SCRIPTO

A new tool for you to talk with LS-PREPOST®[1]

[1] SCRIPTO is supported since version 2.1 or higher, if you find any bugs in the SCRIPTO, errors or typos in this document, please send your corrections or suggestions to roger@lstc.com. LS-DYNA, LS-OPT and LS-PREPOST are registered trademarks of Livermore Software Technology Corporation.
2. Current revision 20070906.

Contents

SCRIPTO Commands in LS-PREPOST

There are several SCRIPTO commands users can evoke from the command parser. Scripts may utilize these commands to communicate with c-parser that SCRIPTO relies on and execute the scripts through a command file. (CFILE)

openc scripto [script_name]

Openc command is the command to open all kinds of files that is recognized by LS-PREPOST. Use **scripto** as first parameter will make LS-PREPOST to read in a script file name **script_name**.

evokescript param_function [param1 param2 param3 ...]

To call a script function from a command file.

- param_function : the function name for a command to call. a script name should be added as a qualifier to a function name.

  Ex: Let there be an a.sco that contains...

  ```
  define:
  void hello(void) { Echo("Hello!"); }
  ```

  To call this function from a command file after the script has been loaded, one can issue:

```
evokescript a::hello;
```

Purpose of a qualifier is to clarify the ambiguity between same function names in different scripts.

A script name may contain up to 20 non-white charaters;

- param1 ... param_n :the argument list for the evoked function. All arguments also has to qualify with their type for command parser to parse them correctly before calling the script function.

  Following is a list of the qualified arguments:

Type	Qualifier	In commands	in script
Integer	%	%1022	1022
Float	#	#25.33	25.33
String	$	$string	"string"
variable	@	@asymbol	asymbol

The 4th type, (variable) is a global variable defined in that script, the value of a variable is defined by the script at the point when command is issued.

```
Ex:
Load in a script "callout.sco" that defines a function
define:
void AreaCircle(Float r) {
  Float pi;
  pi = 4.0 * atan(1.0);
  char areabuf[30];
  sprintf(areabuf, "area = %f", pi * r * r);
  Echo(areabuf);
}

To call this function from a command file, one may issue:

     evokescript callout::AreaCircle #2.5;

This will be equivalent to

  AreaCircle(2.5);

in a script.
```

Global Functions and Variables

Following are the global function and variables defined in SCRIPTO, functions that ties to no objects, in stead, related to the file manipulation or global constants.

`void ExecuteCommand(Char *cmd);`

Execute the command cmd in LS-PREPOST right away.

- cmd : a command that LS-PREPOST understand.

`void Echo(char *sentence);`

To echo a sentence from the script, it will be show at the information box at the bottom panel of the LS-PREPOST.

- sentence : the sentence to be echoed.

`void MsgBox(Generic g);`

This function will inform user with a simple dialog box, the information that has been passed into the message box can be literally anything. MsgBox will translated the value according to the type that has been passed into it.

- g : generic information to show on the simple message box provided by SCRIPTO.

`void DataRefresh(void);`

Refresh the model, by calling redrawing routine inside LS-PREPOST. This function will invalidate the OpenGL window area and redraw the area again.

`void UserError(Pointer errmsg);`

User send an error signal to the SCRIPTO parser, script will stop parsing after receiving the signal, error message will be sent to the dialog area.

- errmsg : error message users send out to LS-PREPOST.

`Int ChangePath(Pointer path);`

Change the working path, or working directory.

- path : the new working directory for the script reading
- returned value : success if not zero.

`void Verbose(Int onoff);`

Change the verbose setting for the parser.

- onoff : to set verbose on, onoff should not be trivial.

This function is obsolete, C-Parser now does not allow verbose mode.

`Pointer TranslateFrom(Pointer name);`

Translate a name of a variable into a pointer to the variable. This function takes the name of a variable as a string and then search the symbols currently understood by c-parser, then return the pointer to the symbol back to the script.

If there is a different symbol that shares the same name in different scopes, this function will match the first symbol it found, and return the pointer to the symbol.

When a symbol is not found, a null pointer will be returned.

- name : name a symbol
- returned value : pointer to a symbol that first match the name.

Utility Classes

Utility classes are a set of objects that provide functionalities for general uses, such as strings and others.

String

String is a class that encapsulates a string. Operations are defined for String objects.

Constructors

`String String(Generic g);`

- g : a value of either integer, string pointer, or a float number, it will give you a String returned.

Member functions

`Pointer c_str();`

This return a c-style string.

- returned value : a constant pointer, null terminated.

`Pointer dup_str();`

This return a c-style string, it will duplicate current content of the String. the difference between c_str() and dup_str() is that c_str() return the pointer itself. so, the whole script shares the same pointer where c_str() used. dup_str() duplicates the content of c_str() and return a new pointer back to caller, very pointer thereafter is independent of the original String.

- return value : an allocated pointer, null terminated.

`Int find(char c);`

Find the character c and return the first occurrence of c.

- c : the character to be found.
- returned value : zero-based index of the position of c in the string contents.

`Pointer left(Int n);`

Take left n characters from the original string and allocate a new string for it.

- n : the number of the characters to be taken.
- returned value : the new string, memory were allocated by the scripto.

`Pointer right(Int n);`

Take n characters from the right of the original string and returned a new string that allocated by the scripto.

- n : the number of the characters to be taken.
- returned value : the new string, memory were allocated by the scripto.

```
Pointer mid(Int start, Int n);
```

Take n characters from the start position of the original string, and returned the allocated string. The memory was allocated by the scripto.

- start : the starting index for the substring
- n : number of the characters to be taken.

```
void shrink();
```

Eat all white spaces (line feed, blank, tab...) of the string from the left and from the right. After shrink(), the string should have no white spaces from the left or right.

Overloaded Operators

```
operator + (String s1, String s2);
```

To use the operator, consider the following example:

```
 String s1, s2, s3;
     s1 = String("This ");
     s2 = String("String.");
     s3 = s1 + s2;
/*
   then we will have
     s3.c_str() == "This String.";
*/
```

```
operator + (String s1, char *ptr);
```

same as operator + (String, String); But this version takes a string and a pointer of a character array;

- s : the string
- ptr : the pointer of a character array

For example:

```
String s, t;
char *p;
p = "a book";
s = String("This is");
t = s + p;
/*
  You may also do this
     t = s+ "a book";
*/
```

```
operator + (String s1, Int i);
```

Same as operator + (String, String); But this version takes a string and a integer;

- s1 : Left hand side of the operator
- i : integer to be added to the string

```
operator + (String s1, Float f);
```

same as operator + (String, String); But this version takes a string and a floating number;

- s1 : the string object
- f : the floating number

```
bool operator == (String s1, String s2);
```

Operator == test if the content of s1 and s2 are the same, it will return true (or 1), false if not.

Toolbox

Toolbox is an object that users may use to retrieve information related to the states of user interface from SCRIPTO. There is only one copy of instance of Toolbox in the SCRIPTO. Following information is available now in the Toolbox:

- Common File Dialog and its status
- Data and Forms for LS-DYNA Keyword manual

Users do not need to create any Toolbox from their scripts; SCRIPTO will create it internally for you. Some Toolbox functions might require you to provide more information before they can be called properly.

Common File Dialog

You use a file dialog to open/save a file from/to the disk drive. Before you can call a Common File Dialog, one should prepare the following data structure to SCRIPTO:

```
struct FileDialogInfo {
     Pointer parent;
     char *title;
     char *style;
     char *init_filter;
     char *filters[];
     Int pressed;
     char *ok_action[];
};
```

- parent : parent of the file dialog box, this parameter suggests which window the dialog should be on top of
- title : a customized title for the file dialog
- style : style for a FileDialog can be the following

FDLG_OPEN	To select a file to read from, user may not use this style with FDLG_SAVE. This is the default style, if style field left empty
FDLG_SAVE	To select a file to write to, user may not use this style with FDLG_OPEN
FDLG_SINGLE	When the style is set, it allows selecting only one file at a time. This is the default style, if style is not set.
FDLG_MULTIPLE	Users allow selecting more than one file at the same time. Return result will be slightly different in format for the file names selected.
FDLG_MUSTEXIST	Users can not create new files

- init_filter : initial filter options, if there are more than one filters installed into the filters options.
- filters : the filters for the dialog. all filter entries will have the following format,
  ```
  "description|format1*;format2*;...;*.ext1;*.ext2..."
  ```
  the examples below are some of the valid formats
  ```
  "binary file output(d3plot*)|d3plot*",
  "script file (*.sco)|*.sco",
  "dyna keyword file (*.k;*.key;*inf)|*.k;*.key;*inf"
  ```
- pressed : can be either scOK or scCANCEL
- ok_action : you can put into this field a series of commands or give a callout function as one would define for an user interface widget

Manipulation Functions

```
void UtilFileDialog(Pointer pfdi);
```

This is the function to get a common file dialog popped up. Users should note that the FileDialog is a modal dialog, which means that until the user has pressed OK or CANCEL button, the process flow will stop here. After the function returned, user can check the pressed field to see if the user actually press OK or CANCEL button.

- pfdi : the pointer to the FileDialogInfo structure

```
Pointer UtilGetLastFileName(void);
```

This function will be valid only right after the FileDialog has been dismissed. File name returned from this function will be cleared every time when user calls UtilFileDialog().

- returned value : the character array for the file name selected. if the file dialog has style of FDLG_MULTIPLE, the filenames will be comma delimitated. Noted that file names will not include the path where the file(s) resides. SCRIPTO does not duplicate the string when returns back from the dialog, only the address of the internal buffer has been assigned, users should not free or store the pointer.

```
Pointer UtilGetLastFilePath(void);
```

This function will be valid only right after the FileDialog has been dismissed. File path returned from this function will be cleared every time when user calls UtilFileDialog().

- return value : the character array for the file path where the file name(s) reside. SCRIPTO does not duplicate the string when returns back from the dialog, only the address of the internal buffer has been assigned, users should not free or store the pointer.

```
Char UtilPathSeparator(void);
```

This function returns the path separator character of the platform where the SCRIPTO runs on.

- returned value : the path separator

Keywords

LS-PREPOST has the latest and most comprehensive keywords support for LS-DYNA. To take advantage of this, SCRIPTO has several keyword related functionality built in to the Toolbox. User may reuse the keyword forms through these functions.

```
struct Keyword {
     char *name;
     char *style;
     Int  id;
     Int formidx;
     Int is_valid;
     Int pressed;
     Int num_entities;
     char *done_action[];
};
```

Keyword structure is used to retrieve the keyword form.

- name : keyword name. please skip the '*' character in front of each keyword.
- style : following are the keyword styles understood by SCRIPTO.

KWD_NOLIST	same keyword might appear more than one time in a keyword deck, LS-PREPOST put them in a list, if you do not want to see the list(sometimes it can be long), the form you are about to popup should include this style. Default style for a keyword form is that it has a list.
KWD_READONLY	with this style, the form you popup up will not have accept button. It might still appeared to be editable, however, it will have no effect on the keyword. Default setting for the keyword form is editable.

- id : user id for the keyword form you want to retrieve, if this field is zero, that means the user is creating a new keyword in the system.

- formidx : index for the keyword form created. Use this id to manipulate the form.
- is_valid : if not zero, the form that just popup is still on-goiong. otherwise, the form have been dismissed.
- pressed : the value of this field can be one of the following:

scDONE	user pressed the Done button without accept the change.
scACCEPT	user pressed the ACCEPT button, the change to the keyword has been commited.

- num_entities : number of entities for the same keyword in the current model.
- done_action : user can have a callout function installed here which will be called after the user dismissed the dialog; or user may list the commands here, and it will be executed accordingly after done button has been pressed.

Manipulation Functions

```
Int UtilPopupKeyword(Pointer pkeyword);
```

Call this function to pop up any keyword understood by LS-DYNA.

- pkeyword : the pointer to Keyword structure that contains the information about the keyword to be retrieved.
- return value : the index number of the Keyword form, it has the same value as the member of Keyword::formidx.

keywordInfo is the other structure that users use to retrieve information about a specific keyword.

```
struct KeywordInfo {
     char *name;
     Int id;
     Int is_valid;
     Int size;
     char *result;
};
```

- name : name of the Keyword, just like it is in structure Keyword, you should not include the '*' character in the name.
- id : keyword cards with this id will be retrieved.
- is_valid : one should always check this value to see if the information in KeywordInfo is valid. if the data is valid, this field will be non-zero, otherwise, it will be zero.
- size : size of the result buffer, in bytes.
- result : any information that would be expected in a buffer about an Keyword card will be written into this field. SCRIPTO allocate

more than enough for the buffer result, its actually size is calculated as followed:

```
size_of_result = ((size + 8) + 7) & ~7;
```

SCRIPTO also zero out all the used bytes of the buffer. Each manipulation function will explain how the result buffer being used.

Int UtilGetKeywordNumber(Pointer pki);

Get the number of the keywords in the model, only name field in the KeywordInfo is used as input.

- pki : pointer to the keywordInfo structure.
- return value : if is_valid is not zero after the function returned, the return value will be number of Keyword in the model.

void UtilGetKeywordContents(Pointer pki);

Get the content of a specific keyword with its id assigned. a non-existed id will not yield any result. result buffer will be packed as follow:

```
*keyword_name
card1_field1   card1_field2   card1_field3...
card2_field1   card2_field2   card2_field3...
....
cardn_field1   cardn_field2   cardn_field3...
```

SCRIPTO inserted each line a linefeed '\n' character and a null '\0' character. so the buffer will look like this:

```
*keyword_name\n\0card1_field1 card1_field2...\n\0\0\0..
```

- pki : pointer to the keywordInfo structure.

void UtilFreeKeywordBuffer(Pointer pki);

Free the allocated result buffer. although SCRIPTO does not require you to free the memory from KeywordInfo to work properly, it is a good practice for script writer to use this function to release the allocated memory back to the system.

- pki : pointer to the keywordInfo structure.

Keyword Access Functions

These are functions that directly access to LS-PREPOST's keyword database.

Int GetCurrentModelID(void);

Retrieve the model ID that is currently active in the system.

- return value: the model id used to switch between models.

`void SwitchModelTo(Int model_id);`

Switch current database retrieving function to focus on a certain model.

- model_id: id for a model to be switched to.

`Int HasKeyword(char *kywd);`

This function tests if a certain keyword exists.

- kywd: the keyword to be tested. Remove *, and _TITLE if needed.
- return value: return 1, if kywd is found in the current model; otherwise 0.

`Int HasKeywordWithID(char *kywd, Int id);`

This function test if a certain keyword with the user id exists. For a keyword that does not have id associated with it; use HasKeyword() instead.

- kywd: the keyword to be tested. Remove *, and _TITLE if needed.
- id: id for the keyword
- return value: return 1, if the keyword is found, otherwise return 0

`Int GetNumKeyword(char *kywd);`

Get the total number of keyword cards for this keyword. That means if in a model there is *NODE for 4 lines

```
*NODE
n1
n2
n3
n4
```

Then this function will return 4.

- kywd: keyword to be counted
- return value: Total number of keywords in the model, if the keyword is not supported or not found by the routine, it returns –1.

`Int GetKeywordIdList(char *keyword, Int *id_list);`

An id list for the keyword will be returned.

A script has to prepared an array that is big enough to save all the ids. one can always use **`GetNumKeyword`** to retrieve the size of a certain keyword before retrieving its id list.

- keyword: the keyword to be retrieved.
- id_list: an array to retrieved the id for this keywords
- return value: size of the array will be returned, it will be the same as **`GetNumKeyword`**().

`Int GetLargestKeywordID(char *keyword);`

The largest id for this keyword is returned. if the keyword was not supported, or not found, 0 will be returned.

- keyword : keyword to be found for its largest id
- returned value : if the keyword is not found or not supported, then 0 will be returned.

```
Int GetKeywordData(char *keyword, Int id);
```

This function retrieves keyword data per field from database. For keywords that does not has user id associate with it; the sequence of the same keywords that read into LS-PREPOST will be the id and for other keywords that can contain only one instance per keyword for each model, id is ignored.

This function will support all the LS-DYNA keywords that is supported by LS-PREPOST and updated accordingly whenever LS-PREPOST update its keyword reader.

- keyword: the keywords to be retrieved
- id: id to identify the keywords.
- return value: total number of fields for this keyword

```
struct KeywordFieldData {
     char      name[32];
     Int       is_type;
     char      val_str[80];
     Int       val_int;
     Float     val_float;
     Int       in_loop;
     Int       at_index;
};
```

KeywordFieldData is an object that script uses to retrieve/set the value for a certain keyword field on a keyword card. Following is the description for each field in the struct.

name	name for the current field, as you may see from LS-DYNA manual
is_type	if 1, it is an integer and val_int is valid if 2, it is a real number, and the val_float is valid if 3, it is a string, then val_str is valid.
val_str	only valid if is_type == 3
val_int	only valid if is_type == 1
val_float	only valid if is_type == 2
in_loop	identify is the field is in an iteration loop, [after July 12, 2007, this field is no longer in use]
at_index	directly access to the keywords through

	internal index.

void GetKeywordField(Pointer pkdf, Int icard, Int ifield);

Retrieve a keyword field from the last call to GetKeywordData(). A script has to provide icard and ifield for LS-PREPOST to identify the field requested.

Each GetKeywordData() will invalidate the previous result of the function call. icard and ifield are the same as they are in the keyword manual/content; for instance,

```
field elform for *SECTION_SHELL,
icard = 1
ifield = 2
```

when querying the database.

- pkdf: a pointer to a KeywordDataField. this function fills up a keyword field object
- icard: a one-based index; identify which card you are about to retrieve.
- ifield: a one-based index; identify which field to retrieve.

void SetKeywordField(Pointer pkdf, Int icard, Int ifield);

Modify a keyword field in the keyword fields retrieved by GetKeywordData(). icard and ifield is the same as it is in the keyword manual/content as they query the database.

Setting a field in a keyword card will not modify the database that actually save the data; a script has to use UpdateKeywordData() to update the database.

- pkdf: a pointer to a KeywordDataField. A script should assign the value to be modified in one of the val_ members in the data structure.
- icard: a one-based index; identify which card you are about to modify
- ifield: a one-based index; idendify which card you are about to modify

void UpdateKeywordData(void);

Update the currently retrieved keyword data to the database. This will modify the keywords from the file permanently. A script should update a keyword whenever it is going to change the content of a database

void ReleaseKeywordData(void);

Free the local memory held by the keyword fields every time it calls GetKeywordData(). GetKeywordData allocates necessary amount of memory to hold the information of a keyword, however, its internal memory holder will be overwritten everytime GetKeywordData() is called. This will create memory leakage if the script does not release the memory. Use this function for the purpose for release the memory.

```
Int CreateKeyword(Pointer keyword_name, Int id,
                  Int ncards, Int *fields_per_card,
                  TypedField *data);
```

There are several ways to create a keyword from a script, this is the plainest and most comprehensive way to do it. It directly interact with the database in LS-PREPOST and bypass all interventions from any user interface.

A new type of object is introduced to carry data for each keyword field.

```
struct TypedField {
     Int typelen;
     DataField df;
};
```

TypedField is a DataField with type information.

Possible **`typelen`** values :

- positive values: if typelen > 0, means that the field contains normal keyword value, first 2 bits (bit#0, and bit#1) decided how to interpret the Datafield.
- negative values: if typelen <0, DataField carries string values. however, the length of the string is not determined by (typelen >> 2). Instead, negative value make LS-PREPOST to look up in the manual. and reassign the length of the field according to the manual and align it to a 4-byte boundary.

typelen & 0x3	valid field in DataField
0	represents a not used field, or an empty field. When a field's typelen is zero, its value in Datafield will be ignored. Attempt to use any of the field in DataField may cause un-desirable results.
1	df.i is valid, df is actually an integer
2	df.f is valid, df is actually a float
3	df.a is valid, df is actually a pointer to a charater array. Size of this character array is **`(typelen >>2)`**;

- keyword_name: the keyword to be created
- id: ID for a new keyword, if a keyword does not have an id associate to it, set id to zero. For any keyword with ID, this id has to be the same as the id field in the data entry as required by LS-DYNA manual.
- ncards: number of cards for this keyword,
- fields_per_card: an array that contains number of fields in each card for this keyword.

- data: an array of data that comprise this keyword.

Remarks:

- Size of the data array will be

$$size = \sum_{i=0}^{ncards-1} fields\_per\_data[i]$$

- Each TypedField if the datafield type is a string, encode the typelen field as

$$typelen = ((size\_of\_string) << 2) \mid 0x3$$

- For negative typelen:

```
Ex

*PART_COMPOSITE
the first field in the manual is a heading. so that
its valid Datafield is a string.

TypedField heading;
heading.df.a = "A Part_Composite Heading";
heading.typelen =
   - (strlen(heading.df.a) << 2) | 0x3;
```

when LS-PREPOST reads the field, it will convert the typelen into positive value, and instead of taking strlen(heading.df.a) as the length to calculate the database entry, it will look up the LS-DYNA manual and replace the size for the heading in manual to this field. This size always align on a 4-byte boundary.

- return value: Following are the possible return values

0 (success)	A new keyword was created, no error occurs
-1(failed)	could not find a requested keyword, check spelling of the keyword
-2(failed)	there is a same entry in the database.
-3(failed)	No such main keyword

```
Int KeywordGetNumberOfCards(Pointer keyword, Int id);
```

Return the number of keyword cards in a keyword.

- keyword: keyword to be asked.
- id: id to query.
- return value:

> 0 (success)	return number of keyword cards in the keyword

-1(failed)	could not find the keyword
-2 (failed)	could not find the id for the keyword

BINOUT Database Access Functions

BINOUT files contains a collection of time history outputs regard of your LS-DYNA simulation. When you turn your BINOUT option on for a simulation, DYNA will provides a family of BINOUT files in the directory it runs. Because of its binary nature, SCRIPTO provides following functions for scripts take advantage of this concise binary time history database collection.

`void BinoutInit(void);`

Before LS-PREPOST is ready for a BINOUT file, it has to initialize itself internally to have all the interpreter ready. this is the function to initialize the reader.

`Int BinoutOpen(char *fname);`

To communicate with a BINOUT database, scripts need to obtain a handle from LS-PREPOST. Use this function to open a BINOUT family.

BINOUT database outputs are slightly different between SMP and MPP version of LS-DYNA; MPP creates a family of BINOUT files while SMP creates only a single BINOUT. LS-PREPOST will try to create a family of binout files assign by fname.

- fname :base BINOUT file name
- return value : a handle returned to the script for communicating purpose. A script use this handle to access different family of BINOUT files. Open failed, if this function returned a negative value.

`Int BinoutSetBranch(Int handle, char *newbranch);`

A BINOUT database might contain more than one ASCII history output branches such as MATSUM, GLSTAT, NODOUT, NODFOR, ELOUT/SHELL... Use this function to point the retrieving function to a certain branch.

- handle : a handle that returned from BinoutOpen().
- newbranch : a valid branch name can be any valid output of LS-DYNA on ASCII database. for a branch that contains sub-branches like ELOUT, "elout" will not be a valid branch, instead, "elout/beam", or "elout/solid" are valid branch names.
- return value : possible return value are

1(success)	A branch has been found in the current handle, and it has been set for retrieving data
0(failed)	requested branch can not be found in the current handle, however, the handle is valid.
-1(failed)	the handle value is not valid, BINOUT database might not open properly

```
struct BinoutTarget {
     Int start_state;
     Int end_state;
     Int id;
     char legend[72];
     char component[256];
};
```

BinoutTarget is the data structure used for communicating when retrieving a specific component for a branch.

There are several ways to retrieve a specific component from a BINOUT branch.

- start_state : the start state for the retrieving function to output the time history data, if not defined, start_state = 1 by default.
- end_state : the end state for the retrieving function to output the time history data; if not defined, end_state = the maximum state of the time history data, by default.
- id : can be 0, or a specific id for a component, if 0, or not defined, retrieving function will match an id based on legend.
- legend : can be left empty, (if id is assigned to a valid entity id); however, if not empty and the branch has "legend" output in the database, LS-PREPOST will match the non-white charaters of the legend and decide which entity to choose to ouput.
- component : the BINOUT component to be retrieved.

```
Ex:
In an input deck, it has the following keywords defined:
*DATABASE_HISTORY_NODE_ID
      8573group1
     10450group2
*DATABSE_NODEOUT
 2.0000E-4         3

Then when retrieving from BINOUT for the NODOUT history
for group2, the script can set BinoutTarget as following:
{
   BinoutSetBranch(handle, "nodout");
   BinoutTarget bt;
   strcpy(bt.legend, "group2");
   strcpy(bt.component, "x_acceleration");
   BinoutSetTarget(handle, &bt);
}
or
{
   BinoutSetBranch(handle, "nodout");
```

```
    BinoutTarget bt;
     bt.id = 10450;
    strcpy(bt.component, "x_acceleration");
    BinoutSetTarget(handle, &bt);
}
to get to same result.
```

`Int BinoutSetTarget(Int handle, BinoutTarget *target);`

Set a target for retrieving the curve data from a BINOUT database.

- handle : handle to the BINOUT database
- target : target to retrieve time history data in the database
- returned value : it can be any of the following:

0 (success)	A target has been set properly, the database is ready for retrieving time history data
-1(failed)	Handle that passed into the function is not valid
-2(failed)	branch was not set properly, it contains zero length in the branch name.
-3(failed)	branch currently pointed to can not be found at the handle that passed into the function.
-4(failed)	legend does not exist for the current branch. the branch that currently assigned for data retrieving does not contain legend information in the database
-5(failed)	there is no valid entity to be targeted in the database. This error occurs either because the legend to be searched was not found, or the id set to be searched was not found.
-6(failed)	can not find such a component in the current branch.

`Int BinoutGetCurve(Int handle, XYPair **curve);`

Retrieve the time history data into the curve.

- handle : the handle to retrieve the database
- curve : XYPair array that will be allocated by this function and filled with the data pointed by BinoutSetBranch() and BinoutSetTarget() functions. A script should free() the xy-pair to release the allocated memory.
- return value : number of pairs in the curve, if successful.

General Selection Mechanism

General Selection is a mechanism installed in the LS-PREPOST for different purpose of selections. General Selection allows user to select Parts, Elements, Segments, and Nodes through various ways. While selecting, objects are placed in a buffer for later use.

SCRIPTO wire this handy functionality such that script writers can take advantage of it. Here are the SCRIPTO structures for hooking up General Selection mechanism.

GenSelOptions

```
struct GenSelOptions {
     char *    target;
     int       clean;
};
```

Users uses this structure to assign what is the target to select.

- target : target strings for the user to start General Selection. Here are the options,

node	Make the selection targets to be nodes
beam	Make the selection targets to be beam elements
shell	Make the selection targets to be shell elements
quadshell	Make the selection targets to be 4-node shell elements
trishell	Make the selection targets to be 3-node shell elements
solid	Make the selection targets to be solid elements
hexa	Make the selection targets to be 8-node solid elements
penta	Make the selection targets to be 6-node solid elements
tetra	Make the selection targets to be 4-node solid elements
tshell	Make the selection targets to be thick shell elements
quadtshell	Make the selection targets to be 4-node thick shell elements
tritshell	Make the selection targets to be 3-node thick shell elements
sphnode	Make the selection targets to be the SPH elements
segment	Make the selection targets to be segments

inertia	Make the selection targets to be inertia elements
mass	Make the selection targets to be mass elements
discrete	Make the selection targets to be the discrete elements
seatbelt	make the selection targets to be the seatbelt elements
part	make the selection targets to be parts

- clean : if not zero, selection buffer will be cleared before the selection is initialized. otherwise, selection buffer will remain as it was before. New selections will be add into the end of the buffer.

GenSelValue

```
struct GenSelValue{
     int num_selection;
     Pointer sel_array;
};
```

- num_selection : the size of the array.
- sel_array : Following code segment should be the guide to cast the pointer correctly,

```
Selection *pselection;
SegmentSelection *psegment;
pselection = sel_array;
if(pselection[0].type == scSEL_SEGMENT)
{
     pselection = NULL;
     psegment = sel_array;
}
```

Selection and SegmentSelection

```
struct Selection {
     Int type;
     Int id;
     Int seq;
};
```

- type :type of the selection, it can be one of the following constants

scSEL_NODE	the selection type is a node
scSEL_SPHELEM	the selection type is a SPH element
scSEL_BEAM	the selection type is a beam element

scSEL_SHELL	the selection type is a shell element
scSEL_QUADSHELL	the selection type is a 4 node-shell element
scSEL_TRISHELL	the selection type is a 3 node-shell element
scSEL_SOLID	the selection type is a solid element
scSEL_HEXA	the selection type is a 8-node solid element
scSEL_PENTA	the selection type is a 6-node solid element
scSEL_TETRA	the selection type is a 4-node solid element
scSEL_TSHELL	the selection type is a thick shell element
scSEL_QUADTSHELL	the selection type is a 8-node thick shell element
scSEL_TRITSHELL	the selection type is a 6-node thick shell element
scSEL_PART	the selection type is a part
scSEL_SEGMENT	the selection type is a segment
scSEL_INERTIA	the selection type is an inertia element
scSEL_MASS	the selection type is a mass
scSEL_DISCRETE	the selection type is a discrete element
scSEL_SEATBELT	the selection type is a seatbelt element

- id : the id of the selection
- seq : sequence of the selection

```
struct SegmentSelection {
     Selection s;
     int conn[4];
};
```

- s : the inherited part of the selection object
- conn : the nodal id for the segment

Manipulation Functions

`void UtilBeginGenSelOptions(Pointer ges);`

Pass in a GenSelOptions and start the general selection process. the bottom panel will be changed into the general selection panel, with a  button.

Users may use the  button to come back to script bottom panel.

`void UtilGetGenSelInfo(Pointer gsif);`

User uses Get Selections from this function. The type of the sel_array in GenSelValue can be determined dynamically as followed:

1. Assign sel_array as a pointer to Selection
2. see if the type is scSEL_SEGMENT
3. and then assign the sel_array accordingly

- gsif : pointer to a GenSelValue

~~**`void UtilClearGenSelection(void);`**~~

The function without a pointer passed in is deprecated. it will no longer be supported in the LS-PREPOST2.2 This API changes its form to the following.

```
void UtilClearGenSelection(Pointer gsv);
```

This will clear the selection for the general selection so that you can use the selection again. The pointer to GenSelValue is needed for cleaning up the selection array.

- gsv : pointer to a GenSelValue

```
void UtilEndGenSel(void);
```

To end the general selection session programmatically, one should use this function, it will switch the general selection panel back to where the customized page.

Bitmaps

Bitmaps are created by through XPM files. A bitmap ID is returned to the scripts for further use.

Manipulation Functions

```
Int UtilCreateBitmap(char *name, char **xpm, Int len);
```

Create a bitmap.
len can be obtained from a C-Parser function

```
len = Length(xpm);
```

- name : name for the bitmap, a bitmap name can have a maximum length of 19 characters.
- xpm : the xpm data for the bitmap
- len : size of the xpm array.
- return value : a bitmap ID.

```
Int UtilSearchBitmap(char *name);
```

Return the first bitmap that matches the name in the bitmap pool.

- name : the bitmap name to be searched
- return value : a bitmap ID found, if not found, -1 will be returned.

```
void UtilBitmapGetSize(Int id, Int *width, Int *height);
```

Get Bitmap size.

- id : bitmap id that the size is to be retrieved.
- width : width of the bitmap, if bitmap is not found, -1 will be returned
- height : height of the bitmap, if bitmap is not found, -1 will be returned

```
void UtilDestroyBitmap(Int id);
```

Destroy a bitmap that has been created, if a bitmap of the assign ID is not found, the function will do nothing.

- id : bitmap to be deleted.

PlotWindow

PlotWindows are one of the utilities provided by LS-PREPOST. Users use Plot windows to show 2-D curves, such as define curves or history plots. First, you open a Plotwindow and then adding curves into the Plotwindow. and all other operations as you usually do in LS-PREPOST can be operate on the newly opened PlotWindow.

PlotWindow also can serve as a data source that provides the curves plotted by LS-PREPOST. By attaching a PlotWindow object to any XY-plots, one can retrieve the curve information in those plots. once you get the curve data, you may perform your own manipulation to the curve and then you can plot your curve into the other PlotWindow, or save it for later process. You should refer to UtilAttachPlotWindow() for more details.

```
struct PlotWindow {
     void open(void);
     void close(void);
     void redraw(void);
     void setcurve(Int npts, Float *x, Float *y);
     Int addcurve(Int npts, Float *x, Float *y);
     Int delcurve(Int idseq);
     Int hascurve(void);
     Int enumcurve(Int *cid);
     Int getcurve(Int cid, Float **x, Float **y);
     void hilightcurve(Int cid, Int onoff);
     Int canmodify(void);
};
```

- open : open a PlotWindow
- close : close a PlotWindow
- redraw : force a redraw to the PlotWindow
- setcurve : set a curve into the PlotWindow, this function will force the window to redraw, it could be costly when there are many curves added into the PlotWindow
    - npts : number of points in the curve
    - x : array that defines value of the abscissa of the curve
    - y : array that defines value of the ordinate of the curve
- addcurve : add an curve into the PlotWindow.
    - npts : number of points in the curve

    - x : array that defines value of the abscissa of the curve
    - y : array that defines value of the ordinate of the curve
    - return value : return the sequence of the added id. when deleting a curve, this is the id to pass into delcurve().
- delcurve : delete a curve in the plotwindow base on its sequence id in the window
    - idseq : sequence id for the curve, the id returns from addcurve()
- hascurve : when this PlotWindow has curves, it returns a non-zero value, otherwise, it returns zero
    - return value : the return value is the number of the curves currently add to the PlotWindow.
- enumcurve : enumerate current curves in the PlotWindow, script writer should allocate enough memory for cid array to retrieve data.
    - cid : curve ids.
    - return value : return the number of curves in the PlotWindow.
- getcurve : retrieve the x-y pairs from a curve. PlotWindow will allocate the memory needed for both abscissa and ordinate arrays.
    - cid : curve id to retrieve an array
    - x : pointer to the array for the abscissa of the curve
    - y : pointer to the array for the ordinate of the curve
    - return value : number of x-y pairs in this curve.
- hilightcurve : to highlight a curve by adding the width of the curve, same function can de-highlight a curve, by setting onoff to zero,
    - cid : curve to be highlighted
    - onoff : if non-zero, the curve will be drawn in the pen width wider than normal (+ 3 points) or until the upper limit is reached (7 points) and if 0, the curve will be drawn in a pen width thinner than normal (-3 points) or until the lower limit is reached (1 point).
- canmodify : test if a PlotWinodw can be modified. that is if one use UtilAttachPlotWindow to associate a PlotWindow object to a plot window, this function will return false, otherwise, it will return true.

Picasso Drawing Routines

A Picasso is a set of drawing routines that can be associated with a 2D Drawing Widgets. When a script handles the paint event of on a Drawing widget, it should provide a callout function with the following prototypes:

```
define:
void call_out_function_for_paint(Picasso p)
{
     /**
          drawing details.
     */
}
```

SCRIPTO will provide the Picasso object for drawing. Scripts should not save this Picasso or copy its content to any other Picasso object. This Picasso will only be valid in the drawing script function, and consider invalid once leave the function border.

Currently Picasso can be associated with Drawing widgets.

```
struct Picasso {
     Int       ready(void);
     void      setbackground(RGB bg);
     void      setforeground(RGB fg);
     void      cleararea(Rectangle area);
     void      setlinewidth(Int width);
     void      drawline(Int x1, Int y1, Int x2, Int y2);
     void      drawrectangle(Rectangle r);
     void      fillrectangle(Rectangle r);
};
```

- ready : function to test if a Picasso is ready. Picassos provided by SCRIPTO's callout mechanism will always be ready.
  - returned value : if a Picasso is ready return 1, otherwise 0.

- setbackground : Set the background color through a RGB object. background color affects several drawing routines, such as cleararea().
  - bg : the RGB object for background

- setforeground : Set the foreground color through a RGB object. foreground color are colors for painting, such as drawline().
  - fg : the RGB object for foreground

- cleararea : clear a rectangle on the canvas where Picasso can draw on. this routine use the background color currently assign to the Picasso to draw the designated rectangle.
  - area : a rectangle area to be cleared

- setlinewidth : set the current line width of the Picasso. The line width will be applied to all line-drawing routines, including drawrectangle and drawline
  - width : witdth of the line

- drawline : draw a line in the foreground color and according to the line width and style assign to it.
  - x1 : x-coordinate for the starting point
  - y1 : y-coordinate for the starting point
  - x2 : x-coordinate for the ending point
  - y2 : y-coordinate for the ending point

- drawrectangle : draw a rectangle border using foreground color and the border width is dictated by setlinewidth().
    - o r : a Rectangle object that defines the area
- fillrectangle : similar to drawrectangle() but with is interior filled with the foreground color
    - o r : a Rectangle object that defines the area

Highlight Mechanism

Highlight mechanism applies to keyword entities created by keyword routines. Entities recognized by DataCenter or created by DataCenter can be highlighted through this utility.

There are 3 general steps to use this mechanism:

- Enable Highlight mechanism
- Call the Highlight routines
- Refresh the model

Highlight mechanism in LS-PREPOST is asynchronous, which means that when a routine is called to highlight or de-highlight an object, it will not be executed at the time the function is called. Instead the function set the internal of the objects to be highlighted, and when next draw event is executed, the picture on the canvas will then be refreshed.

and a call to refresh model ensures a drawing event is issued.

```
void EnableHighlightEntities(Int onoff);
```

Enable or disable this mechanism in the script.

- onoff : if non-trivial, the entities will be highlighted, when set.

```
void DataHighlightSet(Pointer dcpv, Pointer typeid, Int onoff);
```

Highlight a set.

- dcpv : a pointer to a DataCenter where the set is.
- typeid : a pointer to a TypeID object that represents a set
- onoff : turn the internal highlight bit of the set on if non-trivial.

DataCenter

DataCenter represents the Model data of the LS-PREPOST2.1. LS-PREPOST allows you to manipulate more than one model at the same time, this makes every DataCenter manipulation function will have a DataCenter pointer to tell which model the user refers to. DataCenter provides some basic information for the user to understand the model.

These basic information includes the geometric data , the boundary and initial conditions, the controls for outputs and other parameters.

```
struct DataCenter {
     int       id;
     char *    name;
     int       numparts;
     int       numelems;
     int       numnodes;
     int       has_beam;
     int       has_shell;
     int       has_solid;
     int       has_tshell;
     int       has_sph;
     int       has_mass;
     int       has_discrete;
     int       has_seatbelt;
     int       has_inertia;
};
```

- id : Each model has a unique id to identify itself in LS-PREPOST. this id will be used as DataCenter id.
- name : name of the model
- numparts : number of parts in the center
- numelems : number of elements in the center
- numnodes : number of nodes in the center
- has_beam : if not zero, it is the number of beam elements in the model
- has_shell : if not zero, it is the number of shell elements in the model
- has_solid : if not zero, it is the number of solid elements in the model
- has_tshell : if not zero, it is the number of tshell elements in the model
- has_sph : if not zero, it is the number of sph elements in the model
- has_mass : if not zero, it is the number of the mass elements in the model

- has_seatbelt : if not zero, it is the number of the seat belt elements in the model
- has_inertia : if not zero, it is the number of the inertia elements in the model.

All Entity data is retrieved and modified by the manipulation functions.

Manipulation functions

`Int DataGetNumActiveModels(void);`

Inquire the system how many active models are in the application.

- return value : return the number of models in the application so far

`Int *DataGetActiveModelIDList(void);`

Retrieve the id of the active models

- return value : return an array for model id.

`void DataFreeActiveModelIDList(Int *ip);`

Release the memory occupied by the id list retrieved back from the DataGetActiveModelIDList.

- ip : id array for the model id.

`Int DataFindModelByBaseFilename(Pointer fname);`

The base file name of a model is the first file loaded or imported by LS-PREPOST for the model. The parameter can include the path where the file is.

- fname : file name for the model, if path is not included, the index of the first model that has the name will be returned.
- return value : the index for the model with that base file name.

`void DataImportFrom(Pointer dc, Int dc_id);`

Users want to use a DataCenter should use this function to import a DataCeter to the data structure.

- dc : a pointer to a DataCenter
- dc_id : a model id

`Int DataHasModel(Pointer dc);`

The function check if a DataCenter has already linked with a model.

- dc : a dc to be checked
- return value : if negative, this datacenter does not link with a model, otherwise, it is the model id that link with it.

`void DataRelease(Pointer dc);`

Release the current DataCenter reference to a model, after a script called this function, it should not use the DataCenter to retrieve any data.

- dc : a pointer to a DataCenter

```
void DataMakeCurrent(Pointer dc);
```

When there are more than one DataCenter around, user can switch the model that represented by the DataCenter to become the current DataCenter.

- dc : a pointer to a DataCenter

```
Pointer DataGetModelKeywordList(Pointer dc, Int *n);
```

Get a keyword list that is used in the model.

- dc : DataCenter to be counted
- n : number of keywords returned
- return value : pointer to a '\0'-separated character array that contains the list of keywords used in the model.

```
Int DataReadKeyword(Pointer dc, Pointer keyword);
```

Make the datacenter to include the keyword into the database.
This is a cruel way of assigning data, in the keyword string, script can import as many keywords as it likes, and a temporary file is created under users' path to contains the keywords.

- dc : Datacenter where the keywords are to be plugged in.
- keyword : a string that contains all the keywords. Each card of every keyword should be separated by a '\0' character. all keywords are taken as is, no further checking is performed on the cards or keywords. Put two '\0' characters to terminate the input string.
- returned value : non-zero if successful; zero, otherwise.

PartInfo

PartInfo is a structure that contains a part's info. Users may consider it as a vehicle to transport the information in and out of the DataCenter.

```
struct RGB {
  Int red;
  Int green;
  Int blue;
};
struct PartInfo {
     int       id;
     int       mat_id;
     int       type;
     int       plotmode;
     Float     transp;
     RGB       color;
     char *    title;
     char *    grouptag;
```

```
};
```

- id : part user id
- mat_id : material user id, used by the part
- type : type for a Part. can be following values

scETYPE_BEAM	Elements that constitute for the part are beams
scETYPE_SHELL	Elements that constitute for the part are shells
scETYPE_SOLID	Elements that constitute for the part are solids
scETYPE_TSHELL	Elements that constitute for the part are thick shells
scETYPE_SPHELEM	Elements that constitute for the part are SPH elements
scETYPE_MASS	Elements that constitute for the part are Mass elements
scETYPE_DISCRETE	Elements that constitute for the part are discrete elements
scETYPE_INERTIA	Elements that constitute for the part are inertia elements
scETYPE_SEATBELT	Elements that constitute for the part are seat belt elements

- plotmode : plot mode of the current part, it can be the following values.

scPLOTMODE_SHADE	Draw current part in the shading mode, with lighting effect. This is the default appearance of a part if not specified
scPLOTMODE_FRINGE	Fringe the current part with fringe component recently set.
scPLOTMODE_VIEW	Draw current part without lighting effect.
scPLOTMODE_WIRE	Draw each element in the part with nodes and their connectivities
scPLOTMODE_EDGE	Draw the part's outline edges
scPLOTMODE_HIDE	Draw the part in wire mode, but apply the hidden line removal algorithm
scPLOTMODE_FEAT	Draw the part as its feature lines.
scPLOTMODE_GRID	Draw the part in a node cloud
scPLOTMODE_MESH	Draw the part in its mesh lines
scPLOTMODE_SHRINK	move each nodes of elements in the part toward the center of gemoetry

- transp: Transparency of the part. You may assign this value to change the transparency level of each part separately. 0 means opaque, while 1.0 means totally transparent.
- color: Color of the part. Composed by 3 integer numbers -- that is red, green and blue; these 3 numbers should have value in the range of 0 and 255.
- title: Title for the part. for a keyword model, the title is the same as the title found in *PART.
- grouptag: assign a part into a group by giving a name. A name can not have white characters and can not exceed 20 characters in length.

Manipulation Functions

`RGB  RGB(Int red, Int green, Int blue);`

The constructor for a RGB object.

- red : red attribute for the RGB object.
- green : green attribute for the RGB object.
- blue : blue attribute for the RGB object.

`Int DataGetPartIdList(Pointer dc, Int **ids);`

Retrieve the part id back in an integer array. SCRIPTO will allocate memory for array ids; it is a good practice to call free() on ids when ids is no longer needed. size of the Id list will be returned.

- dc : pointer to a DataCenter.
- ids : scripts does not need to allocate memory for the ids. the function will allocate enough memory to hold the data.
- return value : size of the array, it should be the same as the datacenter's numparts member.

`void DataGetPartInfo(Poiner dc, Pointer pi);`

Retrieve the part information by giving the part ID in the PartInfo structure.

- dc : pointer to a DataCenter
- pi : pointer to a PartInfo. to retrieve the information for a part, user has to set id in the PartInfo.

`void DataSetPartInfo(Pointer dc, Pointer pi, char *items);`

Set the information for a part.

- dc : pointer to a DataCenter
- pi : pointer to a PartInfo. User should fill in data for id, and fields that user wants to change for this part.
- items : a string that assign fields contains valid data in the PartInfo structure.

`void DataHighlightPart(Pointer dc, Int partid, Int on_off);`

Highlight or unhighlight a part.

- dc : pointer to a DataCenter
- partid : part id to be hightlighted
- on_off : highlight or un-highlight a part. 1, for highlight; 0, for un-highlight.

```
void DataCreatePart(Pointer dc, Pointer pi);
```

Create a brand new part.

- dc : pointer to a DataCenter
- pi : pointer to a PartInfo. Certain fields of the PartInfo are needed:
  - id : new part id.
  - title : this field is optional, part will have no title if leave blank to this field.

```
Int DataCollectNodesFromPart(Pointer dc, Int pid,
                             Int **nids);
```

Collect the node id from a part and return the number of nodes contained in the part. User should free() the nids after calling this function.

- dc : pointer to a DataCenter
- pid : part id which part's nodes are about to be collected
- nids : pointer to an integer array, where the collected node ids are stored.
- returned value : number of nodes in the part.

```
void DataSetNodeCoord(Pointer dc, Int nid,
                      Float x, Float y, Float z);
```

Set nodal coordinate for a specific node.

- dc : pointer to a DataCenter
- nid : nodal id
- x : x coordinate for the node
- y : y coordinate for the node
- z : z coordinate for the node

```
void DataGetNodeCoord(Pointer dc, Int nid,
                      Float *x, Float *y, Float *z);
```

Get nodal coordinate for a specific node.

- dc : pointer to a DataCenter
- nid : nodal id
- x : a Float pointer where the x-coordinate returned
- y : a Float pointer where the y-coordinate returned
- z : a Float pointer where the z-coordinate returned

```
Int DataCreateNode(Pointer dcpv, Int nid, Float x, Float y,
```

```
                Float z);
```

Create a node. If the supplied nid was already in the model, this function will modify the coordinate of nid without warning.

- dcpv : pointer to a DataCenter
- nid : new nodal id
- x : the x-coordinate of the new node
- y : the y-coordinate of the new node
- z : the z-coordinate of the new node
- returned value : number of new nodes created.

```
Int DataCreateNodes(Pointer dcpv, Int nnp, Int *nids,
                Vector3 *v);
```

Create multiple nodes. If any of the nids has been in the model the new coordinate values in v will set the node to a new location.

- dcpv : pointer to a DataCenter
- nnp : number of nodes in nids and v
- nids : nodal id array
- v : vector that contains the coordinates of the nodes to be created

```
struct Vector3 {
     Float x;
     Float y;
     Float z;
};
```

Vector3 is a simple structure which contains 3 real numbers. and we borrow its data structure to create the vector array when create nodes.

- returned value : number of new nodes created.

```
Int DataGetAllNodeIds(Pointer dcpv, Int **ids);
```

Get all Node Ids in the arrary ids. Scripts that retrieves node id this way should aware that the nodes normally is the biggest group of data in a model.

- dcpv : pointer to a DataCenter
- ids : array that will hold a copy of current node id in the model, user should free() the ids array after use.
- return value : return the number of the nodes in the nodal id array.

```
Int DataGetFirstNodeId(Pointer dcpv);
```

Get the first Node Id in a model. with the other function DataGetNextNodeId(), a script can visit all node id in a model.

- dcpv : pointer to a DataCenter
- returned value : the id of the first node in the database.

```
Int DataGetNextNodeId(Pointer dcpv);
```

Get the next Node Id in a model. This function does not work alone; a script should use DataGetFirstNodeId() first to get the first ID and then use this function to visit all the nodes in a model.

- dcpv : pointer to a DataCenter
- return value : a node id will be return (which has to be positive) if not all nodes in a model has been visited. return 0, when it passed the end of the nodal array.

ElementInfo

ElementInfo contains the information about elements:

```
struct ElementInfo {
     int id;
     int type;
     int conn;
     int nids[10];
};
```

- id : user id for an element
- type : Following are types valid

scETYPE_BEAM	For beam elements
scETYPE_SHELL	For shell elements
scETYPE_SOLID	For solid elements
scETYPE_TSHELL	For tshell elements
scETYPE_SPHELEM	For SPH elements
scETYPE_MASS	For mass elements
scETYPE_DISCRETE	For discrete elements
scETYPE_INERTIA	For inertia elements
scETYPE_SEATBELT	For seat belt elements

- conn :connectivity of an element. this is how we differentiate between different variation of elements of same type. For example the solid elements, there are 4 possible values for this variable
  conn = 4; tetrahedral elements
  conn = 6; 6-nodes solid elements, the wedge.
  conn = 8; hexahedral elements
  conn = 10; 10-nodes tetrahedral solid elements
- nids : node id that constitute the element, the valid size of the array is determined by conn.

Manipulation Functions

```
Int DataCollectElementsFromPart(Pointer dc, Int pid,
                                Int **eids);
```

Collect all element in a part and store them in an element id array

- dc : a pointer to a DataCenter
- pid : part id

- eids : pointer to an integer array, where the collected element ids are stored.
- return : return the number of elements in the part

```
void DataGetElementInfo(Pointer dcpv, Pointer eip);
```

Get the information for an element.

- dcpv : a pointer to a DataCenter
- eip : a pointer to an ElementInfo

To use this function, users should fill in the id field for retrieving the element information.

```
void DataSetElementInfo(Pointer dcpv, Pointer eip);
```

Set the information for an element.

- dcpv : a pointer to a DataCenter
- eip : a pointer to an ElementInfo.

Every field of the elementInfo will be used to change the data contained in an element.

```
Int DataAssignElementToPart(Pointer dcpv, Int eid,
                            Int pid);
```

Assign an element to a new part.

- dcpv : a pointer to a DataCenter
- eid : element id
- pid : new part id
- returned value : successful if non-zero, otherwise, failed.

```
Int DataCreateElement(Pointer dcpv, Int pid, Pointer ei);
```

Create an element. The part this element belongs to has to be existed first. Variation of element types are determined by the conn field in the ElementInfo. This function only takes care of connectivities; all other attributes are empty for now.

- dcpv : a pointer to a DataCenter.
- pid : id of an existed part or a created empty part.
- ei : a pointer to an ElementInfo. All fields are used in the ElementInfo.

```
Int DataCreateElements(Pointer dcpv, Int pid,
                       Int nne, Pointer ei);
```

Create more than an element at a time.

- dcpv : a pointer to a DataCenter
- pid : id of an existed part.
- nne : size of the ei array, number of the elements to be created.
- ei : array for ElementInfo.

```
Int DataGetElementIdList(Pointer dcpv, Int etype,
                         Int **ids);
```

Get an array of the element id in this DataCenter.

- dcpv : a pointer to a DataCenter
- etype : element type for this function to report the element IDs. valid element types include:

scETYPE_BEAM	For beam elements
scETYPE_SHELL	For shell elements
scETYPE_SOLID	For solid elements
scETYPE_TSHELL	For tshell elements
scETYPE_SPHELEM	For SPH elements
scETYPE_MASS	For mass elements
scETYPE_DISCRETE	For discrete elements
scETYPE_INERTIA	For inertia elements
scETYPE_SEATBELT	For seat belt elements

- ids : pointer to the array to be returned. the API will allocate needed memory for the array. Scripts need to free() the memory returned to prevent memory leak.
- returned value : number of elements in the array. if there is no element of such kind, 0 will be return.

MaterialInfo

MaterialInfo is to represent the material that has been used in a specific part.

```
struct MaterialInfo {
     int       part;
     int       mid;
     char*     name;
     char*     title;
};
```

- part : part id of where the material is to be associated with
- mid : material id to the part
- name : name of the material
- title : title for the material

Manipulation Functions

```
void GetMaterialInfo(Pointer dc, Pointer mi);
```

Retrieve the material information from a part id.

- dc : a pointer to a DataCenter
- mi : a pointer to a MaterialInfo. Users should set the 'part' field to the part Id which that the material information is to be retrieved.

TypedID

TypedID is a small structure that represents model entities with or without id. For entities that are without id, the id field will be id of the object that the entity associates with.

For example, for a boundary condition SPC, a node set id that this SPC applied to is the id of the TypedID for the SPC.

```
struct TypedID {
     Int id;
     Int type;
};
```

Set - groups of geometric data

The set data in LS-DYNA is used extensively in a lot of keywords; you may use the following functions to create equivalent set data from your scripts.

Since the number of members of set can always modify later, there are two states for a set. In any give time, once a set is created, it can be realized or un-realized. A set is realized, if it is created in the database. This means that if a set is not realized, even the user tried to write a keyword file, it will ignore an unrealized set.

```
void DataCreateSet(Pointer dc, TypedID typeid);
```

This will create an un-realized set with the assigned id and type. This set will be created with no member in it.

- dc : datacenter where this set data is created in.
- typeid : currently, you may create a set of following element types.

scSEL_BEAM	Create a *SET_BEAM
scSEL_SHELL	Create a *SET_SHELL
scSEL_SOLID	Create a *SET_SOLID
scSEL_TSHELL	Create a *SET_TSHELL
scSEL_NODE	Create a *SET_NODE
scSEL_PART	Create a *SET_PART
scSEL_SEGMENT	Create a *SET_SEGMENT

```
Int DataHasSet(Pointer dc, TypedID set);
```

Check if a DataCenter has the specific set.

- dc : datacenter where the set is queried
- set : the set to be queried
- return value : return 1, if set is found, otherwise 0.

```
Int DataCountSets(Pointer dc, Int type);
```

Return the counts of set that are in the datacenter.

- dc : number of set to be counted in the datacenter

- type : allowed types are the same as the creation function.

```
Int DataSetGetNextID(Pointer dc, Int type);
```

Automatically get the next available Set ID for the model for a given type.

- dc : datacenter where the set ID is acquired
- type : allowed types are the same as the creation function.

```
Int DataSetAddMember(Pointer dc, TypedID set,
                     Int memberid);
```

Add a member into a created set. If the set is realized, it will also update the database in LS-PREPOST. This function can not be used to add an entry for a segment set. To add a segment into a segment set, use **DataSetAddSegment**.

- dc : datacenter where the set is created
- set : a TypedID object that represents the set
- memberid : depends on the set, this can be the id for a node or an element or a part.
- Return value : if successful returns 1, otherwise 0.

```
Int DataSetAddSegment(Pointer dc, TypedID set,
                      SegmentItem item);
```

Add a segment into a created segment set. If the segment set is realized, it will also update the database in LS-PREPOST. This function can not be used to anything else but for a segment set. To add a member to a set other than a segment set, use **DataSetAddMember.**

- dc : datacenter where the set is created;
- set : a TypedID object that represents the set.
- item : a SegmentItem contains the information of a segment.
- return value : return 1, if successful; otherwise, 0.

```
struct SegmentItem {
  Int n[4];
  Int attrib[4];
};
```

SegmentItem is the object that represents a segment. All segment set functions use this object to pass information.

- n : Four nodes for the segment
- attrib : the correspond attribute that you can define for the segment.

```
Int DataSetRemoveMember(Pointer dc, TypedID set,
                        Int memberid);
```

Remove a member from a created set. If the set is realized, it will also update the database in LS-PREPOST. This function will not remove a segment from a segment set. To remove a segment, use **DataSetRemoveSegment**.

- dc : datacenter which contains the set.
- set : a TypedID object that represents the set.
- memberid : depends on the set, this can be the id for a node or an element or a part.
- return value : return 1, if successful; otherwise, 0.

```
Int DataSetRemoveSegment(Pointer dc, TypedID set,
                         SegmentItem segids);
```

Remove a segment from a created segment set. If the segment set is realized, it will also update the database in LS-PREPOST. This function will not remove any member other than segments. To remove a member of other sets, use, **DataSetRemoveMember**.

- dc : datacenter which contains the set.
- set : a TypedID object that represents the set.
- segids : a SegmentItem object that represents the segment.
- return value : return 1, if successful; otherwise, 0.

```
void DataSetAssignAttrib(Pointer dc, TypedID set,
                         Int index, Float attrib);
```

Assign extra data to a set. LS-DYNA *SET allows user to assign 4 extra data to a set. The index assigns which field of the data is to be set. Index is zero based.

- dc : datacenter which contains the set.
- set : a TypedID object that represents the set.
- index : an index to assign which extra data to be set
- attrib : the data for the extra field.

```
void DataSetRealize(Pointer dc, TypedID set);
```

This function is to realize a set. This will create the set into memory of LS-PREPOST's database.

- dc : datacenter that contains the set.
- set : set to be realized.

```
Int DataSetIsRealized(Pointer dc, TypdeID set);
```

A query function. this function check to see if a set is realized.

- dc : datacenter that contains the set.
- set : set to be realized.
- return value : return 1 if it is realized, otherwise, 0.

```
void DataDestroySet(Pointer dc, TypedID set);
```

Destroy a set in the model, if it is realized, the database copy will be removed also.

- dc : datacenter that contains the set.
- set : set to be destroyed.

```
Int DataSetGetCount(Pointer dc, TypedID set);
```

Get numbers of members in a set.

- dc : datacenter that contains the set.
- set : data set to be counted.
- return value : return the number of members of a set.

```
Int DataSetSearchMember(Pointer dc, TypedID set,
                        Int memberid);
```

Search a member in a set. This function does not apply to search a segment set, you have to use **DataSetSearchSegment**

- dc : datacenter that contains the set.
- set : data set where the member is to be searched
- memberid : member id to be searched
- return value : return the index of the location where the member is (zero based), if found, otherwise, -1.

```
Int DataSetSearchSegment(Pointer dc, TypedID set,
                         SegmentItem item);
```

Search a segment in a set. you can not use this function to search in a set other than segment sets. To search a member of other set, use **DataSetSearchMember**.

- dc : datacenter that contains the set
- set : data set to be searched
- item : a SegmentItem item, it contains the data to be searched.
- return value : return the index of the location where the member is (zero based), if found, otherwise, -1.

```
Int DataSetReplaceMember(Pointer dc, TypedID set,
                         Int idex, Int memberid);
```

Replace a member at assigned index in the data set;

- dc : datacenter that contains the set
- set : data set which the member belongs to.
- idex : the index of the array where the memberid is about to replace.
- return value : return 1, when successful, otherwise, 0

```
Int DataSetReplaceSegment(Pointer dc, TypedID set,
                          Int idex, SegmentItem item);
```

Replace a segment at assigned index in the segment set.

- dc : datacenter that contains the set
- set : data set which the segment belongs to.
- idex : the index of the array where the segment is about to replace
- return value : return 1, when successful, otherwise, 0

```
Int DataSetDumpMembers(Pointer dc, TypedID set, Int **id);
```

Dump the whole set members to the id array. This array will be allocated by LS-PREPOST. The script should not provide an allocated array. The script, however, should free the array after using it.

- dc : datacenter that contains the set
- set : dataset for id dump.
- id : pointer to an integer array, that will be allocated with memory and loaded with the member ids for the set.
- return value : number of members, when successful, 0, otherwise.

```
Int DataSetDumpSegments(Pointer dc, TypedID set,
                        SegmentItem **idseg);
```

Dump the whole set of segments in to the SegmentItem array. This array will be allocated by LS-PREPOST. The script should not provide an allocated pointer to the function. The script, however, should free the array after using it.

- dc : datacenter that contains the set.
- set : data set for the segment dump.
- idseg : SegmentItem array allocated by LS-PREPOST
- return value : number of segments, when successful, 0, otherwise.

```
Int *DataGetSetList(Pointer dc, Int type);
```

Collect the set id of the type in a DataCenter.

- dc : datacenter which the set list is to be retrieved.
- type : the type allowed is the same as the creation function
- return value : an allocated array of set ID.

DefineCurve - Two-dimensional XY curves

DefineCurve is the object that encapsulates the keyword *DEFINE_CURVE. The object defines a general purpose curve that does not check if the abscissa value increases monotonically.

DefineCurve is somewhat similar to a set data, since its xy-pair varies in every curve, a curve has to be realized to add the curve into a datacenter and hence can be output by LS-PREPOST and used by other objects.

DefineCurve works closely with a primitive structure **`XYPair`**:

```
struct XYPair {
     Float x;
```

```
    Float y;
};
```

XYPair is the structure that represents a point of a curve, however, this data structure can be used in other occasion as well, as long as it is a pair of two real values.

- x : the abscissa value of the pair
- y : the ordinate value of the pair

```
void DataCreateDefineCurve(Pointer dc, Int idcrv);
```

DefineCurves are id-ed entities. in the functions that manipulate the object, we use its id to identify the curve we are operating, this function create a curve with no point in it.

- dc : datacenter where the curve is created.
- idcrv : ID for the DefineCurve.

```
Int DataDefineCurveAddPoint(Pointer dc, Int idcrv, Float x,
                            Float y);
```

Add a point to the curve. this function will add the current (x, y) to the last point in the curve, if there is no point in the curve, this will be the first point of the curve.

- dc : datacenter that contains the DefineCurve.
- idcrv : ID for the DefineCurve.
- x : x-value for the point
- y : y-value for the point
- return value : if successful, return number of points in the curve, otherwise, 0.

```
Int DataDefineCurveRemovePoint(Pointer dc, Int idcrv,
                               Int idx);
```

Add remove a point at the specific index of the curve.

- dc : datacenter that contains the DefineCurve.
- idcrv : ID for the DefineCurve
- idx : index of the point to be removed. The index is zero-based; that is the idx == 0 for the first point.
- return value : return 1, if successful, 0, otherwise.

```
XYPair DataDefineCurveGetPoint(Pointer dc, Int idcrv,
                               Int idx);
```

Retrieve a point from a DefineCurve in the form of XYPair.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve
- idx : index of the point to be retrieved. The index is zero-based; that is the idx == 0 for the first point.
- reurn value : the XYPair returned from the curve.

```
Int DataDefineCurveInsertPoint(Pointer dc, Int idcrv,
                               Int idx, Float x, Float y);
```

Insert a point at the position idx.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve
- idx : Index for the point to be inserted
- x : x value for the point that is about to be inserted
- y : y value for the point that is about to be inserted
- return value : 1, if successful, 0, otherwise.

```
Int DataDefineCurveGetCount(Pointer dc, Int idcrv);
```

Get the number of points for the curve.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve
- return value : 1, if successful, 0, otherwise.

```
Int DataDefineCurveDumpPoints(Pointer dc, Int idcrv,
                              XYPair **dumpxy);
```

Dump the points of the curve into a XYPair array.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve
- dumpxy : a XYPair array allocated by the LS-PREPOST. The script should provide only a pointer to a pointer of XYPair, to use the function; however, the script should free the pointer after using the array.
- return value : return the number of points in the curve.

```
Int DataDefineCurveSetOffset(Pointer dc, Int idcrv,
                             Float offx, Float offy);
```

Set offsets to both abscissa and ordinate axis for the curve, it applied the same way as it is described in LS-DYNA keyword users manual.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve
- offx : offset apply to the abscissa axis
- offy : offset apply to the ordinate axis
- return value : return 1, when successful, otherwise, 0.

```
Int DataDefineCurveSetScale(Pointer dc, Int idcrv,
                                Float sclx, Float scly);
```

Set scales that will be applied to both abscissa and ordinate axes.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve

- sclx : scale that will apply to the curve to the abscissa axis
- scly : scale that will apply to the curve to the ordinate axis
- return value : 1 if successful, 0, otherwise.

`void DataDefineCurveRealize(Pointer dc, Int idcrv);`

Realize a DefineCurve object.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve.

`Int DataDefineCurveIsRealized(Pointer dc, Int idcrv);`

Report if a DefineCurve is realized.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve.
- return value : 1, if the curve is realized, 0, otherwise.

`void DataDestroyDefineCurve(Pointer, Int idcrv);`

Destroy a DefineCurve. If a DefineCurve is realized, its database copy in LS-PREPOST will also be removed.

- dc : datacenter that contains the DefineCurve
- idcrv : ID for the DefineCurve.

Vector - to define a 3-D direction

Vector is the other kind of ID-entities. Create this kind of entities in a DataCenter will generate *DEFINE_VECTOR keywords in the input deck.

Vector is auto-realized, which means that scripts do not need to realize any vectors as it should for DefineCurve and Set. the creation function create a vector and realize it automatically. Vector that assigns with magnitudes alone will originate from global (0, 0, 0), and there is a member function to offset the vector's origin.

Vector also works closely with structure **`Vector3`**.

`void DataCreateVector(Pointer dc, Int idvec, Vector3 dir);`

Create a vector entity with direction assigned. this vector will be created originates from global origin.

- dc : datacenter that contains the vector
- idvec : ID of the vector to be created
- dir : the predefined vector magnitude that goes along with creation.

`void DataDestroyVector(Pointer dc, Int idvec);`

Destroy a vector entity.

- dc : datacenter that contains the vector
- idvec : ID of the vector to be destroyed

```
Int DataHasVector(Pointer dc, Int idvec);
```

Check if the datacenter has the vector entity.

- dc : datacenter that contains the vector
- idvec : ID of the vector to be checked.
- return value : 1, if exists, 0, otherwise.

```
void DataVectorModify(Pointer dc, Int idvec,
                      Int idx, Float val);
```

Modify a component in direction for a vector.

- dc : datacenter that contains the vector
- idvec : ID of the vector to be motified.
- idx : index (0, 1, 2) to modify the component of the vector direction.
- val : the component value.

```
void DataVectorNormalize(Pointer dc, Int idvec);
```

Normalize vector's length to unity.

- dc : datacenter that contains the vector
- idvec : ID of the vector to be normalized.

```
void DataVectorSetOrigion(Pointer dc, Int idvec,
                          Vector3 origin);
```

Set vector's origin.

- dc : datacenter that contains the vector
- idvec : ID of the vector to set origin.
- origin : new origin for the Vector.

Box - to define a 3-D rectangle box

A box entity encapsulates the *DEFINE_BOX keyword for LS-DYNA. Boxes are also auto-realized, once a box is created, it is always realized. Box entities also work closely with `Vector3`.

```
void DataCreateBox(Pointer dc, Int idbox,
                   Vector3 min, Vector3 max);
```

Create a box with the diagonal points assigned.

- dc : datacenter that contains the vector
- idbox : ID of the box to be created.
- min : corner of the box that contains the minimum coordinates of the box extent
- max : corner of the box that contains the maximum coordinates of the box extent.

```
void DataDestroyBox(Pointer dc, Int idbox);
```

Destroy a box.

- dc : datacenter that contains the vector
- idbox : ID of the box to be destroyed.

```
Int DataHasBox(Pointer dc, Int idbox);
```

Check to see if a box exists.

- dc : datacenter that contains the vector
- idbox : ID of the box to be checked.
- return value : 1, if box exists, 0, otherwise.

```
void DataBoxSetMin(Pointer dc, Int idbox,
                   Int idx, Float val);
```

Modify a box's minimum corner.

- dc : datacenter that contains the vector
- idbox : ID of the box to be modified.
- idx : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified

```
void DataBoxSetMax(Pointer dc, Int idbox,
                   Int idx, Float val);
```

Modify a box's maximum corner.

- dc : datacenter that contains the vector
- idbox : ID of the box to be modified.
- idx : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified

Coordinate - to define a local coordinate system

Coordinate encapsulates the keyword *DEFINE_COORDINATE_SYSTEM for LS-DYNA. Coordinate is another ID-ed auto-realized entity.

A script that utilizes this object should provide a **`Matrix3x3`** object to create the Coordinate. the structure is defined as follow:

```
struct Matrix3x3 {
     Float a11, a12, a13;
     Float a21, a22, a23;
     Float a31, a32, a33;
};
```

```
void DataCreateCoord(Pointer dc, Int idcoord, Matrix3x3 m);
```

Define a local coordinate system that originates at (m.a11, m.a12, m.a13) and other two non-parallel vectors. The sequence will be the same as if you are defining in *DEFINE_COORDINATE_SYSTEM.

- dc : datacenter that contains the coordinate
- idcoord : the ID of the coordinate to be created.
- m : the 3 vectors that defines the coordinate system. vector m.a1x defines the origin, vector m.a2x defines the first vector, while vector m.a3x defines the second vector.

```
void DataDestroyCoord(Pointer dc, Int idcoord);
```

Destroy a local coordinate system.

- dc : datacenter that contains the coordinate
- idcoord : the ID of the coordinate to be destroyed.

```
Int DataHasCoord(Pointer dc, Int idcoord);
```

Check if the local coordinate system exists.

- dc : datacenter that contains the coordinate
- idcoord : the ID of the coordinate to be checked.
- return value : return 1, if exists, 0, otherwise.

```
void DataCoordSetOrigin(Pointer dc, Int idcoord,
                        Int idx, Float val);
```

Set the origin of the coordinate.

- dc : datacenter that contains the coordinate
- idcoord : the ID of the coordinate to be modified.
- idx : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified

```
void DataCoordSetVector1(Pointer dc, Int idcoord,
                         Int idx, Float val);
```

Set the vector1 for the coordinate.

- dc : datacenter that contains the coordinate
- idcoord : the ID of the coordinate to be modified.
- idx : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified

```
void DataCoordSetVector2(Pointer dc, Int idcoord,
                         Int idx, Float val);
```

Set the vector2 of the coordinate.

- dc : datacenter that contains the coordinate
- idcoord : the ID of the coordinate to be modified.
- idx : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified

Ascii - to assign an ASCII output

Ascii is considered an entity in SCRIPTO, it is identified by its option and also encapsulates the keyword *ASCII_{option} of LS-DYNA.

Ascii is the other entity that is auto-realized. There are three different options for a script to define how a ascii file could output from a simulation, that is, traditional ASCII files, BINOUT files, or both.

```
void DataCreateAsciiDatabase(Pointer dc, Int file_opt,
                             Float dt, Int output_method);
```

Create an Ascii entity.

- dc : datacenter that contains the ascii
- file_opt : the ascii file options for the ascii entities. Currently following are supported.

scDAT_BNDOUT	*DATABASE_BNDOUT will be created
scDAT_DEFORC	*DATABASE_DEFORC will be created
scDAT_ELOUT	*DATABASE_ELOUT will be created
scDAT_GCEOUT	*DATABASE_GCEOUT will be created
scDAT_GLSTAT	*DATABASE_GLSTAT will be created
scDAT_JNTFORC	*DATABASE_JNTFORC will be created
scDAT_MATSUM	*DATABASE_MATSUM will be created
scDAT_NCFORC	*DATABASE_NCFORC will be created
scDAT_NODFOR	*DATABASE_NODFOR will be created
scDAT_NODOUT	*DATABASE_NODOUT will be created
scDAT_RBDOUT	*DATABASE_RBDOUT will be created
scDAT_RCFORC	*DATABASE_RCFORC will be created
scDAT_RWFORC	*DATABASE_RWFORC will be created
scDAT_SBTOUT	*DATABASE_SBTOUT will be created
scDAT_SECFORC	*DATABASE_SECFORC will be created
scDAT_SLEOUT	*DATABASE_SLEOUT will be created
scDAT_SPCFORC	*DATABASE_SPCFORC will be created
scDAT_SSSTAT	*DATABASE_SSSTAT will be created
scDAT_SWFORC	*DATABASE_SWFORC will be created

- dt : the time interval between outputs.
- output_method : it can be one of the following:

scDAT_ONLYASCII	only corresponding ASCII files be output, no it binary counter part will be generated
scDAT_ONLYBINOUT	Only BINOUTxxxx will be generated, its original ASCII files are not generated.
scDAT_BINASCII	Both of the ASCII files and the BINOUT files will be generated.

```
Int DataHasAsciiDatabase(Pointer dc, Int file_option);
```

Check if a Ascii entity exists.

- dc : datacenter that contains the ascii entity
- file_option : file option as it is tabulated in the Create function

```
void DataAsciiSetOutTime(Pointer dc, Int file_option,
                         Float dt);
```

Modify the output time for a specific file.

- dc : datacenter that contains the ascii entity
- file_option : file options available as tabulated in the creation function.
- dt : time interval to output for this file option.

```
void DataAsciiSetOutOption(Pointer dc, Int file_option,
                           Int output);
```

Modify the output option for a ascii entity.

- dc : datacenter that contains the ascii entity
- file_option : file options available as tabulated in the creation function.
- output : the output option available as tabulated in the creation function.

```
void DataDestroyAsciiDatabase(Pointer, Int);
```

Destroy an Ascii entity.

- dc : datacenter that contains the ascii entity
- file_option : file options available as tabulated in the creation function.

SPC - nodal single point constraint to assign a boundary condition

SPC encapsulates the nodal single point constraint (SPC) definition in a model, the *BOUNDARY_SPC_NODE and *BOUNDARY_SPC_SET can be generated by this entity. The option of _ID in the LS-DYNA manual is not yet supported.

SPC is auto-realized. A **`TypedID`** object is needed to specify if a spc entity associates with a node or a node set. The ID for this **`TypedID`** object is the node or the node set the SPC associated with. and the type can be one of the following:

scBOUNDARY_NODE	the SPC is associated with a node, and the nodal id is in the ID field.
scBOUNDARY_SET	the SPC is associated with a node set and the node set id is in the ID field.

```
void DataCreateBoundarySPC(Pointer dc, TypedID spc);
```

Create a SPC entity, there is not constraint or local coordinate involved when create a SPC. A script should assign these property later by functions provided.

- dc : datacenter that contains the SPC
- spc : the spc that is to be created.

```
void DataDestroyBoundarySPC(Pointer dc, TypedID spc);
```

Destroy a SPC entity

- dc : datacenter that contains the SPC
- spc : the spc that is to be destroyed.

```
Int DataHasBoundarySPC(Pointer dc, TypedID spc);
```

Check if a SPC entity exists.

- dc : datacenter that contains the SPC
- spc : the spc that is to be checked.
- return value : return 1, if exists, 0, otherwise.

```
void DataBoundarySPCSetDof(Pointer dc, TypedID spc,
                           Int dof);
```

Set the DOF for the SPC

- dc : datacenter that contains the SPC
- spc : the spc that is to be modified.
- dof : DOF that is to assign to the SPC, through script, one can either ADD or REMOVE a specific constraint in/out with a node/node set by pipeline the add/sub options.

scSPC_ADD	to make this modification to the spc be an addition operation. This option is mutually exclusive to the scSCPC_SUB.
scSPC_SUB	to make this modification to the spc be an substraction operation (remove), this option is mutually exclusive to the scSCP_ADD.
scSPC_RX	make changes to the constraint about rotation about x-axis
scSPC_RY	make changes to the constraint about rotation about y-axis
scSPC_RZ	make changes to the constraint about rotation about z-axis.
scSPC_TX	make changes to the constraint about the translation about x-axis
scSPC_TY	make changes to the constraint about the translation about y-axis
scSPC_TZ	make changes to the constraint about the translation about z-axis

EX: one can call the function as follows

```
DataBoundarySPCSetDof(&dc, spc,
                      scSPC_SUB | scSPC_RX |scSPC_TX);
```

the pipelined value says that to remove the constraint of rotation and translation from the spc entity.

```
Int DataBoundarySPCSetCoord(Pointer dc, TypedID spc,
                            Int idcoord);
```

make the constraints applied follow a local coordinate system.

- dc : datacenter that contains the spc
- spc : the spc that is to be modified
- idcoord : coordinate ID for the local coordinate system;

Motion - prescribed motion to set boundary conditions

Motion entities encapsulate the prescribed motion boundary conditions, there are the four keywords of LS-DYNA that can be generated by this entity.

*BOUNDARY_PRESCRIBED_MOTION_NODE
*BOUNDARY_PRESCRIBED_MOTION_SET
*BOUNDARY_PRESCRIBED_MOTION_RIGID
*BOUNDARY_PRESCRIBED_MOTION_RIGID_LOCAL

Motion entities are auto-realized when created. A script should use its member functions to access each field of the entity

The relative displacement option in LS-DYNA keyword manual for this keyword is not supported at the moment.

```
void DataCreateBoundaryMotion(Pointer dc, TypedID motion);
```

Create a motion entity.

- dc : datacenter that contains the motion.
- motion : the type field of this variable can be the following:

scBOUNDARY_NODE	*BOUNDARY_PRESCRIBED_MOTION_NODE will be created
scBOUNDARY_SET	*BOUANDAY_PRESCRIBED_MOTION_SET will be created
scBOUNDARY_RIGID	*BOUDNARY_PRESCRIBED_MOTION_RIGID will be created
scBOUNDARY_RIGID_LOCAL	*BOUNDARY_PRESCRIBED_MOTION_RIGID_LOCAL will be created.

and the id for the motion entity in each type is

scBOUNDARY_NODE	a node ID
scBOUNDARY_SET	a node set ID
scBOUNDARY_RIGID	a part ID
scBOUNDARY_RIGID_LOCAL	a part ID

```
void DataDestroyBoundaryMotion(Pointer dc, TypedID motion);
```

Destroy a motion entity.

- dc : datacenter that contains the motion.
- motion : the motion to be destroyed

```
Int DataHasBoundaryMotion(Pointer dc, TypedID motion);
```

Check if the model has the motion entity.

- dc : datacenter that contains the motion.
- motion : the motion to be checked
- return value : return 1, if exists, 0, otherwise.

```
Int DataBoundaryMotionSetConstraint(Pointer dc,
                                    TypedID motion,
                                    Int constraint_id);
```

Set the constraint ID as described in LS-DYNA users manual.

- dc : datacenter that contains the motion.
- motion : the motion to be modified
- constraint_id : this is the same as DOF field of *BOUNDARY_PRESCRIBED_MOTION_{option} in LS-DYNA users' manual.
- return value : 1, if successful,0, otherwise

```
Int DataBoundaryMotionSetMovingVector(Pointer dc,
                                      TypedID motion,
                                      Int idvec);
```

Set the moving vector of the prescribed motion, if translational.

- dc : datacenter that contains the motion.
- motion : the motion to be modified
- idvec : the vector id of the moving direction.
- return value : 1, if successful, 0, otherwise.

```
Int DataBoundaryMotionSetMotionCurve(Pointer dc,
                                     TypedID motion,
                                     Int idcrv,
                                     Float scale);
```

Set the motion curve of the prescribed motion.

- dc : datacenter that contains the motion.
- motion : the motion to be modified
- idcrv : the DefineCurve ID for the motion curve.
- scale : scale factor for the curve when applied, if leave as 0.0, it will be set to 1.0.
- return value : 1, if successful, 0, otherwise

```
Int DataBoundaryMotionSetDuration(Pointer dc,
                                  TypedID motion,
                                  Float birth,
                                  Float death);
```

Set the birth and death time for the motion curve to apply, if not setting or leave as 0.0, the birth time will be 0.0 and the death time will be set to 1.0e28.

- dc : datacenter that contains the motion.
- motion : the motion to be modified
- birth : the birth time
- death :: the death time.

```
Int DataBoundaryMotionSetOffsets(Pointer dc,
                                 TypedID motion,
                                 Float off1,
                                 Float off2);
```

Set the offset value for the prescribe motion when the constraint id = { -11, -10, -9, 9, 10, 11}. this function is dormant, if the constraint id is any of the 6 options.

- dc : datacenter that contains the motion.
- motion : the motion to be modified
- off1 : the first offset
- off2 : the second offset.
- return value : 1, if successful, 0, otherwise.

```
Int DataBoundaryMotionSetMotionType(Pointer dc,
                                    TypedID motion,
                                   Int motype);
```

Set motion curve's type, this value is the same as it is in LS-DYNA user's manual.

- dc : datacenter that contains the motion.
- motion : the motion to be modified
- motype : value = [0, 3], value = 4 is not yet supported.

InitialVelocity - initial velocity to set initial conditions

InitialVelocity encapsulates the following keywords in LS-DYNA manual:

*INITIAL_VELOCITY
*INITIAL_VELOCITY_NODE
*INITIAL_VELOCITY_RIGID_BODY
*INITIAL_VELOCITY_GENERATION

InitialVelocity entities use the **`TypeID`** object to identify themselves in a model, and they are auto-realized entities.

Following options described in these keywords are not yet implemented into InitialVelocity entities:

For *INITIAL_VELOCITY, excluded node set is not yet implemented.

For *INITIAL_VELOCITY_GENERATION, dynamic relaxation option is not yet implemented.

```
void DataCreateInitialVelocity(Pointer dc, TypedID ini);
```

Create an InitialVelocity entity.

- dc : datacenter that contains this InitialVelocity.
- ini : the type field for the variable can be one of the followings:

scINITIAL_VELOCITY	*INITIAL_VELOCITY will be created, the NSIDEX option is not available.
scINITIAL_VELOCITY_NODE	*INITIAL_VELOCITY_NODE will be created.
scINITIAL_VELOCITY_RIGID	*INITIAL_VELOCITY_RIGID will be created
scINITIAL_VELOCITY_GENERATION	*INITIAL_VELOCITY_GENERATION will be created

And the ID type that will be as followings:

scINITIAL_VELOCITY	Node set ID
scINITIAL_VELOCITY_NODE	Node ID
scINITIAL_VELOCITY_RIGID	Part ID
scINITIAL_VELOCITY_GENERATION	This can be either a part set ID, a part ID or a node set ID.

```
void DataDestroyInitialVelocity(Pointer dc, TypedID ini);
```

Destroy an InitialVelocity entity.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be destroyed.

```
Int DataHasInitialVelocity(Pointer dc, TypedID ini);
```

Check to see if an InitialVelocity entity does exist.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be checked.
- return value : return 1, if exists, 0, otherwise.

```
Int DataInitialVelocitySetTranslation(Pointer dc,
                                      TypedID ini,
                                      Int index,
                                      Float val);
```

Modify the initial translational velocity.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.

- index : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified
- return value : return 1, if successful, 0, otherwise.

```
Int DataInitialVelocitySetRotationDir(Pointer dc,
                                      TypedID ini,
                                      Int index,
                                      Float val);
```

Modify the initial rotational velocity direction, apply only to scINITIAL_VELOCITY_GENERATION type.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.
- index : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified
- return value : return 1, if successful, 0, otherwise.

```
void DataInitialVelocityNormalizeRotDir(Pointer dc,
                                        TypedID ini);
```

Normalize the InitialVelocity's rotation direction, apply only to scINITIAL_VELOCITY_GENERATION type.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.

```
Int DataInitialVelocitySetRotationOrig(Pointer dc,
                                       TypedID ini,
                                       Int index,
                                       Float val);
```

Set initialVelocity's rotational vector origion, apply only to scINITIAL_VELOCITY_GENERATION type.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.
- index : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified
- return value : return 1, if successful, 0, otherwise.

```
Int DataInitialVelocitySetAngularVelocity(Pointer dc,
                                          TypedID ini,
                                          Int index,
                                          Float val);
```

Set InitialVelocity's rotation angular velocity, this function does not apply to the scINITIAL_VELOCITY_GENERATION.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.

- index : index(0, 1, 2) to which axis to be modified.
- val : the component of which axis to be modified
- return value : return 1, if successful, 0, otherwise.

```
Int DataInitialVelocitySetOmega(Pointer dc,
                                TypedID ini,
                                Float w);
```

Set InitialVelocity's rotational velocity, when direction provided, This option applies only to the scINITIAL_VELOCITY_GENERATION type.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.
- w : rotational velocity about the rotational vector.

```
Int DataInitialVelocitySetGenerationType(Pointer dc,
                                TypedID ini, Int gen_type);
```

This option set up what kind of entities this InitialVelocity associated with, this only applies to scINITIAL_VELOCITY_GENERATION type.

- dc : datacenter that contains this InitialVelocity.
- ini : the InitialVelocity entity to be modified.
- gen_type : following are currently supported:

scINITIAL_VELOCITY_PART	the associated objecti a part
scINITIAL_VELOCITY_PARTSET	the associated object are a part
scINITLAL_VELOCITY_NODESET	the associated object are node set

SCRIPTO global functions

```
void CreateWidget(Pointer p);
```

Create a widget instance base on the pointer to the widget description.

It is a factorial constructor. it takes pointers of all different kinds and it will create the instance to the memory space. script writers do not need to worry about what exactly a widget is, the pointer of a widget's specification represents the instance of the widget itself, after it has been created.

- p : pointer to a widget specification.

```
void RefreshWidget(Pointer p);
```

Redraw the whole area of the widget. this will cause a redraw immediately to the widget and all its children.

- p : pointer to a widget to be refreshed.

`void HideWidget(Pointer p);`

Hide a widget. Once a widget is hidden all widgets that in the current widget's children tree will be invisible as well.

- p : pointer to a widget specification.

`void ShowWidget(Pointer p);`

Show a widget. Once a widget is shown, all widget that was not hidden originally will be seen.

- p : pointer to a widget specification

`void MoveWidget(Pointer p, Int x, Int y);`

Move a Widget to the location specified by (x, y);

- p : pointer to a widget specification
- x : x-coordinate of the new location
- y : y-coordinate of the new location

`void SizeWidget(Pointer p, Int w, Int h);`

Size a Widget to the desired size assigned by (w, h);

- p : pointer to a Widget specification
- w :new width of the Widget;
- h :new height of the Widget;

`void SensitiveWidget(Pointer p);`

Sensitize (enable) a widget that it can receive user input.

- p : pointer to a Widget to be sensitized.

`void InsensitiveWidget(Pointer p);`

In-sensitize (disable) a widget that it will not be able to receive the users' input.

- p : pointer to a Widget to be in-sensitized.

`void BubbleTip(Pointer p, Int onoff);`

Turn on or off the bubble tip for a widget.

- p : pointer to a Widget

- onoff : any non zero integer will turn the bubble tip on for the widget

```
void SetWidgetForeground(Pointer widget, Int r, Int g,
                         Int b);
```

Assign the foreground color of a widget, normally this means the color of the text for the widget.

- widget : pointer to the widget to set foreground colour
- r : red intensity of the color, range [0, 255]
- g : green intensity of the color, range [0, 255]
- b : blue intensity of the color, range [0, 255]

```
void SetWidgetBackground(Pointer widget, Int r, Int g,
                         Int b);
```

Assign the background color of a widget, normally this means the color of the underlying background of the widget.

- widget : pointer to the widget to set background colour.
- r : red intensity of the color, range [0, 255]
- g : green intensity of the color, range [0, 255]
- b : blue intensity of the color, range [0, 255]

Object Structures

SCRIPTO are a set of data structures. By declaring and defining SCRIPTO objects, users can customize LS-PREPOST to the way they want.

Currently SCRIPTO understands the following data structures:

- Form
- PushButton
- ToggleButton
- RadioBox
- TextField
- ScrolledText
- Slider
- Separator
- Label
- Menu
- Spin
- ListBox

- Tree
- Tab
- Grid
- MultiColumnList
- Dialog

Besides these widgets there are few functions related to manipulation of the widgets, such as construction, showing and hiding a widget, or moving and sizing a widget, here are these functions:

Following will be a short introduction to each of the data structures:

SCRIPTO classes

Form

Forms are one of the containers in the SCRIPTO families. it provides the other objects a place to build upon. By supplying fractions, Form gives its children a grid-like map to locate themselves.

Data Members

```
struct Form {
     void *parent;
     char *tag;
     Rectangle anchor;
     char *style;
     int fraction;
     int ismanaged;
};
```

- parent : parent of the form.
- tag : description name of this form.
- anchor : location of the form relative to its parent. the Rectangle is composed by 4 integer members, x1, y1, x2, y2.
- style : currently one of the following styles can be used when creating forms.

FS_FRAME	form with decorative frame around it.
FS_CAPIONFRAME	an extra description of the form will be at the upper-left corner of the form.
FS_FITPARENT	the form will be the same size as its parent. Anchor of this form will be ignored.

- fraction : how fine the form will be divided in each direction(both x and y).
- ismanaged : the form is shown after its creation, if this field is clear, all other interfaces that have this form as ancestor will not be shown.

Manipulation Functions

```
void FormSetCaption(Pointer form, Pointer caption);
```

Change the caption of a form that has style of FORM_CAPTIONFRAME.

- form : form widget
- caption : the caption to be set unto the form.

PushButton

PushButton is a simple widget that user can click on the face of the button, and the assigned actions will be executed upon when clicked.

Data Members

```
struct PushButton {
     void *parent;
     char *tag;
     Rectangle anchor;
     char *help;
     char *style;
     char *pix[];
     Int  id_bitmap;
     char *onactivate[];
};
```

- parent : parent of the pushbutton
- tag : tag for the pushbutton
- anchor : location for the pushbutton
- help : comment string for the pushbutton
- style : styles supported by PushButton right now are

PB_XPM	this style turns the pix member on, and script parser will expect to read an icon with the XPM format in.
PB_SHARED	This style can be used only if PB_XPM is used. the XPM picture will not be generated by the PushButton, script should provided a bitmap id to the id_bitmap member. This id is returned by Bitmap creation functions.

- pix : the icon that will show on this PushButton. If there is a tag for the PushButton, its tag will be truncated to only one character long. XPM stands for X Window's Pixmap. One may google how XPM is defined and find the supported icon editors from internet. You should put the output data array of your icon to this member.

- onactivate : this is the action field for user to determine when a pushbutton get clicked (activated), what the application should do. here is the format for the onactivate[] member

```
onactivate[] = { "command_one",
                 "command_two",
                 "command_three",
                 ...
               };
or

onactivate[] = { "@function_name,
[optional user data]" };
```

While user chooses to put a function name as the action assigned for onactivate[], we call the function callout (or callback) function.

Here are rules for a callout function,

1. Only one callout function per interaction. for example, in a pushbutton, it will be one per onactivate[].
2. Several user interfaces may share same callout function with different optional user data assigned.
3. The signature (prototype) of a callout function should look like the following, when defined in the script.

```
void callout_function(DataField pUser,
                      CallStruct *pCall);
```

   - pUser: the type of the optional user data is DataField, this is an union that can be interpreted by the data given to the onactivate[] member. User has the responsibility to use the member of the DataField when it has been passed onto the callout function. However, user should tell the LS-PREPOST how to pack the pUser when read in from the script. Here, some characters has been reserved for SCRIPTO to identify the pUser:
     i. $: the user data is a string, use pUser.a to address to the pointer of the string;

ii. %: the user data is an integer, use pUser.i to retrieve the integer;

iii. #: the user data is a real number, use pUser.f to retrieve the float;

iv. @: the user data is a pointer to a variable, typecast pUser.a to the same type of pointer you passed in the onactivate[] member.

- pCall: a pointer to a CallStruct. If there is no extra information needed, pCall is null. However, if pCall is not null, user should force the pCall to be equal to the other pointer that is a type of the current Widget's CallStruct, and the values of each members in that CallStruct will be mapped accordingly.

For the pushbuttons, there will be no extra CallStruct passed out to the user-defined callout functions.

Manipulation Functions

```
void PushButtonDown(Pointer pushbutton);
```

Use this function when you want to press the pushbutton and held it there.

- pushbutton : a pointer to the pushbutton.

```
void PushButtonUp(Pointer pushbutton);
```

Use this function when you want the held pushbutton to be up.

- pushbutton : a pointer to the pushbutton.

```
Int PushButtonIsDown(Pointer pushbutton);
```

Use this function when you wan to know if a pushbutton is held down.

- pushbutton : a pointer to the pushbutton.

```
void PushButtonChangeBitmap(Pointer pushbutton,
                            Int idbimap);
```

After creating a PushButton with bitmap, you may change the button with a new bitmap ID.

- pushbutton : a pointer to the pushbutton
- idbitmap : an id retrieved from UtilCreateBitmap() function

```
void PushButtonSetTag(Pointer pushbutton, char *newtag);
```

Use this function to change a Pushbuttons' tag.

- pushbutton : a pointer to the pushbutton
- newtag : new tag to the button.

ToggleButton

ToggleButtons are designed to have two states, i.e., checked, or unchecked. and every time when user clicks on them, they will alternate their state accordingly. a command can be installed if a togglebutton was changing states.

```
struct ToggleButton {
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *help;
     char      *oncheck;
};
```

- parent : the parent of this toggle button
- tag : tag that show to the user
- anchor : location where this toggle button suppose to be
- help : comment for this toggle button
- on check : similar as the PushButton::onactivate

CallStruct in the callout function is

```
struct ToggleCallStruct {
     int iset;
};
```

- iset : if the toggle button were checked, iset == 1, otherwise, it is zero.

Manipulation Functions

```
void CheckToggleButton(Pointer togglebutton);
```

Use this function to check the toggle button through program, no callback function will be emitted, if through this function.

- togglebutton: pointer to the togglebutton

```
void UncheckToggleButton(Pointer togglebutton);
```

Use this function to un-check your toggle button off, no callback function will be emitted if through this function

- togglebutton: a pointer to the ToggleButton

```
Int ToggleButtonGetState(Pointer togglebutton);
```

This function will return back the current state of the ToggleButton that you query at.

- Togglebutton : a pointer to the ToggleButton

RadioBox

RadioBox is another container that contains radiobuttons. Normally users use radioboxes to encapsulate the concepts of the one-of-many choices. For example, if someone can only go from one place to the other by a train, by a car or by a airplane; but cannot be on two or more at the same time, then a RadioBox would be a good candidate to implement this concept.

To describe a Radiobox, you need to prepare also a RadioItem array, which describe each of the item that is in the box.

```
struct RadioItem {
     char *tag;
     char *onselect[];
     int  idata;
     char *comment;
};
```

- tag : the label for this radioitem
- idata : item data, used for the callback function, user may use this integer for customization.
- comment : the comment for each radioitem
- onselect : action for the radioitem, when it is get selected, similar as PushButton::onactivate. every change of selection actually generate two times of callout, one is to notify the user an item was unselected. and followed by notifying the other item is selected. typecast to the pointer to CallStruct can be RadioCallStruct shown as followed:

```
struct RadioCallStruct {
     int index;
     int iset;
};
```

- index :index for which item has been selected or unselected.
- iset :if the notification is for the selection of an item this field will be true, otherwise it will be false.

```
struct RadioBox {
     void      *parent;
```

```
    char      *tag;
    Rectangle anchor;
    char      *style;
    int       nitems;
    int       ncolumns;
    int       itemset;
    RadioItem item[];
};
```

- parent : parent of the RadioBox
- tag : tag of the RadioBox, optional.
- anchor : where the RadioBox suppose to be relative to its parent
- style : the following styles are available for a RadioBox

DIR_HORZ	Tile the RadioItems in the RadioBox horizontally first.
DIR_VERT	Tile the RadioItems in the RadioBox vertically first.

- nitems : number of items in the RadioItem array.
- itemset : the item that is selected originally, the item index started from 0.
- item : the RadioItem array for the RadioBox

Manipulation Functions

```
void SetRadioBoxSelection(Pointer radiobox, Int iset);
```

Set the item *iset* to selection, this will deselect the other item that is currently selected, but will not trigger the callback function that associates to it.

- radiobox : a pointer to the RadioBox.
- iset : the item is about to set to selection. Index started from 0.

```
Int GetRadioBoxSelection(Pointer radiobox);
```

Get the current selection for the radiobox. The integer returned is the index of the radioitem now selected.

- radiobox : a pointer to the RadioBox.

- return value : the index to the radio item that is now selected in the RadioBox.

```
void RadioItemSetSensitive(Pointer box, Int item_idx,
                           Int onff);
```

Set a specific item to sensitive or insensitive.

- box : a pointer to the RadioBox
- item_idx : index of the radio item to be sensitized. The index ranges from [0, number_of_items -1]
- onoff : if non-zero, it sensitizes the item.

```
Int IsRadioItemSensitive(Pointer box, Int item_idx);
```

Check if a radio item for a box is sensitive.

- box : a pointer to the RadioBox
- item_idx : index of the radio item to be sensitized. The index ranges from [0, number_of_items -1]
- return value : return 1 if sensitive, 0, if not sensitive.

TextField

TextField is a widget for users to enter a one-line text. It is possible to make TextField to be read-only by setting its related attributes. You can also limit the types of text users enter to a specific TextField, such as integer, floating numbers or character strings only.

```
struct TextField{
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *help;
     char      *style;
     int       noedit;
     int       columns;
     char      *initxt;
     char      *whenentered[];
};
```

- parent : parent of this TextField
- tag : name tag for the TextField
- anchor : location of the text relative to its parent
- help : comments for this TextField
- style : styles available for the TextField are

ES_LEFT	make the text entered align to left
ES_CENTER	make the text entered align at center
ES_RIGHT	make the text entered align to right
ES_FLOATONLY	when this style is assigned, only floating numbers are accepted
ES_INTONLY	when this style is assigned, only the integers are accepted
ES_ALPHAONLY	when this style is assigned, the entered text will be treated as character string

- noedit : when this field is not zero, the TextField will not be editable.

- columns : how many characters are allowed in the TextField. if this is not assigned a default value for the columns allowed is 10 characters more than the length of the initxt
- initxt : Initial text of the TextField, if not assigned, the original TextField will be left blank.
- whenentered : an optional callback action allowed for the TextField, if existed, it will be the same as PushButton::onactivate, there is no CallStruct information passed out to users.

Manipulation Functions

```
void TextFieldSetString(Pointer textfield, Pointer p);
```

Set the TextField to the string give by p.

- textfield : a pointer to the TextField
- p : a character array, (i.e. a string)

```
Pointer TextFieldGetString(Pointer textfield);
```

Get the string from a TextField, a character array will be allocated for caller, it is caller's responsibility to clean up a string.

- textfield : a pointer to the TextField.
- returned value : an allocated string contains the value of the textfield.

```
void TextFieldSetCursorPosition(Pointer p, Int icp);
```

Set cursor position to location icp;

- p : a pointer to the TextField
- icp : location where the cursor should go.

```
Int TextFieldGetCursorPosition(Pointer p);
```

Retrieve the cursor position of a given TextField.

- p : a pointer to the TextField
- returned value : the location of the cursor for the TextField. if the TextField pointer is not correct, -1 will be returned.

```
void TextFieldAddString(Pointer p, Pointer str);
```

Expand the string in the TextField.

- p : a pointer to the TextField
- str : the string to expand to the end of the current string in the TextField.

```
void TextFieldInsertString(Pointer p, Int loc,
                           Pointer string);
```

Insert a string at the location of the cursor in the TextField.

- p : a pointer to the TextField.
- loc : cursor location where the string to insert
- string : the string to insert to the TextField.

```
void TextFieldShowAll(Pointer p);
```

Force the TextField to Show all of it contents, and consequently, move the cursor for the user to input to the end of the TextField.

- p : a pointer to the TextField.

ScrolledText

ScrolledText is a widget that allows users to type in multi-line text. it can also be used to show the user what the information is. This widget does not have callback function installed.

```
struct ScrolledText {
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *help;
     int       candedit;
     int       wordwrap;
     int       hscroll;
};
```

- parent : parent window of this widget
- tag : name of the window
- anchor : location of the scrolled window
- help : description for the window
- canedit : if not zero, the ScrolledText will be editable
- wordwrap : if not zero, the ScrolledText will do a word wrap.
- hscroll : if not zero, the ScrolledText will have a horizontal scroll bar, no matter if the text should be word wrapped or not.

Manipulation functions

```
Pointer ScrolledTextGetContents(Pointer p);
```

Get the contents in the ScrolledText.

- p : a pointer to the ScrolledText
- returned value : string returned, with linefeed character at the end to each line.

```
void ScrolledTextSetContnts(Pointer p, Pointer c);
```

Set the contents in the ScrolledText as c.

- p : a pointer to the ScrolledText
- c :the contents to be set in the ScrolledText

Slider

Use a slider when the user interface is to represent a value that is in a certain range. Sliders can be either vertical or horizontal.

```
struct Slider{
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *help;
     char      *style;
     int       min;
     int       max;
     int       ini;
     char      *whendragged[];
     char      *whenchanged[];
};
```

- parent : parent widget of this slider
- tag : name tag of the slider
- anchor : location of the slider relative to its parent.
- help : comments for the slider.
- style : Sliders have the following styles available.

SLD_HORZ	Make the slider horizontal, this is default setting for the sliders.
SLD_VERT	Make the slider vertical, it is mutually exclusive with the SLD_HORZ
SLD_SHOWVALUE	Show the value of the thumbnail position in the slider.

- min : minimum value for the slider
- max : maximum value for the slider
- ini : initial value for the thumbnail position.
- whendragged : similar to PushButton::onactivate, and a callout function can get called whenever a user is dragging the thumb and changing the value of the slider

- whenchanged : similar to PushButton::onactivate, and a callout function can get called whenever the value of a slider get changed.

Both in both situations, a pointer to SliderCallStruct will be passed out in the pCall parameters.

```
struct SliderCallStruct {
     Int value;
};
```

- value : the current thumb position of the slider.

Manipulation functions

```
Int SliderGetValue(Pointer p);
```

Get the thumb position for slider's current value

- p : a pointer to the slider.
- returned value: the current position of the thumb.

```
void SliderSetValue(Pointer p, int v);
```

move the thumb to the position.

- p : a pointer to the slider
- v : thumb position

Separator

Separator is a decorative widget, to serve for visual purpose only. it can have two different styles, vertical or horizontal.

```
struct Separator {
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *style;
};
```

- parent :parent of the separator
- tag :name tag for the separator
- anchor : location of the separator relative to its parent
- style : here are the available styles for the separator

SEP_HORZ	Create a horizontal separator, this is the default style, when style is not set, it will be horizontal.
SEP_VERT	create a vertical separator.

Manipulation Functions

none.

Label

Label is also a decorative widget; it displays a string on the location where anchor assigned.

```
struct Label {
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *style;
};
```

- parent : parent widget for the label
- tag : this is not only a name, it will be the label been seen on the user interface.
- anchor : location of the label relative to its parent.
- style : following are the available styles for Label widgets

LB_LEFT	adjust the output text to the left hand side of the rectangle. This will be the default style, if there is no style set to the Label
LB_CENTER	adjust the output text to be central aligned.
LB_RIGHT	adjust the output text to align right.

Manipulation Functions

```
Pointer LabelGetText(Pointer p);
```

Get the face name of the Label; the pointer returned is a null terminated string that has been allocated memory before it returned. Callers should free the memory if necessary.

- p : a pointer to the Label

- returned value : the face text of the Label.

```
void LabelSetText(Pointer p, Pointer s);
```

Set s to the face name of the Label.

- p : a pointer to the label
- s : this must be a null terminated string

Menu

Menu is actually pull down menus, or combo boxes. User can select one of the items from the drop down list.

```
struct MenuItem {
     char *label;
     char *data;
};

struct Menu {
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *help;
     int       nitem;
     MenuItem  *item;
     int       inipos;
     char      *whenpulldown[];
     Int       add(MenuItem *);
     Int       insert(MenuItem *, Int);
     Int       remove(char *);
     Int       remove_at(Int);
     void      clear(void);
};
```

- label : the label of each item in the MenuItem
- data : extra data of each item in the MenuItem
- parent : parent of the Menu
- tag : name tag for the Menu
- anchor : location of the menu relative to its parent
- help : comments for the menu
- nitem : number of items for the MenuItem
- item : the array to the menu item.
- inipos : item that should be selected when created
- whenpulldown : similar to PushButton::onactivate, a callback function can be given when user drop down the menu and selected the other item other than the current one.

- add : user may add a MenuItem dynamically to the end of the menu list.
- insert : user may add a Menuitem to an appointed position in the list
- remove : find the text to be removed in the list, and remove the item.
- remove_at : remove the item at appointed position.
- clear : clean up the list.

A new CallStruct is defined as follow:

```
struct MenuCallStruct {
     int selection;
};
```

Callout function will pass MenuCallStruct to the argument of pointer to CallStruct. Users can use the selection field to find out what item has been selected.

The data passed to the pUser argument in the callout function will be the data interpreted from the data field of the item array in the Menu structure.

Manipulation Functions

```
MenuItem MenuItem(Pointer label, Pointer data);
```

The constructor for the MenuItem objects.

- label : the label attribute for the MenuItem
- data : the data attribute for the MenuItem
- returned value : the MenuItem Object

```
Int MenuGetSelection(Pointer p);
```

Return the index of the selected item for the current menu.

- p : a pointer to the Menu widget
- returned value : index of the item currently selected

```
void MenuSetSelection(Pointer p, Int i);
```

Set the selection of the Menu to the item i. This function will not cause a callout function being called.

- p : a pointer to the Menu widget
- i : the index for the item to be selected.

Spin

Spin is a combo widget. it consist at most 4 different parts.

- main TextField
- a PushButton for spinning up.
- a PushButton for spinning down.
- increment TextField

Users use Spin widget to define a specific input for a quantity, like number of grid points along a meshing boundary. Base on the functionality of each part in Spin, there are three different structures to feed in before the widget has been created.

```
struct SpinIncInfo {
     int            size;
     char           *message;
     char           *initext;
};
```

SpinIncInfo structure is used to describe the information in the increment TextField, where user define how much a notch is whenever the user kicks it.

- size : define the size in percentage of the size of the whole Spin in the main direction (horizontal or vertical).
- message : comments for the increment TextField
- initext : initial increment to be put in. if it has not been assigned, when a user spins, the main value will not change.

```
struct SpinBtnInfo {
     int            invert;
     char           *upmessage;
     char           *downmessage;
     char           *whenspinned[];
};
```

SpinBtnInfo structure is used to describe the information about the spin buttons, the up/down (or left/right) buttons. Normally, a spin-up (or spin-right) button will be assigned as increment and the spin-down (or spin-left) button will be decrement, but this is also customizable.

- invert : by default, two spin buttons will act as normally it would be expected (up for increment and down for decrement), if this value has been set as 1, the role of the button will be switched. that means that up will decrease the value in the main TextField, and the down will increase the value in the main TextField.
- upmessage : the comments for the spin-up (or spin-right) button.
- downmessage : the comments for the spin-down (or spin-left button.

- whenspinned : the action for the spin buttons while they were clicked. the spin buttons will first kick the TextField value up or down a notch first, and then if whenspinned is not zero, then it will execute the commands or function assigned here. whenspinned is similar to PushButton::onactivate, on the pointer to the CallStruct will be the following:

```
struct SpinCallStruct {
     int reason;
     DataField value;
};
```

- reason :reason of the call, there are few reasons for a callout to be issued, they are represented by non-zero integers

reason ID, value of the reason field	value of the DataField
1: spin-up (or spin-down) button pushed.	It should be interpreted from the datatype of the spin structure. For example if datatype == "SPN_INTEGER" then the value should be interpreted as integers.
2: Main TextField 's value changed by entering.	

- value :base on what was given on datatype this should be interpreted accordingly in the union of DataField in order to get the value of the field.

```
struct Spin {
     void           *parent;
     char           *tag;
     Rectangle      anchor;
     char           *style;
     int            no_increment;
     SpinIncInfo    *increment;
     SPinBtnInfo    *btninfo;
     char           *message;
     char           *datatype;
     DataField      upper;
     DataField      lower;
     char           *initxt;
     char           *whenentered[];
};
```

- parent :parent of the Spin
- tag :tag for the Spin widget

- anchor :location for the Spin widget with respect to the parent.
- style : following are the styles for a Spin Widget.

SPN_HORZ	the spin button will be left-right button instead of up-down button.
SPN_VERT	the default value of the spin control.
SPN_SEPARATE	normally the spin buttons will be attached together and sandwiched by the main TextField and the increment TextField; however, if this style is assigned, it will instead sandwich the main TextField.
SPN_ALIGNLEFT	this will force the increment TextField widget go to the left if the style is horizontal and at the top if the style is vertical.

- no_increment : when this set to 1, the increment will be ignored.
- increment : this describes the information about the increment
- btninfo : this describes the information about the spin buttons.
- message : comment for the main TextField.
- datatype : the datatype for the main TextField, user has to set this field to one of the following values

SPN_INTEGER	the value for this spin widget will be interpreted as integer
SPN_FLOAT	the value of this spin widget will be interpreted as floating number
SPN_STR	the value of this spin widget will be interpreted as alphabets

- upper : upper limit of the data in the main TextField
- lower : lower limit of the data in the main TextField
- initext : Initial value in the main TextField
- whenentered : similar to PushButton::onactivate, the reason field for the pointer to CallStruct will be 2.

Manipulation functions

```
Pointer SpinGetValue(Pointer p);
```

Get the value string of the Spin.

- p : pointer to the Spin Widget
- return value : value string for spin P.

```
void SpinSetValue(Pointer p, Pointer v);
```

Set a value string to a Spin

- p : pointer to the Spin Widget

- return value : value string for spin p.

ListBox

A ListBox can be used when you have a lot of items that you want to tabulate them. there are different styles of ListBox(es).

```
struct ListBox {
     void      *parent;
     char      *tag;
     Rectangle anchor;
     char      *help;
     char      *style;
     char      *item[];
     int       inipos;
     char      *whenpicked[];
     char      *whendoubleclicked[];
};
```

- parent : parent of ListBox
- tag : tag fro this ListBox
- anchor : location respect to it parent.
- help : comment for this ListBox
- style : following are styles that user can assign for a list

LIST_BROWSE	when a ListBox is LIST_BROWSE, that means it will allow one and one item alone being selected at any given time.
LIST_SINGLE	when a ListBox is LIST_SINGLE, that means it will have at most one item being selected at any given time.
LIST_MULTIPLE	when a ListBox is LIST_MULTIPLE, that means it can have more then one items selected at any given time, and any consecutive selection to the same item will make the item back to its original state. There is no double click event in this style.
LIST_EXTENDED	when a ListBox is LIST_EXTENDED, it allows more than one item being selected at any given time, you can select ranges of items even the positions of the items are not continuous.

- item : an array of quoted strings, they are the items of a list
- inipos : initial position of the selected item in a list

- whenpicked[] : this field is very similar to PushButton::onactivate(). action will be lanched when user change selections.
- whendoubleclicked[]: this field is very similar to PushButton::onactivate(). action will be lanched when user double-clicked an item on the list. However, this action will be sent only after one whenpicked[] event has sent.

the pointer to the CallStruct structure should be type-cast into ListCallStruct as:

```
struct ListCallStruct {
int       reason;
char      *policy;
int       nsel;
int       selpos[];
char      *selitem[];
};
```

- reason :here are the reasons that a ListCallStruct could have,

RSN_LIST_SELECT	callout was caused by list selection
RSN_LIST_DOUBLECLICK	callout was caused by double clicked

- policy : depends on the style of the ListBox, this will be the same as the style you create the ListBox
- nsel : number of selections
- selpos : currently selected position array, position numbering start from 1.
- selitem : selected items, an array of the selected items in the ListBox.

Manipulation Functions

```
Int ListGetItemCount(Pointer p);
```

Get the total Item counts of the ListBox.

- p : pointer to the ListBox which item count is to be retrieved.
- return value : number of items in the ListBox.

```
ListCallStruct ListGetAllItems(Pointer list);
```

Get back all items in a ListBox. A ListCallStruct will carry all the items contain in the ListBox. and member **nsel** will be the number of items in the ListBox; while member array **selitem**[] will be the items in the ListBox.

- list : pointer to the ListBox
- return value : the ListCallStruc object will carry all items in the ListBox.

```
ListCallStruct ListGetSelectedItems(Pointer list);
```

Get back all selected Items and the ListCallStruct will carry the result.

- list : pointer to the list
- return value : item and position of the selected items will be carried by the ListBox.

```
Int ListIsItemSelected(Pointer list, Pointer item);
```

Check if the item is selected in the ListBox.

- list : pointer to the ListBox
- item : item to be tested
- returned value : if returned integer is non-zero, then the item is selected.

```
Int ListIsPositionSelected(Pointer list, Int Position);
```

Check if the item at the position is selected in the ListBox

- list : pointer to the ListBox
- position : position of the item, it is 1-based.
- return value : if returned integer is non-zero, then the position is selected.

```
void ListAddItem(Pointer list, Pointer item);
```

Add an item to the end of the ListBox.

- list : pointer to the ListBox
- item : item to be added into the list

```
void ListInsertItem(Pointer list, Pointer item, Int at);
```

Insert an item to the specific position, if there is an item existed at the location, it will put the original item down one notch in the list.

- list : pointer to the ListBox
- item : item to be inserted
- at : the location the item is about to be inserted.

```
void ListDeleteItem(Pointer list, Pointer item);
```

Delete an item from the ListBox, if the item is not found an exception will be thrown.

- list : pointer to the ListBox
- item : item to be deleted

```
void ListDeletePosition(Pointer list, Int position);
```

Delete an item located at the position.

- list : pointer to the ListBox
- position : the item at the location will be deleted.

```
void ListPurge(Pointer list);
```

Purge the contents of the whole ListBox.

- list : pointer to the ListBox.

```
void ListSelectItem(Pointer list, Pointer item);
```

Selected an item that matches the item.

- list : pointer to the ListBox
- item : item to be selected

```
void ListSelectPosition(Pointer list, Int position);
```

Select a position in the list.

- list : pointer to the ListBox
- position : position to the be selected.

```
void ListDeselectItem(Pointer list, Pointer item);
```

Selected off an item that matches the item.

- list : pointer to the ListBox
- item : item to be selected off, or deselected.

```
void ListDeselectPosition(Pointer list, Int position);
```

Select off a position in the list.

- list : pointer to the ListBox
- position : position to the be selected off or deselected.

```
void ListReplaceItem(Pointer list, Pointer item, Int at);
```

Replace the item in position (at) to become the item passed in.

- list : pointer to the ListBox
- item : item to replace the current item in the at-position.
- at : the position where the item is about to take place.

```
void ListSetPositionVisible(Pointer list, Int position);
```

This will force the position of the ListBox being visible and the user does not need to use the Scrollbar to scroll up and down to find the item.

- list : pointer to the ListBox
- position : position in the ListBox where user wants to show.

Tree

A tree is a list-like widget. Items added into the tree will have a parent and possibly to have children. It acts like a file manager's folder structure.

This file folder structure makes the Tree widget creation and TreeItem manipulation a bit different from other widgets. There is a special structure TreeItem that will be requested and returned from different manipulation functions. Script writers should treat all TreeItem objects as opaque handles when passed in/out tree functions.

```
struct Tree {
     void *parent;
     char *tag;
     Rectangle anchor;
     char *help;
     char *style;
     char *whenactivate[];
     char *whendoubleclicked[];
};
```

- parent : parent of the tree
- tag : a symbolic name for the tree
- anchor : the size of the tree list container
- help : comment for the tree widget
- style : tree widget support the following styles

TREE_SINGLE	the tree widget allows only one selection at a time
TREE_MULTIPLE	the tree widget allows more than one selections at the same time.

- whenactivate : when user clicked the item. this function will be called, behaves as normal event functions.
- whendoubleclicked : when user double-clicks an item , this function will be called, behave as normal event functions.

Manipulation Functions

```
TreeItem TreeAddRoot(Pointer tree, Pointer itmstr);
```

Add a root item to the tree. All items need to have its parent, but root item is an exception. it takes no parent, and the returned TreeItem will become the parent of other items.

Noted that it is possible to have more than one root in a tree widget, but the additional roots do not have any advantage or disadvantage to a tree that has only one root item. And it is possible to not give the root Item a name, since this item will not be shown on the tree widgets.

tree : pointer to the tree widget
itmstr : string for the root item.

returned value : TreeItem handle for the root item

```
TreeItem TreeAddItem(Pointer tree, TreeItem parent,
                     Pointer itmstr);
```

Add an item in the tree. This function will insert this item at the end of the branch, that is, if the parent node has no child, the item will be the first child, and if the parent already has two children, then this item will be the third one.

- tree : tree widget to be added an item.
- parent : the parent TreeItem of the item to be added.
- itmstr : the string of the item
- returned value : the TreeItem handle for the currently added item.

```
TreeItem TreeAddItemSibling(Pointer tree, TreeItem bigbro,
                            Pointer itmstr);
```

Add an item to a tree, and make the current item has the same parent as the bigbro item. Current item will be added as the next brother of the bigbro item.

- tree : tree widget to be added an item.
- bigbro : the direct older brother of the current item.
- itmstr : the string of the current item
- returned value : the TreeItem handle for the current item.

```
TreeItem TreeInsertItem(Pointer tree, TreeItem parent,
                        TreeItem bigbro, Pointer itmstr);
```

Insert an item to a tree, and make the parent, be the parent of the current item, and the bigbro become the direct previous sibling of the current item. This is the only method that one can add an item to become the first child when there are children already for the parent, set the bigbro to NULL, and then current item will be the first child.

- tree : tree widget to be added an item.
- parent : parent of the inserted item.
- bigbro : direct older sibling of the tree item.
- itmstr : the string for the current item.
- return value : the TreeItem handle for the current item.

```
String TreeItemGetString(Pointer tree, TreeItem item);
```

Return the string of the TreeItem.

- tree : tree widget
- item : content of the item to be retrieved.
- return value : returned string in the String object form.

```
void TreeItemSetString(Pointer tree, TreeItem item,
                       Pointer str);
```

Set a new string to the current item in the tree.

- tree : tree widget
- item : item to be set a new string
- str : new string for the new TreeItem.

`void TreeItemSetHasChildren(Pointer tree, TreeItem item);`

To set the current to have children in the future, this is a function that you may assign a button next to the item explicitly.

- tree : tree widget
- item : the item to be set has children in the future.

`Int TreeItemHasChildren(Pointer tree, TreeItem item);`

Test if the treeitem in the tree widget has children or it is a child node.

- tree : tree widget
- item : the item to be tested
- return value : return 1, if the item has children; 0, otherwise.

`void TreeItemSetSelection(Pointer tree, TreeItem item);`

Set current item's selection state to be selected.

- tree : tree widget
- item : the item to set selection.

`void TreeItemDeselection(Pointer tree, TreeItem item);`

Set current item's selection state to be not selected.

- tree : tree widget
- item : item to be deselected.

`TreeItem TreeItemGetParent(Pointer tree, TreeItem item);`

Return the parent of the TreeItem.

- tree : tree widget.
- item : parent of this item will be returned
- return value : the parent of the current TreeItem

`Int IsTreeItemValid(TreeItem item);`

Return if the tree item is valid.

- item : the treeitem to be tested
- return value : return 1, if valid, 0, otherwise.

`void TreeItemDeleteChildren(Pointer tree, TreeItem item);`

Delete the item's children but not the item itself.

- tree : tree widget.
- item : children of this item will be deleted.

`Int TreeItemGetChildrenCount(Pointer tree, TreeItem item,`

```
                                        Int deep);
```

Count the items' children. and if deep is nontrivial, it counts all inheritance as well.

- tree : tree widget
- item : item that the children number to be counted.
- deep : if not zero, all descendants of the tree will be counted; otherwise, only immediate children will be counted.
- return value : number of descendants reported.

```
void TreeDeleteItem(Pointer tree, TreeItem item);
```

Delete the current TreeItem

- tree : tree widget
- item : the item to be deleted

```
void TreePurge(Pointer tree);
```

Delete all items in a tree.

- tree : tree widget to be pruned.

```
void TreeExpandNode(Pointer tree, TreeItem item);
```

Expand the current item.

- tree : tree widget
- item : item to be expanded.

```
void TreeCollapseNode(Pointer tree, TreeItem item);
```

Collapse the current item.

- tree : tree widget
- item : item to be collapse

```
void TreeExpandAll(Pointer tree);
```

Expand all items in the tree.

- tree : tree widget to be expanded

```
void TreeCollapseAll(Pointer tree);
```

Collapse all items in the tree to the first level tree items.

- tree : tree widget to be collapsed.

```
TreeItem TreeGetSelection(Pointer tree);
```

This is a function for a tree widget that has style of TREE_SINGLE.

- tree : tree widget to be queried.
- return value : the selected TreeItem

```
Int TreeGetItemsSelected(Pointer tree, TreeItem **array);
```

Get all selected TreeItems. This function applies only to the tree widget that has style of TREE_MULTIPLE.

- tree : tree widget
- array : a pointer of a TreeItem array to be passed to the function to retrieve the TreeItem.
- return value : number of items that has been selected.

Tab

A tab is one of the few containers in the LS-PREPOST. Think of a tab widget as a notebook with tags on the side and you can then turn your page to where you tagged it and find your information.

A Tab widget can have Tabpage(s) as its children. You should also create Tabpages when using Tabs as your containers.

```
struct Tab {
     void *parent;
     char *tag;
     Rectangle anchor;
     char *help;
     char *style;
     char *whenclicked[];
};
```

- parent : parent of the Tab widget
- tag : tag for the Tab widget
- anchor : the location for the Tab widget
- help : comment for this tab widget
- style : here are the styles a Tab widget can have,

TAB_LEFT	Make all Tabpages in this widget have their labels on the left hand side of the Tab
TAB_RIGHT	Make all Tabpages in this widget have their labels on the right hand side of the Tab
TAB_BOTTOM	Make all Tabpages in this widget have their labels at the bottom of the Tab
TAB_TOP	Make all Tabpages in this widget have their labels at the top of the Tab, this is the default setting, if the style left null.
TAB_MULTIROWS	All Tabpages label will be tiled in the area assigned by any of the previous styles if the space of the Tab is not enough to line the labels in one row.

- whenclicked : if left null, Tab will switch to the page that the user asked. If whenclicked is not trivial, it will act similar to PushButton::onactivate. An object of TabCallStruct will be passed to the callout fuction. the name of the TabPage where the event takes place will be passed into the callout function.

```
struct TabCallStruct {
     int reason;
     char *name;
};
```

- reason : the reason for an event sending to a tab can be the one of the following.

scREASON_PAGELEAVE	the event will be sent when a TabPage is about to be out of focus
scREASON_PAGEENTER	the event will be sent when a TabPage is about to gain focus

- name : name of the TabPage that an event is taking place.

```
struct TabPage {
     void *parent;
     char *tag;
     char *help;
     int fraction;
};
```

- parent : parent of the Tabpage, it has to be a Tab widget
- tag : name of the page, users may use this name to switch between pages programmatically.
- help : comment for the page
- fraction : this is the fraction for children of this Tabpage. its definition is similar to the fraction of a Form widget. fraction has a default value of 100, if user left it as zero.

Manipulation Functions

```
void TabSelectPage(Pointer tab, Pointer name);
```

Select a page by its name.

- tab : pointer to the Tab widget
- name : the name of the Tabpage

```
void TabRenamePage(Pointer tab, Pointer from, Pointer as);
```

Rename a Tabpage to a new name

- tab : pointer to the Tab widget
- from : the old name of the Tabpage to be changed
- as : the new name of the Tabpage to be changed

```
Int TabGetPageIndex(Pointer tab, Pointer name);
```

Get the page index back from the tab. The page index of a TabPage is the position of the page in the Tab, or the sequence a TabPage created in a Tab.

- tab : pointer to the Tab widget
- name : name of a page
- return value : if -1, there is no page with such a name, otherwise, the page index is returned.

```
char *TabGetSelectedPage(Pointer tab);
```

retrieve the name of the TabPage that is currently get the focus in a Tab

- tab : pointer to a Tab widget
- return value : name of the currently selected page.

Dialog

SCRIPTO has two different dialogs; one is standard (modeless) and the other is modal. Currently, only modeless dialog is implemented. After a Dialog is created, it is shown by default, user can un-manage it if that is the requested behavior.

```
struct Dialog {
     void *parent;
     char *tag;
     Rectangle anchor;
     char *style;
     int fraction;
};
```

- parent : parent of the dialog, this field can be left 0, or equals to &FromDialog, since this is the default behavior.
- tag : the title of the dialog
- anchor : in dialog, this field, has a new meaning at its members. First, all values are in pixel units, in stead of the fraction units; and the
  x1 : x-coordinate of the left border of the dialog
  y1 : y-coordinate of the top border of the dialog
  x2 : width in pixel of the dialog
  y2 : height in pixel of the dialog
- style : styles can be one of the following values

DS_STANDARD	this is the default for now, if style is left 0, SCRIPTO will use DS_STANDARD as the Dialog's style.
DS_MODAL	Not yet implemented

- fraction : the fraction for the widgets that use this dialog as their parent. the default value is 100, if this field left 0.

Manipulation Functions

```
void DialogClose(Pointer dialog);
```

Use this function to close a dialog. When a dialog is closed, it is destroyed as well, that means all widgets that take this dialog as an ancestor will be destroyed as well. All Widgets will be invalid after the call.

- dialog : the pointer of a dialog.

```
void DialogMaximize(Pointer dialog);
```

Use this function to maximize a dialog. If a dialog can not be maximized, it will restore it instead.

- dialog : the pointer of a dialog

```
void DialogIconize(Pointer dialog);
```

Use this function to iconize a dialog. If it can not be honored for some reason, the function will do nothing.

- dialog : the pointer to a dialog

```
void DialogRestore(Pointer dialog);
```

Restore a dialog while it can not be maximized and maximize is called. Restore the size of a dialog to when it was, before calling DialogIconize or DialogMaximize

- dialog : the pointer to a dialog

Drawing

A Drawing Widget is a panel that users may use drawing tools to plot a 2D drawing. SCRIPTO provides toolbox object Picasso that can be associated with a Drawing widget in handling its drawing event. Picasso is the tool to draw on the Drawing widget.

```
struct Drawing {
     void *parent;
     char *tag;
     Rectangle anchor;
     char *help;
     char *onpaint[];
     };
```

- parent : a pointer to its parent widget
- tag : name of the drawing widget
- anchor : position and size of the Drawing
- help : the comment for this drawing widget
- onpaint : the is the event handler for the paint event for the Drawing widget. When a Drawing widget needed to be painted or refreshed, the callout function assigned at onpaint will be called.

Drawing widget's callout function on painting is not the same as other function. it has to be like the following

```
define:
void callout(Picasso p){ ... }
```

The Picasso object **p** that passed into the drawing event handler will be provided by SCRIPTO. This Picasso object is the drawing tool for the users to draw on the Drawing widget. However, since **p** will be created by SCRIPTO and it is considered transparent to the scripts. Users should not use this variable and assign or save the value to other Picasso objects after the painting event handler.

Manipulation Functions

```
void DrawingGetSize(Pointer p, Int *xsz, Int *ysz);
```

Get the size of the Drawing widget in pixel.

- p : pointer to the drawing widget
- xsz : width of the Drawing widget
- ysz : height of the Drawing widget

C-Parser Syntax Reference

This is an in-house interpreter that SCRIPTO relies on to parse all the scripts. It's developer is Dr. Trent Eggleston. You may reach him at: trent@lstc.com.

In general, C-parser follows most of the standards of C language. For an experienced C programmer, you can probably omit this reference to understand the demo scripts and develop your own. This part of the document will serve as a reference when LS-PREPOST reports strange errors that do not make sense to you.

Following is a list that shows you how this reference is organized:

- Lexical Conventions
- Basic concepts
- Conversions, Expressions and Statements
- Standard functions

Lexical Conventions

This part introduces some fundamental elements of C-Parser. You use these elements to build statements, definitions, and so on, which in turn you use them to build the whole script.

Comments

Comments are used to explain what you are doing in your codes.

Comments can be assigned in two ways:

```
/*
  this is a comment
*/
```

Or

```
//
// this is also a comment
//
```

Identifiers

An identifier is a string that denotes

- variable names
- structure names
- function names
- object names

To construct an identifier, you can use any of these characters

```
_ a b c d e f g h i j k l m n o p q
r s t u v w x y z A B C D E F G H I
J K L M N O P Q R S T U V W X Y Z
```

And all digits,

```
0 1 2 3 4 5 6 7 8 9
```

The rules are,

- First character has to be a non-digit
- An identifier can not exceed 63 characters.
- An identifier can not be any reserved keywords
- An identifier can not be any predefined symbols, such as those you read in the SCRIPTO documents

Keywords

Keywords are predefined identifiers. Here is the list of it.

break	char	Char	continue
define	else	float	Float
for	if	int	Int
return	struct	union	while
void			

Punctuators

Punctuators are characters that have syntactic or semantic meanings to the parser and are used to compose the statements or operators, however, they themselves alone does not yield any value.

Here is the list

```
~ ! % ^ & * ( ) - + = [ ] { } | : ; " ' < > , . ? / \
```

Operators

Operators specify an evaluation in the script.

Operator	Name	associativity
[ ]	array subscript	L -> R
( )	function call	L -> R
( )	conversion	none
->	member selection (pointer)	L -> R
.	member selection (object)	L -> R
*	dereference	none
&	address of	none
+	unary plus	none
-	arithmetic negation (unary)	none
!	logical NOT	none
~	bitwise complement	none
<<	bitwise shift to left	L -> R
>>	bitwise shift to right	L -> R
^	bitwise XOR	L -> R
&	bitwise AND	L -> R
|	bitwise OR	L -> R
%%	percent	L -> R
**	power	L -> R
*	multiplication	L -> R
/	division	L -> R
%	remainder	L -> R
+	addition	L -> R
-	subtraction	L -> R
==	logical equal	L -> R
!=	logical not equal	L -> R
<	logical less than	L -> R
<=	logical less than or equal to	L -> R
>	logical greater than	L -> R
>=	logical greater than or equal to	L -> R
&&	logical AND	L -> R
||	logical OR	L -> R
=	assignment	R -> L

C-Parser does not have the capability of doing increment, decrement or compound assignment in an operation as C or C++ can do. Scripts have to compose an statement to do this.

```
//this is how you do increment in the script

     a = a + 1;

// or compound assignment

     a = a * 8;

// the following will yield an error

     a++;

// so as this,

     a *= 8;
```

Constants

Constants can be subdivided into 4 categories,

- Integers
- Real numbers
- Characters
- Strings

Integers

For decimal : 0-9

EX:

```
     Int nine = 9;
     Int twenty = 20;
```

For octal : 0[0-7]

EX:

```
     Int o_nine = 011;
     Int o_twenty = 024;
```

For hexadecimal : 0x[0-9,A-F,a-f]

EX:

```
     Int x_nine = 0x9;
     Int x_twenty = 0x14;
```

Real numbers

[+ | -][0-9].[0-9][e | E][+ | -][0-9]

```
EX:
Float pi = 3.1415926;
Float mole = 6.02e23;
Float kappa = 1.38e-23;
Float e = 2.718281828;
```

Characters

Use single quotation to surround one of the source character set; or the following escape characters.

```
\n \' \" \\ \r \t \b \0
```

```
EX:
Char c = 'C';
Char newline = '\n';
Char quote = '\'';
Char dquote = '\"';
```

Strings

Use double quote to surround characters will create a null character-terminated string. the length of this string constant is the sum of the characters between the double quotes mark, however, the memory to hold the string is one character more as it is in C-Language.

```
EX:
    char *str0 = "This is the c-parser reference";
    char *str1 = "here is the \"other\" string";
```

Basic Concepts

These are the vocabulary used when scripting with C-Parser.

Declarations and definitions

Declaration is to tell C-Parser there is a name for a script element.
Definition is to tell C-Parser how the declared script element looks like.

Rules:

- A name has to be declared to be used.
- Declarations can be any where in the script.
- Declarations also serve as a definition in most cases, except when a function or a structure is declared. the definition can be defined somewhere else.

Scope

Names can be used only in a certain region of a script. This region is called "scope" of the name. You can use

```
{ statements; }
 or
functions
```

to define different scope.

However, there is a script scope that served as a global scope. Whatever you defined in a global scope can be used and changed there after and in other scopes that defines after them.

Script Definition

The whole script consists of the script that is in the global scope and the scripts that are in the function definitions.

```
[--> SCRIPTO specific]
```

If a script is included into an existing one, the global scope of the included script will extend the global scope of the previous script; all variables defined in the previous script will be seen immediately the including process is underway.

Including sequence thus become important. to make different scripts work together.

```
[<-- end SCRIPTO specific]
```

Interpreting and compiling

C-Parser is a semi-interpreted script parser. It interprets and executes the script immediately those statements in the global scope; while compiles the statements in the function definitions and executes them only when they are called.

Storage Classes

Inside C-Parser, all variables are assumed as static variables in C language. This is one of the distinct differences from C-language.

When a function is called, script writers are suggested to initialize all of variable, otherwise the static behavior of the variable might create unexpected result from your script.

Types

Here are the different types of objects

- Fundamental types
  These are the Float, float, Char, char, Int, and int
- Derived types

Pointers	address of the fundamental type or the structure type objects.
Arrays	variables that contains a number of specified types of objects.
Functions	objects take none or more variables of a given type and return a specified type of object. A function can return nothing as well, then the type of the return object will be void
Structures	Types that define by the script, it may contain fundamental types and other declared types. The size of a structure is the sum of all components.
Unions	Types that define by the script, it may contain fundamental types and other declared types. The size of a structure is the biggest size of the components

Initialization

An object can be initialized when it is declared. If an initialization is not given by the script, C-Parser will put zeros into the memory of the object. Example:

```
Int c = 2;     // this initializes c to 2
struct A {
     Int m1;
     Float f1;
     char name[20];
};
A aobj;   // this will initialize aobj's member
          // m1 to 0
          // f1 to 0.0
          // each of the character of name to '\0'
```

Noted that since all variables and object instances in C-Parser are classified as static variables, script developer should pay extra attention to initialize and re-initialize the variable before using the variable.

Function definition

To define a function in a C-Parser, the script should insert the keyword

```
define:
```

This is another distinct difference from C-Language.
Example:

```
//
// a function is defined as followed:
//
define:
void afunction (Int opt) {
     if (opt < 0)
          return;
     // other statements...
}
```

An error will be generated by C-Parser if "define:" is missing.

Function Declaration

To declare a function in a C-Parser, the dummy variable can not be included in the prototype of a function.
Example:

```
//
// declare a function prototype
//
void f1(Float, Float *, Int **); //ok
void f2(Float radius, Float *area, Int **rarray); //error
```

Conversions, Expressions and Statements

Conversions happen when evaluating a statement with two different types of objects.

Integer <--> Float conversion

This happens when evaluating between an integer and a Float, all integers converts to Floats as C-Language does.

Pointer <--> Pointer conversion

This is a bit different from C. Two pointers can convert as follow:

```
CallStruct *cp;
ListCallStruct *lcp;

lcp = malloc(sizeof(ListCallStruct));
cp = lcp;
```

The pointer cp here will have the same address of lcp.

Type Casting

Type casting does not exist in c-parser.

Expressions

Expressions are used to evaluate a value from operands and operators. Examples of expressions

```
a + b               //a two operand expression
!a                  //an unary expression
(a > 6 && b <3)     //conditional expression
&a                  //take the address of a variable
*p                  //dereference a pointer
b | x               //bitwise OR operation
NULL                // \0
object->parent      //get a member from a pointer to
                    // a object
(*object).parent    //get a member from an object
```

Statements

Statements are script elements that build the whole script, These statements control the flow of interpreting, compiling and executing. Statements are executed sequentially; however, there are other expressions that can direct how a program flow should go.

Expression Statements

Expression statement causes expressions to be evaluated. All operations in an expression statement will be evaluated and complete before the parser goes on evaluate the next.
Examples of expression statements

```
Float a, b, c;

a = 1.0;
b = 2.5;
c = (a * 5.0 + b* 3.0)/a*b;
```

Null Statement

A null statement is a statement that has nothing in there. A null statement can be plugged into a script when there is a statement needed, and yet nothing should be taken action.

Compound Statement

A compound statement is zero or more statements that enclosed in a curly brace { }.
Example of compound statements:

```
if ( option == 2 )
{                    //this is a compound statement
     a = 1.0;
     b = 2.5;
}
else
     i = 8;
```

Selection Statement

In C-Parser,

```
if
if...else
if...else if...else
```

are the selection statements that provides a means to conditionally execute a part of codes. C-Parser does not provide **`switch...case`** mechanism to do case switching, if a selection has to be made between multiple cases, the only way to implement it is to use **`if...else if...else`** mechanism.
Example of selection statements: (see, the example of compound statements)

```
if (day == 0)       printf("sunday\n");
else if (day == 1)  printf("monday\n");
else if (day == 2)  printf("tuesday\n");
else if (day == 3)  printf("wednesday\n");
else if (day == 4)  printf("thursday\n");
else if (day == 5)  printf("friday\n");
else if (day == 6)  printf("saturday\n");
else                printf("no such a day\n");
```

Iteration Statement

Iteration statements make statements to be executed zero or more times, termination criteria should be provided to exit the iteration loop.

In C-Parser,

```
while
for
```

are the keywords to construct iteration statements.

Example of iteration statements:

```
//while loop
Int i;
i = 0;
while (strcmp(string[i], "node"))
     i = i+1;
//termination condition meets, if any element
// of the char *string[] is "node"

//for loop
Int i;
for(i=0;i<10;i=i+1)
     printf("i = %d, i*i = %d\n", i, i*i);
//
// i = 0 <--- initial value for i
// i < 10 <--- termination condition
// i = i + 1 <-- incremental (or decremental)
//statement
```

Jump Statement

Jump statements provide a means for the execution flow to be interrupted and jump to the place where the statement assigned.

In C-Parser,

```
break
continue
```

return

are the keywords for jump statements

break and **continue** are used with iteration statements. **return** is used in a function definition.

break: can only be used in an iteration statement and it transfers control to the statement immediately following the iteration statement and thus exit the iteration.

continue: can only be used in an iteration statement and it transfer the control immediately to the loop-continuation evaluation statement in the iteration statement.

return: is used to exit a function definition, this will transfer the control back to the calling function immediately. If a function returns a value, return must have an optional expression that evaluates the return value; however, if the function's return type is void, a null statement is inserted instead.

Example:

```
define:
Int f(void) {
     return 1;
}

define:
void g(void) {
     return ;
}
```

Note that C-Parser does not provide "goto" keyword, which implies that unconditional execution flow interruption in a script is forbidden.

Standard Functions

Following is a list of available C library that implemented inside C-Parser. They have the same convention and functionality in C-Parser as they are in C library.

isalpha	isupper	isdigit	isxdigit	isalnum
isspace	ispunct	isprint	isgraph	iscntrl
isascii	atof	atoi	getenv	putenv
remove	unlink	strcasecmp	strncasecmp	strcat
strncat	strchr	strrchr	strcmp	strncmp
strcpy	strncpy	strdup	strlen	memcpy
memmove	memset	malloc	free	exit
fopen	fgets	fputs	fclose	fwrite
fread	fseek	rewind	clearerr	feof
ferror	printf	sprintf	fprintf	strdupf
sleep	open	read	readline	linediscipline
readlinetest	write	lseek	close	filestrerror
fileclear	ftruncate	truncate	stat	access
realloc	ceil	floor		

And their corresponding structures:

FILE	stat			