



Simplifying and Optimizing HTML Construction

By: David Adams

Introduction

Recently it has become commonplace to use 4D to create HyperText Markup Language ("HTML") documents. HTML is the structured document tagging language of the World Wide Web. This technical note illustrates a technique that simplifies HTML construction and maintenance, and optimizes document delivery. This technique is suitable for any of the following settings:

- Constructing static HTML pages for later access by a Web server
- Creating HTML on the fly using 4D as an external processor (CGI) to a Web server
- Using 4D as a Web server

Background

HTML is a verbose language. This makes it superficially easy to read, but a burden to manage. Here is the code for a simple HTML document:

```
<HTML><HEAD><TITLE>Hello world!</TITLE></HEAD>
<BODY><H1>The body text appears here.</H1></BODY>
</HTML>
```

An obvious way to construct HTML pages is to concatenate complete HTML pages in-line, like this:

```
C_TEXT ($tHTMLBlock)
$tHTMLBlock := "<HTML><HEAD><TITLE>Hello world!</TITLE></HEAD>"
$tHTMLBlock := $tHTMLBlock + "<BODY><H1>The body text appears
here.</H1></BODY>"
$tHTMLBlock := $tHTMLBlock + "</HTML>"
```

The disadvantages of this practice should be obvious:

- Constructing code in this way is tedious, awkward, and error prone.
- You need to recompile your database to change the HTML output.
- You will need to rewrite your procedures to support any other tagging language, such as QuarkXPress tags, or FrameMaker Interchange Format.
- Your HTML becomes rigid because it is cumbersome to edit.
- The end user is given no control over the HTML output.

TECHNICAL NOTE 96-

March 19



- The entire text block is reconstructed every time it is used, which is slow.
- It becomes difficult to test your HTML.
- Changing your HTML blocks globally requires editing many 4D procedures.
- Localization of your project will be difficult.

You can eliminate all of these problems with a single file and a few procedures. You will save the time it takes you to implement this solution in a single morning's work constructing HTML.

Keep the Data in Records

4D records are the logical place to store repeatedly used HTML blocks:

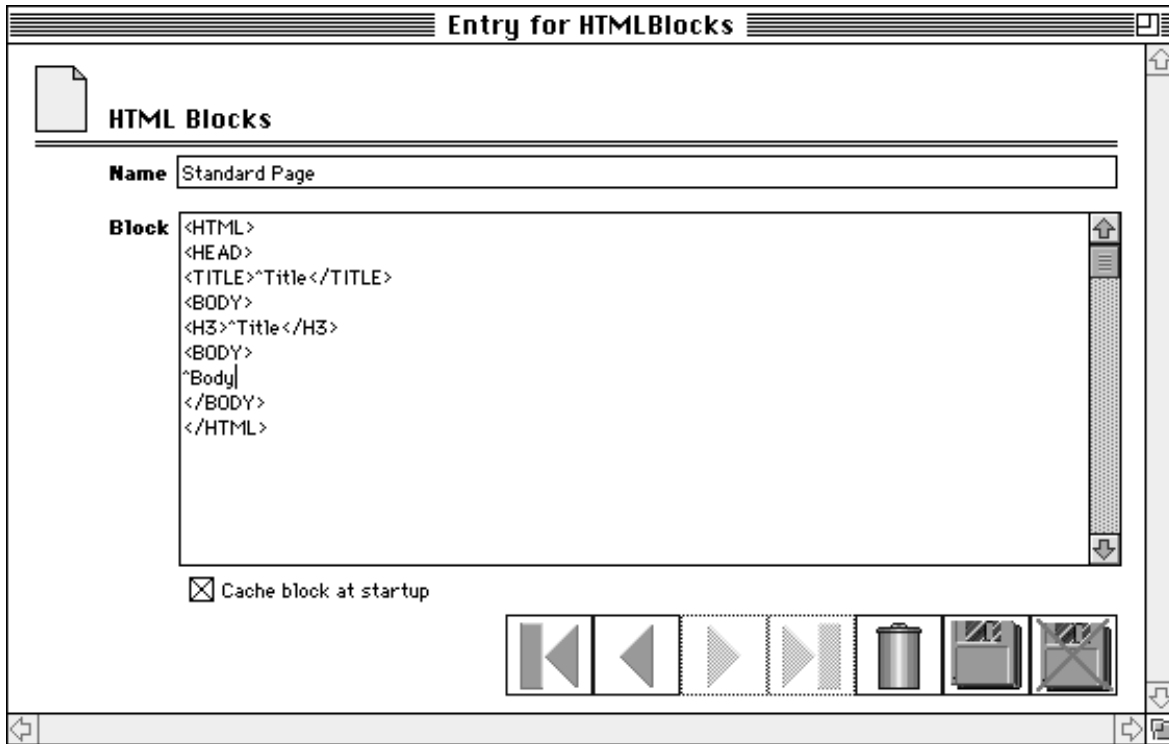
HTMLBlocks	
BlockName	A
HTMLBlock	T
AutoLoad	B

The file structure

You can store your HTML blocks in a text field for easy editing. You can then construct your HTML in an HTML editor, test it in a Web browser and then paste it into 4D once you are satisfied with the results.

TECHNICAL NOTE 96-

March 19



HTML for a standard page stored in a 4D record

As you can see in the sample text pictured above there are some strings embedded in the HTML that start with “^”, for example ^Title and ^Body. These strings stand for dynamic values that you insert when the final page is produced. The “^” symbol is arbitrary, you can use any character or string that allows you to target your substitution strings accurately.

```
C_TEXT ($tHTMLBlock)
$tHTMLBlock := HTMLBlock ("Standard Page")
$tHTMLBlock := Replace string ($tHTMLBlock; "^Title"; "My title")
$tHTMLBlock := Replace string ($tHTMLBlock; "^Body"; "Body text")
```

Here is the output:

```
<HTML>
<HEAD>
<TITLE>My title</TITLE>
<BODY>
<H3>My title</H3>
<BODY>
Body text
</BODY>
</HTML>
```



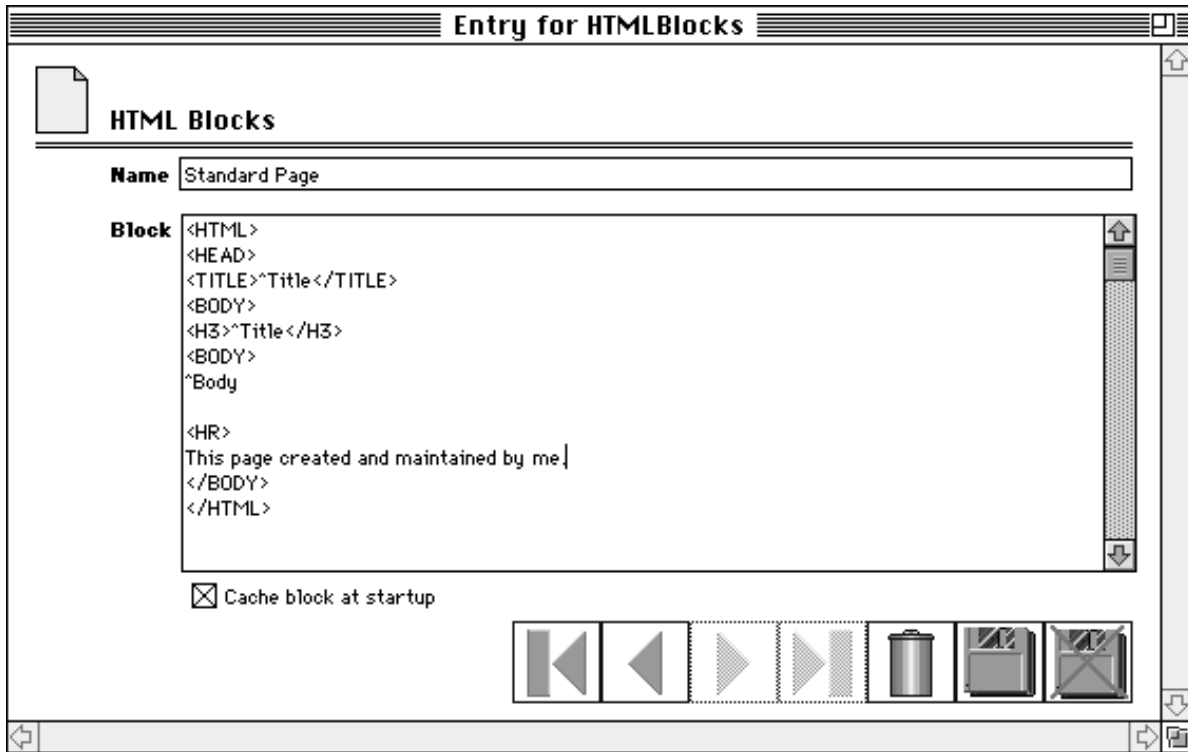
A few items to note about Replace string:

- Replace string is fast. It is faster to replace a few strings in a text block than to construct the entire block from scratch.
- By default, Replace string replaces multiple instances of the same string in the source block. For example, ^Title appears twice in the source block and is changed in both cases with a single call to Replace string. You can control this behavior with the Replace string optional fourth parameter.

In this example the replacement values are hard coded into the 4D code. In actual practice you can supply values from:

- User input
- Another [HTMLBlocks] record
- The results of any number of calculations
- A callback to 4D like String (Current date)
- A disk file
- Any 4D record in your system
- The results of an SQL query
- A variable
- Values provided by the client as a part of a CGI request or HTTP session

Abstracting the code and your data greatly simplifies system maintenance, and eases experimentation. If you want your standard page to include a credit statement at the bottom you need only change a single record, rather than changing many 4D procedures:



The updated HTML for a standard page in a 4D record

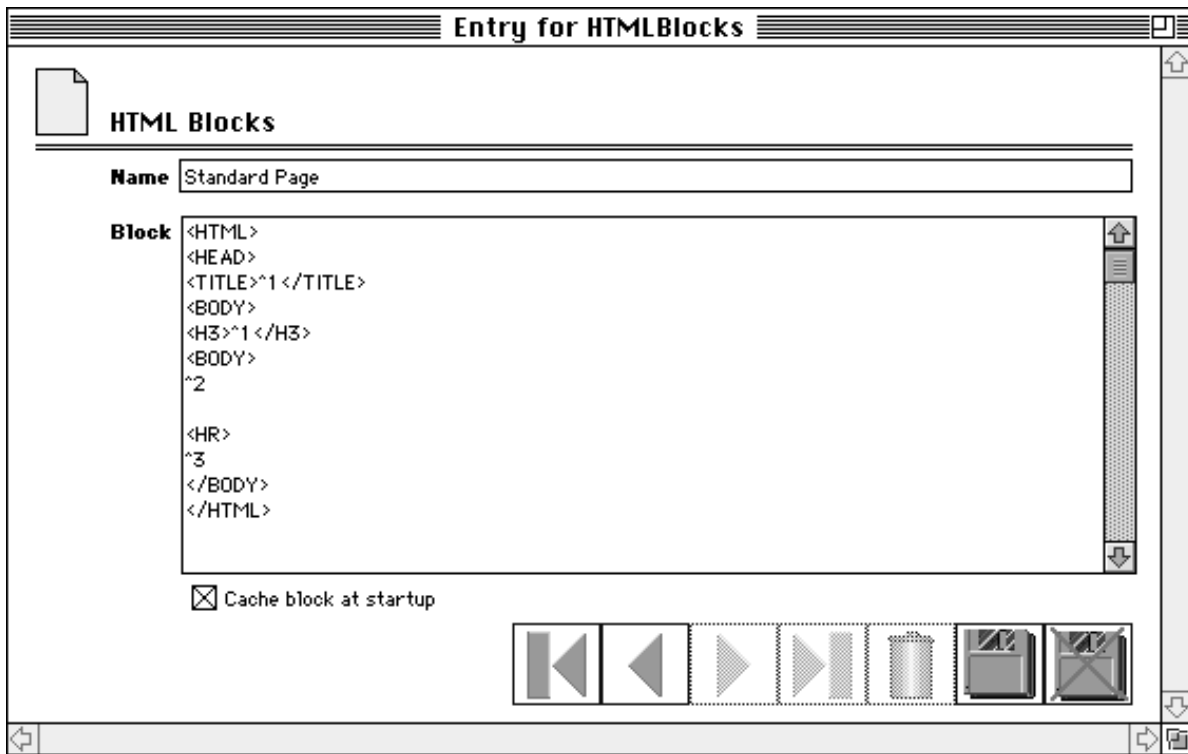
The new text “<HR>This page created and maintained by me.” now appears on each page that uses the Standard Page HTML block without your having to change a single line of 4D code.

Generalized Insertion

In the examples shown so far, Replace string has sought strings with names known to your code, like ^Title. You can create a more generic system with a simple routine that uses parameter indirection to replace any number of strings. Examine the block shown here:

TECHNICAL NOTE 96-

March 19



HTML for a standard page with numbered insertion targets

You can see that this block uses numbered insertion targets instead of named insertion targets. This makes generalized insertion simple to perform. Here is the code of the complete HTMLBlock procedure:

TECHNICAL NOTE 96-

March 19



```
If (False)
  \ Procedure:   HTMLBlock (Block name; Insertion 1; ...; {Insertion n}) ->
Block
  \ By:         David Adams
  \ Created:    02/96
  \ Purpose:    Abstracts HTML coding.
  \ Note:       \<asHTML_Name and \<atHTML_Text initialized by
InitHTMLBlocks at
  \
                startup.
End if

  \ Declare parameters
C_TEXT ($0) \      HTML block, or empty string if block does not exist
C_STRING (80; $1) \      Block name

  \ Parameter indirection in declaration supports n parameters of same type.
C_TEXT (${2}) \      Insertion block 1 through n passed in $2 - $n

  \ Declare local variables
C_LONGINT ($LCurrParm) \      Input parameter being processed
C_LONGINT ($LElement) \      Position of target block in block arrays
C_LONGINT ($LMaxParms) \      Number of insertion strings passed
C_STRING (80; $sBlockName) \      Block name
C_TEXT ($tResultText) \      Completed block

  \ Input parameter reassignment
$sBlockName := $1

$tResultText := ""
$LElement := Find in array (\<asHTML_Name; $sBlockName)

If ($LElement > 0) \      Is the block already cached?
  $tResultText := \<atHTML_Text{$LElement}
Else \      Not cached, search file
  SEARCH ([HTMLBlocks]; [HTMLBlocks]BlockName = $sBlockName)
  $tResultText := [HTMLBlocks]HTMLBlock \ Empty string if no [HTMLBlocks]
found.
End if
```



```

If ($tResultText # "") ` No need to substitute on an empty
string.
    $LMaxParms := Count parameters - 1 ` How many insert parameters were
    passed?

    For ($LCurrParm; 1; $LMaxParms)
        $tResultText := Replace string ($tResultText; "^" + String ($LCurrParm);
        {$LCurrParm + 1})
    End for

End if ` ($tResultText # "")

$0 := $tResultText

` End of procedure
    
```

A call to this procedure looks like this:

```

C_TEXT ($tResultText)
$tResultText := HTMLBlock ("Standard page"; "Hello world!"; "Lots-o-data."; "Page
created: "
    + String (Current date))
    
```

With output like this:

```

<HTML>
<HEAD>
<TITLE>Hello world!</TITLE>
<BODY>
<H3>Hello world!</H3>
<BODY>
Lots-o-data.

<HR>
Page created: 2/26/96
</BODY>
</HTML>
    
```

You can nest calls to HTMLBlock within the limits of the procedure stacks size:

```

C_TEXT ($tResultText)
$tResultText := HTMLBlock ("Standard page"; "Hello world!"; "Lots-o-data.";
HTMLBlock
    ("Page Stamp"; String (Current date); String (Current time); Current user))
    
```

The preceding statement produces this output:



```
<HTML>
<HEAD>
<TITLE>Hello world!</TITLE>
<BODY>
<H3>Hello world!</H3>
<BODY>
Lots-o-data.

<HR>
<!-- This page produced on 2/26/96 at 15:25:04 by Designer. -->
</BODY>
</HTML>
```

About the HTMLBlock Procedure

The HTMLBlock procedure takes advantage of parameter indirection. 4D allows you to pass to a procedure an unknown number of parameters of the same type at the end of the parameter list. Count parameters determines how many parameters have been passed, and each is referenced using the syntax for parameter indirection:

`$(parameter number)`

This phrase is commonly referred to as the generic parameter. Thus, `${1}` refers to parameter 1, and so forth. Since the number can be contained in a variable, this allows you to run through the parameter list in a loop, which is what is happening in the HTMLBlock procedure.

Notice that the HTMLBlock procedure first looks for the target block in a pair of arrays populated at startup by the InitHTMLBlocks procedure. The advantage of loading blocks into arrays is that subsequent access is then considerably faster than finding them on disk. Use these arrays as a repository for all of your repeatedly used HTML blocks. The [HTMLBlocks]AutoLoad field setting determines if a record is loaded into arrays at startup or not. Here is the code for the InitHTMLBlocks routine:

TECHNICAL NOTE 96-

March 19



```
If (False)
  ` Procedure:  InitHTMLBlocks ()
  ` By:        David Adams
  ` Created:   02/96
  ` Used by:   HTMLBlock function
  ` Purpose:   This routine is run at startup to load frequently needed
               HTML blocks into arrays.
End if
```

```
READ ONLY ([HTMLBlocks]) `           This file should be in a read only
state.
```

```
ARRAY STRING (80; ⋄asHTML_Name; 0) ` [HTMLBlocks]BlockName
ARRAY TEXT (⋄atHTML_Text; 0) `       [HTMLBlocks]HTMLBlock
```

```
SEARCH ([HTMLBlocks]; [HTMLBlocks]AutoLoad = True)
SELECTION TO ARRAY ([HTMLBlocks]BlockName; ⋄asHTML_Name;
[HTMLBlocks]HTMLBlock; ⋄atHTML_Text)
```

```
` End of procedure
```

Summary

This technote has described a simple record structure to store blocks of HTML code, and a flexible routine for replacing dynamic elements within stored blocks. This technique is easy to implement, and immediately simplifies HTML coding and code maintenance.