SCRIPTO

A New Tool for Customizing LS-PREPOST

R. Chen

LSTC

© 2006 Copyright by DYNA*more* GmbH

SCRIPTO

A New Tool
for
Customizing LS-PREPOST

LSTC
October 13, 2006

Overview

- Why Bother Customizing?
- SCRIPTO
- C-Parser
- Events
- The Scripting APIs
- The Entry Points
- Documents and Users Group
- Looking Ahead
- Demos, Q & A

Why Bother Customizing ?

- LS-PREPOST is a general purpose pre- and post- processor that provides extensive functionality for all LS-DYNA users to prepare and manipulate their models.
- Yet, the development team still receives the critics as most of the general purpose pre- and post- processors would get:
  - Steep learning curve
  - Scattered functionality

Why Bother Customizing ?

- A Call for a tool that may
  - Flatten the learning curve by redesigning look and feel of the application, so that new users can break in easier
  - Regroup the needed functionality so that users does not need to navigate through multiple steps to get things done.
  - Allow users to refocus back to their problem domains, and eventually leverage the productivity.

Why Bother Customizing ?

- SCRIPTO is the tool designed to have the script developers participated in the customizing process.
- Through SCRIPTO, users may
  - Redesign and re-implement the user interfaces.
  - Regroup and reorganize the existed functionality in the way users want.
  - Easily plug in a new functionality implemented by a third party or the users themselves to manipulate the model.

SCRIPTO – Behind the Scene

- Interactive Model

SCRIPTO – Behind the Scene

- Customized Interactive Model

SCRIPTO – What Is It

- SCRIPTO (SCRIPT-ing O-bjects)
  - Builds on an in-house script parser, C-Parser, that incorporates most of the C language syntax.
  - Is native after interpreted/compiled by C-Parser. Once parsed, the interfaces generated and functionality plugged become part of LS-PREPOST.

© 2006 Copyright by DYNA*more* GmbH

SCRIPTO – What Is It

- SCRIPTO (continued)
  - Has application programming interfaces (APIs) that open to all users. It is now over 270 functions provided by SCRIPTO.
  - APIs Can be categorized into 3 different areas base on the functionality they provide
    - User Interface APIs
    - Model Data APIs
    - Utility APIs

SCRIPTO – What Is It

- SCRIPTO (continued)
  - Can be developed into modules. Script modules can be reused, included and shared.
  - Is supported after LS-PREPOST 2.1 both on Microsoft windows and most Linux* platforms.

*only on wx builds.

SCRIPTO – What Is In It

- User interface APIs
  - There are 17 different classes of user interface supported in SCRIPTO

PushButton	ListBox
ToggleButton	Menu
RadioBox	Tree
TextField	Tab
ScrolledText	Dialog
Form	Spin*
Slider	MultiColumnList*
Separator	Grid*
Label	

SCRIPTO – What Is In It

- Model data APIs
  - Allow you to interface with the model data in LS-PREPOST directly.
  - Allow you to interface with more than one model simultaneously through DataCenter objects.
  - Currently, only the APIs that handle pre-processing data are implemented.

SCRIPTO – What Is In It

- Model data APIs (continued)
  - Can now generate the entities of
    - Basic geometries (nodes, elements, and parts)
    - Sets
    - Define curves
    - Coordinates
    - Vectors
    - SPCs
    - Prescribed motions
    - Initial velocities
    - ASCII output controls
- Keywords may also be imported directly through SCRIPTO functions. For those entities and controls that is currently not implemented, script develop can import them to their models.

SCRIPTO – What Is In It

- Utility APIs
  - Allow the script developers to take advantage of the tools implemented by LS-PREPOST
- Currently utility APIs has the following tools implemented
  - Keyword forms
  - General selection mechanism
  - File Open/Save Dialog

How Can I Get There ?

- Understand C-Parser
- Understand GUI programming concepts
- Understand Scripting APIs

C-Parser – The Language

- C-Parser
  - Is a script parser. It is developed and maintained by Dr. Trent Eggleston.*
  - has been linked to LS-DYNA, LS-OPT and now LS-PREPOST.
  - is more than an interpreter. C-Parser compiles all the subroutines defined by script developers. C-Parser is a half-interpreted and half-compiled parser.

*Dr. Trent Eggleston is an employee of LSTC

C-Parser – The Language

- C-Parser (continued)
  - has a very flat learning curve for users who know C-language.
  - Follows most of the standards of C-Language.
    - It uses similar lexical conventions, tokens, operators, keywords as C-Language does.
    - It comes with predefined functions and data structures that provide same functionality as standard C libraries.

C-Parser – The Language

- … differences (continued)
  - A multi-dimensional array should be initialized as a single-dimensional array.

```
Float ncoord[8][3] =  {0.0, 0.0, 0.0,
                       1.0, 0.0, 0.0,
                       1.0, 1.0, 0.0,
                       0.0, 1.0, 0.0,
                       0.0, 0.0, 1.0,
                       1.0, 0.0, 1.0,
                       1.0, 1.0, 1.0,
                       0.0, 1.0, 1.0};
```

Put all 24 real numbers together as it is initialized in a single dimensional array

```
Float ncoord[8][3] =  { {0.0, 0.0, 0.0},
                        {1.0, 0.0, 0.0},
                        {1.0, 1.0, 0.0},
                        {0.0, 1.0, 0.0},
                        {0.0, 0.0, 1.0},
                        {1.0, 0.0, 1.0},
                        {1.0, 1.0, 1.0},
                        {0.0, 1.0, 1.0} };
```

Put 24 real numbers in a 8x3 fashion, this will yield parsing errors.

C-Parser – The Language

- C-Parser (continued)
  - Allows scripts to
    - Do mathematical expressions evaluation
    - Execute statements conditionally
    - Iterate (Loop) through statements
    - Define function calls
    - Create user-defined data types
    - Use pointers and arrays
    - Include each other

C-Parser – The Language

- … Differences (continued)
  - Condition statements
    - switch…cases is not supported
    - Scripts have to use if…else if…else instead.
  - Iteration statements
    - do… while is not supported
    - Scripts have to use for-loop or while-loop instead.

No do..while or switch..cases blocks allow in c-parser

Correct way is to use while and if..else to implement the same script

```
Int i;
char buf[20];
i = 0;
/*
do {
    switch(i) {
    case 0 : sprintf(buf, " "); break;
    case 1 : sprintf(buf, "0"); break;
    case 2 : sprintf(buf, "0x"); break;
    default : sprintf(buf, "error"); break;
    }
    i = i + 1;
} while(i<3);
*/
while(i<3) {
  if(i==0)
    sprintf(buf, " ");
  else if (i==1)
    sprintf(buf, "0");
  else if (i==2)
    sprintf(buf, "0x");
  else
    sprintf(buf, "error");
  i = i +1 ;
}
```

C-Parser – The Language

```
/*LS-SCRIPT*/
void Func(Int);
define:
void main(void)
{
        Int square[20];
        Int primes[20] = { 2,  3,  5,  7, 11,
                          13, 17, 19, 23, 29,
                          31, 37, 41, 43, 47,
                          53, 59, 61, 67, 71};
        Int i;
        FILE *f;
        f = fopen("c:\\square.txt", "w");
        for ( i = 0; i < 20; i = i + 1) {
           square[i] = primes[i]**2;
           if(primes[i]<50)
             fprintf(f, "first half ");
           else
             fprintf(f, "2nd half ");
           fprintf(f, "prime = %d , square = %d\n",
                          primes[i], square[i]);
          Func(i);
        }
        fclose(f);
}
define:
void Func(Int i)
{
        char demo[40];
        sprintf(demo, "prime = %d , square = %d\n",
                          i, i*i);
        Echo(demo);
}
main();
```

comments – Function prototype – Array initialization – Build-in C structure – System access – Expression statement – Iteration statement, for-block – Condition execution, if-block – Formatted output – Function call – Function definition – Main script

C-Parser – The Language

… Differences (continued)

- There are several operators missing in C-Parser:
  - Increment and decrement : ++, --
  - Combined assignments: +=, -=, *=, /=, …
  - Conditional operator: ?:

```
Int a;
Int b;
Int c;

  a = 0;
  b = 1;

  // a ++;
  a = a + 1;

  // c = ++a * b;
  a = a + 1;
  c = a * b;

  // c += a * b;
  c = c + a * b;

  char buf[20];

  // sprintf(buf, " %d\n", c>2 ? a : b);

  if(c>2)
    sprintf(buf, " %d\n", a);
  else
    sprintf(buf, " %d\n", b);
```

Neither prefix, nor postfix increment are allowed

So as compound assignments

And no conditional operator either.

C-Parser – The Language

- Here is a list of differences between C-Parser and C-language.
  - All variables are static
    - Variables which are initialized, will not be re-initialized again in a defined function.

j is initialized only at the first call.

All successive calls will add the current i into j and the result looks like

0
1
3
6
10
15
21
28
36
45

```
/*LS-SCRIPT*/
void func(int);
define:
void main(void) {
  Int i;
  i = 0;
  while(i<10) {
    func(i);
    i = i+1;
  }
}

define:
void func(int i ) {
  char buf[20];
  Int j = 0;
  j = j + i;
  sprintf(buf, "%d\n", j);
  Echo(buf);
}
main();
```

C-Parser – The Language

- … Differences (continued)
  - The semi-interpret nature makes functions called by main script have to be defined first

```
/*LS-SCRIPT*/
Int sub(Int, Int);
define:
void main(void) {
  Int a;
  Int b;
  Int c;
  a = 20;
  b = 25;
  c = sub(a, b);
}
main();
define:
Int sub(Int a, Int b) { return a-b; }
```

Script starts from here.

•Left : Error. Although sub() has a prototype, the parser will not know the definition until main() has been executed.

•Right : OK.

```
/*LS-SCRIPT*/
Int sub(Int, Int);
define:
void main(void) {
  Int a;
  Int b;
  Int c;
  a = 20;
  b = 25;
  c = sub(a, b);
}
define:
Int sub(Int a, Int b) { return a-b; }
main();
```


© 2006 Copyright by DYNA*more* GmbH

Events

- Most GUI applications, when running, is in an idle state.
- GUI applications react only when users activate it through input devices or when the OS signals the user interfaces to change their internal states.
- We call these incidences of activation and signals - events.

Events

Object Type	Member Name (Event)	The Callback function is ...
PushButton	.onactivate	Called when a PushButton is clicked
ToggleButton	.oncheck	Called when a ToggleButton is checked
RadioItem	.onselect	Called when a RadioItem is selected
TextField	.whenentered	Called when the Textfield is in focus, and enter received
Slider	.whendragged	Called when the slider thumb is dragged
	.whenchanged	Called when the position of a slider thumb has changed.
Menu	.whenpulldown	Called when the menu has been pull down and a different selection has made by the user
SpinBtnInfo	.whenspinned	Called when the spin buttons are clicked
Spin	.whenentered	Called when the TextField part of spin received enter
ListBox	.whenpicked	Called when a different item has been selected
	.whendoubleclicked	Called when an item has been double-clicked
Tree	.whenactivate	Called when a different tree item has been selected
	.whendoubleclicked	Called when a tree item has been double-clicked
Tab	.whenclicked	Called when a page in the Tab is selected
Grid	.whenselected	Called when a grid cell is selected

Events

- In respond to an event, a GUI control can provide a function for the OS to connect them. Applications then participate in the execution flow this way.
- We call these reacting functions, the "Callback" functions for controls in responding of the events.

Events

- SCRIPTO allows two types of callbacks
  - Command Series

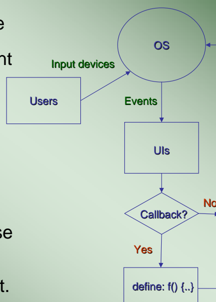
```
button1.onactivate = {"background 1.0 1.0 1.0",
                      "textcolor 0.0 0.0 0.0",
                      "labelcolor 0.0 0.0 0.0"};
```

  - Script defined callout functions

```
...
popdlg.onactivate = {"@popdlgfn"};
...

define:
void popdlgfn(DataField df, CallStruct *cs) {
  ...
  ...
}
```

Events

- In order to work with these users interfaces, script develops should implement these callback functions.
- Callback functions are assigned to the "on-" or "when-" members during the creation in SCRIPTO.
- LS-PREPOST calls these functions when the event happens.
- After the execution of these functions, the execution flow goes back to the OS and wait for the next event.



Events

- Events handled through command series
  - This option allows users to make customized widgets acts like mini cfile playback in LS-PREPOST
  - There is no limit the numbers of commands can be installed into a command series.

Events

- These "on-" and "when-" members are called the event members of the control.
- Not every type of UI control has event members; yet, some of the UI controls may have more than one event members.
- Script developers may choose to implement none or all event members.
- Following is the member name of events for each type of UI controls:

Events

- Events handled through callout function
  - Only one function is allowed to connect to an event.
  - User may assign an extra parameter to a callout function.
    .event = {"@function_name[, [@][#][$][%]data]"};
  - The leading character tells SCRIPTO how to pass the data to callout functions
    - @ : data interpreted as a pointer
    - # : data interpreted as an float
    - $: data interpreted as an string
    - % : data interpreted as an integer

Events

- …Callout function: (continued)
  - All callout functions have the same function prototype:

    void co_function(DataField, CallStruct *);

    - DataField :the extra parameter passed by users. Data Field is a union of a Pointer, an Integer, and a Float.
    - CallStruct * :the information provided by LS-PREPOST about the UI control participated in the event. This pointer must be converted to a derived CallStruct pointer before it can be used.

Events

Object Type	CallStruct * converted to
PushButton	Not used
ToggleButton	ToggleCallStruct *
RadioBox	RadioCallStruct *
Slider	SliderCallStruct *
Menu	MenuCallStruct *
Spin	SpinCallStruct *
ListBox	ListCallStruct *
Tab	TabCallStruct *
Tree	TreeCallStruct *
Keyword*	KeywordCallStruct *

The Scripting APIs

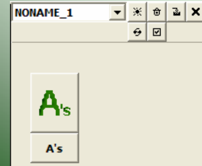
- In order to utilize the functionality provided by LS-PREPOST, different API functions ask for different information from script developers.
- User Interface APIs
  - Construction of a UI control
  - Setting/Retrieving states of a UI control.
  - Callback functions to an event occurrence.
- Model Data APIs
  - Model to connect to
  - Entity types to create and modify.
  - Keywords to import
- Utility APIs
  - Information an utility tool needed before you call.
  - Information retrieving after a request is completed.

User Interface APIs

- Construction
  - In general, UI APIs need following information to create an UI control
    - parent : every widget needs a parent to address where they come from. There are 3 root parents defined by SCRIPTO where the customization starts.
    - anchor : the real estate occupied by the widget.
    - style : describes how a widget look like
    - tag : the 'face name' of the widget
    - help : a descriptive sentence that tells users how the widget works.
    - other info : other properties for the widget to be created, and they are widget dependant.
    - events : the event member of the widget, some widget does not accept events, then they do not have this field.

User Interface APIs

- The listing on the right gives the result below.
- This short script demonstrates how style member would change the look and feel of an UI.
- SCRIPTO.pdf has the details of each member for different UIs; developers should take advantage of the information when scripting for LS-PREPOST.




```
/*LS-SCRIPT*/
Include("asxpm.xpm");
PushButton abutton;

  abutton.parent = &FromRight;
  abutton.anchor = {10, 5, 35, 15};
  abutton.style = "PB_XPM";
  abutton.help = "demonstrate a xpm button";
  abutton.pix = asxpm_xpm ;
  abutton.tag = "as_icon";

  CreateWidget(&abutton);

PushButton bbutton;

  bbutton.parent = &FromRight;
  bbutton.anchor = {10, 15, 35, 20};
  bbutton.tag = "A's";
  bbutton.help = "demonstrate a normal button";

  CreateWidget(&bbutton);
```

Model Data APIs

- Here are the general guidelines for a script to hook up a DataCenter:
  - Declare a DataCenter
  - Look up a model
  - Import a model to the DataCenter
  - Use manipulation functions to create entities in the DataCenter
  - Refresh the DataCenter

- Script to the right will create a single solid element with unit nodal coordinates.

Declare a DataCenter

Query for a model index

Import from the model

Create nodes, elements and parts, (DataCenter manipulation functions)

Refresh DataCenter

```
/*LS-SCRIPT*/
DataCenter dc;
Int *model_idx;
  model_idx = DataGetActiveModelIDList();
  DataImportFrom(&dc, model_idx[0]);
  free(model_idx);
Float ncoord[8][3] =  {0.0, 0.0, 0.0,
                       1.0, 0.0, 0.0,
                       1.0, 1.0, 0.0,
                       0.0, 1.0, 0.0,
                       0.0, 0.0, 1.0,
                       1.0, 0.0, 1.0,
                       1.0, 1.0, 1.0,
                       0.0, 1.0, 1.0};
PartInfo pi;
  pi.id = 1;
  pi.type = scETYPE_SOLID;
  pi.title = "a solid part";
  DataCreatePart(&dc, &pi);
  Int i;
  Int nid[8];
  for(i=0;i<8;i=i+1) {
    nid[i] = i+101;
    DataCreateNode(&dc, nid[i],
          ncoord[i][0], ncoord[i][1], ncoord[i][2]);

  }
ElementInfo ei;
  ei.id = 51;
  ei.type = scETYPE_SOLID;
  ei.conn = 8;
  for(i=0;i<8;i=i+1)
    ei.nids[i] = nid[i];
  DataCreateElement(&dc, 1, &ei);

  DataRefresh();
```

Utility APIs

- To use open/save file dialog Utility
  - Declare an object FileDialogInfo
  - Fill in the information needed by FileDialogInfo
  - Assign an event function for the dialog.
  - Call UtilFileDialog() functions
  - Use GetLastFileName() to retrieve the file selected by users.
- Once UtilFileDialog() is called, the execution will not continue until user 'ok' or 'cancel' the action.

© 2006 Copyright by DYNA*more* GmbH

Declare a FileDialogInfo

Primary filter

Allowed filters

Ok action

Call UtilFileDialog()

Since FileDialog is a modal dialog, execution flow stop here, until user responds.

Get the file name selected by the user

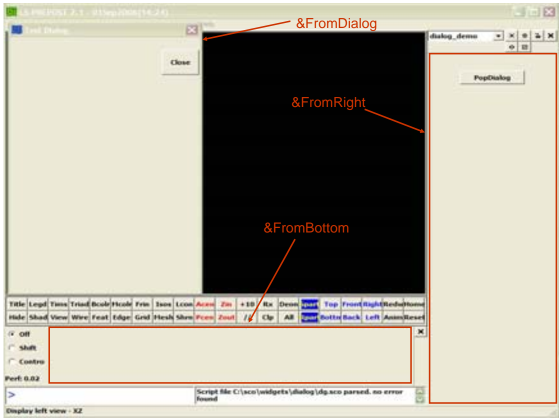
```
/*LS-SCRIPT:filedialogtest*/

void fdok(DataField, CallStruct *);
define:
void main(void){
FileDialogInfo fdi;

   fdi.parent = NULL;
   fdi.title = "open a scripto file";
   fdi.init_filter = "Scripto File(*.sco)|*.sco";
   fdi.filters = { "Binary Plot File (d3plot*)|d3plot*",
                   "keyword File (*.k;*.inf;*.key)|*.k;*.inf;*.key",
                   "Scripto File(*.sco)|*.sco"};
   fdi.ok_action = {"@fdok"};

   UtilFileDialog(&fdi);

  if(fdi.pressed != scOK)
    return;
  Echo("select a file");
}
define:
void fdok(DataField df, CallStruct *cs) {
   char *name;

   name = UtilGetLastFileName();
   Echo(name);
}
main();
```

The Entry Points

- There are 3 different globally defined root widgets.
- They are FromRight, FromBottom, and FromDialog
  - FromRight: is a Form, with a default fraction of 100.
  - FromBottom: is a Form, with a default fraction of 100.
  - FromDialog: is a place holder, its value is not important. All dialogs need to create from it.
- All widgets created by the script should have an ancestor of one of the 3 root widgets
- Root widgets can not be destroyed by the scripts.

More Model Data and Utility APIs

- There are few other model data and utility APIs, they all have been documented in SCRIPTO.pdf file.
- There are still other model data and utility APIs that yet to be implemented, so that users interested in customization may take advantage of them.



The Entry Points

- The signature
  - Any script that is a lawful SCRIPTO script has to have a comment line /*LS-SCRIPT[:script_name]*/
  - LS-PREPOST will not read any script that does not have a SCRIPTO signature.
  - A signature can have an optional script name. A script name can be up to 20 characters.
  - Whenever a script is loaded, a default script name "nonamex" will be given if user does not assign one.
- The verbose mode
  - Default is on.
  - User may turn off verbose setting on C-Parser, this will make LS-PREPOST not report the errors/warnings found.

The Entry Points

- Load a script
  - A script can be named in any fashion as long as the OS accepts.
  - However the .sco file extension (.sco pronounced as 'dot sko'), are recognized by LS-PREPOST

```
C:\sco\widgets\tree>
C:\sco\widgets\tree>
C:\sco\widgets\tree>lsprepost2_1_pc tr.sco

C:\sco\widgets\tree>
```

LS-PREPOST will launch a script called "tr.sco" in the current directory.

- On the right is a script template.
- Scripts organized in this order make debugging easier

Signature: mandatory

Function prototypes, and global variables

Change path to the include files

Include files

The main function: this is optional. All statements are assumed in the main, unless it is defined in a define: function.

Other define: functions

Call main function

```
/*LS-SCRIPT:script_name_here*/

/**
 prototypes
 */
void f1(DataField, CallStruct *);
void f2(DataField, CallStruct *);
...
/**
 * logistics
 */
Verbose(0);
ChangePath("\\my scripts\\");
...
/**
 * Includes
 */
Include("a.xpm");
Include("b.xpm");
Include("c.xpm");
...
/**
 * main script
 */
define:
void main(void)
{
  /**
   * ui creation, and others ..
   */
 ...
}
define:
void f1(DataField df, CallStruct *cs) { /*...*/ }
define:
void f2(DataField df, CallStruct *cs) { /*...*/ }

main();
```

The Entry Points

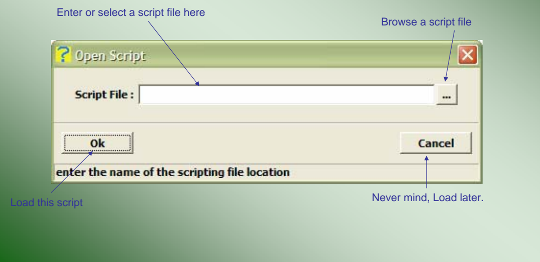
- Load a Script (Continued)
  - One may also load from the menu
    - [Applications]->[Customize]

File Misc. Toggle Background Applications Settings Help

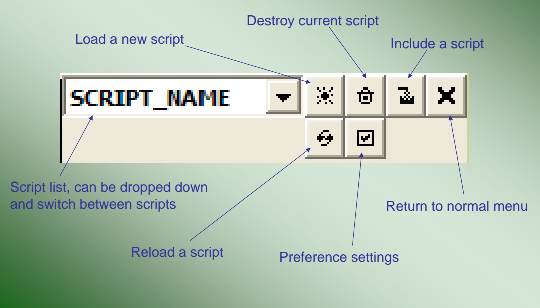
General
Forming
AirBag Folding
Occupant Position
Customize
MetalForming

The Entry Points

- In the “Open Script” dialog, load the script.



The Entry Points



Documents and Users group

- SCRIPTO.pdf
  - Contains every API for SCRIPTO provides by LS-PREPOST
  - Contains also a syntax reference to C-Parser.
  - You may download it from LSTC’s ftp site.
  - This document will be updated as frequently as further development of SCRIPTO goes.

Documents and Users group

- There is a user’s group on Google. Users are encouraged to join as a member to discuss, exchange scripts.

  http://groups.google.com/group/scripto

Looking Ahead

- SCRIPTO
  - is a customization tool provides by LS-PREPOST to all users to extend their experience with LS-PREPOST.
  - Gives users a means to better manipulate and organize their models.
  - Invites the users of LS-PREPOST to give it a new look
  - provides script developers the tool to impart their knowledge to LS-PREPOST and make LS-PREPOST from a one-size-fit-all general purpose pre- and post-processor to a one-of-a-kind tailor-made application.

Looking Ahead

- SCRIPTO (continued)
  - Is still improving. More functionality should be added into the API functions
    - Picking/Selecting mechanism
    - Plotting mechanism
    - More Entity Creation
    - Post-processing capabilities

Warnings and Reminders

- C-Parser is a powerful parser. System libraries it provides give script developers handy tools to develop their scripts. However, these function may also raise some concern of the security to a system whenever a script is loaded.
- Script developers and users are advised to read all scripts from a third party before using it.
- LSTC will not be responsible for any loss or damage caused by loading a SCRIPTO script.

© 2006 Copyright by DYNA*more* GmbH