SCRIPTO

A New Tool
for
Customizing LS-PREPOST

LSTC
October 13, 2006

Overview

- Why Bother Customizing
- SCRIPTO
- How Can I Get There
- Getting Started
- Support Materials
- Looking Ahead
- Demos, Q & A

Why Bother Customizing

- LS-PREPOST is a general purpose pre- and post- processor that provides multiple functions for LS-DYNA users to prepare and manipulate their models
- The development team still receives complaints as most of the general purpose pre- and post- processors have:
  - A steep learning curve
  - Scattered functionality

Why Bother Customizing

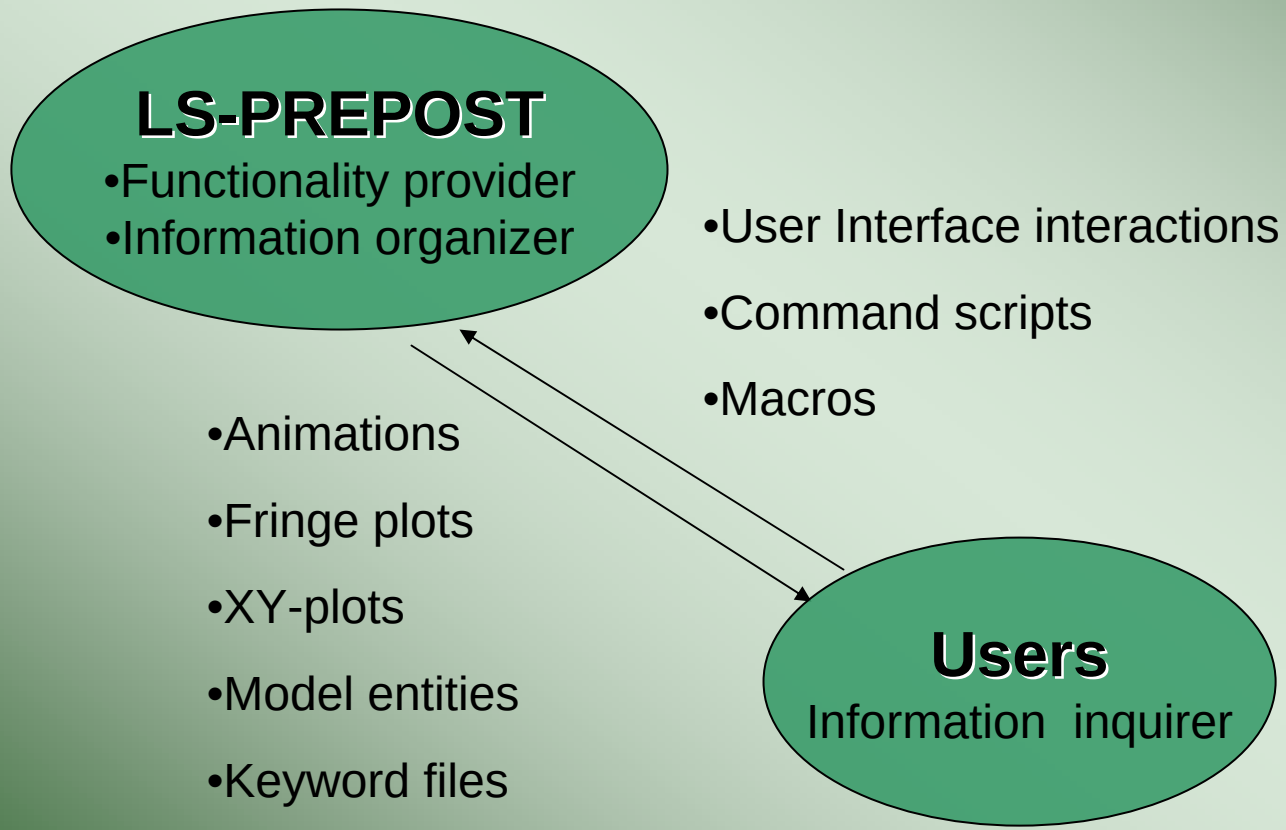
- There is a call for a tool that may
  - Flatten the learning curve by redesigning the look and feel of the application, so that new users can break in easier
  - Regroup the needed functions so that users do not need to navigate through multiple steps to get things done
  - Allow users to put the focus back on their problem domains, and eventually improve the productivity

Why Bother Customizing

- SCRIPTO is the tool designed to have the script writers participate in the customizing process
- Through SCRIPTO users may
  - Redesign and re-implement the user interfaces
  - Regroup and reorganize the existing functions
  - Easily plug in a new function implemented by a third party or themselves to manipulate the model

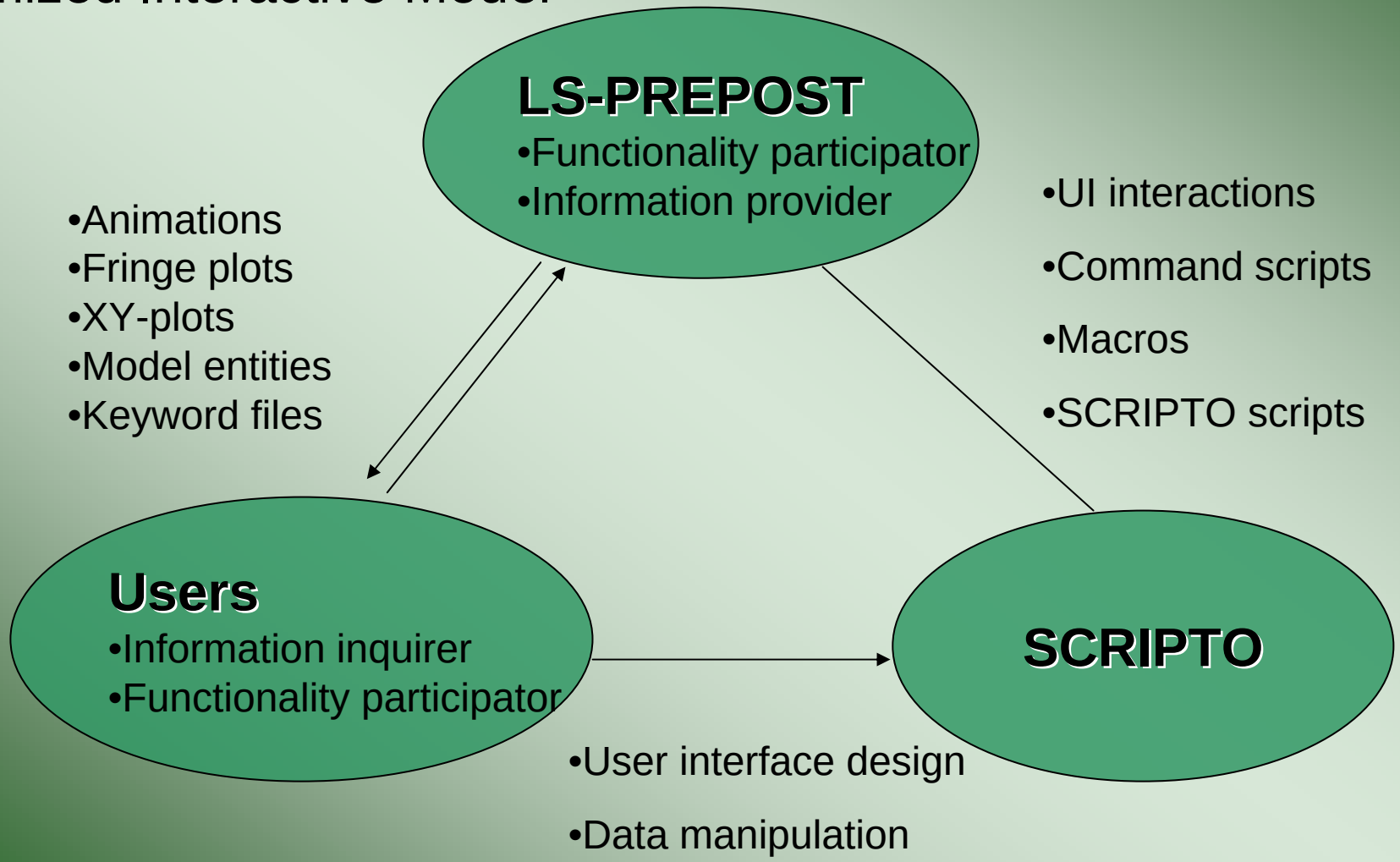
SCRIPTO – Behind the Scenes

Interactive Model



SCRIPTO – Behind the Scenes

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 functions added become part of LS-PREPOST

SCRIPTO – What Is It

- SCRIPTO (continued)
  - Has application programming interfaces (APIs) that are open to all script writers. There are now over 270 functions provided by SCRIPTO
  - APIs can be categorized into 3 different areas based on the functions they provide
    1. User Interface
    2. Model Data
    3. Utility

SCRIPTO – What Is It

- SCRIPTO (continued)
  - Can be developed into modules. Script modules can be reused, included, and/or shared
  - Is supported in 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 interfaces supported by SCRIPTO

PushButton
ToggleButton
RadioBox
TextField
ScrolledText
Form
Slider
Separator
Label

ListBox
Menu
Tree
Tab
Dialog
Spin*
MultiColumnList*
Grid*

*The manipulation functions are limited.

SCRIPTO – What Is In It

- Model Data APIs
  - Allow script writer to interface with the model data in LS-PREPOST directly
  - Allow script writer to interface with more than one model simultaneously through DataCenter objects
  - Are only partially implemented for handling pre-processing information

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

SCRIPTO – What Is In It

- Model Data APIs (continued)
  - Also provide functions to import LS-DYNA keywords data directly within the script

SCRIPTO – What Is In It

- Utility APIs
  - Allow the script writers to take advantage of the tools implemented in LS-PREPOST
  - Have the following tools:
    - Keyword forms
    - General selection mechanism
    - File Open/Save Dialog

How Can I Get There

- Understand
  - C-Parser
  - Events
  - Scripting APIs

C-Parser – The Language

- C-Parser
  - Is a script parser
  - Is developed and maintained by LSTC
  - Has been linked to LS-DYNA, LS-OPT and now LS-PREPOST
  - Is a semi-interpreted and semi-compiled parser
  - Compiles all the subroutines defined by script writers

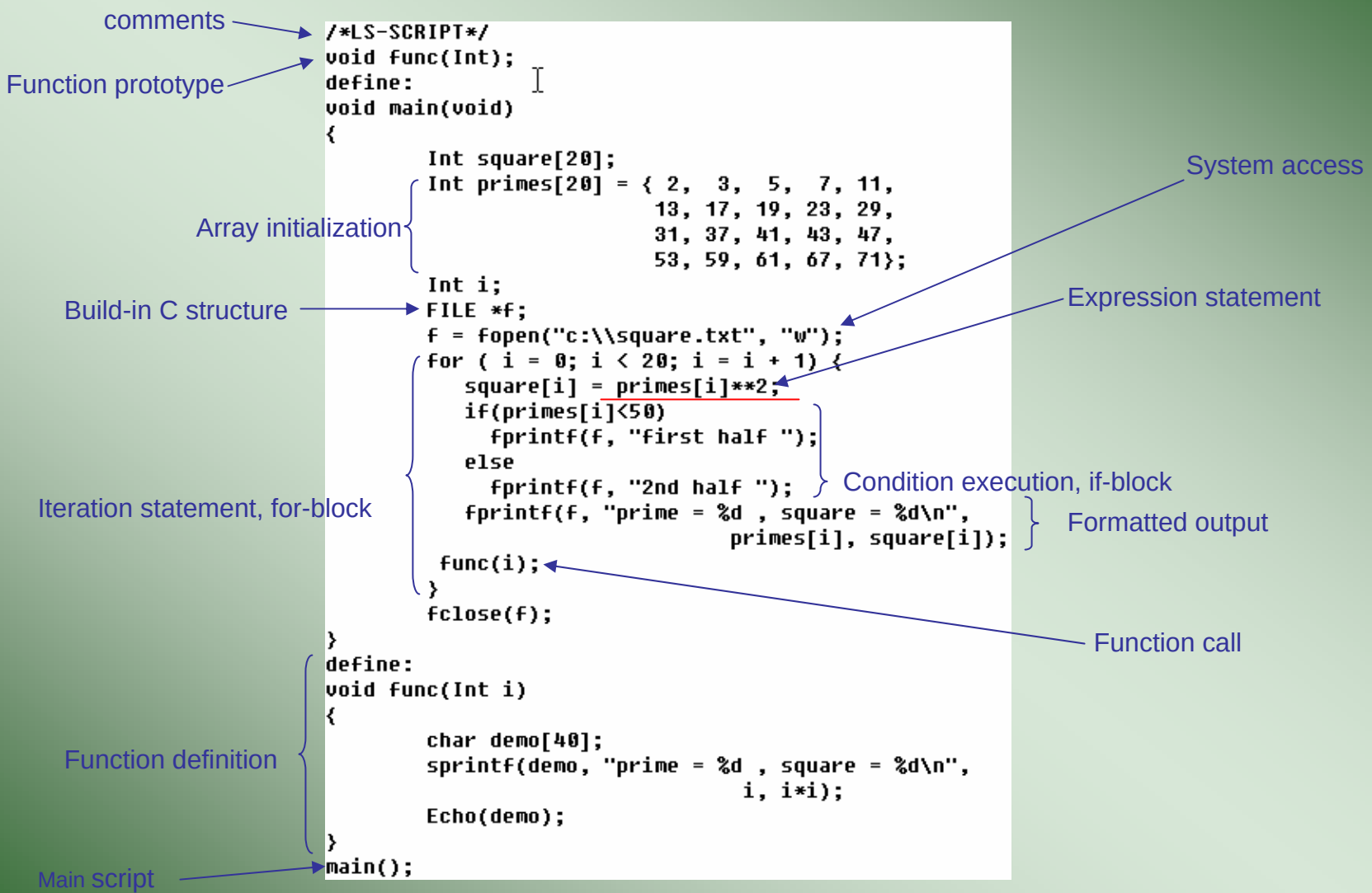
C-Parser – The Language

- C-Parser (continued)
  - Has a flat learning curve for the script writer who is familiar with C-Language
  - Follows most of the standards of C-Language
    - It uses similar lexical conventions, tokens, operators, and keywords that C-Language uses
    - It comes with predefined functions and data structures that provide the same functions as standard C libraries

C-Parser – The Language

- C-Parser (continued)
  - Allows scripts to
    - Do mathematical expression 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



C-Parser – The Language

- C-Parser differs from C-Language in that:
  - 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

- … differs in that: (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 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

- … Differs in that: (continued)
  - Condition statements
    - switch…case 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 allowed in c-parser

Use while and if..else to implement instead

```
 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

```
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);
```

… Differs in that (continued)

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

Neither prefix, nor postfix increments are allowed

Or compound assignments

Or conditional operator

C-Parser – The Language

- … Differs in that (continued)
  - The semi-interpret nature requires functions called by main script to be defined first

ERROR

```
/*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 here.

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

•Right : OK.

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();
```

Events

- Most GUI applications, when running, are in an idle state
- GUI applications react only when
  - users activate them through input devices
  - the OS notifies the UI to change its internal state
- We call these activations and notifications - events

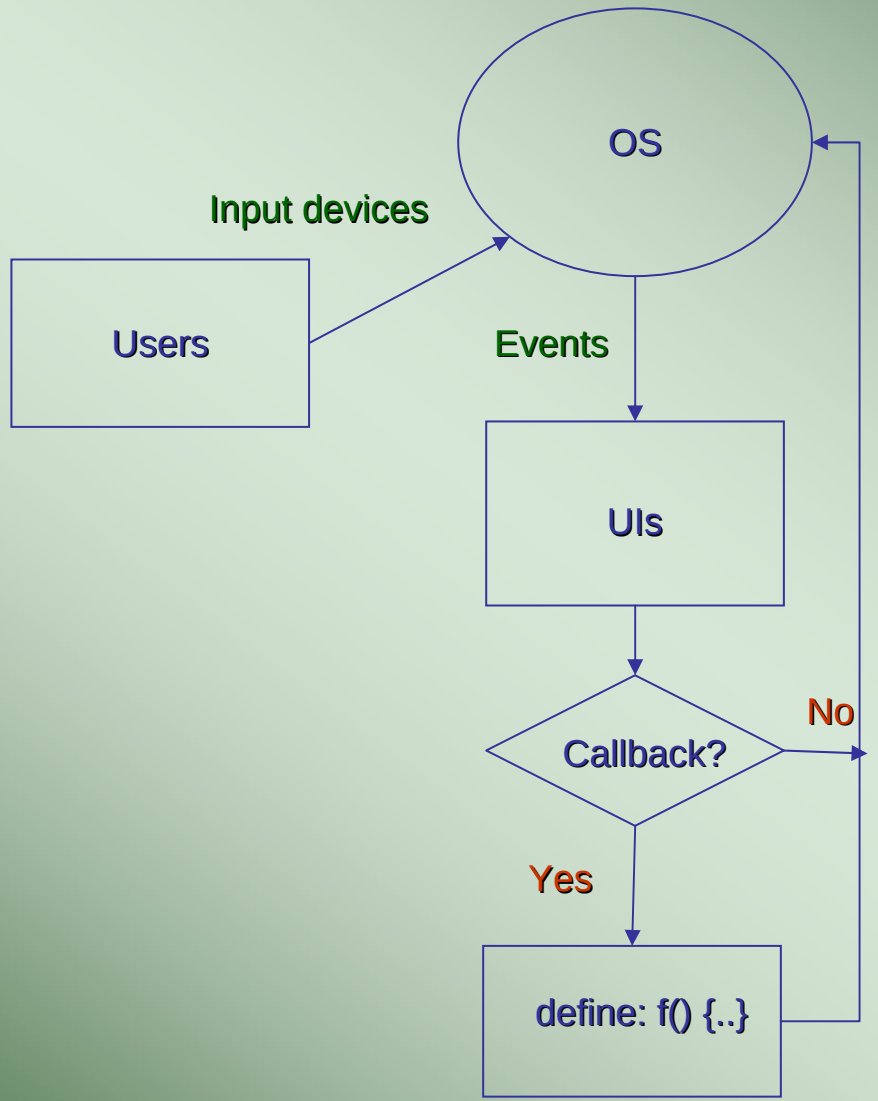
Events

- In response to an event, a GUI control can provide a function for the OS to connect to it
- This function then allows the application to participate in the execution flow
- We call the reacting function, the “Callback” function for the control in response to an event

Events

- Callback functions are assigned to the “on-” or “when-” members during creation
- LS-PREPOST calls the callback function when an event occurs
- After the execution of callback functions, the execution flow goes back to the OS and waits for the next event

Events



Events

- These “on-” and “when-” members are called the event members of the control
- Not every type of UI control has event members; however, some of the UI controls may have more than one event member
- Script writers may choose to implement none or all event members
- Following is the list of event members for each type of UI control:

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 or un-checked
RadioItem	.onselect	Called when a RadioItem is selected
TextField	.whenentered	Called when the Textfield is in focus, and the [enter] is 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 is pulled down and a different selection is made by the user
SpinBtnInfo	.whenspinned	Called when the spin buttons are clicked
Spin	.whenentered	Called when the TextField part of a Spin receives the [enter]
ListBox	.whenpicked	Called when a different item is selected
	.whendoubleclicked	Called when an item is double-clicked
Tree	.whenactivate	Called when a different tree item is selected
	.whendoubleclicked	Called when a tree item is double-clicked
Tab	.whenclicked	Called when a page in the Tab is selected
Grid	.whenselected	Called when a grid cell is selected

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

- Events handled through a command series
  - allow users to customize widgets to serve as mini-cfile playbacks in LS-PREPOST
  - Can use an unlimited number of commands for a single callback

Events

- Events handled through callout functions possess the following characteristics:
  - Only one function is allowed to connect to an event
  - Script writers may assign an extra parameter to a callout function

    .event = {“@function_name[, [@][#][$][%]data]”};
  - The leading character of the extra parameter directs SCRIPTO in passing the data to callout functions
    - @ : data interpreted as a pointer
    - # : data interpreted as a float
    - $: data interpreted as a string
    - % : data interpreted as an integer

Events

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

    void co_function(DataField, CallStruct *);

    - DataField :the extra parameter passed by script writers which is a union of a Pointer, an Integer, and a Float object
    - CallStruct * :the information provided by LS-PREPOST about the UI control participating in the event

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 *

Scripting APIs

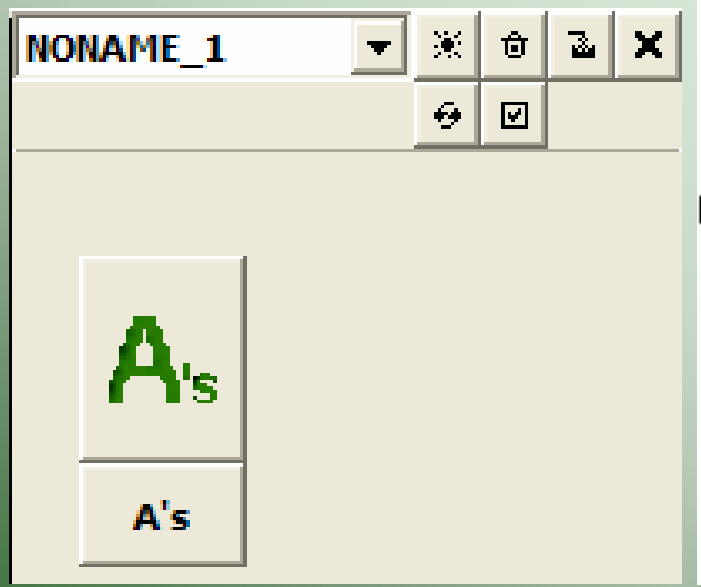
- User Interface APIs provide the functions of:
  - Construction of a UI control
  - Setting/Retrieving states of a UI control

Scripting APIs

- Construction
  - UI APIs need the following information to create a widget
    - parent : a widget that the current widget is based on
    - anchor : the position and size used by the widget
    - style : describes a widget's appearance
    - tag : the 'face name' of the widget
    - help : a descriptive sentence that tells users how the widget works
    - other info : other properties needed for the widget to be created
    - event : the event member of the widget

Scripting APIs

- This short script demonstrates how the style member may change the look and feel of a UI




```
/*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);
```

Scripting APIs

- Model Data APIs:
  - Use DataCenter objects to connect to models
  - Allow script writers/users to manipulate their model through creating/modifying entities

Scripting APIs

- Guidelines for a model 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

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();
```

Scripting APIs

- Utility APIs
  - Create/utilize sharable objects/mechanisms in LS-PREPOST

Scripting APIs

- To use open/save file dialog utility
  - Declare a FileDialogInfo object
  - Fill in the information needed by FileDialogInfo
  - Assign an event function for the dialog
  - Call UtilFileDialog()
  - Use GetLastFileName() to retrieve the file selected by users

Declare a FileDialogInfo

Primary filter

Allowed filters

Ok action

Call UtilFileDialog()

Since FileDialog is a modal dialog, execution flow stops here until the 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();
```

Getting Started

- Signatures of your scripts
- Verbose mode
- Organize your scripts
- Customizable areas
- Loading a script
- The script control panel

Getting Started

- The signature
  - A SCRIPTO script must have a signature as the first line:

    /*LS-SCRIPT[:script_name]*/
  - A signature can have an optional script name
  - A script name can be up to 20 characters
  - Script file name will be used as the script name if script writers do not assign one

Getting Started

- The verbose mode
  - If on, provides syntax or other error/warning messages to the user from LS-PREPOST
  - Default setting is ‘on’
  - Users may turn off verbose setting on LS-PREPOST

- Organize your scripts : a script template

Signature: mandatory

Function prototypes and global variables

Change path to where the include files are located

Include files

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

Other define: functions

Call main function

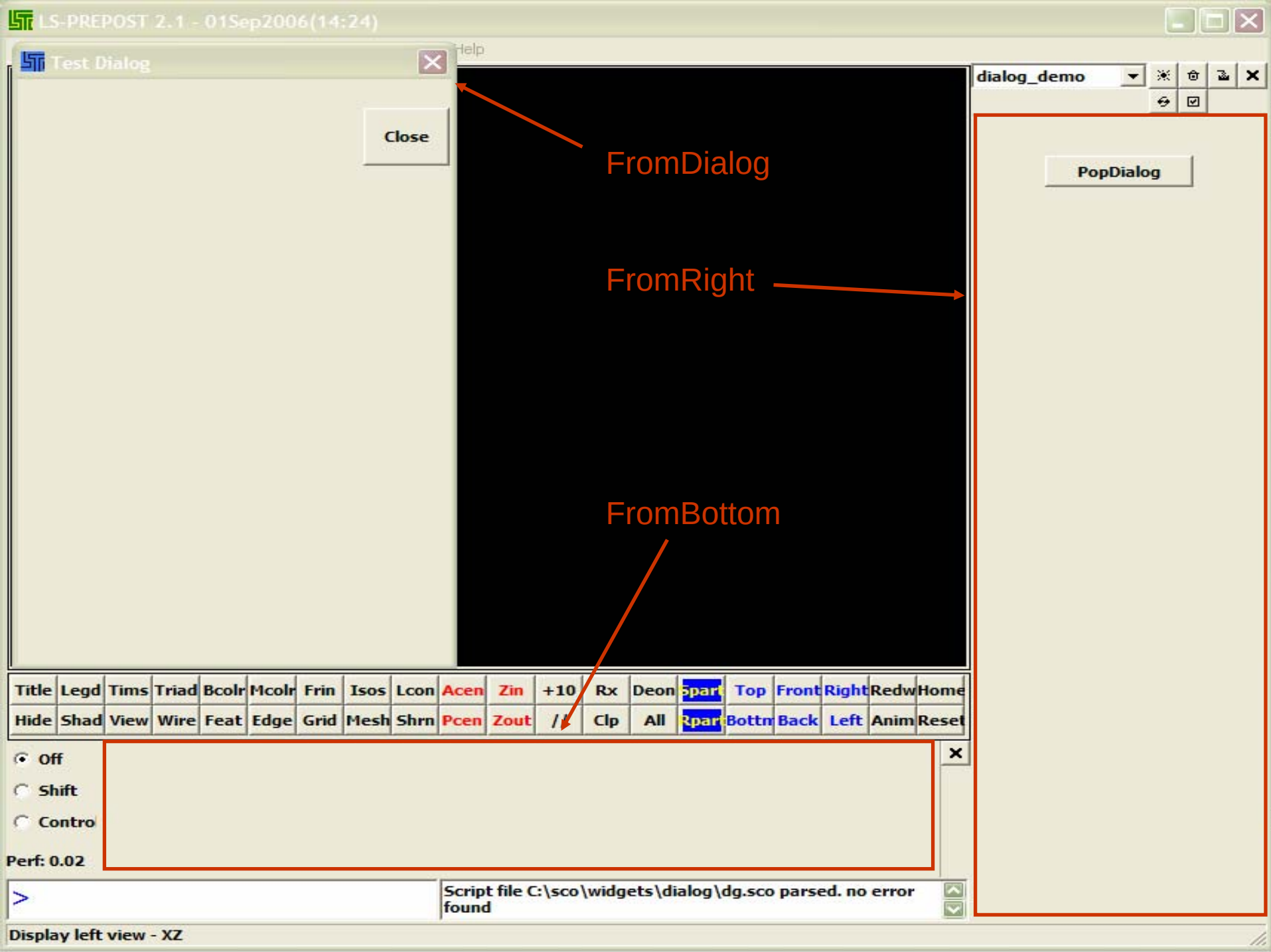
```
/*LS-SCRIPT:script_name_here*/

/**
 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();
```

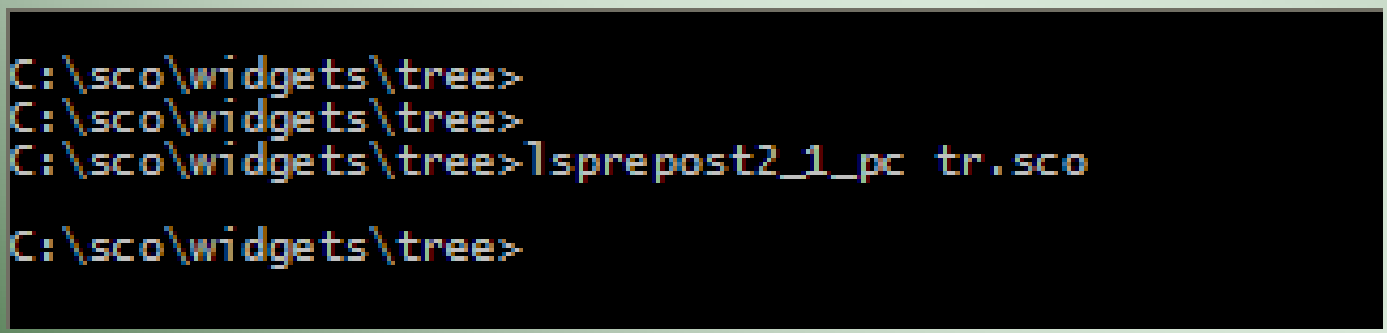
Getting Started

- Customizable areas
  - FromRight, FromBottom, and FromDialog are 3 different globally defined root widgets
    - FromRight: a Form, fraction = 100
    - FromBottom: a Form, fraction = 100
    - FromDialog: a place holder from which a dialog must originate
  - 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


LS-PREPOST 2.1 - 01Sep2006(14:24)
Help
Test Dialog
Close
FromDialog
FromRight
FromBottom
dialog_demo
PopDialog
Title Legd Tims Triad Bcolr Mcolr Frin Isos Lcon Acen Zin +10 Rx Deon Spart Top Front Right Redw Home
Hide Shad View Wire Feat Edge Grid Mesh Shrn Pcen Zout /1 Clp All Rpart Bottm Back Left Anim Reset
Off
Shift
Contro
Perf: 0.02
>
Script file C:\sco\widgets\dialog\dg.sco parsed. no error found
Display left view - XZ

Getting Started

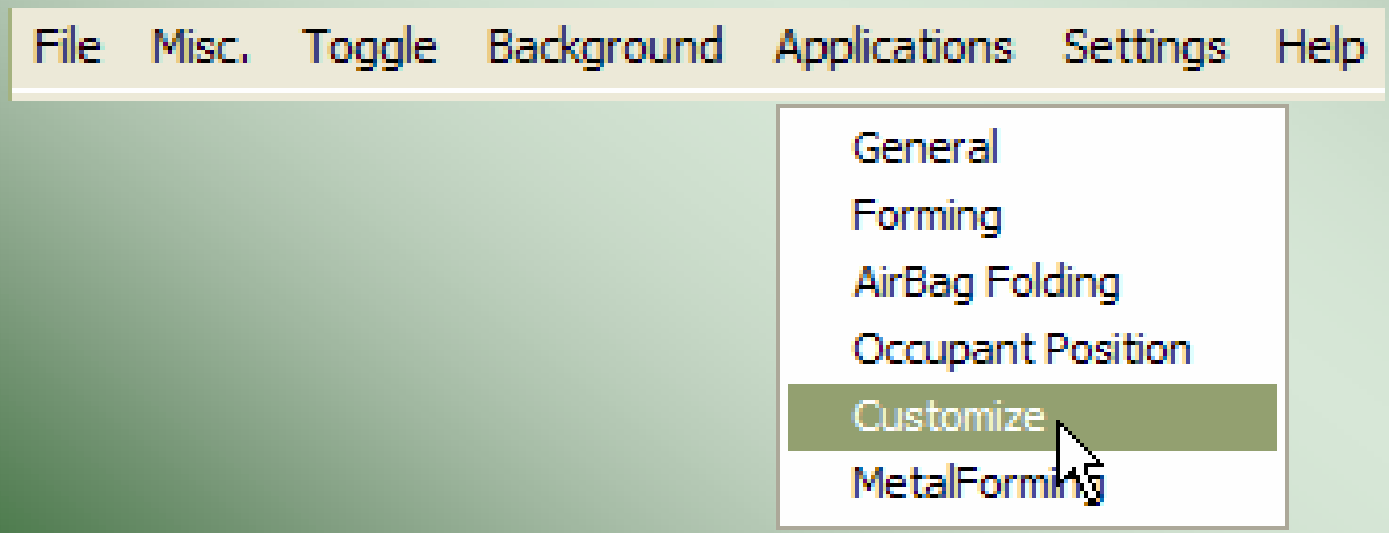
- Loading a script
  - A script can be named in any fashion as long as the OS accepts it
  - LS-PREPOST recognizes the .sco extension (pronounced ‘dot sko’) and will load the script so named automatically (as shown below)



LS-PREPOST will launch a script called “tr.sco” from the current directory

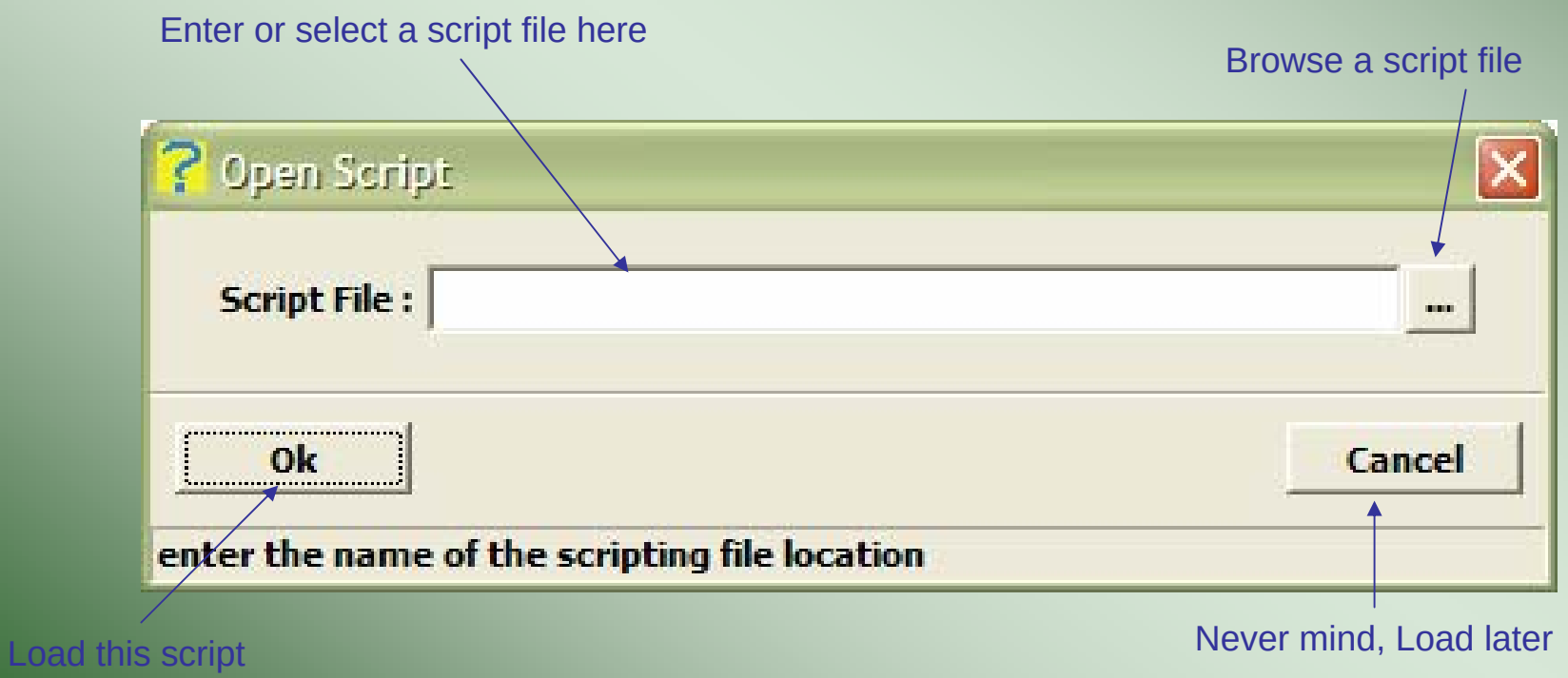
Getting Started

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



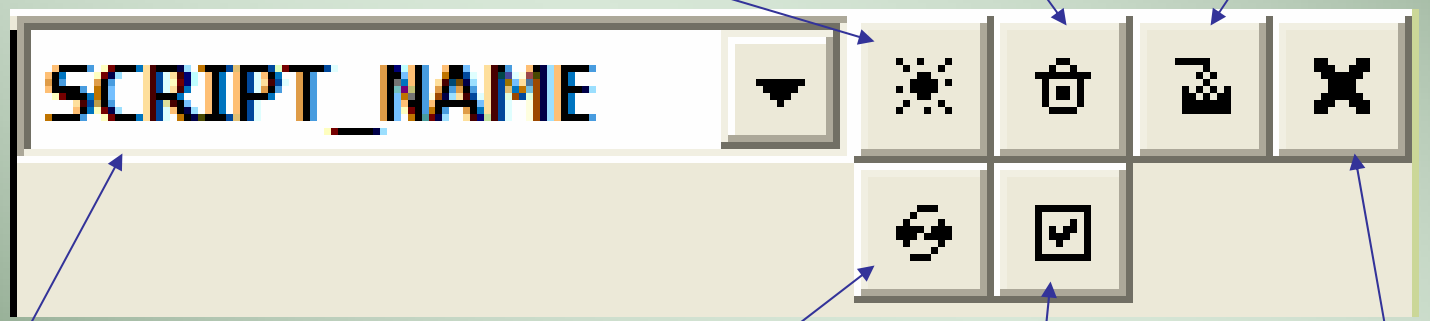
Getting Started

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



Getting Started

- The script control panel



Load a new script

Destroy current script

Include a script

Script list, can be dropped down to switch between scripts

Reload a script

Preference settings

Return to normal menu

Support Materials

- The Document - SCRIPTO.pdf
  - Contains every API for SCRIPTO provided by LS-PREPOST
  - Contains a syntax reference to C-Parser
  - May be downloaded from LSTC’s FTP site
  - Will be updated further as the development of SCRIPTO continues

Support Materials

- A users' group

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

  Provides a place for…
  - Q & A
  - Bug reports
  - Suggestions
  - Update Announcements
  - Script sharing

Looking Ahead

- SCRIPTO
  - Is a customization tool for all user to expand 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 writers a tool to impart their knowledge to LS-PREPOST

Looking Ahead

- SCRIPTO (continued)
  - Transforms LS-PREPOST from a one-size-fit-all general purpose pre- and post-processor to a one-of-a-kind tailor-made application
  - Is still improving as more API functions continue to be added
    - Picking/Selecting mechanism
    - Plotting mechanism
    - More entity creation functions
    - Post-processing capabilities

Warnings and Reminders

- System libraries provided by C-Parser give script writers handy tools to develop their scripts
- These functions may raise the security concerns to a system whenever a script is loaded
- Script writers and users are advised to read all scripts from a third party before using them
- Care should be exercised so that scripts will not cause damage to the system