COMPEL
Command Based Interpreter

By

Elias Bachaalany

A Senior Project Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science

AMERICAN UNIVERSITY COLLEGE OF SCIENCE & TECHNOLOGY

11/11/2006

Table of contents

Chapter 1 Built-In extensions reference

Outline

Welcome.compel script

This is a simple COMPEL script to get you started. It demonstrates how to show message boxes, do an IF/ELSE and how to trigger a “YES/NO” dialog.

```
var $name
var $gender
var $mstatus

inputbox $name "What is your name?" "Enter information"

yesnobox $gender "Are you a male?" "Gender"

if $gender != 1
{
  yesnobox $mstatus "Are you married" "Martial Status"
}

// Is it a male?
if $gender == 1
{
  var $gender "Mr."
}
// Female?
else
{
  // Is she married?
  if $mstatus == 1
  {
    var $gender "Mrs."
  }
  else
  {
    var $gender "Ms."
  }
}

msgbox "Welcome " $gender " " $name

end
```

Variables Terminology

COMPEL engine recognizes variables and objects embedded in a command and replaces them with their appropriate values. COMPEL recognizes variables only if they are preceded by "$" sign. Variables preceded with "&" are not parsed directly and considered a call by reference similarly to other high level languages.

Value substitution

Let us consider a simple script:

```
var $age 18
echo "The age is " $age "\n"
```

This script will cause the output:

```
The age is 18
```

Object substitution

Objects are replaced by their "to_string()" representation. Base object of type "object_t" has a default "to_string()" which simply displays all the attributes of the object and their values.

```
var $info.

var $info.name "elias"
var $info.lastname "b."
echo "The info is:" $info "\n"
```

This will cause an output similar to:

```
The info is: {name: "elias", lastname: "b."}
```

List of built-in extensions

COMPEL comes with a set of built-in extension created to enrich the scripting engine and to diminish the need for developers to create basic extensions. Such built-in extensions deal with variable management, Control Structure and Arithmetic operations.

Arithmetic	Allows you to do basic arithmetic operations
Comments	Adds support for single and multi-line C++/PHP style comments
Comparison	Adds assembly style binary comparison support
COMPEL_LIB	Allows the script to manage the compel library
CRT	Screen and keyboard related operations
Directory	Directory and file system enumeration operations
File I/O	File manipulation
Include	Inclusion of other scripts
Memory objects	Memory buffer and objects manipulation
Scopes	Adds the support of repetition structures and C style conditional branching operations
Shell	Adds support to run commands through system shell
Strings	String manipulation operations
Threads	Thread creation operations
Time	Time management operations
Unconditional Branching	Unconditional branching and GOTO support
Variables	Variables and objects creation and manipulation

Variables - Extensions

The variable extension's purpose is to provide variables creation and manipulation

Command	Description
alias	Creates a variable or command alias
assign	Similar to "var" command but does not create instead it just update the value.
unvar	Deletes a variable or an object
var	- Creates a variable - Creates an object - Creates an attributes within an object - Updates a value

Namespace: none

Check the fcd_vars*.compel sample series

CRT - Extensions

Command	Description
`delay`	Causes the execution of the script to delay a specified number of milliseconds
echo	Outputs a value to the console window. If no console was there, the first call to this command will create a console
echoln	`This command is similar to calling "echo" then writing a "\n" (new line)`
`echoxy`	This command is the combination of "echo" and "gotoxy"
`getch`	Waits for a key press inside the console window
`getxy`	Returns the console's X and Y coordinates
`gotoxy`	Sets the cursor location in the console window. Where X (horizontal axis) and Y (vertical axis) has (1,1) as origin
`inputbox`	Prompts for a value using a graphical input box
`inputline`	Waits for a line input inside the console window
`msgbox`	Displays a message using a message box.
`textattr`	Changes the text foreground and background color
`yesnobox`	This command displays a graphical message box with "Yes" and "No" prompts

Namespace: "crt"

Check the fcd_crt*.compel sample series

Arithmetic - Extensions

Command	Description
`add/sub/div/mul`	Achieves addition, substraction, integer division, and multiplication
expr	This is a full expression evaluator allowing the following operators: • \ : integer division • * : multiplication

	• % : Modulo • + : Addition • - : Substraction • ^ : Bitwise XOR • & : Bitwise AND • | : Bitwise OR • && : Logical AND • || : Logical OR • !| : Logical NOR • !& : Logical NAND • == : Integer equality test • != : Not equal • << : Bitwise shift left • >> : Bitwise shift right • < : Less than • > : Greater than • >= : Less than or equal • <= : Greater than or equal

The “expr” command may allow complex expressions combined with variables, such as:
expr $result ($delta + 1900) * $height / $width

Namespace: “math”

Check the fcd_arith*.compel sample series

Comments - Extensions

The comments extension is responsible for introducing commands with no side-effects such as comments in High-Level functions.
For example the command:
This is a command
The “#” does nothing and ignores its parameters.

The following comment syntax is supported: “//”, “#” for single line comment, and “/*” and “*/” for multi-line comment like in C language

Check the fcd_comment*.compel sample series

COMPEL_LIB - Extensions

This extension allows the script to talk with the COMPEL_LIB directly.

Only one command is exposed at the moment.

”dl”: Load external extensions.

Check the “canvas” examples in the “binaries” folder

Mandatory Namespace: “compellib”

Directory - Extensions

Command	Description
`enumfiles`	Enumerate files in the specified directory
`enumdrives`	Enumerates system drives
`Getdir`	Returns the current directory
`chdir`	Changes the current working directory
`fileexists`	Checks if a file exists
`direxists`	Checks if a directory exists

Namespace: “dirs”

Check the fcd_dirs*.compel sample series

File I/O - Extensions

Command	Description
`fopen`	Opens a file
`fclose`	Closes a file
`fread`	Reads from a file
`fwrite`	Writes to a file
`fseek`	Seeks to a new file position

When you open a file, a new variable is introduced to the symbol table.
The variable is an object of type file object.

The following attributes are present in the file object:

ok	A Boolean variable designating whether the last operation succeed or failed.

	Applies for all file operations
read	Contains the total read count
name	The name of the opened file
eof	A Boolean variable designating whether the end of file has been reached or not
write	The total number of bytes written so far
pos	The current file position. The position is automatically updated after a read/write or seek operation

Namespace: “fileio”

Check the fcd_fobj*.compel sample series

Include - Extensions

The include extension allows one script to include into its line another script.
It works similarly to other high-level languages.
The command is: “include”

Namespace: none

Check the fcd_include*.compel sample series

Memory objects - Extensions

```
Memory objects/extensions are closely related to the file objects /
extensions.
```

Command	Description
`alloc`	Allocates memory
`mfree`	Frees allocated memory

When memory is allocated, a new memory object is created.
The memory object has the following attributes:

ptr	The system address where the new allocated memory resides
size	The size of the allocated memory
length	The length of the zero terminated string located at “ptr”

The memory object’s to_string() method will return a C string of its contents.

However, a memory object will preserve its binary contents when used with File I/O routines.

Namespace: none

Check the fcd_mem*.compel and fcd_fobj*.compel sample series

Scopes - Extensions

The scopes extension provides a large set of commands:

Command	Description
`{`	Initiates a scope
`}`	Closes a scope
`for`	Creates a repetition scope by providing a starting count value and ending count value. It supports “to” and “downto” counting, meaning forward counting and backward counting
`command`	Creates a user command scope
`if`	Initiates a conditionally executed scope
`else`	Optionally used with “if” command to create an alternative scope for an “if” scope
`return`	Jumps to the end of a command scope
`break`	Jumps to the end of a scope
`continue`	Jumps to the beginning of a scope

A scope is simply a bunch of script lines enclosed within opening and ending curly braces.

```
1:{
2:   Echo “Hello world!\n”
3: }
```

Only one line exists in this scope. The scope begins at line 1 and ends at line 3.

Scopes can be nested, that means, scopes can exist within each other:

```
1: {
2:  {
3:    echo “Hi”
4:  }
5: }
```

Now we have two scopes; scope 1 [line 1-5] and scope 2 [line 2-4].

A user command scope, is a scope that names a given scope, so that when that name is used, the scope contents are executed. This is similar to procedures/functions in high-level commands.

Namespace: none

Check the fcd_ifelse*.compel, and fcd_for*.compel, fcd_recursive*.compel and other scripts that may contain the { and } commands.

Shell - Extensions

The shell extension allows one to interact with the system.
Such as shelling (or executing) commands.

Only one command is exposed: “shell”

Namespace: “shell”

Check the fcd_shell*.compel sample series

Strings - Extensions

The string extension provides string manipulation routines. One command is exposed: “tokenize” which is supposed to tokenize (break down string) into tokens.

Namespace: “string”

Check the fcd_string*.compel sample series

Threads - Extensions

This is an experimental extension.

It will allow the execution of a user command in parallel with the main script execution.

For example, you may want to create a program that constantly displays the time.

Mandatory Namespace: “thread”

Check the fcd_shell*.compel sample series

Time - Extensions

This extension allows you to retrieve date and time. It creates a basic object and fills it with time values.

Command	Description
`getdatetime`	Returns date and time into a basic object
`gettickcount`	Returns the system's ticks count (in milliseconds) since the system is started. This is useful to calculate how long the system was up.
`rand`	Returns a random number
`randomize`	Randomizes the random number generator

Namespace: "time"

Check the fcd_time*.compel sample series

Conditional and Unconditional Branching - Extensions

`Conditional branching means that we go to a new execution point only if a condition is met. Unconditional branching means we always go to a new execution point with no prior conditions.`

Command	Description
`goto`	Jumps to a labeled script position
`gotoline`	Allows you to jump to a script line directly if you know it is number. This is a dangerous call. It is advised that you used named labels with "goto" command
`end`	Ends the script's execution. It is similar to C's "exit()" and Pascal's "halt()" functions.
`label`	Creates a named label at the given line. A label will be a symbolic name holding the current line number
`if_gt/if_gte/if_lt/ if_lte/if_eq/if_neq`	Conditionally jumps to a given label if the condition is met. "gt" stands for "greater than" and so on.

Namespace: none

Check the fcd_count*.compel, fcd_branch*.compel sample serie

Chapter 2 COMPEL_LIB C++ Reference

Outline

List of examples

In this section we will be giving a set of examples on how to use COMPEL LIB in the C++ language. The aim of this section is to demonstrate how to get started with COMPEL from the developer point of view.

Script Initialization / De-Initialization

The following script will illustrate how to initialize the scripting engine, how to set up the required extensions and how to get started with COMPEL.

```
void example_initialize(compel_script_t &script)
{
  compel_init_t init = {0};
  init.b_usefullns = false;

  init.b_dbgout_script = true;

  // get all the avail extensions
  init.extensions = compel_script_avail_extensions();

  // something like this:
  //
"binary_arith;binary_comparison;comments;crt;fileio;include;memory;scop
es;shell;unconditional_branching;vars"
  // is returned.

  // initialze the script engine
  script = compel_script_init(&init);

  // do something...
  // ...
  // ...
  //

  // De-initialize script
  compel_script_deinit(script);
}
```

Script Running

The following script will illustrate how to run a single script line using COMPEL or how to load and run a complete set of script lines.

```
int example_run_script(compel_script_t &script)
{
  // script lines
  char *script_lines =
    "echoln \"Hello world\";"
    "echoln \"Welcome to COMPEL\";"
    "echoln internal:> $_COMPEL <\n;";

  compel_script_load_lines(script, script_lines, ";");
  int err = compel_script_run(script);

  return err;
}
```

Script Pausing example

This example shows you how to pause the script and change its state from running to paused. This is an important aspect and functionality of the scripting engine, this function is used inside the COMPEL IDE debugger.

```
static DWORD __stdcall PauseThread(compel_script_t script)
{
  _getch();
  printf("will pause...");
  compel_internal(script, compel_internal_pause, 0, 0, 0);
  printf("paused...\n");
  return 0;
}

void example_pause_script(compel_script_t script)
{
  ::CreateThread(0, 0, PauseThread, (LPVOID)script, 0, 0);
  compel_script_load_lines(script, "var $i 0|for $i 0 to 1000|{|echoln
\"i=\" $i|}|", "|");
  compel_script_run(script);
}
```

Value Manipulation

This sample will illustrate how to create variables and how to manipulate existing

variables.

```
void example_value(compel_script_t &script)
{
  compel_script_interpret_line(script, "var $s value!");

  // find the value "$"
  compel_value_t val = compel_value_find(script, "$s");

  // get the variable's value
  std::string s = compel_value_get(script, val);

  // patch string - enclose in angle brackets
  s = "<" + s + ">";

  // update the variable's value
  compel_value_set(script, val, s.c_str());

  // display the "$s" value
  compel_script_interpret_line(script, "echoln s= $s");

  // create a new variable
  compel_value_create(script, "$el", "elias");

  // show the value of the newly created variable
  compel_script_interpret_line(script, "echoln el = $el");
}
```

Object Manipulation example

This sample will illustrate how to create objects and how to manipulate existing objects. Notice that an object can be considered as a variable however it can hold multiple values in its different attributes.

```
void example_object(compel_script_t &script)
{
  // create OBJ1 using SCRIPT
  compel_script_load_lines(script, "var $obj1.|var $obj1.name john|var
$obj1.lastname doe", "|");
  compel_script_run(script);

  // create another object "obj2"
  compel_object_t obj = compel_object_create(script, "$obj2");

  // create attribute in obj2
  for (int i=0;i<10;i++)
```

```
  {
    char attr[20];
    sprintf(attr, "attr%d", i);
    compel_object_add_attr(script, obj, attr, attr);
  }

  compel_script_interpret_line(script, "echoln obj2: $obj2");

  // delete the object "obj2"
  compel_object_destroy(script, "$obj2");

  // adjust an attribute
  obj = compel_object_find(script, "$obj1");
  compel_value_t val = compel_object_find_attr(
    script,
    obj,
    "name");

  // adjust the attribute's value
  compel_value_set(script, val, "elias");

  // show object info
  compel_script_interpret_line(script, "echoln before_lastname_remove:
$obj1");

  // remove an attribute
  compel_object_remove_attr(script, obj, "lastname");

  // show object info - after removing an attribute
  compel_script_interpret_line(script, "echoln after_lastname_remove:
$obj1");

  // find another object
  obj = compel_object_find(script, "$_COMPEL");
}

void example_object_2(compel_script_t &script)
{
  compel_script_load_lines(script, "var $o.;var $o.a a;var $o.b b",
";");
  compel_script_run(script);

  char *s;
  s = compel_object_to_string(script, "$o");
  compel_string_destroy(s);

  s = compel_script_evaluate_expression(script, "$o.b . $o.a", false,
0);
  compel_string_destroy(s);

  compel_script_interpret_line(script, "echoln $o");
}
```

User Command example

This sample illustrates how to extend the COMPEL commands by creating and registering additional “user command”. The sample shows how a “lowercase” command was registered and shows how the body of the function is coded within the C++ language.

```
/*!
\brief lowercase user command
This command takes the first parameter by value and returns the
lowercase version
*/
int COMPEL_API my_lowercase(compel_script_t compel_script, int argc,
char *argv[])
{
  size_t len = strlen(argv[0]);
  char *s = new char[len+1];
  strcpy(s, argv[0]);
  strlwr(s);

  compel_lu_cmd_set_retval(compel_script, s);

  delete [] s;

  return 0;
}

/*!
\brief uppercase user command by reference
Takes two arguments, the first one is the variable's name that will
hold the uppercase
The 2nd argument is the string that needs to be uppercased
*/
int COMPEL_API my_uppercase(compel_script_t script, int argc, char
*argv[])
{
  compel_value_t v = compel_value_find(script, argv[0]);
  if (v == 0)
    return compel_error_symbol_expected;

  size_t len = strlen(argv[1]);
  char *s = new char[len+1];
  strcpy(s, argv[1]);
  strupr(s);

  compel_value_set(script, v, s);

  delete [] s;
```

```
  return compel_error_success;
}

void example_user_command(compel_script_t &script)
{
  int err;

  // register a command, the long way
  lib_usercommand_info_t cmd = {0};
  cmd.cb = my_lowercase;
  cmd.minargs = 1;
  cmd.maxargs = 1;
  cmd.name = "my_lowercase";
  err = compel_lu_cmd_register(script, &cmd);

  assert(err == compel_error_success);

  // register another command the quick way
  err = compel_lu_cmd_register2(script, my_uppercase, "my_uppercase",
2, 2);

  assert(err == compel_error_success);

  char *lines =
    "var $s1 \"tHiS iS a StRiNg\";"
    "var $s2 $s1;"
    "echoln \"before user command calls: \" $s1;"
    "my_lowercase $s1;"
    "echoln \"lowercase: \" $my_lowercase;"
    "my_uppercase &$s2 $s1;"
    "echoln \"uppercase: \" $s2;"
    ;

  // load the test script
  compel_script_load_lines(script, lines, ";");

  // run the script
  compel_script_run(script);
}
```

External extension loading

This sample illustrates how to extend the scripting engine by loading external extensions. External extensions are special user commands that reside in external DLL files. External extensions are very useful if you want to distribute binaries to end-users without giving the them the source code of your extension.

```
int example_user_ext(compel_script_t &script)
{
  int err;

  err = compel_script_load_lines(script,
    "var $s \"HElLo WoRld\";"
    "ext_lowercase $s;"
    "echoln \"x=\" $ext_lowercase;"
    ,
    ";"
    );
  err = compel_extension_load(script, "ext", "..\\Debug\\ext_dll.dll");

  err = compel_script_run(script);

  return err;
}
```

Script Error handling

Error handling sample will illustrate how the programmer can write an error handler to catch all scripting errors and decided what to do when the script error occures. This sort of error handling is also implemented within the COMPEL IDE debugger.

```
// the error handler
int COMPEL_API script_error_handler(compel_script_t script, size_t
lineno, int liberr)
{
  const char *faulty_code = compel_script_get_line(script, lineno);

  printf("error handler caught error %d @ %d:\n>%s<\n", liberr, lineno,
faulty_code);

  return compel_error_success;
}

// shows how to install an error handler to handle script errors
void example_error_handler(compel_script_t script)
{
  compel_script_set_error_handler(script, script_error_handler);
  compel_script_load_lines(script,
    "echoln okay;"
    "some_bad_command;"
    "echoln after_bad_command",
    ";");
  compel_script_run(script);
}
```

Script Internal commands

COMPEL library allows the developer to issue high-level commands via the normally documented commands and also allows low-level commands via the compel_internal() function call. The compel_internal() function allows the user to accomplish multiple functionalities by selecting which functionality via the internal command code.

```
void example_internal_script_load_and_show(compel_script_t &script,
char *fn)
{
  compel_script_load_file(script, fn);
  compel_internal(script, compel_internal_showlines, 0, 0, 0);
}

void example_internal_multiple_scripts(compel_script_t &script)
{
  const int max_scripts = 4;
  char *scripts[max_scripts] =
  {
    "lib_test1.compel",
    "lib_test2.compel",
    "lib_test3.compel",
    "lib_test4.compel"
  };

  compel_script_clear_lines(script);

  for (int i=0;i<max_scripts;i++)
    example_internal_script_load_and_show(script, scripts[i]);
}

void example_internal_1(compel_script_t &script)
{
  compel_script_load_file(script, "lib_test1.compel");

  compel_internal(script, compel_internal_setdbgout, 1, 0, 0);

  compel_script_set_lineno(script, 0);
  compel_script_run(script);

  compel_internal(script, compel_internal_writeraw, 0, 0, 0);

  printf("execution ended @ line: %d in (%d msecs)\n",
    compel_script_get_lineno(script),
    compel_internal(script, compel_internal_exectime, 0, 0, 0)
    );
}
```

Tokenizer example

COMPEL library also provides some utility functions such as the string tokenizer, and the following example demonstrates how to use the tokenizer.

```
void example_tokenizer()
{
  compel_tokenizer_t tok;

  tok = compel_tokenize_init("a;b;c;d", ";", "\"", 0);

  size_t pcount = compel_tokenize_parsed_count(tok);

  for (size_t i=0;i<pcount;i++)
  {
    printf("@%d=%s\n", i, compel_tokenize_get(tok, i));
  }

  compel_tokenize_free(tok);
}
```