User's Guide

HTML Preprocessor

Version 1.1

a product of

Software Analysis & Forensic Engineering Corporation

Table of Contents

1 Overview

The HTML Preprocessor is a software application developed by SAFE Corporation to help research engineers use SAFE CodeSuite to find copied code in web pages. It runs on Microsoft Windows-based PCs and transforms HTML files and other web site files into a form amenable to analysis by CodeSuite.

Figure 1shows a high level diagram of the Preprocessor. A research engineer may use either the Preprocessor Windows Application or the Preprocessor Command Line Program. The former provides a Windows-based graphical user interface, while the latter is a Windows console application that provides a command line user interface. When using either program the user specifies an input directory containing one or more web files to analyze, a set of input filename patterns, and a directory where output files should be stored.

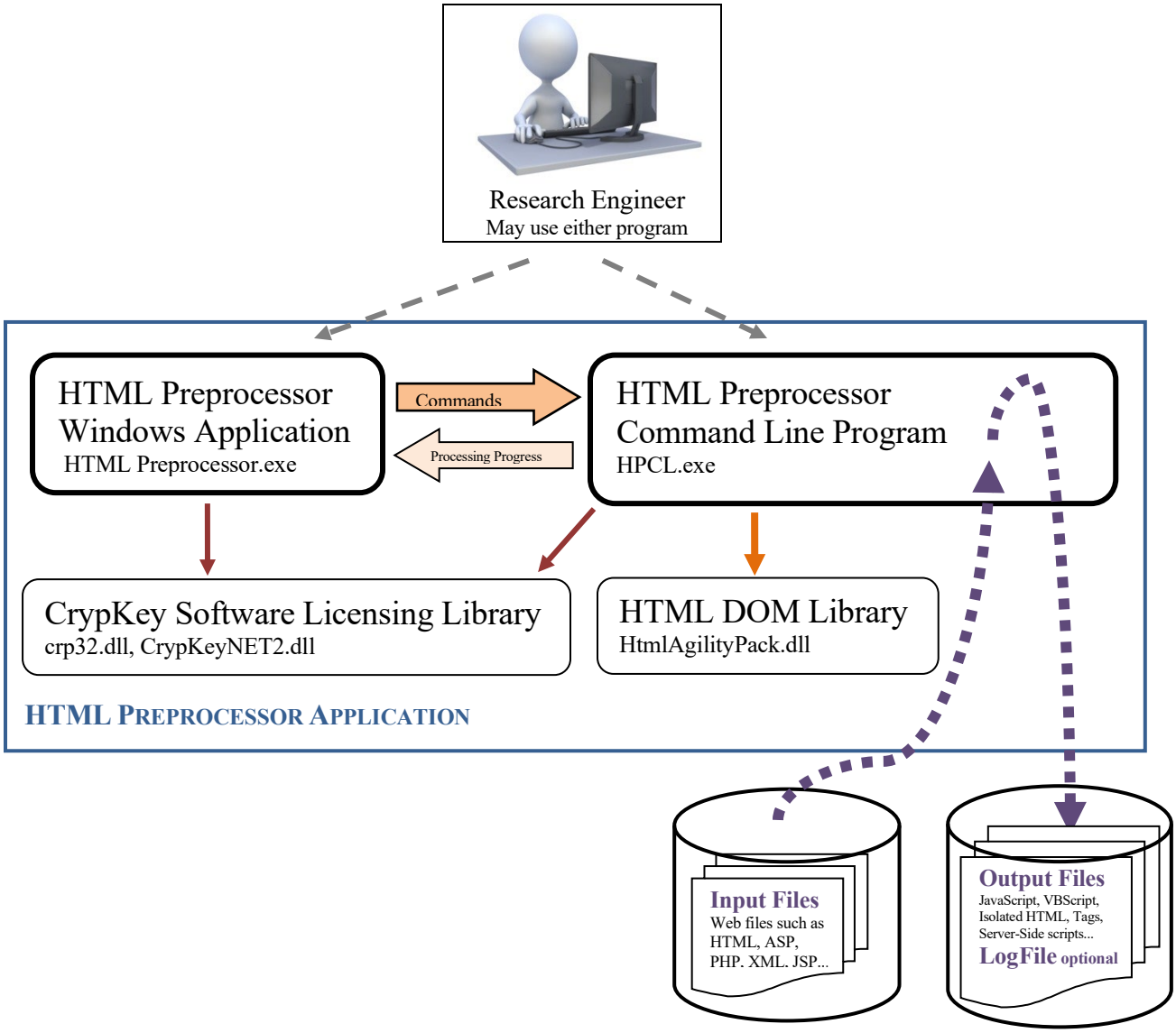


Figure 1. HTML Preprocessor High Level Architecture

The application analyzes all input files in the input directory that match the filename pattern, and saves the corresponding output files to the output directory. The user may also tell the program to recursively walk the input directory and its subdirectories and perform the analysis on each file in the input directory tree where the input file name matches the filename pattern. When the recursion option is used, the program creates an output directory tree that mirrors the input directory tree, and stores output files in corresponding subdirectories of the output tree.

After the HTML Preprocessor has processed all input files, the research engineer performs the next stage of analysis by using the Preprocessor's output files as inputs to CodeMatch, DocMate, or possibly other CodeSuite analysis tools. Subsequent analysis of the Preprocessor's output files using CodeSuite tools is beyond the scope of this document. See the SAFE Corporation document *"Website Analysis Procedure"* for best practices when using the HTML Preprocessor in conjunction with CodeSuite to find copied web code.

2 Supported Input and Output File Types and File Naming Scheme

While reading this section it may be helpful to review Appendix A, which shows examples of several types of input files and the corresponding output files generated by the Preprocessor.

2.1 Supported Input File Types

The HTML Preprocessor can process several different types of files. The Preprocessor was developed primarily for transforming HTML files into a set of files that CodeSuite can analyze. It can also be used to transform server-side web files, such as Active Server Pages and JSP pages, and other types of files such as XML files, into files that CodeSuite and/or DocMate can analyze.

2.2 Output File Naming Scheme

The HTML Preprocessor extracts various kinds of information from input files and produces different types of output files depending on an input file's contents. When the Preprocessor analyzes an HTML file it produces up to 8 different types of output files and may produce multiple instances of each file type. The Preprocessor can generate an even wider variety of file types for each server-side file. To make it easier for the user to identify which input file a given output file came from, the Preprocessor uses the following naming scheme for all output files:

Output File Name = Input File Name + Input File Extension(s) + ".out"
+ *Output-file-type Extension(s)*

This scheme preserves the input file name and all of its extensions so that unique output file names will be used even in scenarios where two input files have identical file names and some identical file extensions. This situation can occur in Active Server Pages where `Foo.asp` may have an associated code-behind file `Foo.asp.cs`.

2.3 Output Files Generated for an HTML Input File

Table 1 lists the different types of output files the Preprocessor may produce for an HTML input file and also shows how output files are named.

Consider an HTML file *Foo.html* that contains two cascading style sheet definitions and three blocks of JavaScript code. The Preprocessor would generate:

- Two CSS output files that contain the respective cascading style sheet definitions;
- Three Java Script output files that contain the respective blocks of JavaScript code;
- An "ExternalScripts" HTML file in which *Foo.html*'s embedded style sheet definitions and java script code are replaced with references to the above style sheet and script output files;
- An HTML Tags file which contains the sequence of HTML element as they occur in *Foo.html*;
- A text file containing all the comment strings that were found in *Foo.html*;
- A text file containing all the user-visible message strings that were found in *Foo.html*;
- And a C-code file that contains a pseudo-code representation of the HTML markup in *Foo.html*.

Input File Name	Output File Types and Names
Foo.html	Text Filename: **Foo.html.out.txt** Contains text from **Foo.html**.
	Comments Filename: **Foo.html.out.comments.txt** Contains comments from **Foo.html**.
	HTML Tags Filename: **Foo.html.out.tags.txt** Contains the HTML tags from **Foo.html**.
	Styles Filenames: **Foo.html.out.Internal_A.css** **Foo.html.out.Internal_B.css** Contains cascading style sheet definitions from **Foo.html**.
	JavaScript Filenames: **Foo.html.out.Internal_A.js** **Foo.html.out.Internal_B.js** **etc.** Each file contains one java-script code block from **Foo.html**.
	VBScript Filenames: Each file contains one VB-script code block from **Foo.html**.
	Pseudocode Filename: **Foo.html.out.c** Contains a C-code translation of the HTML from **Foo.html**.
	HTML with External Scripts and Style Blocks replaced with references Filename: **Foo.html.out.ExternalScripts.html** Contains the HTML code from **Foo.html** with all style sheets, JavaScript and VBScript removed, and with references to them replaced with references to the CSS, JavaScript and VBScript output files described above.

Table 1. Possible Output Files for an HTML Input File

2.4 Output Files Generated for Server-Side Input Files

Server-side web files typically contain a combination of HTML and server-side script. The server-side script typically executes on a web server and produces HTML that is then downloaded to the client's browser. When the HTML Preprocessor analyzes a server-side file it first splits the file into two files:

- A server-side script file, which it stores as an output file with the extension ".SSX", and
- A file containing the client-side elements, which it stores as an output file with the extension ".HTM".

The Preprocessor then feeds the HTML file back through the Preprocessor and processes it just as an input HTML file would have been processed.

The Preprocessor performs this analysis on each of the following types of server-side files:

- Active Server Page, extension .ASP
- .NET Active Server Page, extension .ASPX
- PHP Hypertext Preprocessor, extension .PHP.[1]
- Java Server Page, extension .JSP

Table 2 lists the different types of output files the Preprocessor may produce for an .ASPX server-side file. Table 2 shows a nested table of HTML output files to represent the output files that are generated when the HTML output file is fed back through the Preprocessor. Although Table 2 only shows an .ASPX input file, an analogous set of output files and output file names would be generated for .ASP, .PHP and .JSP input files.

Current Limitation and Bugs

Sometimes the HTML Preprocessor does not correctly identify a script block as server-side code. For example, it does not properly recognize the "`runat=server`" tag in an HTML `SCRIPT` element. Appendix A includes an example of this situation. This is a bug that will hopefully be fixed in a future drop.

Also, the HTML Preprocessor currently leaves server-side code that was embedded in the server side file, such as C# or VB.NET code in an .ASP file, in the .SSX output file. Support for splitting out server-side code into separate output files may be added in a future drop.

[1] The PHP website says PHP is a recursive acronym for "PHP: Hypertext Preprocessor".
See: http://us3.php.net/manual/en/faq.general.php

<table>
<tr><th>Input File Name</th><th>Output File Types and Names</th></tr>
<tr><td>Foo.aspx</td><td>
Server-Side Script
Filename: Foo.aspx.out.ssx
Contains the portions of Foo.aspx file that are server-side code blocks.
The HTML Agility Pack DOM is used to parse the input file and determine what parts of it are server-side code blocks.
<hr>
Client-Side Elements
Filename: Foo.aspx.out.htm
Contains everything that isn't put into <code>Foo.aspx.out.ssx.</code>

<code>Foo.aspx.out.htm</code> is then automatically fed back through the Preprocessor and run through the typical HTML Input File preprocessing steps, as shown here:
<table>
<tr><th>Input File Name</th><th>Output File Types and Names</th></tr>
<tr><td>Foo.aspx.out.htm</td><td>
Text
Filename: Foo.aspx.out.txt
<hr>
Comments
Filename: Foo.aspx.out.comments.txt
<hr>
HTML Tags
Filename: Foo.aspx.out.tags.txt
<hr>
Styles
Filenames: Foo.html.out.Internal_A.css,…
<hr>
JavaScript
Filenames: Foo.html.out.Internal_A.js, …
<hr>
VBScript
Filenames: - TBD I haven't tested this yet
<hr>
Pseudocode
Filename: Foo.aspx.out.c
<hr>
HTML with External Scripts and Style Blocks replaced with references
Filename: Foo.aspx.out.ExternalScripts.htm
</td></tr>
</table>
</td></tr>
</table>

Table 2. Possible Output Files for a Server-Side Input File

3 Setup

A setup program consisting of two files, `Setup.exe` and `HTML Preprocessor 1.1.0.msi`, installs version 1.1 of the HTML Preprocessor on a Windows computer. Separate setup programs are used to install the debug or release build of the program. By default, setup installs the program to this directory:

```
c:\Program Files (x86)\SAFE\HTML Preprocessor
```

A user can run the HTML Preprocessor windows application by clicking on the HTML Preprocessor item located in the SAFE Corporation program folder found under Windows Start / All Programs.

A user can run the HTML Preprocessor command line program by opening a command window and running `hpcl.exe` from the above installation directory.

4 Licensing

The HTML Preprocessor uses a software licensing scheme which disables the main features of the program unless it has been licensed by SAFE Corporation. If a user attempts to process files with an unlicensed copy of the HTML Preprocessor, the program will display an authorization warning dialog box rather than process the input files. The message in the dialog says "The program is not authorized. Contact SAFE Corporation for a license."

License checking is performed in the Release build of the HTML Preprocessor. If you wish to run a Release build of the program you must first license it by doing the following:

1. Run the HTML Preprocessor windows application. The main application window, shown in Figure 4, will come up even though the program is not licensed.

2. Click the License button on the lower left of the main application window.

3. The Authorization dialog, shown in Figure 2, will appear. Provide Bob Zeidman the site code from the Authorization dialog.

4. Get an authorization key from Bob Zeidman.

5. Enter the authorization key into the license dialog.

6. Click Authorize.

After completing the above steps you will be licensed to run the program an unlimited number of times. If you click the License button again, the License type field of the Authorization dialog will say "Unlimited".

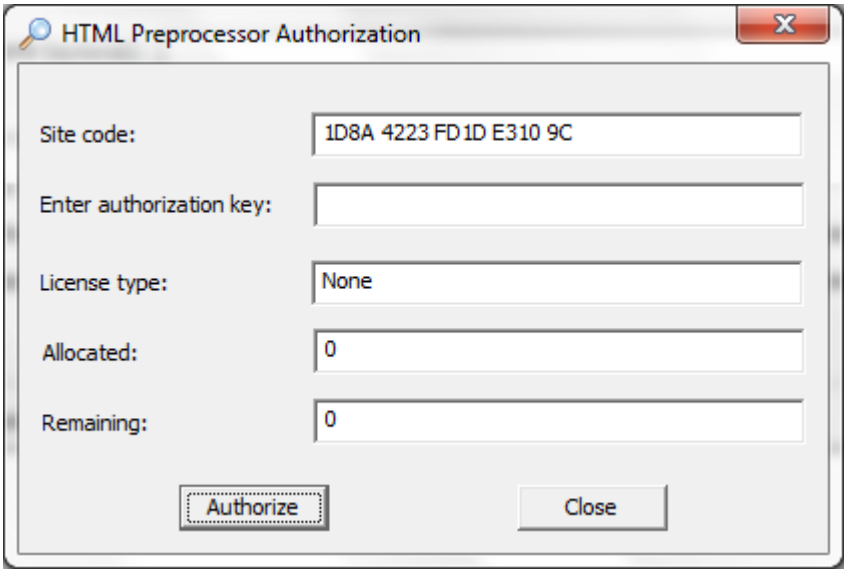


Figure 2. Authorization Dialog with a typical site code prior to authorization

5 The HTML Preprocessor Windows Application

5.1 Starting the HTML Preprocessor Windows Application

The user starts the HTML Preprocessor Windows Application from the SAFE Corporation program folder under the Windows Start menu / All Programs. The main application window shown in Figure 3 should then appear.

5.2 Specifying Options in the Main Application Window

The user enters the following information in the main application window before processing files:

- **Input Directory**
  The HTML Preprocessor gets its input files from the input directory. If *Parse files in subdirectories* is checked, it gets input files from a directory tree rooted at the input directory.

  The user must enter the full path to the input directory in the *Input directory* field. The user may browse to the input directory by clicking the adjacent *Browse...* button.

- **Parse Subdirectories**
  The *Parse files in subdirectories* checkbox is located under the *Input directory* field. When checked, the HTML Preprocessor gets its input files from the input directory and its subdirectories, and stores output files to an output directory tree that mirrors the input directory tree. If not checked, the Preprocessor gets its input files from the input directory only and writes output files to the output directory only. It is checked by default.

- **Output Directory**
  The HTML Preprocessor stores its output files in the output directory. If *Parse files in subdirectories* is checked, it stores output files in a directory tree rooted at the output directory.

  The user must enter the full path to the output directory in the *Output directory* field. The user may browse to the output directory by clicking the adjacent *Browse...* button.

- **Log File**
  The user may request the HTML Preprocessor to log information about each input file it processes. If a user would like a log file to be generated, the user must enter the full path of the log file in the field labeled *Log file name (optional)*. The user may browse to the desired log file by clicking the adjacent *Browse...* button.

  If the user specifies the name of an existing log file, when the user clicks the *Run* button the Preprocessor will first delete the old log file and then create the new log file at that location.

  If the user requested a log file, as the Preprocessor processes each input file it writes to the log file the full path of the file and any errors or warnings that occurred while processing that file. It also provides a summary at the end of the log file stating how many input files were processed and how many errors and warnings occurred. If the user tells the Preprocessor to stop processing part way through a large set of input files, the user determine which files were processed by looking at the log file.

- **File Pattern**

The *File pattern* field contains one or more filename pattern strings. If more than one filename pattern string is provided, each pattern string must be separated by a semicolon.

After the user clicks the *Run* button, the Preprocessor examines the name of each file in the input directory or input directory tree and will only processes the file if its name matches at least one of the file name patterns contained in the *File pattern* field.

By default, the File pattern field contains the string "*.htm*". This pattern would match, for example, files "Foo.htm" and "Bar.html" but would not match a file named "MyProgram.c". By contrast, the pattern string "MyProgram.*;*.htm" would match all three files.

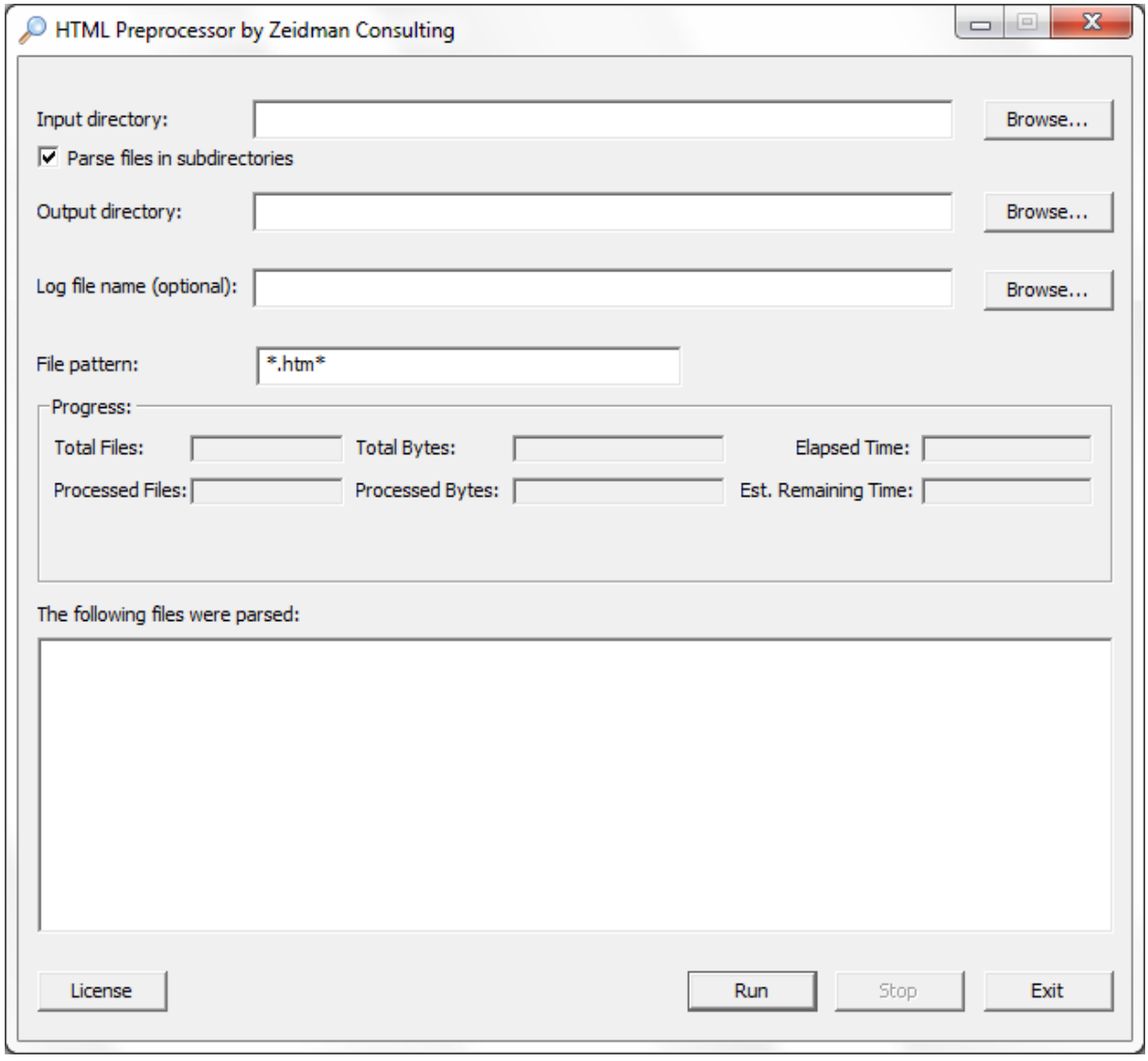


Figure 3. HTML Preprocessor, Main Application Window

5.3 Running the Preprocessor

Once the user has specified the desired options in the above fields, the user clicks the *Run* button at the lower right of the main window to start processing files. The Preprocessor will then calculate the total number of input files and total bytes in the input files and display that information in the Progress region

of the main window. It displays "Calculating..." in the Total Files and Total Bytes fields while calculating the totals.

The HTML Preprocessor then begins processing input files. As it processes files it periodically updates the following fields in the Progress region, as shown in Figure 4:

- *Progress Bar* - Indicates the percent of total input bytes that have been processed.
- *Processed Files* - Shows the number of files processed so far.
- *Processed Bytes* - Shows the number of bytes processed in *Processed Files*.
- *Elapsed Time* - Shows the amount of time that has elapsed since the user clicked the Run button. This is shown in terms of hours, minutes and seconds, displayed in hh:mm:ss format. If the program has been processing for more than one day, this field will show the number of days elapsed followed by hours, minutes and seconds.
- *Est. Remaining Time* - Shows the estimated time remaining until processing completes. This time value is formatted the same way as Elapsed Time.

Whenever the Preprocessor finishes processing all the files in a directory, it also writes a line to a list box that says the input directory's file path and how many files in it were processed.

5.4 Preprocessing Completed Notification

When the Preprocessor finishes processing all input files it displays a message box stating that processing has completed and writes a final line to the list box showing how many files were parsed.

5.5 Stopping the Preprocessor

While the Preprocessor is processing files the user may stop the preprocessor by clicking the *Stop* button in the lower right corner of the main window. When the user clicks *Stop* a confirmation dialog comes up that asks "Are you sure you want to stop processing files?". While the confirmation dialog is up the Preprocessor will continue to process files until the user either clicks the *Yes* button or it finishes processing all input files. If the user clicks the *No* button the confirmation dialog is dismissed and the Preprocessor continues processing files.

After the user clicks the Yes button, the Preprocessor will stop processing as soon as it finishes the input file it is currently working on. If the Preprocessor happens to be working on a large and complex input file, there may be a slight delay before it stops processing. Once the Preprocessor stops processing it displays a dialog box that says "File parsing stopped by user". The Progress area of the main window displays how many files and bytes were processed at the point when processing stopped.

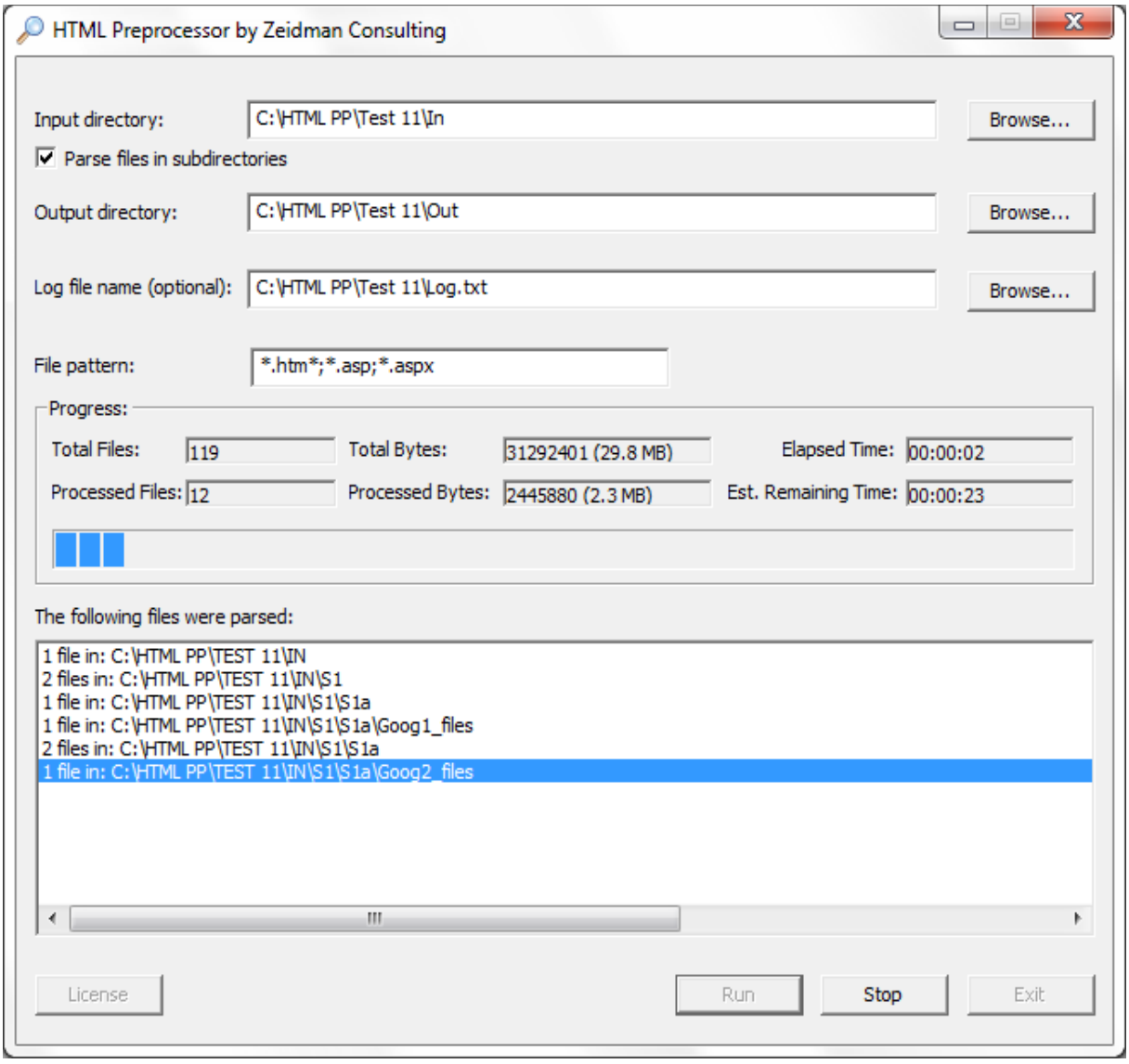


Figure 4. HTML Preprocessor, Processing Progress UI

6 The HTML Preprocessor Command Line Program

6.1 Capabilities of the HTML Preprocessor Command Line Program

A user can run the HTML Preprocessor as a console application rather than as a windows application. The console application is called the HTML Preprocessor Command Line program and its executable file is `hpcl.exe`**.** It is installed in the same directory as the windows application, typically `c:\Program Files (x86)\SAFE\HTML Preprocessor`.

Users can do almost everything through the command line program that they can through the windows application. However, users can only enter a license authorization key through the windows application.

Conversely, the Preprocessor command line program has the following capabilities that the windows application lacks, and which may make it a better choice for power users and developers:

- Like any command line program, it can be invoked from a batch file.
- The command line program can be passed the path of an input command file which contains multiple command lines, each of which causes the Preprocessor command line program to recursively invoke itself with the given command line.
- Finally, it is possible for another application to launch the command line program as a separate process and communicate with it through inter-process communication mechanisms. The HTML Preprocessor windows application is in fact implemented on top of the command line program by doing just that, as described in a section below.

6.2 Command Line Interface Documentation

The text below describes the parameters that can be passed to the command line program. This text is displayed on the command line when a user runs hpcl.exe with the help option, -h.

```
HTML Preprocessor
Version 1.1.0
(c) 2009-2011 SAFE Corporation

To run the HTML Preprocessor on one or more input files, type:

     hpcl [-option] InputDir Pattern OutputDir [LogFile]

where:
     InputDir:    Path or relative path name of the input directory.
                  Each file found in the input directory with a file name
                  that matches Pattern will be processed.
                  Corresponding output files will be written to OutputDir.

     Pattern:     Pattern of file names in InputDir to process.
                  Multiple semicolon-separated pattern strings
                  may be provided.

     OutputDir:   Path or relative path name of the output directory.
                  The Preprocessor will create the output directory
                  and subdirectories if they don't already exist.

     LogFile      Used to specify the path or relative path name
```

```
                    of an optional log file.
                    The HTML Preprocessor will write status messages
                    and error messages to the log file while it runs.
                    If LogFile already exists, it will be deleted and
                    a new log file created.
                    A log file is created only if LogFile is specified.

     If relative path names are provided, the current working directory
     will be used.

To run a series of HTML Preprocessor functions from a command file, type:

     hpcl CommandFile

where:
     CommandFile: Path to a command file that lists one or more
                  lines of arguments and options.
                  The hpcl program will be invoked once per line.

To see help on the HTML Preprocessor, type:

     hpcl -h

options:
     -h     Prints this help message.
     -LCn   Specifies how much status information should be logged to the
            console window
            n = 0  Does not write any status information to the console
                   window.
            n = 1  Writes only summary status information to the console
                   window, namely the total number of files processed and
                   the program's exit code. This is the default if no -LC
                   option is specified.
            n = 2  Writes all the status information to the console window
                   that would be written to a log file.
     -RSn   Recursively examine subdirectories of InputDir
            n = 0  No recursion: process only files in InputDir.
                   If RSn is not specified, it defaults to -RS0.
            n = 1  Process files in InputDir and, recursively,
                   its subdirectories.
                   Processes each input file in InputDir and its
                   subdirectories which has a file name that matches
                   Pattern, and saves its output file(s) to
                   OutputDir or a corresponding subdirectory of OutputDir.
     -NP0   ProgressPipeName
            If specified, the HTML Preprocessor will write progress
            messages to a Windows named pipe while processing input
            file(s). The name of the pipe, ProgressPipeName, must be
            provided as a parameter. The calling application is responsible
            for creating the named pipe. When -NP0 is specified, the HTML
            Preprocessor will write a progress message to the
            ProgressPipeName named pipe each time 1% of the total number of
            bytes in all input files has finished processing,
            and when all the files in a directory have finished processing.
            The calling application should consume each progress message;
            the HTML Preprocessor may suspend processing if the pipe
            is not drained.
```

```
            For progress message details, see the HTML Preprocessor
            design document.
      -NP1  StopMsgPipeName
            Used to specify that the HTML Preprocessor should monitor
            the pipe named StopMsgPipeName. If the string "STOP" appears
            on this pipe, the HTML Preprocessor will stop processing files
            and return the PROCESSING_STOPPED_BY_USER exit code.

Returns:
      Value Exit message written to      Indicates
            log file
      0     "SUCCESS"                     Program completed without errors.
      1     "INVALID_ARG"                 Invalid command line argument(s).
      2     "UNRECOVERABLE_ERROR"         Unrecoverable error occurred.
      3     "NOT_LICENSED"                Program could not run;
                                          it is not licensed.
      4     "LICENSING_ERROR"             Program could not run;
                                          couldn't check license.
      5     "PROCESSING_STOPPED_BY_USER"  Program stopped processing
                                          because "STOP" received on the
                                          StopMsgPipeName named pipe.

Usage Examples:
```

`hpcl c:\MyIputDir SomeFile.html c:\MyOutputDir`

```
      Processes one input file, c:\MyInputDir\SomeFile.html.
      Puts the corresponding output files in the c:\MyOutputDir
      directory.
      For example, a file named "SomeFile.html.out.c", which contains
      the "C" pseudo-code equivalent of the HTML in SomeFile.html,
      would be created in c:\MyOutputDir.
```

`hpcl  -RS1 c:\MyWebsite *.* c:\MyOutDir c:\MyLogfile.txt`
`      -NP0 HTMLPreProcUI_pipe -NP1 MyStopMsgPipe`

```
      Processes all files in the c:\MyWebsite directory and its
      subdirectories which have a filename extension of .htm or .asp.
      Writes output files to c:\MyOutdir and its subdirectories.
      Periodically writes a progress message to the
      "HTMLPreprocessorUI_pipe" named pipe while processing files.
      Stops processing if the calling program writes
      "STOP" to the "MyStopMsgPipe" named pipe.
      Writes a message about each input file it processes to
      c:\MyLogfile.txt, or, if an error was encountered,
      writes a message describing the error.
```

7 Related Specifications

7.1 HTML Standard Specifications

Worldwide Web Consortium (W3C) HTML specifications:

W3C HTML 2.0	http://www.w3.org/MarkUp/html-spec
W3C HTML 3.2	http://www.w3.org/TR/REC-html32
W3C HTML 4.0	http://www.w3.org/TR/REC-html40
W3C HTML 4.01	http://www.w3.org/TR/html401
W3C XHTML	http://www.w3.org/TR/xhtml1
W3C HTML5	http://dev.w3.org/html5/spec/Overview.html

Web Hypertext Application Technology Working Group (WAHTWG) specifications:

HTML5 Living Standard:	http://www.whatwg.org/specs/web-apps/current-work/multipage/

7.2 CSS Specifications

Links to specifications for Cascading Style Sheets 1 and 2, as well as the working draft for CSS 3, can be found on the W3C CSS Home Page:

CSS Home Page:	http://www.w3.org/Style/CSS/

7.3 ASP.NET Web Page Overview

The following page provides an overview of the components of ASP.NET web pages:

http://msdn.microsoft.com/en-us/library/428509ah.aspx

7.4 JSP Specifications

The following page has links to documentation on the structure of Java Server Pages:

http://java.sun.com/products/jsp/reference/api/index.html

7.5 PHP Specifications

The PHP Manual can be found here:

http://www.php.net/manual/en/index.php

8 Contacting SAFE Corporation

Software Analysis and Forensic Engineering Corporation

Web: www.SAFE-corp.com

Email: Support@SAFE-corp.com

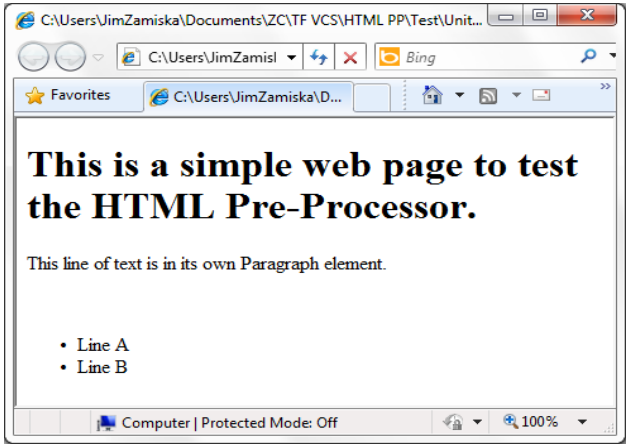
Appendix A: Examples of Input Files and Output Files

1. A Simple HTML Input File

In this example, the input file Foo.html contains:

- One line of text in an HTML header
- One line of text in a HTML paragraph tag
- Two HTML list elements

The screen shot at the right shows how Foo.html would look when opened in Internet Explorer 8.



Input File: Foo.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
</head>
<body>
    <h1>
        This is a simple web page to test the HTML Pre-Processor.</h1>
    <p>
        This line of text is in its own Paragraph element.</p>
    <p>
         </p>
    <ul>
        <li>Line A</li>
        <li>Line B</li>
    </ul>
</body>
```

Output Files: 5 are generated

Foo.html.out.ExternalScripts.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
</head>
<body>
    <h1>
        This is a simple web page to test the HTML Pre-Processor.</h1>
    <p>
        This line of text is in its own Paragraph element.</p>
    <p>
         </p>
    <ul>
        <li>Line A</li>
        <li>Line B</li>
    </ul>
</body>
```

```
</html>
```

```
</html>
```

Foo.html.out.txt

```
This is a simple web page to test the HTML Pre-Processor.
This line of text is in its own Paragraph element.
 
Line A
Line B
```

Foo.html.out.comments.txt

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Foo.html.out.tags.txt

```
html
head
title/
/head
body
h1 /h1
p /p
p /p
ul
li /li
li /li
/ul
/body
/html
```

Foo.html.out.c

```
void title()
{
}

void head()
{
          title();
}

void h1()
{
          char* value = "
       This is a simple web page to test the HTML Pre-Processor.";
}
```

```
void p()
{
          char* value = "
        This line of text is in its own Paragraph element.";
}

void p_1()
{
          char* value = "
         ";
}

void li()
{
          char* value = "Line A";
}

void li_1()
{
          char* value = "Line B";
}

void ul()
{
          li();
          li_1();
}

void body()
{
          h1();
          p();
          p_1();
          ul();
}

void html()
{
         char* xmlns =  "http://www.w3.org/1999/xhtml";
          head();
          body();
}

int main(int argc, char* argv[])
{
/* <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> */
          html();
}
```

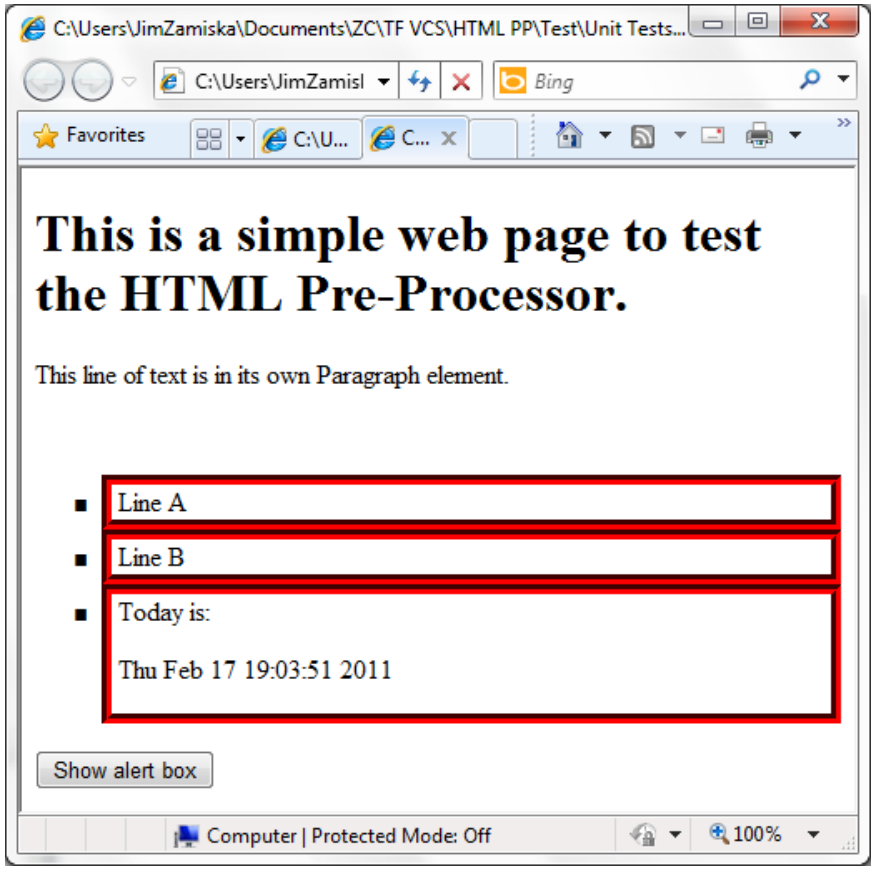
2. An HTML Input File with Comments, Styles and JavaScript

In this example, Foo.html contains:

- Two Cascading Style Sheet style definitions
- One line of body text in an HTML header
- One line of body text in an HTML paragraph tag
- Two bullet lines that use the two styles
- Two comment lines
- Two JavaScript function definitions and HTML elements that call them

Also, Foo.html has some of its <h1>, <p> and <li> tags on the same line.

The screen shot at the right shows how Foo.html would look when opened in Internet Explorer 8.



Input File: Foo.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

 <!-- This is comment 1 -->
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <!-- This is comment 2 -->
    <style type="text/css">
        .style1
        {
            list-style-type: square;
        }
        .style2
        {
            border: 6px groove #FF0000;
            padding: 1px 4px;
        }
    </style>
    <script type="text/javascript">
        function show_alert() {
            alert("Hello! I am an alert box!");
        }
    </script>
</head>
<body>
    <h1>
This is a simple web page to test the HTML Pre-Processor.</h1><p>This line
of text is in its own Paragraph element.</p>
    <p>
         </p>
    <ul class="style1">
        <li  class="style2">Line  A</li><li  class="style2">Line  B</li><li
class="style2">Today is:
            <script type="text/javascript">
                document.write("<p>" + Date() + "</p>");
            </script>
        </li>
    </ul>

    <input type="button" onclick="show_alert()" value="Show alert box" />
</body>
</html>
```

Output Files: 8 Output Files Are Generated

Foo.html.out.Internal_A.css

```
        .style1
        {
            list-style-type: square;
        }
        .style2
        {
            border: 6px groove #FF0000;
            padding: 1px 4px;
        }
```

Foo.html.out.Internal_A.js

```
        function show_alert() {
            alert("Hello! I am an alert box!");
        }
```

Foo.html.out.Internal_B.js

```
                document.write("<p>" + Date() + "</p>");
```

Foo.html.out.comments.txt

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- This is comment 1 -->
<!-- This is comment 2 -->
```

Foo.html.out.txt

```
This is a simple web page to test the HTML Pre-Processor.
This line of text is in its own Paragraph element.
 
Line A
Line B
Today is:
```

Foo.html.out.ExternalScripts.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

 <!-- This is comment 1 -->
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <!-- This is comment 2 -->
    <link                type="text/css"                rel="stylesheet"
href="file://~/Foo.html.out.Internal_A.css">
    <script                                        type="text/javascript"
src="file://~/Foo.html.out.Internal_A.js"></script>
</head>
<body>
    <h1>
This is a simple web page to test the HTML Pre-Processor.</h1><p>This line
of text is in its own Paragraph element.</p>
    <p>
         </p>
    <ul class="style1">
        <li  class="style2">Line  A</li><li  class="style2">Line  B</li><li
class="style2">Today is:
            <script                                        type="text/javascript"
src="file://~/Foo.html.out.Internal_B.js"></script>
        </li>
    </ul>

    <input type="button" onclick="show_alert()" value="Show alert box">
</body>
</html>
```

Foo.html.out.tags.txt

```
html
head
title/
style
/style
script
/script
/head
body
h1 /h1
p /p
p /p
ul
li /li li /li li
script
/script
/li
/ul
input/
/body
/html
```

Foo.html.out.c

```
void title()
{
}

void link()
{
        char* type =  "text/css";
        char* rel =  "stylesheet";
        char* href =  "file://~/Foo.html.out.Internal_A.css";
}

void script()
{
        char* type =  "text/javascript";
        char* src =  "file://~/Foo.html.out.Internal_A.js";
}

void head()
{
         title();
/* <!-- This is comment 2 --> */
         link();
         script();
```

```
}

void h1()
{
         char* value = "
This is a simple web page to test the HTML Pre-Processor.";
}

void p()
{
         char*  value  =  "This  line  of  text  is  in  its  own  Paragraph
element.";
}

void p_1()
{
         char* value = "
         ";
}

void li()
{
        char* class =  "style2";
         char* value = "Line A";
}

void li_1()
{
        char* class =  "style2";
         char* value = "Line B";
}

void script_1()
{
        char* type =  "text/javascript";
        char* src =  "file://~/Foo.html.out.Internal_B.js";
}

void li_2()
{
        char* class =  "style2";
         char* value = "Today is:
           ";
         script_1();
}

void ul()
{
```

```
        char* class =  "style1";
         li();
         li_1();
         li_2();
}

void input()
{
        char* type =  "button";
        char* onclick =  "show_alert()";
        char* value =  "Show alert box";
}

void body()
{
         h1();
         p();
         p_1();
         ul();
         input();
}

void html()
{
        char* xmlns =  "http://www.w3.org/1999/xhtml";
         head();
         body();
}

int main(int argc, char* argv[])
{
/* <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> */
/* <!-- This is comment 1 --> */
         html();
}
```

3. A PHP Input File containing HTML and Server-Side Script

In this example, MiniChat.php is a PHP file that that implement a chat window. It contains both HTML and server-side PHP script.

This example demonstrates how the Preprocessor splits a server-side file into its HTML and server-side components. The following tables show the MiniChat.php source file and the output files the Preprocessor generates.

Although the `miniChat()` PHP function contains code that emits HTML, that function runs on the server. The top level HTML starts almost at the end of MiniChat.php and is highlighted in green to make it easier to spot; the Preprocessor identifies that HTML as client-side code.

This sample PHP code comes from the Happy Codings website: http://www.php.happycodings.com/Other/code27.html.

Input File: MiniChat.php

```
<?php
// Contains db constants
require "cst.inc";

/*
CREATE TABLE MINICHAT (
  NB tinyint(4) NOT NULL auto_increment,
  LOGIN varchar(20) NOT NULL default '',
  MESSAGE varchar(255) NOT NULL default '',
  ITSTIME varchar(10) NOT NULL default '',
) TYPE=MyISAM;
*/

// Number of messages to print
define( "_NB_MSGS_", 10 );

// Connect
mysql_connect( $db_hostname, $db_username, $db_password );
mysql_selectdb( $db_database );

// Add message into minichat
function addMessage( $login, $message ) {

   @setcookie( "minichatlogin", strip_tags( $login ) );

   $login = $_COOKIE['minichatlogin'] ? $_COOKIE['minichatlogin'] :
      mysql_escape_string( strip_tags( $login ) );
   $message = mysql_escape_string( strip_tags( $message, '<a><b><i><u>') );
   mysql_query( "INSERT INTO MINICHAT ( NB, ITSTIME, LOGIN, MESSAGE )
VALUES ( "
   ._NB_MSGS_.", '".
```

Output Files: 6 Output Files Are Generated

MiniChat.php.out.ssx

```
<?php
// Contains db constants
require "cst.inc";

/*
CREATE TABLE MINICHAT (
  NB tinyint(4) NOT NULL auto_increment,
  LOGIN varchar(20) NOT NULL default '',
  MESSAGE varchar(255) NOT NULL default '',
  ITSTIME varchar(10) NOT NULL default '',
) TYPE=MyISAM;
*/

// Number of messages to print
define( "_NB_MSGS_", 10 );

// Connect
mysql_connect( $db_hostname, $db_username, $db_password );
mysql_selectdb( $db_database );

// Add message into minichat
function addMessage( $login, $message ) {

   @setcookie( "minichatlogin", strip_tags( $login ) );

   $login = $_COOKIE['minichatlogin'] ? $_COOKIE['minichatlogin'] :
      mysql_escape_string( strip_tags( $login ) );
   $message = mysql_escape_string( strip_tags( $message, '<a><b><i><u>') );
   mysql_query( "INSERT INTO MINICHAT ( NB, ITSTIME, LOGIN, MESSAGE ) VALUES
( "
   ._NB_MSGS_.", '".
```

```
     time()."', '".$login."','".$message."' )" );
   mysql_query( "UPDATE MINICHAT SET NB=NB-1" );
   mysql_query( "DELETE FROM MINICHAT WHERE NB < 1" );
}

// Returns messages
function getMessages() {
   $rs = mysql_query( "SELECT * FROM MINICHAT ORDER BY NB" );

   $ret = Array();
   while ( $msg = mysql_fetch_array( $rs )) {
      $ret[] = date( 'h:m', $msg['ITSTIME'] )." ".$msg['LOGIN']."
>".$msg['MESSAGE'];
   }

   return $ret;
}

// Prints mini chat
function miniChat() {

   $msgs = getMessages();
   @reset( $msgs );

   echo '<form method="post">
         <table border="0" bgcolor="#000000" cellpadding="1">
         <tr><td>
         <table border="0" bgcolor="#ffffff" cellpadding="1">
   ';

   while ( list(,$msg) = each( $msgs ))
      echo "<tr><td>$msg</td></tr>";

   if ( !$_COOKIE['minichatlogin'] ) {
      if ( !$_POST['login'] )
         echo '<tr><td>Login:<input type="text" name="login"
size="6"></td></tr>';
      else
         echo '<input type="hidden" name="login"
value="'.$_POST['login'].'">';
   }
   echo '
   <tr><td><input type="text" name="msg" size="10"></td></tr>
   <tr><td align="center"><input type="submit" value="Send"></td></tr>
   </table></tr></td></table></form>';
}
```

```
     time()."', '".$login."','".$message."' )" );
   mysql_query( "UPDATE MINICHAT SET NB=NB-1" );
   mysql_query( "DELETE FROM MINICHAT WHERE NB < 1" );
}

// Returns messages
function getMessages() {
   $rs = mysql_query( "SELECT * FROM MINICHAT ORDER BY NB" );

   $ret = Array();
   while ( $msg = mysql_fetch_array( $rs )) {
      $ret[] = date( 'h:m', $msg['ITSTIME'] )." ".$msg['LOGIN']."
>".$msg['MESSAGE'];
   }

   return $ret;
}

// Prints mini chat
function miniChat() {

   $msgs = getMessages();
   @reset( $msgs );

   echo '<form method="post">
         <table border="0" bgcolor="#000000" cellpadding="1">
         <tr><td>
         <table border="0" bgcolor="#ffffff" cellpadding="1">
   ';

   while ( list(,$msg) = each( $msgs ))
      echo "<tr><td>$msg</td></tr>";

   if ( !$_COOKIE['minichatlogin'] ) {
      if ( !$_POST['login'] )
         echo '<tr><td>Login:<input type="text" name="login"
size="6"></td></tr>';
      else
         echo '<input type="hidden" name="login"
value="'.$_POST['login'].'">';
   }
   echo '
   <tr><td><input type="text" name="msg" size="10"></td></tr>
   <tr><td align="center"><input type="submit" value="Send"></td></tr>
   </table></tr></td></table></form>';
}
```

```
// Message Posted ?
if ( isset( $_POST['msg'] )) {
   addMessage( $_POST['login'], $_POST['msg'] );
}
?>
<html>
<head>
<title>Mini Chat Sample</title>
</head>
<body>
<center>
<?php miniChat(); ?>
<hr>
</center>
</body>
</html>
```

```
// Message Posted ?
if ( isset( $_POST['msg'] )) {
   addMessage( $_POST['login'], $_POST['msg'] );
}
?>
<?php miniChat(); ?>
```

MiniChat.php.out.htm

```
<html>
<head>
<title>Mini Chat Sample</title>
</head>
<body>
<center>

<hr>
</center>
</body>
</html>
```

MiniChat.aspx.out.ExternalScripts.htm

```
<html>
<head>
<title>Mini Chat Sample</title>
</head>
<body>
<center>

<hr>
</center>
</body>
</html
```

MiniChat.aspx.out.tags.txt

```
html
head
title /title
/head
body
center
hr/
/center
/body
/html
```

MiniChat.aspx.out.txt

```
Mini Chat Sample
```

MiniChat.aspx.out.c

```c
void title()
{
	char* value = "Mini Chat Sample";
}

void head()
{
	title();
}

void hr()
{
}

void center()
{
	hr();
}

void body()
{
	center();
}

void html()
{
	head();
	body();
}

int main(int argc, char* argv[])
{
	html();
}
```