compartment *pTo;    /* Compartment ...
   Crew *holding;       /* Crewmember  ...
   Portal *airlock;     /* Pointer to i...

   float public_flooding; /* flooding e...
   Roid public_fromid; /* fromid eoc */
   Roid public_toid; /* toid eoc */
   /*Begin bitmap*/
   unsigned int public_jammed    :4; /*...
   unsigned int public_open    :4; /* o...
   unsigned int public_reserved    :1; ...
};
```

Example: The C implementation of the portal structure

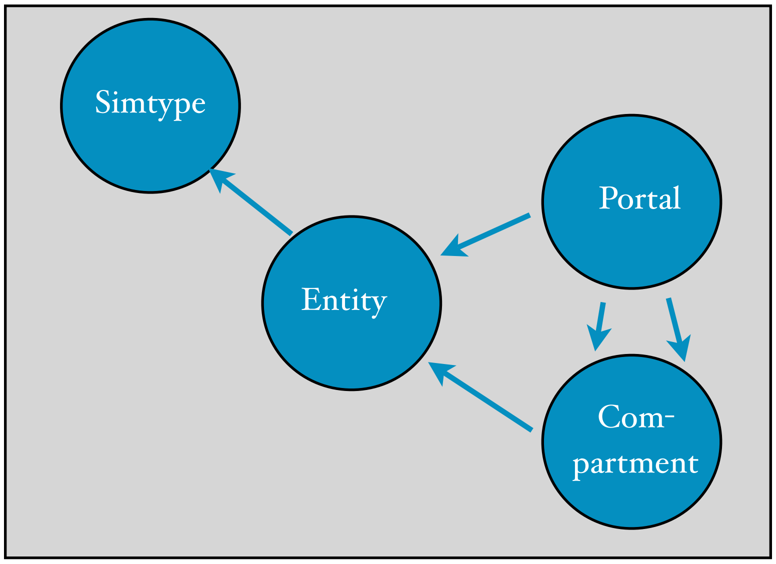
Data Interface

Of course, simply generating a data structure is not particularly useful in of itself. So CTHULHU has another few tricks up it's sleeve. In particular, as suite of tools to handle the interchange of data between Tcl and C.

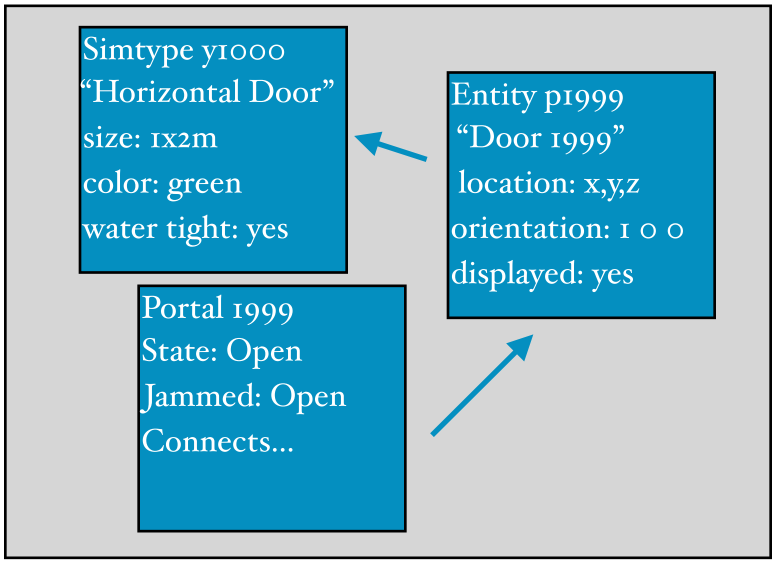
Because it is machine generated, every field is given it's own handler to manage the unique constraints of each field. For instance, CTHULHU knows that a boolean field should use Tcl_GetBooleanFromObj, and fall back to Tcl_GetDoubleFromObj if the user was uncouth enough to put a decimal point in to represent 1.0 or 0.0. Because the machine is doing all the brute force programming, we can afford to be fancy because we only have to debug the prototype. Our work will be carried over and repeated N times for N different fields.

Properties, Entity and SimTypes

You may have noticed the pointer to a structure called **Entity** in the Portal structure. Entity is a container CTHULHU provides as a place to store properties. If you have a ship full of doors, many will be similar. Some can be opened faster than others. Some are taller, some are wider, but generally, there is usually more than one of each type of door in a model.



The SimType system allows you to store those properties once for all of the nodes that share it. Because we are making such efficient use of space by eliminating copies, there's very little need to break the properties up into C structures. So, we don't bother. Both SimType and Entity use a single Tcl_Obj that is formatted as a dict.



It is also handy to store information about a node that isn't part of the state C needs to access occasionally if at all. So it's nice to store that sort of information separately, with C being able to access on demand. In other cases, this is information that is needed for a Tcl representation, with no impact on the model running in C. Rather than write 1,000,001 handlers for all of the different data types, I decided to write the **Entity** data structure to handle this ephemeral data, and have all the other structures link to it.

Ok... I think I need to break out a graphic to explain what is going on. On the right you can see a chart of which data structure is pointing to which. **Portal** contains 5 links. One to an entry in **Entity**, two links to **Compartment**, one to another Portal, and the fifth to a structure called **Crew**. We won't explore crew or compartment for now.

The only link CTHULHU is concerned with is the link to Entity. When portal looks for a property, it looks to Entity. Every entry in the portal container has a corresponding entry in the Entity container.

If a property hasn't been specified for a particular entity, our search routing will route next to the Entity entry's SimType.

In our C model, let's assume the simulation itself doesn't care about things like object_size. We just need to know if the door is open, and if it can be opened or closed.

When we draw the portal on the screen, though, we do have to be concerned about size, location and direction. Our GUI can also suppress the display of the portal.

On the right you can see a sample conversation between a user at a console, the simtype, entity, and portal containers.

Delta

CTHULHU was designed with discrete time simulations in mind. One handy thing to know is what changed during the timestep. The pointer to a type of Portal called **delta** is a copy of this data structure from the last time step.

The current timestep is copied into the delta during the call to **cstruct_advance_all. cstruct_advance_all** is called after data logging has taken place, but before the next timestep kicks off. We can get away with copying the entire state of every node in the system, of course, because we represent it in such a compact form.

```
# See our portal with no type
% portal get p1
  open 1 jammed 2 fromid 100 \
  toid 101 flooding 1.0
# Create a type and stuff it with
# properties
% simtype create 1000
% simtype put 1000 {
  fullname {20” portal}
  object_size {1000 100 2000 mm}
}
# Set our portal to that new typeid
% portal typeid 1000
% portal get p1
  open 1 jammed 2 fromid 100 \
  toid 101 flooding 1.0 \
  fullname {20” portal} \
  object_size {1000 100 2000 mm}
# We also have a representation in
# entity. It stores properties,
# but has it’s own state
% entity get p1
  visible 1 hidden 0 ...(etc)...
  fullname {20” portal} \
  object_size {1000 100 2000 mm}
% portal create p2
% portal typeid p2 1000
# If we add a property to portal,
# it’s actually stored in entity
% portal put p2 \
   {dclocator 01-400-4}
% portal get p2
  open 0 jammed 0 fromid 0 \
  toid 0 flooding 0.0 \
  fullname {20” portal} \
  object_size {1000 100 2000 mm} \
  dc_locator 01-400-4
% entity get p2
  visible 1 hidden 0 ...(etc)...
  fullname {20” portal} \
  object_size {1000 100 2000 mm} \
  dc_locator 01-400-4
# If we add a property to entity,
# portal immediately sees it
% entity put p2 {open_time 10}
% portal get p2 open_time
  10
```

Example: Entities and Types in action

Programming CTHULHU

CTHULHU is intended to provide utility to C code. It's not a complete extension in of itself. It depends on external C code to maintain the hash tables the nodes are in, for instance. It assumes something else is doing the high-level integration of the ensemble. It's a tool with one job, and I'd like to think it does the job well: take care of the mundane exchange of data between Tcl and C.

The basic process is:

1. Document your structures in the cStructureDefine markup language, one file per structure, in the cstruct/ path in your source directory.
2. Call cstructBuild.tcl to generate your C source files: cstruct.c and cstruct.h
3. Include ctruct.c and cstruct.h in your build process
4. Include the cstruct definition files in your Tcl application

Modifications to TEA

The standard distribution of CTHULHU uses a modified version of the Tcl Extension Architecture. Certain paths have a special meaning, and some scripts have been incorporated into the makefiles.

build/

To make it easier to spot C files that are auto-generated, all of the products of the build process are placed in the build/ directory. It's designed so you can safely throw away that one path and start the build process from scratch.

conf/

The conf path is designed to store Tcl files that contain databases that will be incorporated into C and published to the resulting Tcl program as well. These files are sourced at various times during the build process, and Each instance of the cStructDefine command encountered during this pass is turned into a new data structure.

scripts/

Contains the Tcl scripts used by the modified Tea to auto-generate sources and/or chunks of the build system.

Other modifications

CTHULHU uses scripts to auto-generate the makefile lines needed to produce a .o file in the build directory for every C file it encounters in the generic/ directory. It also wraps those resulting .o files into a library automatically. This requires a custom Makefile, If you have additional makefile requirements, place them in static.mk Other than that, to build your extension is simply "./configure" and "make install"

CTHULHU Structure Descriptions

The markup language for CTHULHU is based on dicts. Each field is a signal for the meta-compiler. Here are the major fields implemented:

keytype

Tells CTHULU what kind of key type the container uses. If the keytype is int CTHULU will auto-generate a *StructName*_**Identify** and *StructName*_**TypeId** function. Otherwise those functions will need to be provided by the developer.

hashtype

Tells CTHULU what type of hash table the container uses to indexing nodes. The only supported hashtype at the moment is **tcl**.

dictstore

Used to differentiate between containers like **Entity** that store dict information, and others who link to entity for that function.

prefix

A one letter prefix that is used to build the node's ID in the Entity container. For instance, portal 199 is simply 199 in the portal container. But in entity it is p199, to differentiate it from, say e199 or v199.

key

Identifies the static field in the data structure that represents the node's identity.

name

A human readable name for the container

location

True or false. When true, add fields and methods to track and manipulate the location of the node within the model.

static

Block of code that is added, verbatim to the top of the data structure

static_bitfield

Block of code that is added after the bitfield fields in the data structure. Usually other sub-byte flags.

delta

True or False. When true, this structure tracks a delta

alloc_free

When non-empty, a block of C code to replace the standard *StructName*_Generic_Alloc and *StructName*_Generic_Free functions.

comment

Human readable comment

fields

A dict containing the definitions for the public fields of the structure.

Field names CTHULHU interacts with are all mapped to lower case, and all prepended with **public_**. If you see **public_foo**, the user in Tcl methods will address the field as **foo**.

CTHULHU Field Descriptions

The markup language for CTHULHU fields are also based on dicts. CTHULHU builds the structure based on the content of the following fields: (other fields are ignored, and assumed to be important to your app.)

storage

Tells CTHULHU how to represent the field in C. CTHULHU understands the values in the table on the right. Fields with a storage other than those understood by CTHULHU are treated as short integer.

Type	C Type	Tcl Type
Bitmask	Tcl_WideInt	wide integer
bool	unsigned int :1	boolean
blob	*stored as a property	
char	char	integer
double	double	double
float	float	double
matrix	*stored as a property	
real	float	double
Roid	unsigned int	integer
script	*stored as a property	
string	*stored as a property	
text	*stored as a property	
u1	unsigned int :1	boolean
u2	unsigned int :2	integer
u3	unsigned int :3	integer
u4	unsigned int :4	integer
u8	unsigned char	integer
u16	unsigned short	integer
u32	unsigned int	wide int
u64	Tcl_WideInt	wide integer

CTHULHU Storage Types

values

Values are mapped as a series of triplets: literal, code, description. A define statement for every literal/code mapping is created for the programmer's convenience.

Calls of the CTHULHU

The following are functions that interact with entities, simtypes, and properties. They can be used on any CTHULHU generated structure.

SimType_GetReal

```
double SimType_GetReal(Entity *pType, const char *field);
```

Searches for a property named **field** and returns the value as a double precision floating point.

SimType_GetInt

```
int SimType_GetInt(Entity *pType, const char *field);
```

Searches for a property named **field** and returns the value as an integer.

SimType_GetWideInt

```
Tcl_WideInt SimType_GetWideInt(Entity *pType, const char *field);
```

Searches for a property named **field** and returns the value as a wide integer.

SimType_GetTclObj

```
Tcl_Obj *SimType_GetTclObj(Entity *pType, const char *field);
```

Searches for a property named **field** and returns the value as a Tcl_Obj structure.

SimType_SetInt

```
void SimType_SetInt(Entity *pType, const char *field, int value);
```

Inserts or modifies a property named **field**, using the int value.

SimType_SetReal

```
void SimType_SetInt(Entity *pType, const char *field, double value);
```

Inserts or modifies a property named **field**, using the double value.

SimType_SetWideInt

```
void SimType_SetInt(Entity *pType, const char *field, Tcl_WideInt value);
```

Inserts or modifies a property named **field**, using the wide int value.

SimType_SetFromObj

```
int SimType_SetFromObj(Tcl_Interp *interp,Entity *pType, Tcl_Obj *field,
    Tcl_Obj *value);
```

Inserts or modifies a property named **field**, using the Tcl_Obj **value**. Returns TCL_OK on success, or TCL_ERROR otherwise.

SimType_GetFromObj

```
Tcl_Obj *SimType_SetFromObj(Tcl_Interp *interp,Entity *pType, Tcl_Obj *field);
```

Retrieves a property named **field**. Returns NULL if the field was not found.

SimType_GetAll

```
Tcl_Obj *SimType_GetAll(Tcl_Interp *interp,SimType *p,Tcl_Obj *local);
```

Combines all of the properties from the dict stored in *SimType* with the properties in *local* and returns them as a single dict.

Structure Functions of CTHULHU

CTHULHU defines a series of functions that govern the interactions on the C level of containers.

---

*StructName*ById

```
StructName *StructName ById( Roid id, int createflag);
```

Returns a structure given an integer ID, or NULL of the ID does not exist. If the createflag is true, generate a new structure if ID does not exist. **Roid** is typedef set aside for Record IDs. By default it's defined as a 32 bit *unsigned int*.

`StructName`ByID is provided by the developer.

---

*StructName*_Generic_Alloc

```
void StructName_Generic_Alloc(StructName *pNode);
```

A function external to CTHULHU that allocates the and initializes the structure. For structures with a key-type of int, CTHULHU provides a generic version.

---

*StructName*_Generic_Free

```
void StructName_Generic_Free(StructName *pNode);
```

A function external to CTHULHU that frees any memory allocated within the structure.

For structures with a keytype of int, CTHULHU provides a generic version.

---

*StructName*_ApplySettings

```
void StructName_Apply_Settings(StructName *pNode);
```

Called after a setting has been changed by the Tcl interface to CTHULHU, allowing the C code a chance to be notified and or alter other fields in response.

*StructName*_Identify

```
Tcl_Obj *StructName_Identify(StructName *pNode);
```

Returns a Tcl object that represents the identity of this node, and is capable of being read back in by *StructName*_**FromTclObj**. If the container's **keytype** is int, this function is generated by CTHULU.

*StructName*_TypeId

```
Tcl_Obj *StructName_TypeId(StructName *pNode);
```

Returns a Tcl object that represents the identity of this node's type. Or zero if this structure has no type.

If the container's **keytype** is int, this function is generated by CTHULU.

*StructName*_FromTclObj

```
int StructName_FromTclObj(Tcl_Interp*, Tcl_Obj*, StructName **pNode);
```

Converts the ID of a node given in a Tcl object to a pointer to a structure. Return TCL_OK on success, or TCL_ERROR otherwise.

*StructName*_ToDict

```
int StructName_FromTclObj(Tcl_Interp*, Tcl_Obj*,StructName **pNode);
```

Converts the ID of a node given in a Tcl object to a pointer to a structure. Return TCL_OK on success, or TCL_ERROR otherwise.

*StructName*_SetType

```
void StructName_SetType(StructName *p,SimType *pType);
```

Associates an CTHULHU node with a type given by the a pointer of **SimType**.

*StructName*_GetType

```
SimType *StructName_GetType(StructName *p);
```

Returns either a the pointer to the **SimType** structure that contains properties for this node, or NULL if no type has been associated with it.

*StructName*_DictPut

```
void StructName_DictPut(StructName *p,Tcl_Obj *field,Tcl_Obj *value);
```

*StructName*_DictGet

`Tcl_Obj *StructName_DictGet(StructName *p,Tcl_Obj *field);`

*StructName*_ValueOffset

`int StructName_ValueOffset(Tcl_Interp *interp, Tcl_Obj *pObj, int *pIndex, int *pType);`

*StructName*_FromTclObj

`int StructName_FromTclObj(Tcl_Interp *interp, Tcl_Obj *nodeid, StructName **pNode);`

Using the information about a node given in the Tcl_Obj **nodeid**, map the pointer **pNode** to the data structure that represents it. If the node does not exist, **pNode** will be null. If a problem was encountered converting **nodeid** to a useable form, the function will return **TCL_ERROR**. Otherwise, it will return **TCL_OK**.

*StructName*_ToDict

`Tcl_Obj *StructName_ToDict(Tcl_Interp *interp, StructName *p);`

Converts `StructName` to a dict that is capable of being exported to Tcl.

*StructName*_nodeeval

`int StructName_nodeeval(Tcl_Interp*, StructName*, Tcl_Obj *body);`

Loads the fields and values from `StructName` to the local interpreter, evaluates the block of code given in **body** and returns the standard Tcl reply codes: **TCL_OK, TCL_CONTINUE, TCL_BREAK,** or **TCL_ERROR**. On replies other the **TCL_ERROR**, changes made to the variables derived from `StructName` are saved back to the structure.

Tcl Commands Provided by CTHULHU

In addition to routines intended to be used by C, CTHULHU also adds ready-made Tcl commands. To include them into the interpreter, one simply needs to add them with your desired name using **Tcl_CreateObjCommand**.

*StructName*_method_count

`Arguments: (none)`

Returns a headcount of all of the nodes in the container

*StructName*_method_exists

`Arguments: nodeid`

Returns true if *nodeid* was found in the container, false otherwise.

*StructName*_method_for

Arguments: keyvar ?valvar? body

Operates like a foreach loop, with the ID of the node stored in the keyvar, and optionally, a dict representation of the node stored in valvar.

Example:

```
%structname% for nodeid nodeinfo { puts [list $nodeid $nodeinfo] }
```

---

*StructName*_method_foreach

Arguments: body

The ID number of the node is loaded into the interpreter as the local variable "id." All fields are loaded into the local interpreter as variables of the same name. Any modifications to those variables will write the changes back to the data structure.

Example:

```
%structname% foreach { set hidden 1 }
```

---

*StructName*_method_list

Arguments: (none)

Returns a list of all of the nodes in the container

---

*StructName*_method_nodedelta

Arguments: nodeid ?field?

Returns a list of parameters that have changed since the last instance of "changes." Optionally, return a true or falseif a specific field has changed.

---

*StructName*_method_nodeget

Arguments: nodeid ?field?

If *field* is given, return the value for *field*. If *field* is not in the data structure, the system searches for a value in the following order:

1. In the local dict of the node
2. In the dict of the node's type
3. Null

If *field* is not given, return a key/value list containing all of the values:

1. Published from the data structure
2. Written into the local dict of the node
3. Written into the dict of the node's type

---

*StructName*_method_nodeput

Arguments: nodeid {*field* value ?*field* value? ...}

Writes values to node. If a *field* matches a field in the data structure, the data structure is updated. Fields that do not match a field in the data structure are written to a local dict.

---

*StructName*_method_nodewith

Arguments: nodeid body

Operates in a similar manner to *dict with*. The state and properties of the node are fed into the interpreter as local variables. If the script modifies one or more of the variable representing state, the change is written back into the structure.

---

*StructName*_method_setting

Arguments: nodeid field ?value?

When value is given, update the *field* in the data structure. *field* must be on of the fields of the data structure, or an exception is returned.

When value is not given, return the value for *field.* In this case *field* is permitted to be a field in the structure, or any property that is expected to be passed from the node's type.

---

*StructName*_method_typeid

Arguments: nodeid ?typeid?

If typeid is given, set the type for *nodeid* to the simtype identified by *typeid*.

If *typeid* is omitted, return the id of the current type for *nodeid*, or 0 if none has been set.

Getting CTHULHU

A public release for CTHULHU is available on T&E Solution's open source site:

http://oss.tnesolutions.net/home

CTHULHU is released under a BSD-style license, modeled after the Tcl core.

You can contact Sean Woods for more details at yoda@etoyoc.com

Credits

Cover Art: Möbius strip, http://en.wikipedia.org/wiki/File:Möbius_strip.jpg

The Tcl Extension Architecture (TEA) is published at: http://www.tcl.tk/doc/tea/