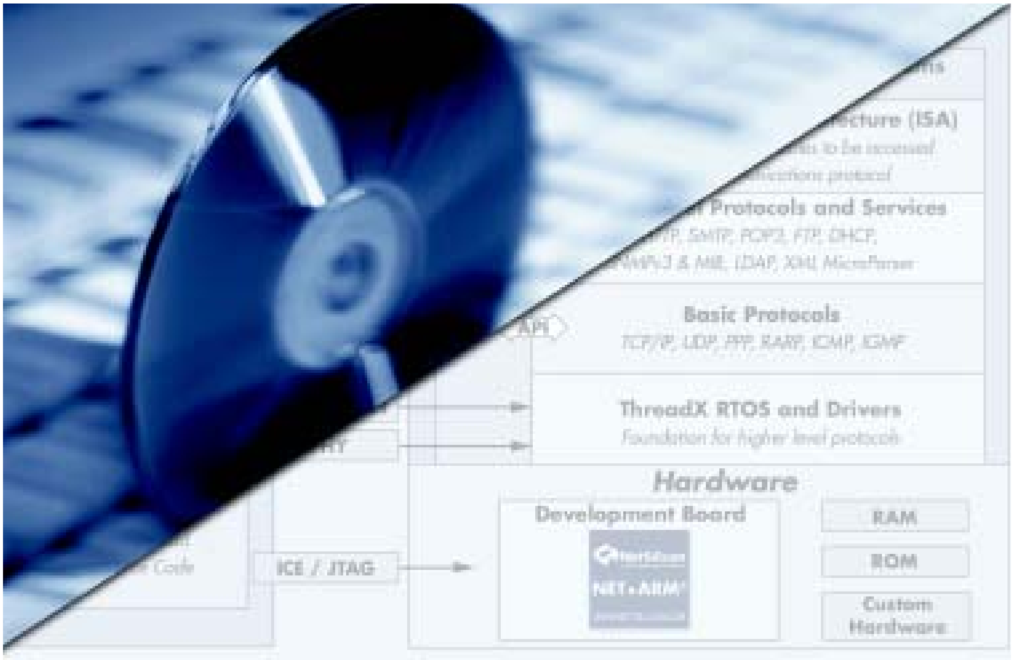
NetSilicon
A Digi International® Company

Advanced Web Server Toolkit



NET+OS 5.0
8833241D

Advanced Web Server Toolkit

Operating system/version: NET+OS 5.0
Part number/version: 8833241D
Release date: July 2002
www.netsilicon.com

©2001-2002 NetSilicon, Inc.

Printed in the United States of America. All rights reserved.

NetSilicon, NET+Works, and NET+OS are trademarks of NetSilicon, Inc. ARM Is a registered trademark of ARM limited. NET+ARM is a registered trademark of ARM limited and is exclusively sublicensed to NetSilicon. Digi and Digi International are trademarks or registered trademarks of Digi International Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

NetSilicon makes no representations or warranties regarding the contents of this document. Information in this document is subject to change without notice and does not represent a commitment on the part of NetSilicon. This document is protected by United States copyright law, and may not be copied, reproduced, transmitted, or distributed in whole or in part, without the express prior written permission of NetSilicon. No title to or ownership of the products described in this document or any of its parts, including patents, copyrights, and trade secrets, is transferred to customers. NetSilicon reserves the right to make changes to products without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

NETSILICON PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES, OR SYSTEMS, OR OTHER CRITICAL APPLICATIONS.

NetSilicon assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does NetSilicon warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of NetSilicon covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

Using This Guide

Review this section for basic information about the guide you are using, as well as general support and contact information.

About this guide

This guide describes the NET+OS Web Application Toolkit for the Advanced Web Server. It is based on the *RomPager Advanced Web Server Programming Reference*, Version 4.00, from Allegro Software Development Corporation. NET+OS, a network software suite optimized for the NET+ARM chip, is part of the NET+Works integrated product family.

Conventions used in this guide

Convention	Description
Italic type	Used for emphasis and for book and manual titles, and reference documents. In format statements and some command line examples, variables are also show in italics.
Bold type	Indicates specific choices within instructions in procedures. For example, Click on **Software Installation**.
`<TAG>`	Source code, including HTML tags, are shown in this font. Note that angle brackets are part of the tag name and do not indicate a placeholder.
`Monospace`	Indicates filenames, pathnames, commands, and so on.

Related documentation

- *NET+OS Getting Started Guide* explains how to install NET+OS with Green Hills or with GNU tools, and how to build your first application.
- *NET+OS User's Guide* describes how to use NET+OS to develop programs for your application and hardware.
- *NET+OS BSP Porting Guide* describes how to port the board support package (BSP) to a new hardware application, with either Green Hills Software or GNU tools.
- *NET+OS Application Software Reference Guide* describes the NET+OS software application programming interfaces (APIs).
- *NET+OS BSP Software Reference Guide* describes the board support package APIs.
- *NET+OS Kernel User's Guide* describes the real-time NET+OS kernel services.
- Review the documentation CD-ROM that came with your development kit for information on third-party products and other components.
- Refer to the NET+Works hardware documentation for information appropriate to the chip you are using.

Customer support

To get help with a question or technical problem with this product, or to make comments and recommendations about our products or documentation, use the contact information listed in this table:

Technical support	**Telephone:** (800) 243–2333 or (781) 647–1234 **Fax:** (781) 893–1338 **E-mail:** *tech_support@netsilicon.com*
Documentation comments	*techpubs@netsilicon.com*
NetSilicon home page	*www.netsilicon.com*
Online problem reporting	*www.netsilicon.com/EmbWeb/Support/forms/bugreport.asp* An engineer will analyze the information you provide and call you about the problem.

Development Process Overview

The process of building Web pages for an embedded device running the Advanced Web Server is as follows:

1. Design the Web pages for the device.
2. Write the HTML code for these Web pages.
3. Identify the variables to be accessed.
4. Define the HTML links to the variables.
5. Generate C source code from the HTML using the PBuilder compiler.
6. Flesh out the C source code for variable access.
7. Compile and link the C source code for the target device.

Step 1. Design the Web pages

The first step in developing Web pages for an embedded device is laying out the actual Web pages. What will the pages look like? Will a page have forms for data entry or just be a display page? What user interface elements will be employed? Which parts will be static and which will be dynamic? The dynamic components are particularly important since they need to be interfaced to the device.

Page design is beyond the scope of this manual, but an excellent reference book is *Teach Yourself Web Publishing with HTML in 14 Days* by Laura Lemay (SamsNet). It covers page design and HTML coding in an easy-to-read style, and includes a CD-ROM with helpful utilities.

Step 2. Write the HTML code

You can use any text editor to create HTML pages, because the storage format is a standard text file. Tools such as Adobe PageMill and Microsoft FrontPage can speed up the page creation process a great deal, since they provide WYSIWYG editing, as opposed to working directly with the HTML tags. You should know some HTML to port the pages efficiently. Most browsers have a View Source command, which you can use to look at the HTML of pages you've seen on other servers. Before integrating your pages, you should preview the pages. Place them in a single directory on your disk and use the Open File option of your browser to preview the pages.

For pages with forms, check the HTML pages to make sure the form actions (URLs) for each page are specified and unique, and that the form item tags have unique names specified. Also make sure that the form method (GET or POST) is the one you want to use. The POST method passes the form arguments in the HTTP request body; they are invisible to the user. The GET method appends the form arguments to the URL; they are visible. Sometimes this is desirable for repeated (or bookmarked) queries, but usually you want the forms to be submitted with the POST method.

Step 3. Identify the variables to be accessed

Once you have designed and coded your HTML pages, the key engineering task is to identify the device variables used in these pages. Each HTML page to be displayed may contain dynamic data retrieved from the device. Additionally, the device can be controlled by using HTML forms that pass data from the browser to the device.

Examples of dynamic data display are the status of ports on an Ethernet switch or the level of toner remaining in a laser printer. Examples of controlling a device are specifying the initial configuration of a router or changing the temperature setting of a building HVAC system. Each variable to be displayed or written needs to have the access to that variable defined. The RomPager engine uses a variable access structure for each variable. The variable access structure contains pointers to the direct memory location of the variable or to an access routine, as well as type information about the variable.

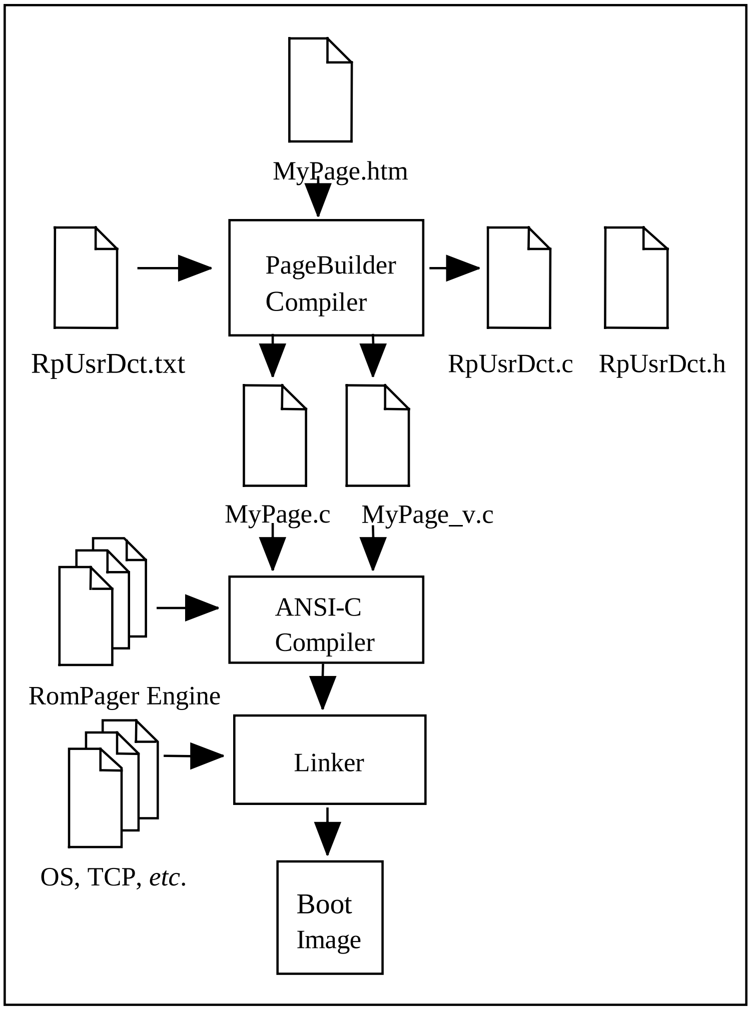
The access routines used to read/write the variables are referred to as Get/Set routines throughout this manual because of their resemblance to SNMP Get/Set calls. These calls may indeed be the same Get/Set routines as used by SNMP if the device is SNMP manageable. Alternatively, they may be new calls written just to support the Web-based management of the device. The variable access structures provide the link between the HTML pages and forms and the Get/Set routines.

Step 4. Define the HTML links to the variables

The variable access structures are defined by using special RomPager HTML comment tags that can be incorporated into the HTML file that defines a page. These comment tags, which are integrated with the HTML code of a page, are used to pass information to the PBuilder compiler about how the variables will be accessed. The compiler generates the ANSI C source code version of the HTML code. The combined HTML and comment tags describe the page layout and the access structures to the variables, while keeping the actual code that accesses the variables in separately compiled modules.

Step 5. Generate C source code using the PBuilder compiler

The next step in the process is to pass the HTML pages through the PBuilder compiler to produce C source code. The following figure shows the flow of the modules that are developed and passed through compile and linking steps.



The PBuilder compiler uses the HTML input file to build two output files:

- The C source code that represents the compressed internal format of the HTML page with structures for both static text and variable access
- A set of stub routines that can be fleshed out to access the variables

The compiler uses a built-in system dictionary for HTML and an optional user provided dictionary to compress commonly used character strings and phrases. See the Phrase Dictionaries section for further details.

If you have more than one HTML file, you should put them in a batch file and process this batch file (**`.pbb`**) using PBuilder, rather than processing them individually. Using a batch file ensures the correct **`RpPages.c`** file is generated for your application.

Step 6. Flesh out the C source code for variable access

The stub routines produced by the PBuilder compiler need to be fleshed out to access the variables, or they can be discarded, if the variable access structures point to existing access routines.

Step 7. Compile and link the C source code

The final step in the process of building the Web pages for the device is compiling the C source code. Since the PBuilder compiler produces standard ANSI C code, you can use any compliant compiler to generate the object code. The object code is then linked into the same ROM image as the engine and the Get/Set routines.

The next sections of the manual describe the HTML Comment Tags used to define pages, how to use the compiler, and the other information necessary to build a Web management application.

Note: Using some PBuilder functions requires calling `RpHSGetServerData`, which returns a handle to the internal data structure maintained on the server. See the *NET+OS Application Software Reference Guide.*

HTML Comment Tags

HTML comment tag formats

The PBuilder compiler prepares the internal format of an HTML page by examining the page and storing the information into internal C-format structures that are used by the RomPager engine to serve the page. The internal structures consist of static elements for storing the regular HTML and dynamic elements for storing information about how to create the dynamic HTML at runtime. RomPager comment tags are a special form of standard HTML comment tags that are used to provide the compiler instructions for creating the RomPager internal formats. Since most HTML editors support HTML comment tags, the RomPager internal format information may be stored in the HTML file along with the normal HTML for displaying a page.

The form of an HTML comment tag is:

```
<!–– the comment ––>
```

The RomPager comment tags are used in this manner:

```
<!–– rompagercommenttag keyword1=value1 keyword2=value2 …––>
<P>Some Sample HTML</P>
<!-- RpEnd -->
```

The tag provides the compiler with the information it needs to prepare the internal RomPager structures. Normally, there will be sample HTML following this tag, so that when the page is previewed by an offline browser or HTML page editor, the offline page will show what the dynamic page will look like when it is served. The page editor ignores the RomPager comment tags and displays the sample HTML. The `RpEnd` tag tells the compiler where the regular HTML starts up again.

For the cases where there is no sample HTML, the `RpNoSample` keyword can be inserted into any comment tag. In this case, the statement:

```
<!–– rompagercommenttag keyword1=value1 keyword2=value2 … RpNoSample ––>
```

is equivalent to:

```
<!–– rompagercommenttag keyword1=value1 keyword2=value2 … ––><!-- RpEnd -->
```

Most of the dynamic RomPager elements have a variable access structure that contains pointers to access the dynamic information. The pointers are qualified with a type to indicate how the pointer is used to access the variable. The pointers can point to a memory location, or a function routine. Most variable access structures have methods both for retrieving a value (the Get pointer) and for setting a value (the Set pointer).

The comment tag keywords are used to specify the values of the variable access structures. The pointer types are indicated by the `RpGetType` and `RpSetType` keywords. The allowed values for these keywords are:

```
Direct
Function
Complex
Null
```

If the pointer type is `Direct`, then `RpGetPtr` or `RpSetPtr` specifies the name of the memory location of the variable. If the pointer type is `Function` or `Complex`, the `RpGetPtr` or `RpSetPtr` keyword specifies the name of the routine to the variable. If the pointer type is `RpGetPtr` or `RpSetPtr`, customer-provided routines are used to access the variable.

If the comment tag is describing a form element that is only used to set a value and has no Get function, a keyword value of Null can be specified for RpGetType. If the RpGetType or RpSetType keywords are not specified, a value of Direct is used. If RpGetPtr or RpSetPtr is not specified and the comment tag contains a Name or RpName keyword, the pointer name is created from the Name or RpName keyword.

```
[RpGetType, RpSetType] = [Direct, Function, Complex, Null]
```

When the RomPager engine serves a page or processes a form, it provides conversion between the HTML ASCII string format and the device internal format. The type of conversion requested is indicated by using the RpTextType keyword in the various RomPager comment tags. The allowed values are:

```
ASCII
ASCIIExtended
ASCIIFixed
Hex
HexColonForm
DotForm
Signed8
Signed16
Signed32
Signed64
Unsigned8
Unsigned16
Unsigned32
Unsigned64
```

If the RpTextType keyword is not specified, ASCII is used.

```
RpTextType = [ASCII, ASCIIExtended, ASCIIFixed, Hex, HexColonForm, DotForm,
    Signed8, Signed16, Signed32, Signed64,Unsigned8, Unsigned16, Unsigned32,
    Unsigned64]
```

The internal elements that PBuilder creates all have default identifiers. The RpIdentifier keyword can be used with most RomPager comment tags to define a specific internal identifier. Memory savings can be achieved by using common elements among more than one page. If an element is to be used more than once, a specific identifier can be provided by using the RpIdentifier keyword. The RpUseIdentifier comment tag (which is discussed later) is used to refer to the additional uses of the common element. Normally, the internal elements that are defined are inserted into a page/form element list as well as being individually defined. If the elements being defined are common elements that are included independently, the Independent keyword can be set to prevent the element from being included in the page/form element list.

The Name keyword can be used in any RomPager comment tag to define the HTML name that is used to refer to the internal element. The RomPager comment tags that are used to define <INPUT> tags or form elements all require the Name keyword, but this keyword can be used with any RomPager comment tag to set up a name association. Since the name takes storage in device memory, a developer with a limited resource device should use name associations only when needed, and short names are preferable to longer names.

If you are using JavaScript in form elements, both PBuilder and the RomPager engine must be configured to store and display the JavaScript along with the rest of the internal elements. For PBuilder, you must enable the UseJavaScript keyword in the **PbSetUp.txt** file. PBuilder then assumes that all unrecognized keywords are JavaScript and assigns storage in the structure for the JavaScript. The RomPager engine can recreate the JavaScript when the element is displayed. See the <INPUT TYPE=Button> example to see the use of JavaScript in RomPager comment tags.

Static text

Static text is the building block used for the bulk of an HTML page. It consists of all the text that does not vary as pages are served. PBuilder analyzes static text elements and compresses them using the built-in system phrase dictionary and a user-supplied phrase dictionary.

HTML comment tags

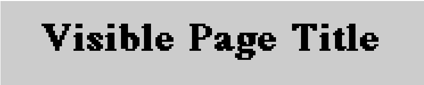
```
<!–– RpDataZeroTerminated ––>
<!–– RpDZT ––>
<!–– RpEnd ––>
```

The `RpDataZeroTerminated` tag and the `RpEnd` tag bracket HTML static text and are used to define the internal elements that generate the text when the page is served. These tags are not normally needed in a page, since PBuilder processes all unrecognized text as though it were bracketed by these tags. They can be used for clarity to define the HTML text not specified by the other RomPager comment tags. `RpDZT` is an alias for `RpDataZeroTerminated`.

Function prototypes

None.

Sample



Visible Page Title

Commented HTML

```
<!–– RpDataZeroTerminated ––><HTML><HEAD>
<TITLE>HTML Page Title</TITLE>
</HEAD><BODY><!-- RpEnd -->
<!–– RpDZT ––><P><H2><CENTER>Visible Page Title</CENTER>
</H2></P><!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated, &PgSample_Item_1 },
   { eRpItemType_DataZeroTerminated, &PgSample_Item_2 },
/* Structures */
static char PgSample_Item_1[] =
   C_oHTML_oHEAD_oTITLE “HTML Page Title” C_xTITLE_xHEAD_oBODY;
static char PgSample_Item_2[] =
   C_oP C_oH2 C_oCENTER “Visible Page Title”
   C_xCENTER C_xH2 C_xP;
```

Sample generated HTML

```
<HTML><HEAD><TITLE>HTML Page Title</TITLE></HEAD><BODY>
<P><H2><CENTER>Visible Page Title</CENTER></H2></P>
```

Dynamic values

A dynamic value can be inserted anywhere in a page by pointing to the variable location or a function to provide the value.

HTML comment tags

```
<!–– RpDisplayText RpGetType=gt RpGetPtr=gp RpTextType=t Size=s ––>
<!–– RpNamedDisplayText Name=n RpGetType=gt RpGetPtr=gp
   RpTextType=t Size=s ––>
```

These tags define the location of the variable to display or point to a function to retrieve that variable. The value of the variable is dynamically inserted into the HTML page being displayed. If you specify `RpTextType`, the RomPager engine converts non-ASCII values at runtime. The `Size` keyword is used for conversions when `RpTextType` is `ASCIIFixed`, `Hex`, `HexColonForm`, or `DotForm` to specify the number of characters that should appear on the page for the field. The `Name` keyword is used with `RpNamedDisplayText` and specifies the HTML name to be passed into complex Get routines.

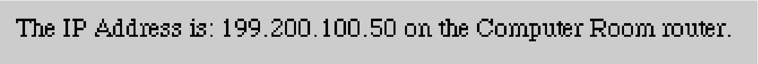
These tags give you the equivalent of “Server-side includes” in that if you want to have C functions that are called to insert arbitrary display data into the middle of a page, you can use these tags with the default `RpTextType` of `ASCII`.

Function prototypes

```
GetType = Function - TextType = ASCII, ASCIIFixed, Hex, HexColonForm,
                  DotForm
typedef char * (*rpFetchBytesFuncPtr)(void);
GetType = Complex - TextType = ASCII, ASCIIFixed, Hex, HexColonForm, DotForm
typedef char * (*rpFetchBytesComplexPtr)(void *theServerDataPtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);
GetType = Function - TextType = Unsigned8, Unsigned16, Unsigned32,
                  Unsigned64
typedef Unsigned8    (*rpFetchUnsigned8FuncPtr) (void);
typedef Unsigned16   (*rpFetchUnsigned16FuncPtr)(void);
typedef Unsigned32   (*rpFetchUnsigned32FuncPtr)(void);
typedef Unsigned64   (*rpFetchUnsigned64FuncPtr)(void);
GetType = Complex - TextType = Unsigned8, Unsigned16, Unsigned32, Unsigned64
typedef Unsigned8 (*rpFetchUnsigned8ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Unsigned16 (*rpFetchUnsigned16ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Unsigned32 (*rpFetchUnsigned32ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Unsigned64 (*rpFetchUnsigned64ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
GetType = Function - TextType = Signed8, Signed16, Signed32, Signed64

typedef Signed8      (*rpFetchSigned8FuncPtr)(void);
typedef Signed16  (*rpFetchSigned16FuncPtr)(void);
typedef Signed32  (*rpFetchSigned32FuncPtr)(void);
typedef Signed64  (*rpFetchSigned64FuncPtr)(void);
GetType = Complex - TextType = Signed8, Signed16, Signed32, Signed64
```

```
typedef Signed8 (*rpFetchSigned8ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Signed16 (*rpFetchSigned16ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Signed32 (*rpFetchSigned32ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Signed64 (*rpFetchSigned64ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
```

Sample



The IP Address is: 199.200.100.50 on the Computer Room router.

Commented HTML

```
<!-- RpDisplayText RpGetType=Function RpGetPtr=SomeFunction -->
The IP Address is: <!-- RpEnd -->
<!-- RpDisplayText RpGetType=Direct RpGetPtr=theIpAddress
RpTextType=DotForm Size=15 -->199.200.100.50<!-- RpEnd -->
<!-- RpNamedDisplayText RpName=ThisName RpGetType=Complex
RpGetPtr=AnotherFunction --> on the Computer Room router.<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DisplayText,       &PgSample_Item_1 },
   { eRpItemType_DisplayText,       &PgSample_Item_2 },
   { eRpItemType_NamedDisplayText,     &PgSample_Item_3 },
/* Structures */
rpTextDisplayItem PgSample_Item_1 = {
   SomeFunction,
   eRpVarType_Function
   eRpTextType_ASCII,
   0
};
rpTextDisplayItem PgSample_Item_2 = {
   &theIpAddress,
   eRpVarType_Direct,
   eRpTextType_DotForm,
   15
};
```

Sample generated HTML

```
<P>The IP Address is: 199.200.100.50 on the Computer Room router.</P>
```

HTML <INPUT> elements

HTML forms use `<INPUT>` tags to describe information that can be entered in a form. The `RpFormInput` tag is used to provide PBuilder with the information it needs to prepare the internal structures to support the `<INPUT>` tags.

HTML comment tag

```
<!-- RpFormInput TYPE=t NAME=n VALUE=v RpGetType=gt RpGetPtr=gp
            RpSetType=st RpSetPtr=sp RpTextType=tt MaxLength=m
            Size=s RpItemNumber=in Dynamic -->
TYPE = [text, password, hidden, checkbox, radio, file, button, reset, submit]
```

This tag contains the information for both the generation of an `<INPUT>` tag and for processing the form value. The `TYPE` keyword can be `text`, `password`, `hidden`, `checkbox`, `radio`, `file`, `button`, `reset`, or `submit` indicating the value for the `<INPUT>` tag `Type` keyword. If this keyword is not specified, `text` is used.

The `Size` keyword is used for the `<INPUT>` tag `SIZE` keyword and for data type conversions when `RpTextType` is `ASCIIFixed`, `Hex`, `HexColonForm`, or `DotForm` to specify the number of characters that should appear on the page for the field.

`MaxLength` is used for the `<INPUT>` tag `MAXLENGTH` keyword.

The `VALUE` keyword is used for the `<INPUT>` tag `Value` keyword when `TYPE` is `radio`, `reset`, or `submit`.

If `TYPE` is `radio`, the `RpItemNumber` keyword can supply a value to which the radio button group variable is set if this radio button is chosen. If `RpItemNumber` is not specified, the value for the radio button becomes one greater than the last radio button.

The `Dynamic` keyword is used if the form element will be used inside a repeat group. For further information, see the section on Repeat Groups.

<INPUT TYPE=TEXT>

HTML comment tag

```
<!-- RpFormInput TYPE=text NAME=n RpGetType=gt RpGetPtr=gp
         RpSetType=st RpSetPtr=sp RpTextType=tt MaxLength=m Size=s -->
```

This tag is used in a form to define text entry fields. The HTML name of the checkbox is specified by the `NAME` keyword. The `Size` keyword determines the display size of the field and the `MaxLength` keyword determines the maximum number of characters that can be entered in the field. The RomPager engine provides automatic data conversion from the HTML/form string format depending on the value of the `RpTextType` keyword. The function prototypes used to access and store the values are shown as follows.

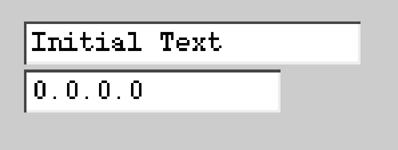
Function prototypes

```
GetType = Function - TextType = ASCII, ASCIIFixed, Hex, HexColonForm, DotForm
typedef char * (*rpFetchBytesFuncPtr)(void);

GetType = Complex - TextType = ASCII, ASCIIFixed, Hex, HexColonForm, DotForm

typedef char * (*rpFetchBytesComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
GetType = Function - TextType = Unsigned8, Unsigned16, Unsigned32, Unsigned64

typedef Unsigned8    (*rpFetchUnsigned8FuncPtr)(void);
typedef Unsigned16   (*rpFetchUnsigned16FuncPtr)(void);
typedef Unsigned32   (*rpFetchUnsigned32FuncPtr)(void);
typedef Unsigned64   (*rpFetchUnsigned64FuncPtr)(void);
GetType = Complex - TextType = Unsigned8, Unsigned16, Unsigned32, Unsigned64

typedef Unsigned8 (*rpFetchUnsigned8ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Unsigned16 (*rpFetchUnsigned16ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Unsigned32 (*rpFetchUnsigned32ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Unsigned64 (*rpFetchUnsigned64ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);

GetType = Function - TextType = Signed8, Signed16, Signed32, Signed64

typedef Signed8   (*rpFetchSigned8FuncPtr)(void);
typedef Signed16  (*rpFetchSigned16FuncPtr)(void);
typedef Signed32  (*rpFetchSigned32FuncPtr)(void);
typedef Signed64  (*rpFetchSigned64FuncPtr)(void);
GetType = Complex - TextType = Signed8, Signed16, Signed32, Signed64

typedef Signed8 (*rpFetchSigned8ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Signed16 (*rpFetchSigned16ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef Signed32 (*rpFetchSigned32ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);
```

```
typedef Signed64 (*rpFetchSigned64ComplexPtr)(void *theServerDataPtr,
               char *theNamePtr, Signed16Ptr theIndexValuesPtr);

SetType = Function - TextType = ASCII, ASCIIFixed, Hex, HexColonForm, DotForm

typedef void (*rpStoreAsciiTextFuncPtr)(char *theValuePtr);

SetType = Complex - TextType = ASCII, ASCIIFixed, Hex, HexColonForm, DotForm

typedef void (*rpStoreAsciiTextComplexPtr)(void *theServerDataPtr,
char *theValuePtr, char * theNamePtr,
Signed16Ptr theIndexValuesPtr);

SetType = Function - TextType = Unsigned8, Unsigned16, Unsigned32,
                               Unsigned64

typedef void (*rpStoreUnsigned8FuncPtr)(Unsigned8 theValue);
typedef void (*rpStoreUnsigned16FuncPtr)(Unsigned16 theValue);
typedef void (*rpStoreUnsigned32FuncPtr)(Unsigned32 theValue);
typedef void (*rpStoreUnsigned64FuncPtr)(Unsigned64 theValue);

SetType = Complex - TextType = Unsigned8, Unsigned16, Unsigned32, Unsigned64

typedef void (*rpStoreUnsigned8ComplexPtr)(void *theServerDataPtr,
Unsigned8 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
typedef void (*rpStoreUnsigned16ComplexPtr)(void *theServerDataPtr,
Unsigned16 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
typedef void (*rpStoreUnsigned32ComplexPtr)(void *theServerDataPtr,
               Unsigned32 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
typedef void (*rpStoreUnsigned64ComplexPtr)(void *theServerDataPtr,
Unsigned64 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);

SetType = Function - TextType = Signed8, Signed16, Signed32, Signed64

typedef void (*rpStoreSigned8FuncPtr)(Signed8 theValue);
typedef void (*rpStoreSigned16FuncPtr)(Signed16 theValue);
typedef void (*rpStoreSigned32FuncPtr)(Signed32 theValue);
typedef void (*rpStoreSigned64FuncPtr)(Signed64 theValue);

SetType = Complex - TextType = Signed8, Signed16, Signed32, Signed64

typedef void (*rpStoreSigned8ComplexPtr)(void *theServerDataPtr,
Signed8 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
typedef void (*rpStoreSigned16ComplexPtr)(void *theServerDataPtr,
Signed16 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
```

```
typedef void (*rpStoreSigned32ComplexPtr)(void *theServerDataPtr,
Signed32 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
typedef void (*rpStoreSigned64ComplexPtr)(void *theServerDataPtr,
Signed64 theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
```

Sample



Initial Text

0.0.0.0

Commented HTML

```
<!-- RpFormInput TYPE=“TEXT” NAME=“TextValue” SIZE=“20” MAXLENGTH=“20” -->
<INPUT TYPE=“TEXT” NAME=“TextValue” VALUE=“Initial Text”
   SIZE=“20” MAXLENGTH=“20” >
<!-- RpEnd -->
<!-- RpFormInput TYPE=“TEXT” NAME=“DefaultGateway” SIZE=“15” MAXLENGTH=“15”
   RpGetType=Complex RpGetPtr=GetDefaultGateway
   RpSetType=Complex RpSetPtr=SetDefaultGateway RpTextType=“DotForm” -->
<INPUT TYPE=“TEXT” NAME=“DefaultGateway” VALUE=“0.0.0.0”
   SIZE=“15” MAXLENGTH=“15” >
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormAsciiText, &PgSample_Item_1 },
   { eRpItemType_FormAsciiText, &PgSample_Item_2 },

/* Structures */

static rpTextFormItem PgSample_Item_1 = {
   “TextValue”,
   &TextValue,
   &TextValue,
   eRpVarType_Direct,
   eRpVarType_Direct,
   eRpTextType_ASCII,
   20,
   20
};
```

```
static rpTextFormItem PgSample_Item_2 = {
   “DefaultGateway”,
   GetDefaultGateway,
   SetDefaultGateway,
   eRpVarType_Complex,
   eRpVarType_Complex,
   eRpTextType_DotForm,
   15,
   15
};

/* Variables and Functions */

char  TextValue[21] = “Initial Text”;
char  theGatewayAddress[4] = { 0, 0, 0, 0 };

char *GetDefaultGateway(void *theServerDataPtr, char *theNamePtr,
   Signed16Ptr theIndexValuesPtr) {

   return theGatewayAddress;
}

void SetDefaultGateway(void *theServerDataPtr, char *theValuePtr,
                     char *theNamePtr, Signed16Ptr theIndexValuesPtr) {

memcpy(theGatewayAddress, theValuePtr, 4);
}
```

Sample generated HTML

```
<INPUT TYPE=“TEXT” NAME=“TextValue” SIZE=“20” MAXLENGTH=“20”
   VALUE=“Initial Text”>
<INPUT TYPE=“TEXT” NAME=“DefaultGateway” SIZE=“15” MAXLENGTH=“15”
   VALUE=“0.0.0.0”>
```

<INPUT TYPE=PASSWORD>
<INPUT TYPE=HIDDEN>

HTML comment tag

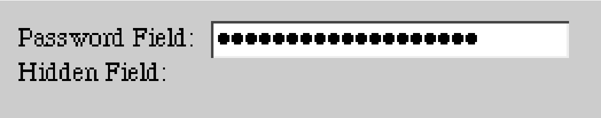
```
<!–– RpFormInput TYPE=password NAME=n RpGetType=gt RpGetPtr=gp
   RpSetType=st RpSetPtr=sp RpTextType=tt MaxLength=m Size=s ––>

<!–– RpFormInput TYPE=hidden NAME=n RpGetType=gt RpGetPtr=gp
   RpSetType=st RpSetPtr=sp RpTextType=tt MaxLength=m Size=s ––>
```

These tags are used in a form to define password and hidden fields. Password fields behave the same way as text entry fields except that the browser does not display the value that is keyed in. Hidden fields are not displayed by the browser, but are still submitted with other fields in a form. They can be used by applications for state passing. The keywords for these tags work the same way as in the `TYPE=text` tags.

Function prototypes

These comment tags use the same functions for accessing variables as the `TYPE=text` tags.

Sample



Password Field: ●●●●●●●●●●●●●●●●●●●●
Hidden Field:

Commented HTML

```
<!-- RpDataZeroTerminated --><P>Password Field:&nbsp;&nbsp;<!-- RpEnd -->
<!-- RpFormInput TYPE=PASSWORD NAME=Password SIZE=25 MAXLENGTH=25
RpTextType=ASCII
   RpGetType=Direct RpGetPtr=gPassword
   RpSetType=Direct RpSetPtr=gPassword -->
<INPUT TYPE=“PASSWORD” NAME=“Password” VALUE=“Initial Password Text”
   SIZE=“25” MAXLENGTH=“25” >
<!-- RpEnd -->
<!-- RpDataZeroTerminated --><BR>Hidden Field:<!-- RpEnd -->
<!-- RpFormInput TYPE=HIDDEN NAME=Hidden RpTextType=ASCII
   RpGetType=Direct RpGetPtr=gHidden
   RpSetType=Direct RpSetPtr=gHidden -->
<INPUT TYPE=“HIDDEN” NAME=“Hidden” VALUE=“Initial Hidden Text”>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_FormPasswordText,  &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },
   { eRpItemType_FormHiddenText,    &PgSample_Item_4 },
/* Structures */
static char PgSample_Item_1[] =
   C_oP “Password Field:” C_NBSP C_NBSP;
static rpTextFormItem PgSample_Item_2 = {
   “Password”,
   &gPassword,
   &gPassword,
   eRpVarType_Direct,
   eRpVarType_Direct,
   eRpTextType_ASCII,
   25,
   25,
};

static char PgSample_Item_3[] =
   C_oBR “Hidden Field: “;

static rpTextFormItem PgSample_Item_4 = {
   “Hidden”,
   &gHidden,
   &gHidden,
   eRpVarType_Direct,
   eRpVarType_Direct,
   eRpTextType_ASCII,
   20,
   20,
};

/* Variables and Functions */

static char gPassword[26] = “Initial Password Text”;
static char gHidden[26] = “Initial Hidden Text”;
```

Sample generated HTML

```
<P>Password Field:&nbsp;&nbsp;<INPUT TYPE=“PASSWORD” NAME=“Password”  SIZE=“25”
MAXLENGTH=“25” VALUE=“Initial Password Text”><BR>
Hidden Field:<INPUT TYPE=“HIDDEN” NAME=“Hidden” VALUE=“Initial Hidden Text”>
```

<INPUT TYPE=CHECKBOX>

HTML comment tag

```
   <!–– RpFormInput TYPE=checkbox NAME=n RpGetType=gt RpGetPtr=gp
            RpSetType=st RpSetPtr=sp ––>
```

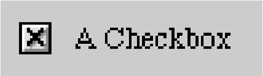
This tag is used in a form to define checkbox elements. The HTML name of the checkbox is specified by the `NAME` keyword. The functions described as follows can be used to access and store a Boolean value that determines whether the box is checked or not.

Since HTML forms only submit the checkbox items that are checked on a form, all checkboxes on a form will be turned off by the RomPager engine at the beginning of form processing. Then as the form is processed, the individual checkbox items that are checked, will be set to on. Both the reset and set operations use the set function specified by the comment tag.

Function prototypes

```
   GetType = Function
   typedef Boolean (*rpFetchBooleanFuncPtr)(void);
   GetType = Complex
   typedef Boolean (*rpFetchBooleanComplexPtr)(void *theServerDataPtr,
                  char *theNamePtr, Signed16Ptr theIndexValuesPtr);

   SetType = Function

   typedef void (*rpStoreBooleanFuncPtr)(Boolean theValue);

   SetType = Complex

   typedef void (*rpStoreBooleanComplexPtr)(void *theServerDataPtr,
   Boolean theValue, char *theNamePtr,
   Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
         <!-- RpFormInput TYPE=CHECKBOX NAME=CheckBox
            RpGetType=Function RpGetPtr=GetCheckBox
            RpSetType=Function RpSetPtr=SetCheckBox -->
         <INPUT TYPE=“CHECKBOX” NAME=“CheckBox”>
         <!-- RpEnd -->
         <!-- RpDataZeroTerminated -->A Checkbox<BR><!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormCheckbox,         &PgSample_Item_1 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_2 },
/* Structures */
static rpCheckboxFormItem PgSample_Item_1 = {
   “CheckBox”,
   GetCheckBox,
   SetCheckBox,
   eRpVarType_Function,
   eRpVarType_Function
};

static char PgSample_Item_2[] =
   “A Checkbox” C_oBR;

/* Variables and Functions */
Boolean theCheckBoxValue = True;
Boolean GetCheckBox(void) {
   return theCheckBoxValue;
}

void SetCheckBox(Boolean theValue) {

   theCheckBoxValue = theValue;
}
```

Sample generated HTML

```
<INPUT TYPE=“CHECKBOX” NAME=“CheckBox” CHECKED>A Checkbox<BR>
```

<INPUT TYPE=RADIO>

HTML comment tag

```
<!–– RpRadioButtonGroup NAME=n RpGetType=gt RpGetPtr=gp
   RpSetType=st RpSetPtr=sp ––>
<!–– RpFormInput TYPE=radio NAME=n VALUE=v RpItemNumber=in ––>
```

These tags are used in a form to define radio button groups. Radio buttons are used to specify groups of values, only one of which can be selected at a time. The `RpRadioButtonGroup` tag contains the information that is common for all radio buttons of a radio button group. It contains the information for both the generation of the button display values and for processing the form value. The `Name` keyword specifies the identification of the radio button group. This tag must precede the `<!–– RpFormInput TYPE=radio ––>` tags for a given radio button group. If it is missing, PBuilder creates a radio group structure with default values.

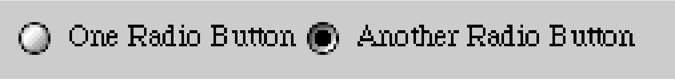
The `RpFormInput` tag for a radio button uses the `Name` keyword to specify the radio button group it belongs to, uses the `Value` keyword to specify the HTML value and uses the `RpItemNumber` keyword to specify the item number used for internal processing. If the `RpItemNumber` keywords are not specified, PBuilder supplies default values starting at 0.

When a page is displayed, the RomPager engine uses the access functions described as follows to retrieve a value. It matches the retrieved value against the item number of each button in the group to determine which button to highlight. When a form is submitted, the RomPager engine matches the HTML value of the submitted button against the buttons in the radio group to determine which item number to pass to the storage functions.

Function prototypes

```
GetType = Function
typedef rpOneOfSeveral (*rpFetchRadioGroupFuncPtr)(void);

GetType = Complex
typedef rpOneOfSeveral (*rpFetchRadioGroupComplexPtr)
(void *theServerDataPtr, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
SetType = Function

typedef void (*rpStoreRadioGroupFuncPtr)(rpOneOfSeveral theValue);

SetType = Complex

typedef void (*rpStoreRadioGroupComplexPtr)(void *theServerDataPtr,
rpOneOfSeveral theValue, char *theNamePtr,
Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
<!-- RpRadioButtonGroup NAME=RadioButtonGroup
        RpGetType=Direct RpGetPtr=gRadioButtonGroup RpSetType=Direct
RpSetPtr=gRadioButtonGroup -->
<!-- RpFormInput TYPE=RADIO NAME=RadioButtonGroup VALUE=One RpItemNumber=0 -->
    <INPUT TYPE="RADIO" NAME="RadioButtonGroup" VALUE="One">
<!-- RpEnd -->
<!-- RpDZT -->One Radio Button<!-- RpEnd -->
<!-- RpFormInput TYPE=RADIO NAME=RadioButtonGroup VALUE=Another
RpItemNumber=1 -->
    <INPUT TYPE="RADIO" NAME="RadioButtonGroup" VALUE="Another" CHECKED>
<!-- RpEnd -->
<!-- RpDZT -->Another Radio Button<BR><!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormRadioButton,   &PgSample_Item_1 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_2 },
   { eRpItemType_FormRadioButton,   &PgSample_Item_3 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_4 },

/* Structures */

static rpRadioGroupInfo PgSample_Item_0 = {
   "RadioButtonGroup",
   &gRadioButtonGroup,
   &gRadioButtonGroup,
   eRpVarType_Direct,
   eRpVarType_Direct
};

static rpRadioButtonFormItem PgSample_Item_1 = {
   &PgSample_Item_0,
   "One",
   0
};

static char PgSample_Item_2[] =
   "One Radio Button\n";

static rpRadioButtonFormItem PgSample_Item_3 = {
   &PgSample_Item_0,
   "Another",
   1
};

static char PgSample_Item_4[] =
   "Another Radio Button" C_oBR;
```

```
/* Variables and Functions */

rpOneOfSeveral gRadioButtonGroup = 1;
```

Sample generated HTML

```
<INPUT TYPE=“RADIO” NAME=“RadioButtonGroup” VALUE=“One”>
One Radio Button
<INPUT TYPE=“RADIO” NAME=“RadioButtonGroup” VALUE=“Another” CHECKED>
Another Radio Button<BR>
```

<INPUT TYPE=FILE>

HTML comment tag

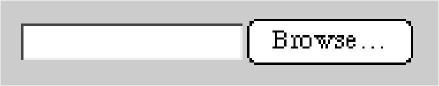
```
<!–– RpFormInput TYPE=file NAME=n MaxLength=m Size=s ––>
```

This tag is used in a form to define the field for selecting a file to upload. In order for HTTP file uploads to work, the `Enctype` keyword in the `RpFormHeader` tag must be specified as MULTIPART/FORM-DATA.

Function prototypes

There are no access functions for this element type.

Sample



Commented HTML

```
<!-- RpFormInput TYPE=“file” NAME=“FileUpload” SIZE=“15” MAXLENGTH=“15” -->
<INPUT TYPE=“FILE” NAME=“FileUpload” SIZE=“15” MAXLENGTH=“15”>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormFile, &PgSample_Item_1 },
/* Structures */
static rpFileFormItem PgSample_Item_1 = {
   “FileUpload”,
   15,
   15
};
```

Sample generated HTML

```
<INPUT TYPE=“FILE” NAME=“FileUpload” SIZE=“15” MAXLENGTH=“15”>
```

<INPUT TYPE=BUTTON>

HTML comment tag

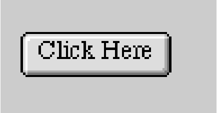
```
<!–– RpFormInput TYPE=button NAME=n RpGetType=gt RpGetPtr=gp
      RpSetType=st RpSetPtr=sp ––>
```

These input elements are normally used as placeholders for JavaScript commands. The HTML name of the button is specified by the `NAME` keyword. The value displayed on the button is determined by the text functions described as follows. If the `UseJavaScript` keyword is enabled in the **`PbSetUp.txt`** file, PBuilder assumes that all unrecognized keywords are JavaScript and assigns storage in the structure for the JavaScript. The RomPager engine recreates the JavaScript when the element is displayed.

Function prototypes

```
GetType = Function
typedef char * (*rpFetchBytesFuncPtr)(void);
GetType = Complex

typedef char * (*rpFetchBytesComplexPtr)(void *theServerDataPtr,
char *theNamePtr, Signed16Ptr theIndexValuesPtr);
SetType = Function

typedef void (*rpStoreAsciiTextFuncPtr)(char *theValuePtr);

SetType = Complex

typedef void (*rpStoreAsciiTextComplexPtr)(void *theServerDataPtr,
char *theValuePtr, char * theNamePtr,
Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
<!-- RpFormInput TYPE=BUTTON NAME=MyButton RpGetType=Direct RpGetPtr=gMyButton
   RpSetType=Direct RpSetPtr=gMyButton onclick=ButtonClick(this.form) -->
<INPUT TYPE=“BUTTON” NAME=“MyButton” VALUE=“Click Here”
   onclick=ButtonClick(this.form)>
   <!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormButton, &PgSample_Item_1 },

/* Structures */

static rpCheckboxFormItem PgSample_Item_1 = {
   “MyButton”,
   &gMyButton,
   &gMyButton,
   eRpVarType_Direct,
   eRpVarType_Direct,
   gJavaScript
};

/* Variables and Functions */

static char gMyButton[] = “Click Here”;
static char gJavaScript[] = “ onclick=ButtonClick(this.form)”;
```

Sample generated HTML

```
<INPUT TYPE=“BUTTON” NAME=“MyButton” VALUE=“Click Here”
   onclick=ButtonClick(this.form)>
```

<INPUT TYPE=RESET>
<INPUT TYPE=SUBMIT>

HTML comment tag

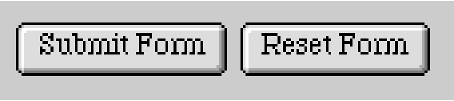
```
<!–– RpFormInput TYPE=submit VALUE=v ––>
<!–– RpFormInput TYPE=reset VALUE=v ––>
```

These tags are used to define HTML Submit and Reset buttons. The `VALUE` keyword defines the label displayed on the button. Reset buttons are used for functions local to the browser and never trigger server actions. Submit buttons are recognized by the RomPager engine and the value is stored in the engine’s data structure. If the device needs to know which of multiple Submit buttons was pressed, it can use the `RpGetSubmitButtonValue` callback routine to retrieve the stored value.

Function prototypes

There are no function calls with these elements.

Sample



Commented HTML

```
<!-- RpFormInput TYPE=SUBMIT VALUE=“Submit Form”-->
    <INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit Form”>
<!-- RpEnd -->
<!-- RpFormInput TYPE=RESET VALUE=“Reset Form” -->
    <INPUT TYPE=“RESET” VALUE=“Reset Form”>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormSubmit,  &PgSample_Item_1 },
   { eRpItemType_FormReset,   &PgSample_Item_2 },
/* Structures */

rpButtonFormItem PgSample_Item_1 = {
   “Submit Form”
};
rpButtonFormItem PgSample_Item_2 = {
   “Reset Form”
};
```

Sample generated HTML

```
<INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit Form”>
<INPUT TYPE=“RESET” VALUE=“Reset Form”>
```

<INPUT TYPE=SUBMIT>

HTML comment tag

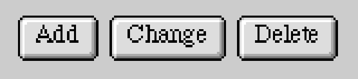
```
<!–– RpFormNamedSubmit NAME=n RpGetType=gt RpGetPtr=gp
            RpSetType=st RpSetPtr=sp ––>
```

Defines HTML Submit buttons, where the label displayed on the button is determined at run time. This tag is also useful to define individually named buttons that can be referred to by JavaScript. The HTML name of the button is specified by the `NAME` keyword. The value that will be displayed on the button is determined by the text functions described as follows.

Function prototypes

```
GetType = Function
typedef char * (*rpFetchBytesFuncPtr)(void);

GetType = Complex
typedef char * (*rpFetchBytesComplexPtr)(void *theServerDataPtr,
 char *theNamePtr, Signed16Ptr theIndexValuesPtr);
SetType = Function
typedef void (*rpStoreAsciiTextFuncPtr)(char *theValuePtr);

SetType = Complex
typedef void (*rpStoreAsciiTextComplexPtr)(void *theServerDataPtr,
 char *theValuePtr, char * theNamePtr,
 Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
<!-- RpFormNamedSubmit NAME=Submit1 RpGetType=Direct RpGetPtr=gAddName
   RpSetType=Direct RpSetPtr=gAddName -->
    <INPUT TYPE="SUBMIT" NAME="Submit1" VALUE="Add">
<!-- RpEnd -->
<!-- RpFormNamedSubmit NAME=Submit2 RpGetType=Function RpGetPtr=GetChangeName
   RpSetType=Function RpSetPtr=SetChangeName -->
    <INPUT TYPE="SUBMIT" NAME="Submit2" VALUE="Change">
<!-- RpEnd -->
<!-- RpFormNamedSubmit NAME=Submit3 RpGetType=Complex RpGetPtr=GetDeleteName
   RpSetType=Complex RpSetPtr=SetDeleteName -->
    <INPUT TYPE="SUBMIT" NAME="Submit3" VALUE="Delete">
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormNamedSubmit, &PgSample_Item_1 },
   { eRpItemType_FormNamedSubmit, &PgSample_Item_2 },
   { eRpItemType_FormNamedSubmit, &PgSample_Item_3 },
/* Structures */

static rpCheckboxFormItem PgSample_Item_1[] = {
   “Submit1”,
   &gAddName,
   &gAddName,
   eRpVarType_Direct,
   eRpVarType_Direct
};
static rpCheckboxFormItem PgSample_Item_2[] = {
   “Submit2”,
   GetChangeName,
   SetChangeName,
   eRpVarType_Function,
   eRpVarType_Function
};
static rpCheckboxFormItem PgSample_Item_3[] = {
   “Submit3”,
   GetDeleteName,
   SetDeleteName,
   eRpVarType_Complex,
   eRpVarType_Complex
};

/* Variables and Functions */
static char gAddName[15] = “Add”;
static char gChangeName[15] = “Change”;
static char gDeleteName[15] = “Delete”;
char *GetChangeName(void) {
   return gChangeName;
}
void SetChangeName(char *theValuePtr) {
   strcpy(gChangeName, theValuePtr);
}
char *GetDeleteName(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   return gDeleteName;
}
void SetDeleteName(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr) {
   strcpy(gDeleteName, theValuePtr);
}
```

Sample generated HTML

```
<INPUT TYPE=“SUBMIT” NAME=“Submit1” VALUE=“Add”>
<INPUT TYPE=“SUBMIT” NAME=“Submit2” VALUE=“Change”>
<INPUT TYPE=“SUBMIT” NAME=“Submit3” VALUE=“Delete”>
```

<SELECT><OPTION>
(Fixed List — Single Select)

HTML comment tag

```
<!-- RpFormSingleSelect NAME=n Size=s RpGetType=gt RpGetPtr=gp
         RpSetType=st RpSetPtr=sp Dynamic -->
<!-- RpSingleSelectOption VALUE=v RpItemNumber=i -->
<!-- RpEndSelect -->
```

These tags are used for creating `<SELECT>` menus where the list of menu choices is known at compile time and only a single choice can be made from the menu.

The `RpFormSingleSelect` tag contains the information for the generation of the `<SELECT>` tag and for processing the form value. The `Name` keyword specifies the HTML name of the item. The `Size` keyword is used for the `<SELECT>` tag `SIZE` keyword, which controls the size of the menu list. If the keyword is not specified, 1 is used. The access functions are used to retrieve and store a value that corresponds with a given option. The `Dynamic` keyword is used if the form element will be used inside a repeat group.

The `RpSingleSelectOption` comment tag provides the specifications of the options for the `<SELECT>` list. The tag uses the `Value` keyword to specify the display string of the option and uses the `RpItemNumber` keyword to specify the item number used for internal processing. If the `RpItemNumber` keywords are not specified, PBuilder supplies default values starting at 1.

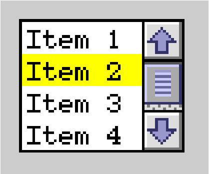
The `RpEndSelect` comment tag is used to terminate the options list of the `<SELECT>` menu.

When a page is displayed, the RomPager engine uses the access functions described as follows to retrieve a value. It matches the retrieved value against the item number of each option in the `<SELECT>` list to determine which option to highlight. When a form is submitted, the RomPager engine uses the HTML name of the submitted option to determine the value to pass to the storage function.

Function prototypes

```
GetType = Function
typedef  Unsigned8 (*rpFetchRadioGroupFuncPtr)(void);

GetType = Complex
typedef  Unsigned8 (*rpFetchRadioGroupComplexPtr)(void *theServerDataPtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);

SetType = Function

typedef  void (*rpStoreRadioGroupFuncPtr)(Unsigned8 theValue);

SetType = Complex

typedef  void (*rpStoreRadioGroupComplexPtr)(void *theServerDataPtr,
      Unsigned8 theValue, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
<!-- RpFormSingleSelect NAME=“Fixed Single Select” SIZE=4
   RpGetType=Direct RpGetPtr=gFixedSingleSelect
   RpSetType=Direct RpSetPtr=gFixedSingleSelect -->
   <SELECT NAME=“Fixed Single Select” SIZE=“4”>
<!-- RpEnd -->
<!-- RpSingleSelectOption VALUE=“Item 1” RpItemNumber=1 -->
   <OPTION>Item 1
<!-- RpEnd -->
<!-- RpSingleSelectOption VALUE=“Item 2” RpItemNumber=2 -->
   <OPTION SELECTED>Item 2
<!-- RpEnd -->
<!-- RpSingleSelectOption VALUE=“Item 3” RpItemNumber=3 -->
   <OPTION>Item 3
<!-- RpEnd -->
<!-- RpSingleSelectOption VALUE=“Item 4” RpItemNumber=4 -->
   <OPTION>Item 4
<!-- RpEnd -->
<!-- RpSingleSelectOption VALUE=“Item 5” RpItemNumber=5 -->
   <OPTION>Item 5
<!-- RpEnd -->
<!-- RpEndSelect -->
   </SELECT>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormFixedSingleSelect,   &PgSample_Item_1 },
/* Structures */

static rpOption_OneSelect PgSample_Item_1_Option5 = {
   (rpOption_OneSelectPtr) 0,
   “Item 5”,
   5
};

static rpOption_OneSelect PgSample_Item_1_Option4 = {
   &PgSample_Item_1_Option5,
   “Item 4”,
   4
};
```

```
static rpOption_OneSelect PgSample_Item_1_Option3 = {
   &PgSample_Item_1_Option4,
   “Item 3”,
   3
};

static rpOption_OneSelect PgSample_Item_1_Option2 = {
   &PgSample_Item_1_Option3,
   “Item 2”,
   2
};

static rpOption_OneSelect PgSample_Item_1_Option1 = {
   &PgSample_Item_1_Option2,
   “Item 1”,
   1
};

static rpFixedSingleSelectFormItem PgSample_Item_1 = {
   “Fixed Single Select”,
   &PgSample_Item_1_Option1,
   &gFixedSingleSelect,
   &gFixedSingleSelect,
   eRpVarType_Direct,
   eRpVarType_Direct,
   4
};

/* Variables and Functions */

rpOneOfSeveral gFixedSingleSelect = 2;
```

Sample generated HTML

```
<SELECT NAME=“Fixed Single Select” SIZE=“4”>
<OPTION VALUE=1>Item 1
<OPTION VALUE=2 SELECTED>Item 2
<OPTION VALUE=3>Item 3
<OPTION VALUE=4>Item 4
<OPTION VALUE=5>Item 5
</SELECT>
```

<SELECT><OPTION>
(Fixed List — Multiple Select)

HTML comment tag

```
<!-- RpFormMultiSelect NAME=n Size=s Dynamic -->
<!-- RpMultiSelectOption VALUE=v RpGetType=gt RpGetPtr=gp
         RpSetType=st RpSetPtr=sp -->
<!-- RpEndSelect -->
```

These tags are used for creating `<SELECT>` menus where the list of menu choices is known at compile time and multiple choices can be made from the menu.

The `RpFormMultiSelect` tag contains the information for the generation of the `<SELECT MULTIPLE>` tag and for processing the form value. The `Name` keyword specifies the HTML name of the item. The `Size` keyword is used for the `<SELECT>` tag `SIZE` keyword, which controls the size of the menu list. If the keyword is not specified, 1 is used. The `Dynamic` keyword is used if the form element will be used inside a repeat group.

The `RpMultiSelectOption` comment tag provides the specifications of the options for the `<SELECT>` list. The tag uses the `Value` keyword to specify the display string of the option. It uses the functions described as follows to access a Boolean value that determines whether an option is selected or not.

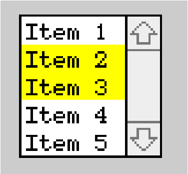
The `RpEndSelect` comment tag is used to terminate the options list of the `<SELECT>` menu.

Similar to checkboxes, an HTML form with a `<SELECT MULTIPLE>` item will submit only the options selected in the list. So, the RomPager engine will set all options in the list to off at the beginning of form processing. Then as the form is processed, the individual options that are selected will be set to on. Both the reset and set operations use the set function by specifying `RpTextType` specified by the comment tag.

Function prototypes

```
GetType = Function
typedef  Boolean (*rpFetchBooleanFuncPtr)(void);

GetType = Complex
typedef  Boolean (*rpFetchBooleanComplexPtr)(void *theServerDataPtr,
         char *theNamePtr, Signed16Ptr theIndexValuesPtr);

SetType = Function
typedef void (*rpStoreBooleanFuncPtr)(Boolean theValue);

SetType = Complex
typedef  void (*rpStoreBooleanComplexPtr)(void *theServerDataPtr,
         Boolean theValue, char *theNamePtr,
         Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
<!-- RpFormMultiSelect NAME=FixedMultiple SIZE=0 -->
    <SELECT MULTIPLE NAME=“FixedMultiple”>
<!-- RpEnd -->
<!-- RpMultiSelectOption VALUE=“Item 1” RpGetType=Direct RpGetPtr=gItem1
RpSetType=Direct RpSetPtr=gItem1 -->
    <OPTION>Item 1
<!-- RpEnd -->
<!-- RpMultiSelectOption VALUE=“Item 2” RpGetType=Direct RpGetPtr=gItem2
RpSetType=Direct RpSetPtr=gItem2 -->
    <OPTION SELECTED>Item 2
<!-- RpEnd -->
<!-- RpMultiSelectOption VALUE=“Item 3” RpGetType=Direct RpGetPtr=gItem3
RpSetType=Direct RpSetPtr=gItem3 -->
    <OPTION SELECTED>Item 3
<!-- RpEnd -->
<!-- RpMultiSelectOption VALUE=“Item 4” RpGetType=Direct RpGetPtr=gItem4
RpSetType=Direct RpSetPtr=gItem4 -->
    <OPTION>Item 4
<!-- RpEnd -->
<!-- RpMultiSelectOption VALUE=“Item 5” RpGetType=Direct RpGetPtr=gItem5
RpSetType=Direct RpSetPtr=gItem5 -->
    <OPTION>Item 5
<!-- RpEnd -->
<!-- RpEndSelect -->
   </SELECT>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormFixedMultiSelect, &PgSample_Item_1 },

/* Structures */

static rpOption_MultiSelect PgSample_Item_1_Option5 = {
   (rpOption_MultiSelectPtr) 0,
   “Item 5”,
   &gItem5,
   &gItem5,
   eRpVarType_Direct,
   eRpVarType_Direct
};

static rpOption_MultiSelect PgSample_Item_1_Option4 = {
   &PgSample_Item_1_Option5,
   “Item 4”,
   &gItem4,
   &gItem4,
   eRpVarType_Direct,
   eRpVarType_Direct
};
```

```
static rpOption_MultiSelect PgSample_Item_1_Option3 = {
   &PgSample_Item_1_Option4,
   “Item 3”,
   &gItem3,
   &gItem3,
   eRpVarType_Direct,
   eRpVarType_Direct
};

static rpOption_MultiSelect PgSample_Item_1_Option2 = {
   &PgSample_Item_1_Option3,
   “Item 2”,
   &gItem2,
   &gItem2,
   eRpVarType_Direct,
   eRpVarType_Direct
};

static rpOption_MultiSelect PgSample_Item_1_Option1 = {
   &PgSample_Item_1_Option2,
   “Item 1”,
   &gItem1,
   &gItem1,
   eRpVarType_Direct,
   eRpVarType_Direct
};

static rpFixedMultiSelectFormItem PgSample_Item_1 = {
   “FixedMultiple”,
   &PgSample_Item_1_Option1,
   0
};
/* Variables and Functions */

static Boolean gItem1 = False;
static Boolean gItem2 = True;
static Boolean gItem3 = True;
static Boolean gItem4 = False;
static Boolean gItem5 = False;
```

Sample generated HTML

```
<SELECT MULTIPLE NAME=“FixedMultiple”>
<OPTION>Item 1
<OPTION SELECTED>Item 2
<OPTION SELECTED>Item 3
<OPTION>Item 4
<OPTION>Item 5
</SELECT>
```

<SELECT><OPTION>
(Variable List — String Values)

HTML comment tag

```
<!–– RpFormVariableSelect NAME=n MULTIPLE RpResetPtr=r
          RpGetType=gt RpGetPtr=gp RpSetType=st RpSetPtr=sp
          Size=s RpTextType=t RpFieldWidth=f Dynamic ––>
```

This tag is used for creating <SELECT> menus where the list of menu choices is created at run time. The values retrieved by the access functions are used to create the list of choices and the selected values are returned to the storage functions when the form is processed.

The RpFormVariableSelect tag contains the information for the generation of the <SELECT> and <OPTION> tags and for processing the form value. The Name keyword specifies the HTML name of the item. The Size keyword is used for the <SELECT> tag SIZE keyword, which controls the size of the menu list. If the keyword is not specified, 1 is used. The MULTIPLE keyword specifies that the MULTIPLE keyword is to be generated for the <SELECT> tag, which allows multiple options to be selected from the list. If the MULTIPLE keyword is present, the RpResetPtr keyword must also be supplied to specify a pointer to a function that is called to reset the options. The RpFieldWidth keyword is used for conversions when RpTextType is ASCIIFixed, Hex, HexColonForm, or DotForm to specify the number of characters that should appear on the page for the field. The Dynamic keyword is used if the form element is used inside a repeat group.

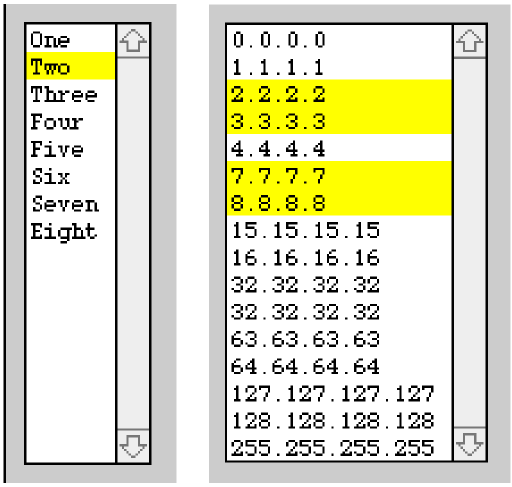
The engine calls the Get function once to get the value to display for each <OPTION> string until the function returns a null pointer to signal that there are no more <OPTION> strings. The engine increments theItemNumber each time the Get function is called. The values returned to the Set function during form processing are the values that the Get function supplied for the selected options.

Function prototypes

```
GetType = Function
typedef  void * (*rpFetchOptionFuncPtr)(Unsigned8 theItemNumber,
               Boolean *theOptionSelectedFlag);

GetType = Complex
typedef  void * (*rpFetchOptionComplexPtr)(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr,
               Unsigned8 theItemNumber,
               Boolean *theOptionSelectedFlag);
Reset Function

typedef  void (*rpResetVarSelectPtr)(void *theServerDataPtr);
```

The prototypes used for setting the value of the selected option are the same as the SetType prototypes used by the <INPUT TYPE=TEXT> tags. The actual prototype used depends on the value of the RpSetType and RpTextType keywords.

Sample



Commented HTML

```
<!-- RpFormVariableSelect NAME=VariableSingle SIZE=16 RpTextType=ASCII
   RpGetType=Function RpGetPtr=VariableSingleGet
   RpSetType=Function RpSetPtr=VariableSinglePut -->
    <SELECT NAME=“VariableSingle” SIZE=“16”>
    <OPTION>One
    <OPTION SELECTED>Two
    <OPTION>Three
    <OPTION>Four
    <OPTION>Five
    <OPTION>Six
    <OPTION>Seven
    <OPTION>Eight
    </SELECT>
<!-- RpEnd -->

<!-- RpFormVariableSelect NAME=VariableMultiple MULTIPLE SIZE=0
   RpFieldWidth=15 RpTextType=DotForm RpResetPtr=VariableMultipleReset
   RpGetType=Function RpGetPtr=VariableMultipleGet
   RpSetType=Function RpSetPtr=VariableMultiplePut -->
    <SELECT MULTIPLE NAME=“VariableMultiple”>
    <OPTION>0.0.0.0
    <OPTION>1.1.1.1
    <OPTION SELECTED>2.2.2.2
    <OPTION SELECTED>3.3.3.3
    <OPTION>4.4.4.4
    <OPTION SELECTED>7.7.7.7
    <OPTION SELECTED>8.8.8.8
    <OPTION>15.15.15.15
    <OPTION>16.16.16.16
    <OPTION>32.32.32.32
    <OPTION>32.32.32.32
    <OPTION>63.63.63.63
    <OPTION>64.64.64.64
    <OPTION>127.127.127.127
    <OPTION>128.128.128.128
```

```
        <OPTION>255.255.255.255
        </SELECT>
    <!-- RpEnd -->
```

Internal structures

```
    /* Element List Items */
       { eRpItemType_FormVariableSingleSelect,   &PgSample_Item_1 },
       { eRpItemType_FormVariableMultiSelect, &PgSample_Item_2 },

    /* Structures */

    static rpVariableSelectFormItem PgSample_Item_1 = {
       “VariableSingle”,
       (rpResetVarSelectPtr) 0,
       VariableSingleGet,
       VariableSinglePut,
       eRpVarType_Function,
       eRpVarType_Function,
       eRpTextType_ASCII,
       0,
       16
    };
    static rpVariableSelectFormItem PgSample_Item_2 = {
       “VariableMultiple”,
       VariableMultipleReset,
       VariableMultipleGet,
       VariableMultiplePut,
       eRpVarType_Function,
       eRpVarType_Function,
       eRpTextType_DotForm,
       15,
       0
    };

    /* Variables and Functions */

    #define kVariableSingleCount 8
    #define kVariableMultipleCount   16

    static Unsigned8 gVariableSingleSelect = 2;
    static char *gVariableSingleSelectItems[kVariableSingleCount] = {
       “One”,
       “Two”,
       “Three”,
       “Four”,
       “Five”,
       “Six”,
       “Seven”,
       “Eight”
    };
```

```
typedef  struct {
         Boolean        fSelected;
         Unsigned8      fValue[4];
} VariableMultiSelect, *VariableMultiSelectPtr;

static VariableMultiSelect gVariableMultiSelect[kVariableMultipleCount] = {
   { False, {   0,   0,   0,   0 } },
   { False, {   1,   1,   1,   1 } },
   { True,  {   2,   2,   2,   2 } },
   { True,  {   3,   3,   3,   3 } },
   { False, {   4,   4,   4,   4 } },
   { True,  {   7,   7,   7,   7 } },
   { True,  {   8,   8,   8,   8 } },
   { False, {  15,  15,  15,  15 } },
   { False, {  16,  16,  16,  16 } },
   { False, {  32,  32,  32,  32 } },
   { False, {  32,  32,  32,  32 } },
   { False, {  63,  63,  63,  63 } },
   { False, {  64,  64,  64,  64 } },
   { False, { 127, 127, 127, 127 } },
   { False, { 128, 128, 128, 128 } },
   { False, { 255, 255, 255, 255 } }
};
static void *VariableSingleGet(Unsigned8 theItemNumber,
   Boolean *theOptionSelectedFlag) {
   *theOptionSelectedFlag = (theItemNumber + 1) == gVariableSingleSelect;
   return (theItemNumber < kVariableSingleCount) ?
   gVariableSingleSelectItems[theItemNumber] : (void *) 0;
}
static void VariableSinglePut(char *theValuePtr) {
   int theItemsIndex;
   theItemsIndex = 0;
   while (theItemsIndex < kVariableSingleCount &&
      strcmp(theValuePtr, gVariableSingleSelectItems[theItemsIndex])) {
   theItemsIndex += 1;
   }
   gVariableSingleSelect = theItemsIndex + 1;
   return;
}
static void *VariableMultipleGet(Unsigned8 theItemNumber,
      Boolean *theOptionSelectedFlag) {
      *theOptionSelectedFlag = gVariableMultiSelect[theItemNumber].fSelected;
   return (theItemNumber < kVariableMultipleCount) ?
      &gVariableMultiSelect[theItemNumber].fValue : (void *) 0;
}
static void VariableMultipleReset(void *theServerDataPtr) {
   int   theIndex;
   VariableMultiSelectPtr  theVariableMultiSelectPtr;
   theVariableMultiSelectPtr = gVariableMultiSelect;
   for (theIndex = 0; theIndex < kVariableMultipleCount; theIndex += 1) {
      theVariableMultiSelectPtr->fSelected = False;
      theVariableMultiSelectPtr += 1;
   }
   return;

static void VariableMultiplePut(char *theValuePtr) {
   int   theIndex;
   VariableMultiSelectPtr  theVariableMultiSelectPtr;
   Boolean  theFoundFlag;
   theVariableMultiSelectPtr = gVariableMultiSelect;
   theIndex = 0;
   theFoundFlag = False;
   while (!theFoundFlag && theIndex < kVariableMultipleCount) {
      if (!memcmp(theValuePtr, theVariableMultiSelectPtr->fValue,
               sizeof(theVariableMultiSelectPtr->fValue))) {
         theVariableMultiSelectPtr->fSelected = True;
         theFoundFlag = True;
      }
      else {
         theVariableMultiSelectPtr += 1;
         theIndex                  += 1;
      }
   }
   return;
}
```

Sample generated HTML

```
<SELECT NAME=“VariableSingle” SIZE=“16”>
<OPTION>One
<OPTION SELECTED>Two
<OPTION>Three
<OPTION>Four
<OPTION>Five
<OPTION>Six
<OPTION>Seven
<OPTION>Eight
</SELECT>
<SELECT MULTIPLE NAME=“VariableMultiple”>
<OPTION>0.0.0.0
<OPTION>1.1.1.1
<OPTION SELECTED>2.2.2.2
<OPTION SELECTED>3.3.3.3
<OPTION>4.4.4.4
<OPTION SELECTED>7.7.7.7
<OPTION SELECTED>8.8.8.8
<OPTION>15.15.15.15
<OPTION>16.16.16.16
<OPTION>32.32.32.32
<OPTION>32.32.32.32
<OPTION>63.63.63.63
```

```
<OPTION>64.64.64.64
<OPTION>127.127.127.127
<OPTION>128.128.128.128
<OPTION>255.255.255.255
</SELECT>
```

<SELECT><OPTION>
(Variable List — Item Number Values)

HTML comment tag

```
<!–– RpFormVarValueSelect NAME=n MULTIPLE RpResetPtr=r
         RpGetType=gt RpGetPtr=gp RpSetType=st RpSetPtr=sp
         Size=s RpTextType=t RpFieldWidth=f Dynamic ––>
```

This tag is used for creating <SELECT> menus where the list of menu choices is created at run time. The values retrieved by the access functions are used to create the list of choices and the item number of the choice is returned to the storage function when the form is processed.

The RpFormVarValueSelect tag contains the information for the generation of the <SELECT> and <OPTION> tags and for processing the form value. The Name keyword specifies the HTML name of the item. The Size keyword is used for the <SELECT> tag SIZE keyword, which controls the size of the menu list. If the keyword is not specified, 1 is used. The MULTIPLE keyword specifies that the MULTIPLE keyword is to be generated for the <SELECT> tag, which allows multiple options to be selected from the list. If the MULTIPLE keyword is present, the RpResetPtr keyword must also be supplied to specify a pointer to a function that is called to reset the options. The RpFieldWidth keyword is used for conversions when RpTextType is ASCIIFixed, Hex, HexColonForm, or DotForm to specify the number of characters that should appear on the page for the field. The Dynamic keyword is used if the form element is used inside a repeat group.

The engine calls the application’s Get function to retrieve the label to display for the <OPTION> string. The Get function will be called once for each <OPTION> string until the function returns a null pointer to signal that there are no more <OPTION> strings. The engine increments theItemNumber each time the Get function is called. When the application’s Get function returns the label to display for the <OPTION> string, it also returns an arbitrary unsigned 32-bit value that will be associated with the display string. This unsigned 32-bit value can be a numeric, a structure pointer, or any other value that the application finds useful. The engine encodes the value using 8 hex digits in the VALUE keyword of the <OPTION> tag.

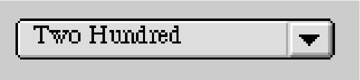
When the browser submits the form, the engine takes the 8 hex digits and turns it back into the unsigned 32 bit value. In turn, this value is passed to the Set function during form processing for the application to handle. With some machines, the encoded value can appear to be byte-swapped, but the value will be reassembled correctly at form processing time. In this way, the value returned to the Set function is the unsigned 32-bit value associated with the <OPTION> display string rather than the display string. This capability can be used to simplify the form processing routines used for variable list menus.

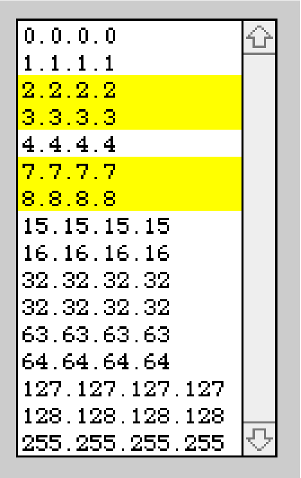
Function prototypes

```
GetType = Function
typedef void * (*rpFetchOptionValueFuncPtr)(Unsigned8 theItemNumber,
                  Boolean *theOptionSelectedFlag,
                  Unsigned32Ptr theValuePtr);

GetType = Complex

typedef void * (*rpFetchOptionValueComplexPtr)(void *theServerDataPtr,
                  char *theNamePtr,
                  Signed16Ptr theIndexValuesPtr,
                  Unsigned8 theItemNumber,
                  Boolean *theOptionSelectedFlag,
                  Unsigned32Ptr theValuePtr);
```

```
Reset Function

typedef void (*rpResetVarSelectPtr)(void *theServerDataPtr);

SetType = Function

typedef void (*rpStoreUnsigned32FuncPtr)(Unsigned32 theValue);

SetType = Complex

typedef void (*rpStoreUnsigned32ComplexPtr)(void *theServerDataPtr,
        Unsigned32 theValue, char *theNamePtr, Signed16Ptr theIndexValuesPtr);
```

Sample





Commented HTML

```
<!-- RpFormVarValueSelect NAME=VariableValueSingle SIZE=1 RpTextType=ASCII
         RpGetType=Function RpGetPtr=VariableValueSingleGet
         RpSetType=Function RpSetPtr=VariableValueSinglePut -->
    <SELECT NAME=“VariableValueSingle” SIZE=“1”>
    <OPTION VALUE=00000001>One Hundred
    <OPTION VALUE=00000002 SELECTED>Two Hundred
    <OPTION VALUE=00000003>Three Hundred
    <OPTION VALUE=00000004>Four Hundred
    <OPTION VALUE=00000005>Five Hundred
    </SELECT>
<!-- RpEnd -->
<!-- RpFormVarValueSelect NAME=VariableValueMultiple MULTIPLE SIZE=0
   RpFieldWidth=15 RpTextType=DotForm RpResetPtr=VarValueMultipleReset
   RpGetType=Function RpGetPtr=VarValueMultipleGet
   RpSetType=Function RpSetPtr=VarValueMultiplePut -->
    <SELECT MULTIPLE NAME=“VariableValueMultiple”>
    <OPTION VALUE=00000000>0.0.0.0
    <OPTION VALUE=00000001>1.1.1.1
    <OPTION VALUE=00000002 SELECTED>2.2.2.2
    <OPTION VALUE=00000003 SELECTED>3.3.3.3
    <OPTION VALUE=00000004>4.4.4.4
    <OPTION VALUE=00000005 SELECTED>7.7.7.7
    <OPTION VALUE=00000006 SELECTED>8.8.8.8
    <OPTION VALUE=00000007>15.15.15.15
    <OPTION VALUE=00000008>16.16.16.16
    <OPTION VALUE=00000009>32.32.32.32
    <OPTION VALUE=0000000a>32.32.32.32
    <OPTION VALUE=0000000b>63.63.63.63
    <OPTION VALUE=0000000c>64.64.64.64
    <OPTION VALUE=0000000d>127.127.127.127
    <OPTION VALUE=0000000e>128.128.128.128
    <OPTION VALUE=0000000f>255.255.255.255
    </SELECT>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormVarValueSingleSelect,   &PgSample_Item_1 },
   { eRpItemType_FormVarValueMultiSelect, &PgSample_Item_2 },

/* Structures */

static rpVariableSelectFormItem PgSample_Item_1 = {
   “VariableValueSingle”,
   (rpResetVarSelectPtr) 0,
   VariableValueSingleGet,
   VariableValueSinglePut,
   eRpVarType_Function,
   eRpVarType_Function,
   eRpTextType_ASCII,
   0,
   1
};
static rpVariableSelectFormItem PgSample_Item_2 = {
   “VariableValueMultiple”,
   VarValueMultipleReset,
   VarValueMultipleGet,
   VarValueMultiplePut,
   eRpVarType_Function,
   eRpVarType_Function,
   eRpTextType_DotForm,
   15,
   0
};

/* Variables and Functions */

#define kVariableValueSingleCount   5
#define kVariableMultipleCount   16

static Unsigned8 gVarValueSingleSelect = 2;
static char *gVarValueSingleSelectItems[kVariableValueSingleCount] = {
   “One Hundred”,
   “Two Hundred”,
   “Three Hundred”,
   “Four Hundred”,
   “Five Hundred”,
};

typedef struct {
   Boolean  fSelected;
   Unsigned8   fValue[4];
} VariableMultiSelect, *VariableMultiSelectPtr;

static VariableMultiSelect gVariableMultiSelect[kVariableMultipleCount] = {
   { False, {   0,   0,   0,   0 } },
   { False, {   1,   1,   1,   1 } },
   { True,  {   2,   2,   2,   2 } },
   { True,  {   3,   3,   3,   3 } },
   { False, {   4,   4,   4,   4 } },
   { True,  {   7,   7,   7,   7 } },
   { True,  {   8,   8,   8,   8 } },
   { False, {  15,  15,  15,  15 } },
   { False, {  16,  16,  16,  16 } },
   { False, {  32,  32,  32,  32 } },
   { False, {  32,  32,  32,  32 } },
   { False, {  63,  63,  63,  63 } },
   { False, {  64,  64,  64,  64 } },
   { False, { 127, 127, 127, 127 } },
   { False, { 128, 128, 128, 128 } },
   { False, { 255, 255, 255, 255 } }
};

static void *VariableValueSingleGet(Unsigned8 theItemNumber,
                  Boolean *theOptionSelectedFlag,
                  Unsigned32Ptr theValuePtr) {
   *theOptionSelectedFlag = (theItemNumber + 1) == gVarValueSingleSelect;
   *theValuePtr = theItemNumber + 1;
   return (theItemNumber < kVariableSingleCount) ?
      gVarValueSingleSelectItems[theItemNumber] : (void *) 0;
}

static void VariableValueSinglePut(Unsigned32 theValue) {

   gVarValueSingleSelect = (Unsigned8) theValue;
   return;
}
```

```
static void *VarValueMultipleGet(Unsigned8 theItemNumber,
                  Boolean *theOptionSelectedFlag,
                  Unsigned32Ptr theValuePtr) {
   *theOptionSelectedFlag = gVariableMultiSelect[theItemNumber].fSelected;
   *theValuePtr = theItemNumber;
   return (theItemNumber < kVariableMultipleCount) ?
         &gVariableMultiSelect[theItemNumber].fValue : (void *) 0;
}
static void VarValueMultipleReset(void *theServerDataPtr) {
   int            theIndex;
   VariableMultiSelectPtr  theVariableMultiSelectPtr;

   theVariableMultiSelectPtr = gVariableMultiSelect;
   for (theIndex = 0; theIndex < kVariableMultipleCount; theIndex += 1) {
      theVariableMultiSelectPtr->fSelected = False;
      theVariableMultiSelectPtr += 1;
   }
   return;
}
static void VarValueMultiplePut(Unsigned32 theValue) {
   gVariableMultiSelect[theValue].fSelected = True;
   return;
}
```

Sample generated HTML

```
<SELECT NAME=“VariableValueSingle” SIZE=“1”>
<OPTION VALUE=00000001>One Hundred
<OPTION VALUE=00000002 SELECTED>Two Hundred
<OPTION VALUE=00000003>Three Hundred
<OPTION VALUE=00000004>Four Hundred
<OPTION VALUE=00000005>Five Hundred
</SELECT>
<SELECT MULTIPLE NAME=“VariableValueMultiple”>
<OPTION VALUE=00000000>0.0.0.0
<OPTION VALUE=00000001>1.1.1.1
<OPTION VALUE=00000002 SELECTED>2.2.2.2
<OPTION VALUE=00000003 SELECTED>3.3.3.3
<OPTION VALUE=00000004>4.4.4.4
<OPTION VALUE=00000005 SELECTED>7.7.7.7
<OPTION VALUE=00000006 SELECTED>8.8.8.8
<OPTION VALUE=00000007>15.15.15.15
<OPTION VALUE=00000008>16.16.16.16
<OPTION VALUE=00000009>32.32.32.32
<OPTION VALUE=0000000a>32.32.32.32
<OPTION VALUE=0000000b>63.63.63.63
<OPTION VALUE=0000000c>64.64.64.64
<OPTION VALUE=0000000d>127.127.127.127
<OPTION VALUE=0000000e>128.128.128.128
<OPTION VALUE=0000000f>255.255.255.255
</SELECT>
```

<TEXTAREA>

HTML comment tags

```
<!-- RpFormTextArea NAME=n RpGetType=gt RpGetPtr=gp ROWS=r COLS=c -->

<!-- RpFormTextAreaBuf NAME=n RpGetType=gt RpGetPtr=gp
         RpSetType=gt RpSetPtr=gp ROWS=r COLS=c -->
```

These tags are used in a form to define `<TEXTAREA>` elements. The HTML name of the text area is specified by the `NAME` keyword. The `ROWS` keyword is used for the `<TEXTAREA>` tag `ROWS` keyword. If the keyword is not specified, 1 is used. The `COLS` keyword is used for the `<TEXTAREA>` tag `COLS` keyword. If the keyword is not specified, 40 is used.

The `RpFormTextArea` tag is used to build a display-only text area. The `RpGetType` keyword value must be either `Function` or `Complex` and the `RpGetPtr` keyword points to a function that retrieves one line at a time. The RomPager engine calls the Get function once to get each display line of the text area until the function returns a null pointer to signal that there are no more display lines. The engine increments `theItemNumber` each time the Get function is called.

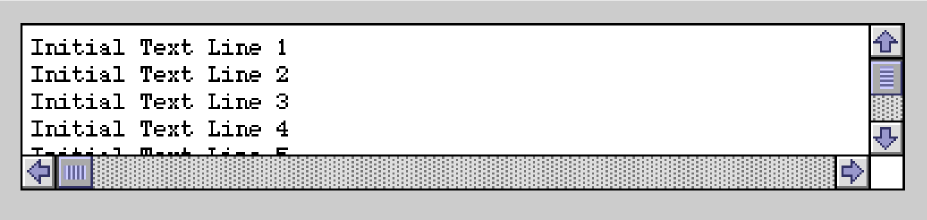
The `RpFormTextAreaBuf` tag is used to build a text area by using a display buffer. The `RpGetType` and `RpSetType` keywords value must be either `Function` or `Complex`. The `RpGetPtr` and `RpSetPtr` keywords point to functions that access and store the complete text area buffer.

Function prototypes

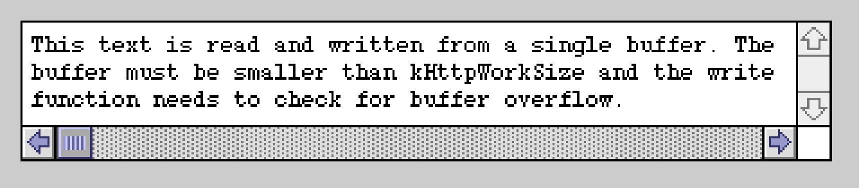
```
GetType = Function - RpFormTextArea

typedef char * (*rpFetchTextFuncPtr)(Unsigned16 theItemNumber);

GetType = Complex - RpFormTextArea

typedef char * (*rpFetchTextComplexPtr)(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr,
               Unsigned16 theItemNumber);
GetType = Function - RpFormTextAreaBuf

typedef char * (*rpFetchBytesFuncPtr)(void);

GetType = Complex - RpFormTextAreaBuf

typedef char * (*rpFetchBytesComplexPtr)(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr);
SetType = Function - RpFormTextAreaBuf

typedef void (*rpStoreAsciiTextFuncPtr)(char *theValuePtr);

SetType = Complex - RpFormTextAreaBuf
```

```
typedef void   (*rpStoreAsciiTextComplexPtr)(void *theServerDataPtr,
               char *theValuePtr,
               char * theNamePtr,
               Signed16Ptr theIndexValuesPtr);
```

Sample



Initial Text Line 1
Initial Text Line 2
Initial Text Line 3
Initial Text Line 4



This text is read and written from a single buffer. The
buffer must be smaller than kHttpWorkSize and the write
function needs to check for buffer overflow.

Commented HTML

```
<!-- RpFormTextArea NAME=TextArea ROWS=4 COLS=60
   RpGetType=Function RpGetPtr=TextAreaGet -->
    <TEXTAREA NAME=“TextArea” ROWS=“4” COLS=“60”>
    Initial Text Line 1
    Initial Text Line 2
    Initial Text Line 3
    Initial Text Line 4
    Initial Text Line 5
    Initial Text Line 6
    </TEXTAREA>
<!-- RpEnd -->

<!-- RpFormTextAreaBuf NAME=TextArea ROWS=3 COLS=55
   RpGetType=Function RpGetPtr=TextAreaBufGet
   RpSetType=Function RpSetPtr=TextAreaBufPut -->
    <TEXTAREA NAME=“TextArea” ROWS=“3” COLS=“55”>
    This text is read and written from a single buffer. The
    buffer must be smaller than kHttpWorkSize and the write
    function needs to check for buffer overflow.
    </TEXTAREA>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_FormTextArea,      &PgSample_Item_1 },
   { eRpItemType_FormTextAreaBuf,   &PgSample_Item_2 },

/* Structures */

static rpTextAreaFormItem PgSample_Item_1 = {
   “TextArea”,
   TextAreaGet,
   (void *) 0,
   eRpVarType_Function,
   eRpVarType_Direct,
   4,
   60
};
static rpTextAreaFormItem PgSample_Item_2 = {
   “TextArea”,
   TextAreaBufGet,
   TextAreaBufPut,
   eRpVarType_Function,
   eRpVarType_Function,
   3,
   55
};
/* Variables and Functions */

#define kTextAreaCount        7
#define kTextAreaBufferSize      200

static char *TextArea[kTextAreaCount] = {
   “Initial Text Line 1”,
   “Initial Text Line 2”,
   “Initial Text Line 3”,
   “Initial Text Line 4”,
   “Initial Text Line 5”,
   “Initial Text Line 6”
};
static char TextAreaBufBuffer[kTextAreaBufferSize] =
    “This text is read and written from a single buffer. The\n”
    “buffer must be smaller than kHttpWorkSize and the write\n”
    “function needs to check for buffer overflow.”;

char *TextAreaGet(Unsigned16 theItemNumber) {
   return TextArea[theItemNumber];
}

char *TextAreaBufGet(void) {
   return TextAreaBufBuffer;
}
```

```
void TextAreaBufPut(char *theValue) {
   Unsigned16  theInputSize;

   theInputSize = strlen(theValue);
   if (theInputSize > kTextAreaBufferSize) {
      *(theValue + kTextAreaBufferSize - 1) = ‘\0’;
   }
   strcpy(TextAreaBufBuffer, theValue);

}
```

Sample generated HTML

```
<TEXTAREA NAME=“TextArea” ROWS=“4” COLS=“60”>
Initial Text Line 1
Initial Text Line 2
Initial Text Line 3
Initial Text Line 4
Initial Text Line 5
Initial Text Line 6
</TEXTAREA>

<TEXTAREA NAME=“TextArea” ROWS=“3” COLS=“55”>
This text is read and written from a single buffer. The
buffer must be smaller than kHttpWorkSize and the write
function needs to check for buffer overflow.
</TEXTAREA>
```

Item groups

At times it may be useful to group a series of elements that are used on more than one page.

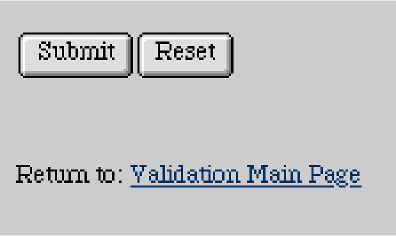
HTML comment tags

```
<!–– RpItemGroup RpIdentifier=i ––>

<!–– RpLastItemInGroup ––>
```

These tags define a group of items that are used more than once. By grouping common sets of items, a significant amount of page storage memory can be saved. Any other page element including static text, dynamic text and form elements can be placed in an element group. The `RpItemGroup` tag defines the beginning of a common element group and the `RpLastItemInGroup` tag defines the end of the common group.

You can place element groups within element groups for complex structures. The nesting level is 5. No checking for recursion is done, so it is possible to set up items groups that point to themselves thus generating an infinite amount of HTML and browser user dismay.

Sample



Return to: Validation Main Page

Commented HTML

```
<!–– RpItemGroup RpIdentifier=PgEndItems ––>
   <!-- RpDataZeroTerminated -->
   <BR>
   <BR>
   <!-- RpEnd -->
   <!-- RpFormInput TYPE=SUBMIT NAME=Submit VALUE=Submit -->
   <INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit”>
   <!-- RpEnd -->
   <!-- RpFormInput TYPE=RESET VALUE=Reset -->
   <INPUT TYPE=“RESET” VALUE=“Reset”>
   <!-- RpEnd -->

   <!-- RpDataZeroTerminated -->
   <P>&nbsp;</P>
   Return to: <A HREF=“ValidationSuite”>Validation Main Page</A>
   </FORM>
   </BODY>
   </HTML>
   <!-- RpEnd -->
<!–– RpLastItemInGroup ––>
```

Internal structures

```
/* Element List Items */
   { eRpItemType_ItemGroup,      &PgEndItems },

/* Structures */

static rpItem PgEndItems[] = {
   { eRpItemType_DataZeroTerminated,   &PgItemGroup_1 },
   { eRpItemType_FormSubmit,     &PgItemGroup_2 },
   { eRpItemType_FormReset,      &PgItemGroup_3 },
   { eRpItemType_DataZeroTerminated,   &PgItemGroup_4 },
   { eRpItemType_LastItemInList }
};

static char PgItemGroup_1[] =
   C_oBR C_oBR;

static rpButtonFormItem PgItemGroup_2 = {
   “Submit”
};
```

```
static rpButtonFormItem PgItemGroup_3 = {
   “Reset”
};

static char PgItemGroup_4[] =
   C_oP_NBSP_xP
   “Return to: C_oANCHOR_HREF “ValidationSuite\”>Validation Main Page” C_xANCHOR
   C_xFORM C_xBODY_xHTML;
```

Sample generated HTML

```
<BR>
<BR>
<INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit”>
<INPUT TYPE=“RESET” VALUE=“Reset”>
<P>&nbsp;</P>
Return to: <A HREF=“ValidationSuite”>Validation Main Page</A>
</FORM>
</BODY>
</HTML>
```

Dynamic HTML

Using the RomPager dynamic HTML capability allows a page to display different HTML at different times depending on the current values in the device. This can be useful for status message display and dynamic table creation.

HTML comment tag

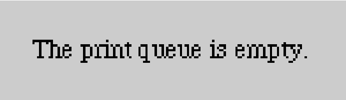
```
<!-- RpDynamicDisplay RpItemCount=n RpGetType=gt RpGetPtr=gp RpGroupPtr=g -->
<!-- RpLastItemInGroup -->
```

The `RpDynamicDisplay` tag is used to generate different HTML depending on the state of an internal variable accessed by the Get function. The Get function returns a variable that is used as an index into a group of items, thereby choosing which HTML to generate. The `RpItemCount` keyword specifies the number of elements in the display list and is used for internal validation of the value returned by the Get function. The group of display list items can be specified by items that follow the `RpDynamicDisplay` tag with the `RpLastItemInGroup` tag ending the group. If the `RpGroupPtr` keyword is present, it specifies the identifier of a previously defined display list group of items.

Function prototypes

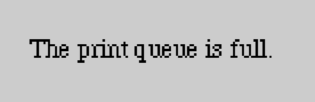
```
GetType = Function
typedef Unsigned8 (*rpFetchIndexFuncPtr)(void);
GetType = Complex
typedef Unsigned8 (*rpFetchIndexComplexPtr)(void *theServerDataPtr,
                     Signed16Ptr theIndexValuesPtr);
```

Samples



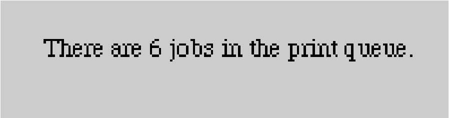
The print queue is empty.

or



The print queue is full.

or



There are 6 jobs in the print queue.

Commented HTML

```
<!–– RpDynamicDisplay RpItemCount=3
   RpGetType=Function RpGetPtr=GetPrintQueueStatus ––>
<!-- RpDataZeroTerminated -->
    <BR>The print queue is empty.<BR>
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
    <BR>The print queue is full.<BR>
<!-- RpEnd -->
   <!–– RpItemGroup ––>
   <!-- RpDataZeroTerminated -->
        <BR>There are
   <!-- RpEnd -->
   <!–– RpDisplayText RpGetPtr=PrintQueueCount RpTextType=Unsigned16 ––>
   6
   <!-- RpEnd -->
   <!-- RpDataZeroTerminated -->
      jobs in the print queue.<BR>
   <!-- RpEnd -->
   <!–– RpLastItemInGroup ––>
<!–– RpLastItemInGroup ––>
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DynamicDisplay,    &PgSample_Item_1 },
/* Structures */

static rpDynamicDisplayItem PgSample_Item_1 = {
   GetPrintQueueStatus,
   eRpVarType_Function,
   3,
   PgSample_Item_1_Group
};
```

```
static rpItem PgSample_Item_1_Group[] = {
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },
   { eRpItemType_ItemGroup,      &PgSample_Item_4 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_2[] =
   C_oBR “The print queue is empty.” C_oBR;

static char PgSample_Item_3[] =
   C_oBR “The print queue is full.” C_oBR;

static rpItem PgSample_Item_4[] = {
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_5 },
   { eRpItemType_DisplayText,    &PgSample_Item_6 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_7 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_5[] =
   C_oBR “There are “;

static rpTextDisplayItem PgSample_Item_6 = {
   &PrintQueueCount,
   eRpVarType_Direct,
   eRpTextType_Unsigned16,
   20
};

static char PgSample_Item_7[] =
   “ jobs in the print queue.”
   C_oBR;

/* Variables and Functions */

#define kMaxPrintQueueCount      300

Unsigned8 GetPrintQueueStatus(void) {

   if (PrintQueueCount == 0) {
      return 0;
   }
   else if (PrintQueueCount >= kMaxPrintQueueCount) {
      return 1;
   }
   else {
      return 2;
   }
}
```

Sample generated HTML

```
<BR>The print queue is empty.<BR>
```

or

```
<BR>The print queue is full.<BR>
```

or

```
<BR>There are 6 jobs in the print queue.<BR>
```

Repeat groups

Groups of items such as those that define table rows and columns can be defined once and repeated as necessary to generate a page. Using repeat groups to generate common sets of page items can lead to substantial page storage savings.

HTML comment tags

```
<!–– RpRepeatGroup RpStart=s RpLimit=l RpIncrement=inc RpGroupPtr=g ––>

<!–– RpRepeatGroupDynamic RpFunctionPtr=fp RpGroupPtr=gp RpMaxItems=m ––>

<!–– RpRepeatGroupWhile RpFunctionPtr=fp RpGroupPtr=gp RpMaxItems=m ––>
```

The `RpRepeatGroup` tag is used to define a repeating group of items where the parameters to define the repeat loop are known at compile time. This item provides the equivalent of a “for” loop. The `RpStart`, `RpLimit`, and `RpIncrement` keywords specify the parameters of the “for” loop. If the `RpIncrement` keyword is not specified, 1 is used.

The `RpRepeatGroupDynamic` tag is used to define a repeating group of items where the parameters to define the repeat loop are determined at run time at the beginning of the loop. This item provides the equivalent of a “for” loop with dynamic initialization. The `RpFunctionPtr` keyword specifies a function that returns the start, limit and increment values for the repeat group.

The `RpRepeatGroupWhile` tag is used to define a repeating group of items where the parameters to define the repeat loop are determined at run time. This item provides the equivalent of a “while” loop. The `RpFunctionPtr` keyword specifies a function that indicates when the looping is to terminate. The function is passed a pointer to a location it can use as a loop index. The function is also passed a pointer to an arbitrary value that the device routine can use. Initially, the index value is passed as 0 and the arbitrary value is passed to the device routine as a (void *) 0. On subsequent calls, the index and the arbitrary value passed to the device routine will be whatever the device passed back on the previous call. When the device routine returns a value of a (void *) 0, it signals to the RomPager engine that the loop is complete.

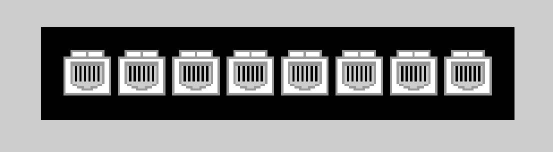
The group of items to repeat can be specified by items that follow the repeat group tag with the `RpLastItemInGroup` tag ending the group. If the `RpGroupPtr` keyword is present, it specifies the identifier of a previously defined group of repeat items.

If the Structure Access form of variable retrieval is used, the `RpMaxItems` keyword is used to tell the PBuilder compiler how much room to reserve in the structures it creates. See the Structured Access section for more details.

Function prototypes

```
RpRepeatGroupDynamic

typedef void (*rpDynamicRepeatFuncPtr)(void *theServerDataPtr,
                     Signed16Ptr theStart,
                     Signed16Ptr theLimit,
                     Signed16Ptr theIncrement);
RpRepeatGroupWhile

typedef void (*rpRepeatWhileFuncPtr)(void *theServerDataPtr,
                     Signed16Ptr theIndexPtr,
                     void ** theRepeatGroupValuePtr);
```

Sample



Commented HTML

```
<!-- RpDataZeroTerminated -->
   <TABLE BORDER=“0” CELLPADDING = “0” CELLSPACING= “0” BGCOLOR=#000000>
   <TR>
<!-- RpEnd -->
<!–– RpRepeatGroup RpStart=1 RpLimit=8 RpIncrement=1 ––>
   <!-- RpDataZeroTerminated -->
      <TD><IMG SRC=“/Images/10BT_Port”></TD>
   <!-- RpEnd -->
<!–– RpLastItemInGroup ––>
<!-- RpDataZeroTerminated -->
   </TR></TABLE>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_RepeatGroup,       &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },

/* Structures */

static char PgSample_Item_1[] =
   C_oTABLE_BORDER “\”0\”“ C_CELLPADDING “ \”0\”“ C_CELLSPACING “ \”0\” “
   “BGCOLOR=#000000>\n”
   C_oTR “\n”;
```

```
static rpRepeatGroupItem PgSample_Item_2 = {
   1,
   8,
   1,
   PgSample_Item_2_Group
};

static char PgSample_Item_3[] =
   C_xTR
   C_xTABLE;

static rpItem PgSample_Item_2_Group[] = {
   { eRpItemType_DataZeroTerminated, &PgSample_Item_4 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_4[] =
   C_oTD C_oIMG_SRC “/Images/10BT_Port\”>“
   C_xTD;
```

Commented HTML

```
   <!-- RpDataZeroTerminated -->
      <TABLE BORDER=“0” CELLPADDING = “0” CELLSPACING= “0” BGCOLOR=#000000>
      <TR>
   <!-- RpEnd -->
   <!–– RpRepeatGroupDynamic RpFunctionPtr=GetRepeatGroupLimits ––>
      <!-- RpDataZeroTerminated -->
         <TD><IMG SRC=“/Images/10BT_Port”></TD>
      <!-- RpEnd -->
   <!–– RpLastItemInGroup ––>
   <!-- RpDataZeroTerminated --></TR></TABLE><!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_RepeatGroupDynamic,   &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },

/* Structures */

static char PgSample_Item_1[] =
   C_oTABLE_BORDER “\”0\”“ C_CELLPADDING “ \”0\”“ C_CELLSPACING “ \”0\” “
   “BGCOLOR=#000000>\n”
   C_oTR “\n”;

static rpRepeatGroupDynItem PgSample_Item_2 = {
   GetRepeatGroupLimits,
   PgSample_Item_2_Group
};
```

```
static char PgSample_Item_3[] =
   C_xTR
   C_xTABLE;

static rpItem PgSample_Item_2_Group[] = {
   { eRpItemType_DataZeroTerminated, &PgSample_Item_4 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_4[] =
   C_oTD C_oIMG_SRC “/Images/10BT_Port\”>“
   C_xTD;

/* Variables and Functions */

void GetRepeatGroupLimits(void *theServerDataPtr, Signed16Ptr theStart,
      Signed16Ptr theLimit, Signed16Ptr theIncrement) {

   *theStart = 1;
   *theLimit = 8;
   *theIncrement = 1;
}
```

Commented HTML

```
<!-- RpDataZeroTerminated -->
   <TABLE BORDER=“0” CELLPADDING = “0” CELLSPACING= “0” BGCOLOR=#000000>
   <TR>
<!-- RpEnd -->
<!–– RpRepeatGroupWhile RpFunctionPtr=TestPortRepeatGroup ––>
   <!-- RpDataZeroTerminated -->
      <TD><IMG SRC=“/Images/10BT_Port”></TD>
   <!-- RpEnd -->
<!–– RpLastItemInGroup ––>
<!-- RpDataZeroTerminated -->
   </TR></TABLE>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_RepeatGroupWhile,  &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },

/* Structures */

static char PgSample_Item_1[] =
   C_oTABLE_BORDER “\”0\”“ C_CELLPADDING “ \”0\”“ C_CELLSPACING “ \”0\” “
   “BGCOLOR=#000000>\n”
   C_oTR “\n”;
```

```
static rpRepeatGroupDynItem PgSample_Item_2 = {
   TestPortRepeatGroup,
   PgSample_Item_2_Group
};

static char PgSample_Item_3[] =
   C_xTR
   C_xTABLE;

static rpItem PgSample_Item_2_Group[] = {
   { eRpItemType_DataZeroTerminated, &PgSample_Item_4 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_4[] =
   C_oTD C_oIMG_SRC “/Images/10BT_Port\”>“
   C_xTD;

/* Variables and Functions */

#define kPortMax = 8

void TestPortRepeatGroup(void *theServerDataPtr,
               Signed16Ptr theIndexPtr,
               void **theRepeatGroupValuePtr) {
   Signed16 theIndex;

   theIndex = *theIndexPtr;
   if (theIndex < kPortMax) {
      *theRepeatGroupValuePtr = thePortName[theIndex];
   }
   else {
      *theRepeatGroupValuePtr = (void *) 0;
   }
   theIndex += 1;
   *theIndexPtr = theIndex;
}
```

Sample generated HTML

```
<TABLE BORDER=“0” CELLPADDING = “0” CELLSPACING= “0” BGCOLOR=#000000><TR>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
<TD><IMG SRC=“/Images/10BT_Port”></TD>
</TR></TABLE>
```

Repeat groups — Index display items

Within repeat groups, it is sometimes useful to display HTML that uses the current index settings of the repeat group.

HTML comment tags

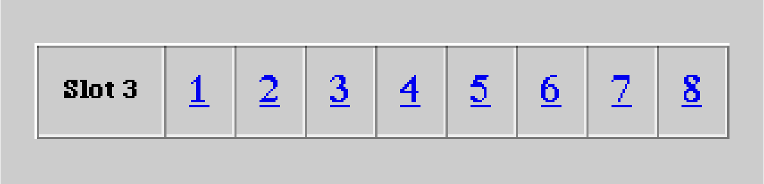
```
<!-- RpIndexDisplay_0 RpText=t -->
<!-- RpIndexDisplay_1 RpText=t -->
<!-- RpIndexDisplay_2 RpText=t -->
<!-- RpIndexDisplay_3 RpText=t -->
<!-- RpIndexDisplay_4 RpText=t -->
<!-- RpIndexDisplay_5 RpText=t -->
<!-- RpQueryDisplay RpText=t -->
```

These tags are used within repeat groups to display the current value of an index. `RpIndexDisplay_0` references the current index for the current repeat group. When nested repeat groups are used, `RpIndexDisplay_1` references the current index for the repeat group that contains the current repeat group. If you have a 3-level repeat group (*x*, *y*, *z*), and you are in the inner most group, `RpIndexDisplay_0` refers to the *z* value, `RpIndexDisplay_1` refers to the *y* value and `RpIndexDisplay_2` refers to the *x* value. The string value of the appropriate index is appended to the text that is defined by the `RpText` keyword.

Index values are generated automatically within repeat groups. They can also be passed to other pages by using query strings. When the RomPager engine receives a URL that has an appended query string of the form “?*x*,*y*,*z*”, it stores the incoming values in the current index array.

The `RpQueryDisplay` tag is used within repeat groups to display a query string that includes the current value of all indices. The query string is appended to the text that is defined by the `RpText` keyword.

Sample



Slot 3	1	2	3	4	5	6	7	8

Commented HTML

```
<!-- RpDataZeroTerminated -->
   <TABLE BORDER=“1” CELLPADDING = “10” CELLSPACING= “0”><TR>
<!-- RpEnd -->
<!-- RpIndexDisplay_0 RpText=“<TD><B>Slot “ -->
   <TD><B>Slot 3
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
   </B></TD>
<!-- RpEnd -->
<!–– RpRepeatGroup RpStart=1 RpLimit=8 RpIncrement=1 ––>
   <!-- RpQueryDisplay RpText=“<TD><FONT SIZE=+2><A HREF=\”/PortInfo” -->
      <TD><FONT SIZE=+2><A HREF=“/PortInfo?3,1
   <!-- RpEnd -->
```

```
<!-- RpIndexDisplay_0 RpText=“\”>“ -->
         “>1
   <!-- RpEnd -->
   <!-- RpDataZeroTerminated -->
         </A></FONT></TD>
   <!-- RpEnd -->
<!–– RpLastItemInGroup ––>
<!-- RpDataZeroTerminated -->
   </TR></TABLE>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_IndexDisplay_0,    &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },
   { eRpItemType_RepeatGroup,       &PgSample_Item_4 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_5 },

/* Structures */

static char PgSample_Item_1[] =
   C_oTABLE_BORDER “\”1\”“ C_CELLPADDING “ \”10\”“ C_CELLSPACING “ \”0\”>“
   C_oTR “\n”;

static char PgSample_Item_2[] =
   C_oTD C_oB “Slot “;

static char PgSample_Item_3[] =
   C_xB
   C_xTD;

static rpRepeatGroupItem PgSample_Item_4 = {
   1,
   8,
   1,
   PgSample_Item_4_Group
};
static char PgSample_Item_5[] =
   C_xTR
   C_xTABLE;

static rpItem PgSample_Item_4_Group[] = {
   { eRpItemType_QueryDisplay,      &PgSample_Item_6 },
   { eRpItemType_IndexDisplay_0,    &PgSample_Item_7 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_8 },
   { eRpItemType_LastItemInList }
};
```

```
static char PgSample_Item_6[] =
   C_oTD C_oFONT_SIZE “+2>“ C_oANCHOR_HREF “/PortInfo”;

static char PgSample_Item_7[] =
   “\”>“;

static char PgSample_Item_8[] =
   C_xANCHOR C_xFONT C_xTD;
```

Sample generated HTML

```
<TABLE BORDER=“1” CELLPADDING = “10” CELLSPACING= “0”><TR>
<TD><B>Slot 3</B></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,1”>1</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,2”>2</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,3”>3</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,4”>4</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,5”>5</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,6”>6</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,7”>7</FONT></TD>
<TD><FONT SIZE=+2><A HREF=“/PortInfo?3,8”>8</FONT></TD>
</TR></TABLE>
```

Repeat groups — Dynamic form items

Within a repeat group, it is often desirable to use common internal routines for processing array values for form items such as the `<INPUT TYPE=`*`xxxx`*`>` items and `<SELECT>` items.

HTML comment tags

```
<!–– RpFormInput TYPE=t NAME=n VALUE=v RpGetPtr=gp RpSetPtr=sp RpTextType=tt
   MaxLength=m Size=s RpItemNumber=in Dynamic RpResetPtr=rp ––>

<!–– RpFormSingleSelect NAME=n Size=s RpGetPtr=gp RpSetPtr=sp Dynamic ––>

<!–– RpFormMultiSelect NAME=n Size=s Dynamic RpResetPtr=rp ––>

<!–– RpFormVariableSelect NAME=n MULTIPLE RpResetPtr=r
   RpGetPtr=gp RpSetPtr=sp Size=s RpTextType=t RpFieldWidth=f Dynamic ––>

<!–– RpFormVarValueSelect NAME=n MULTIPLE RpResetPtr=r
   RpGetPtr=gp RpSetPtr=sp Size=s RpTextType=t RpFieldWidth=f Dynamic ––>
```

If the `Dynamic` keyword is used with any of these tags, the PBuilder compiler generates different internal structures so that the array values can be accessed by common routines. The access functions are assumed to be of type Complex so that the functions can access the index values of the repeat group. Therefore, the `RpGetType` and `RpSetType` keywords need not be specified.

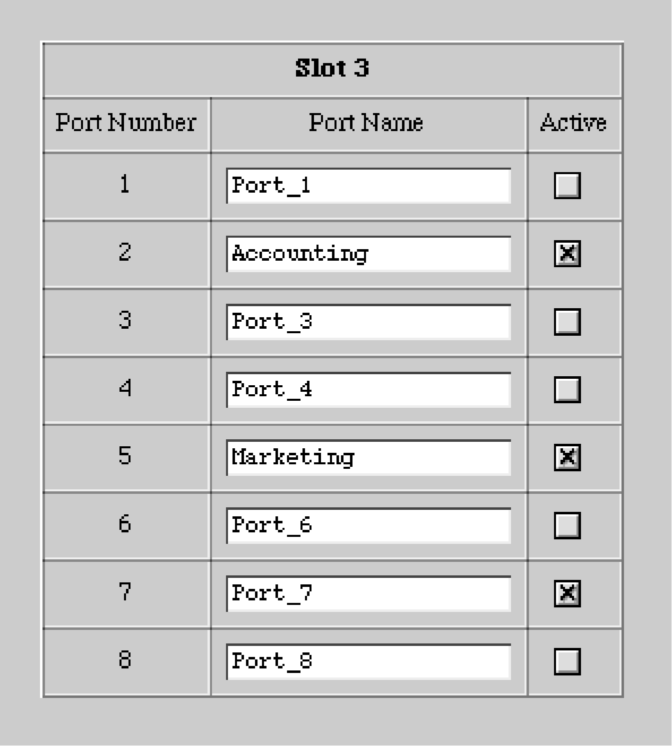
Function prototypes

The function prototypes for the repeat group versions of the form elements are the same as for the regular complex type form elements.

In the case of the checkbox and the multi-select items, an additional function is defined using the `RpResetPtr` keyword. This function is used to reset the entire array prior to repeat group processing.

```
<!–– RpFormInput TYPE=checkbox -->
<!–– RpFormMultiSelect -->

typedef void (*rpResetCheckboxArrayPtr) (void *theServerDataPtr,
                  char *theNamePtr,
                  Signed16Ptr theIndexValuesPtr);
```

Sample



Slot 3		
Port Number	Port Name	Active
1	Port_1	☐
2	Accounting	☒
3	Port_3	☐
4	Port_4	☐
5	Marketing	☒
6	Port_6	☐
7	Port_7	☒
8	Port_8	☐

Commented HTML

```
<!-- RpDataZeroTerminated -->
   <TABLE BORDER=“1” CELLPADDING = “5” CELLSPACING= “0”>
   <TR ALIGN=CENTER><TD COLSPAN=3>
<!-- RpEnd -->
<!-- RpIndexDisplay_0 text=“<B>Slot “ -->
   <B>Slot 3
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
   </B></TD></TR>
   <TR ALIGN=CENTER><TD>Port Number</TD>
   <TD>Port Name</TD><TD>Active</TD></TR>
<!-- RpEnd -->
```

```
<!–– RpRepeatGroup RpStart=1 RpLimit=8 RpIncrement=1 ––>
   <!-- RpIndexDisplay_0 text=“<TR ALIGN=CENTER><TD>“ -->
      <TR ALIGN=CENTER><TD>1
   <!-- RpEnd -->
   <!-- RpDataZeroTerminated RpIdentifier=PgCellSeparator -->
      </TD><TD>
   <!-- RpEnd -->
   <!-- RpEnd -->
   <!-- RpUseId RpId=PgCellSeparator RpItemType=DZT -->
      </TD><TD>
   <!-- RpEnd -->
   <!–– RpFormInput TYPE=checkbox NAME=PortActive
      RpGetPtr=GetPortStatus RpSetPtr=SetPortStatus
      Dynamic RpResetPtr=ResetPortStatus ––>
      <INPUT TYPE=“CHECKBOX” NAME=“PortActive”>
   <!-- RpEnd -->
   <!-- RpDataZeroTerminated -->
      </TD></TR>
   <!-- RpEnd -->
<!–– RpLastItemInGroup ––>
<!-- RpDataZeroTerminated -->
   </TABLE>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_IndexDisplay_0,    &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },
   { eRpItemType_RepeatGroup,       &PgSample_Item_4 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_5 },

/* Structures */

static char PgSample_Item_1[] =
   C_oTABLE_BORDER “\”1\”“ C_CELLPADDING “ \”5\”“ C_CELLSPACING “ \”0\”>\n”
   C_oTR_ALIGN_CENTER “>“ C_oTD_COLSPAN “3>\n”;

static char PgSample_Item_2[] =
   C_oB “Slot “;

static char PgSample_Item_3[] =
   C_xB C_xTD C_xTR C_oTR_ALIGN_CENTER “>“ C_oTD “Port Number” C_xTD
   C_oTD “Port Name” C_xTD C_oTD “Active” C_xTD C_xTR;

static rpRepeatGroupItem PgSample_Item_4 = {
   1,
   8,
   1,
   PgSample_Item_4_Group
};
```

```
static char PgSample_Item_5[] =
   C_xTABLE;

static rpItem PgSample_Item_4_Group[] = {
   { eRpItemType_IndexDisplay_0,    &PgSample_Item_6 },
   { eRpItemType_DataZeroTerminated,   &PgCellSeparator },
   { eRpItemType_FormTextDyn,       &PgSample_Item_7 },
   { eRpItemType_DataZeroTerminated,   &PgCellSeparator },
   { eRpItemType_FormCheckboxDyn,   &PgSample_Item_8 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_9 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_6[] =
   C_oTR_ALIGN_CENTER “>\n”  C_oTD;

char PgCellSeparator[] =
   C_xTD C_oTD;

static rpTextFormItem PgSample_Item_7 = {
   “PortName”,
   GetPortName,
   SetPortName,
   eRpVarType_Complex,
   eRpVarType_Complex,
   eRpTextType_ASCII,
   20,
   20
};

static char PgSample_Item_8[] =
   C_xANCHOR C_xFONT C_xTD;
   C_oBR;

static rpDynCheckboxFormItem PgSample_Item_8 = {
   “PortActive”,
   GetPortStatus,
   SetPortStatus,
   eRpVarType_Complex,
   eRpVarType_Complex,
   ResetPortStatus
};

static char PgSample_Item_9[] =
   C_xTD C_xTR;
/* Variables and Functions */

Boolean thePortStatusArray[4][8];
char thePortNameArray[4][8][21];

char *GetPortName(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr) {
```

```
   Signed16 theSlotIndex = *theIndexValuesPtr;
   Signed16 thePortIndex = *(theIndexValuesPtr + 1);

   return thePortNameArray[theSlotIndex][thePortIndex];
}

extern void SetPortName(void *theServerDataPtr,
               char *theValuePtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr) {
   Signed16 theSlotIndex = *theIndexValuesPtr;
   Signed16 thePortIndex = *(theIndexValuesPtr + 1);

   strcpy (thePortStatusArray[theSlotIndex][thePortIndex], theValuePtr);
}

Boolean GetPortStatus(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr) {
   Signed16 theSlotIndex = *theIndexValuesPtr;
   Signed16 thePortIndex = *(theIndexValuesPtr + 1);

   return thePortStatusArray[theSlotIndex][thePortIndex];
}
void SetPortStatus(void *theServerDataPtr,
               Boolean theValue,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr) {
   Signed16 theSlotIndex = *theIndexValuesPtr;
   Signed16 thePortIndex = *(theIndexValuesPtr + 1);

   thePortStatusArray[theSlotIndex][thePortIndex] = theValue;
}
void ResetPortStatus(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr) {
   Signed16 theSlotIndex;
   Signed16 thePortIndex;

   for (theSlotIndex = 0; theSlotIndex < 4; theSlotIndex += 1) {
      for (thePortIndex = 0; thePortIndex < 8; thePortIndex += 1) {
         thePortStatusArray[theSlotIndex][thePortIndex] = False;
      }
   }
}
```

Sample generated HTML

```
<TABLE BORDER=“1” CELLPADDING = “5” CELLSPACING= “0”>
<TR ALIGN=CENTER><TD COLSPAN=3><B>Slot 3</B></TD></TR>
<TR ALIGN=CENTER><TD>Port Number</TD><TD>Port Name</TD><TD>Active</TD></TR>
<TR ALIGN=CENTER><TD>1</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,1” Value=“Port_1” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,1”></TD></TR>
<TR ALIGN=CENTER><TD>2</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,2” Value=“Accounting” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,2” CHECKED></TD></TR>
<TR ALIGN=CENTER><TD>3</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,3” Value=“Port_3” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,3”></TD></TR>
<TR ALIGN=CENTER><TD>4</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,4” Value=“Port_4” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,4”></TD></TR>
<TR ALIGN=CENTER><TD>5</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,5” Value=“Marketing” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,5” CHECKED></TD></TR>
<TR ALIGN=CENTER><TD>6</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,6” Value=“Port_6” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,6”></TD></TR>
<TR ALIGN=CENTER><TD>7</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,7” Value=“Port_7” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,7” CHECKED></TD></TR>
<TR ALIGN=CENTER><TD>8</TD>
<TD><INPUT TYPE=“Text” NAME=“PortName?3,8” Value=“Port_8” Size=20></TD>
<TD><INPUT TYPE=“CHECKBOX” NAME=“PortActive?3,8”></TD></TR>
</TABLE>
```

Repeat groups — Radio buttons

Within a repeat group, it may be desirable to have a radio button group with one button for each member of the group. This capability can be useful for selecting an item within a repeat group.

HTML comment tag

```
<!–– RpFormRadioGroupDyn NAME=n RpGetType=gt RpGetPtr=gp
   RpSetType=st RpSetPtr=sp ––>
```

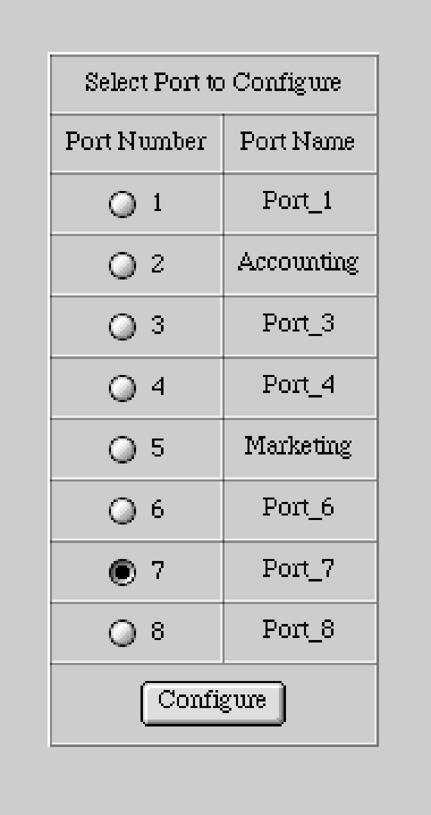
The `RpFormRadioGroupDyn` tag specifies a radio button group that spans the range of the repeat group with one button associated with each index value of the repeat group. The `NAME` keyword specifies the HTML name of the radio button group.

When a page is displayed, the RomPager engine can use the access functions described as follows to retrieve a value. If the value matches the current index of the repeat group, the button is highlighted. When a form is submitted, the RomPager engine passes the value of the submitted button (the repeat group index) to the storage functions.

Function prototypes

```
GetType = Function

typedef rpOneOfSeveral (*rpFetchRadioGroupFuncPtr)(void);

GetType = Complex

typedef rpOneOfSeveral (*rpFetchRadioGroupComplexPtr)(void *theServerDataPtr,
            char *theNamePtr, Signed16Ptr theIndexValuesPtr);

SetType = Function

typedef void (*rpStoreRadioGroupFuncPtr)(Unsigned8 theValue);

SetType = Complex

typedef void (*rpStoreRadioGroupComplexPtr)(void *theServerDataPtr,
            Unsigned8 theValue, char *theNamePtr,
            Signed16Ptr theIndexValuesPtr);
```

Sample



Select Port to Configure

Port Number	Port Name
○ 1	Port_1
○ 2	Accounting
○ 3	Port_3
○ 4	Port_4
○ 5	Marketing
○ 6	Port_6
● 7	Port_7
○ 8	Port_8

Commented HTML

```
<!-- RpDataZeroTerminated -->
   <TABLE BORDER=“1” CELLPADDING = “5” CELLSPACING= “0”>
   <TR ALIGN=CENTER><TD COLSPAN=2>Select Port to Configure</TD></TR>
   <TR ALIGN=CENTER><TD>Port Number</TD>
   <TD>Port Name</TD></TR>
<!-- RpEnd -->
<!-- RpRepeatGroup RpStart=1 RpLimit=8 RpIncrement=1 -->
   <!-- RpDataZeroTerminated -->
      <TR ALIGN=CENTER><TD>
   <!-- RpEnd -->
   <!-- RpFormRadioGroupDyn NAME=SelPort
      RpGetPtr=SelectedPort RpSetPtr=SelectedPort -->
   <!-- RpIndexDisplay_0 text=“&nbsp;”-->
      &nbsp;
   <!-- RpEnd -->

   <!-- RpDataZeroTerminated -->
      </TD><TD>
   <!-- RpEnd -->
   <!-- RpDisplayText RpGetType=Complex RpGetPtr=GetPortName -->
      Port_1
   <!-- RpEnd -->
   <!-- RpDataZeroTerminated -->
      </TD></TR>
   <!-- RpEnd -->
<!-- RpLastItemInGroup -->
<!-- RpDataZeroTerminated -->
   <TR ALIGN=CENTER><TD COLSPAN=2>
<!-- RpEnd -->
   <INPUT TYPE=“Submit” VALUE=“Configure”>
<!-- RpDataZeroTerminated -->
   </TD></TR>
   </TABLE>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_RepeatGroup,       &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },
   { eRpItemType_FormSubmit,     &PgSample_Item_4 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_5 },

/* Structures */

static char PgSample_Item_1[] =
   C_oTABLE_BORDER “\”1\”“ C_CELLPADDING “ \”5\”“ C_CELLSPACING “ \”0\”>\n”
   C_oTR_ALIGN_CENTER “>“ C_oTD_COLSPAN “2>Select Port to Configure”
   C_xTD C_xTR C_oTR_ALIGN_CENTER “>“ C_oTD “Port Number” C_xTD C_oTD
   “Port Name” C_xTD
   C_xTR;

static rpRepeatGroupItem PgSample_Item_2 = {
   1,
   8,
   1,
   PgSample_Item_2_Group
};

static char PgSample_Item_3[] =
   C_oTR_ALIGN_CENTER “>“ C_oTD_COLSPAN “2>\n”;

static rpButtonFormItem PgSample_Item_4 = {
   “Configure”
};

static char PgSample_Item_5[] =
   C_xTD C_xTR C_xTABLE;

static rpItem PgSample_Item_2_Group[] = {
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_6 },
   { eRpItemType_FormRadioGroupDyn, &PgSample_Item_7 },
   { eRpItemType_IndexDisplay_0,    &PgSample_Item_8 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_9 },
   { eRpItemType_DisplayText,       &PgSample_Item_10 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_11 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_6[] =
   C_oTR_ALIGN_CENTER “>“ C_oTD;

static rpRadioGroupInfo PgSample_Item_7 = {
   “SelPort”,
   &SelectedPort,
   &SelectedPort,
   eRpVarType_Direct,
   eRpVarType_Direct
};

static char PgSample_Item_8[] =
   C_NBSP;

static char PgSample_Item_9[] =
   C_xTD C_oTD;

static rpTextDisplayItem PgSample_Item_10 = {
   GetPortName,
   eRpVarType_Complex,
   eRpTextType_ASCII,
   20
};
static char PgSample_Item_11[] =
   C_xTD C_xTR;

/* Variables and Functions */

rpOneOfSeveral SelectedPort = 7;

char thePortNameArray[4][8][21];

char *GetPortName(void *theServerDataPtr,
               char *theNamePtr,
               Signed16Ptr theIndexValuesPtr) {
   Signed16 theSlotIndex = *theIndexValuesPtr;
   Signed16 thePortIndex = *(theIndexValuesPtr + 1);

   return thePortNameArray[theSlotIndex][thePortIndex];
   }
```

Sample generated HTML

```
<TABLE BORDER=“1” CELLPADDING = “6” CELLSPACING= “0”>
<TR ALIGN=CENTER><TD COLSPAN=2>Select Port to Configure</TD></TR>
<TR ALIGN=CENTER><TD>Port Number</TD><TD>Port Name</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“1”>&nbsp;1</TD>
<TD>Port_1</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“2”>&nbsp;2</TD>
<TD>Accounting</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“3”>&nbsp;3</TD>
<TD>Port_3</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“4”>&nbsp;4</TD>
<TD>Port_4</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“5”>&nbsp;5</TD>
<TD>Marketing</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“6”>&nbsp;6</TD>
<TD>Port_6</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“7” CHECKED>&nbsp;7</TD>
<TD>Port_7</TD></TR>
<TR ALIGN=CENTER>
<TD><INPUT TYPE=“RADIO” NAME=“SelPort” VALUE=“8”>&nbsp;8</TD>
<TD>Port_8</TD></TR>
<TR ALIGN=CENTER>
<TD COLSPAN=2><INPUT TYPE=“Submit” VALUE=“Configure”></TD></TR>
</TABLE>
```

Page and form objects

The RomPager engine uses object structures to manage the ROM-based pages. When a browser requests a URL, the server finds the object in a master object list and determines how to process the request. The PBuilder compiler creates object structures as part of compiling an HTML page. A page without a form will create a single object structure. A page with a form will create two object structures — one for the page and one for the form. The headers described as follows are optional and used for overriding the default values that the compiler creates. Certain types of special processing require these tags so that the appropriate flags are set in the object structure.

HTML comment tag

```
<!-- RpPageHeader RpUrl=url RpObjectType=ot
      RpRefreshTime=rt RpRefreshPage=pgid RpServerPush RpAccess=aa
      RpFunctionPtr=fp RpStructuredAccess DebugFlow Disposition -->

<!-- RpFormHeader Action=url Enctype=et Method=m RpNextPage=pgid RpAccess=aa
      RpFunctionPtr=fp RpDirect -->
```

The `RpPageHeader` tag is used to supply information that is stored in the page object structure. The `RpUrl` keyword is used to specify the URL that is used to match a request from a browser. If the `RpUrl` keyword is not specified, the PBuilder compiler creates the URL in the form `/`*`xxxxx`* where *xxxxx* is the name of the page (***`xxxxx`*****`.html`**). The `RpObjectType` keyword specifies how the page should be cached, the allowed values are Static and Dynamic. If the keyword is not specified, Dynamic is used. Dynamic pages are served to the browser with HTTP headers to indicate that the page should not be cached. Static pages are served to the browser with HTTP headers indicating that the page was created on the ROM image date of the device and can be cached.

Pages can be served dynamically using client pull techniques. The `RpRefreshTime` keyword specifies that dynamic page techniques should be used on this page and the time to wait before a page is served again. If the `RpRefreshPage` keyword is specified, it identifies the page to be served when the time expires. If the `RpRefreshPage` keyword is not specified, the current page is redisplayed when the time expires. The `pgid` parameter for the `RpRefreshPage` keyword specifies the internal PBuilder generated name of the next page to be served. This name is of the form `Pg`*`xxxxx`* where *xxxxx* is the name of the page.

If the `RpStructuredAccess` keyword is defined, the PBuilder compiler creates structured access routines and storage for accessing and storing the variables. See the *Stub Routines* section and the *Configure Network Info Page* (Example 2 in Appendix A) for more details.

If the `Disposition` keyword is defined, the compiler sets the `kRpObjFlag_Disposition` flag for the page object structure. This flag is used to identify a page object that should be treated as an attachment. If the flag is set, the RomPager engine will serve the page with a `Content-Disposition`: attachment header. Some browsers recognize this header and save the page to the user’s disk rather than interpreting the HTML and displaying the page.

The `RpFormHeader` tag is used to supply information that is stored in the form object structure. The `Action` keyword is used to specify the URL that is used with the Action keyword of the `<FORM>` tag. This URL is sent by the browser when a form is submitted and used by the server to find the object that handles the form items. If the `Action` keyword is not specified, the PBuilder compiler creates the URL in the form `/Forms/`*`xxxxx`* where *xxxxx* is the name of the page (***`xxxxx`*****`.html`**).

The `Enctype` keyword specifies the encoding of the form data. The allowed values are `APPLICATION/X-WWW-FORM-URLENCODED` (the default) and `MULTIPART/FORM-DATA`. The `Method` keyword specifies the HTTP method for sending the form data. The allowed values are POST (the default) and GET.

The RpNextPage keyword specifies the page to be served after the form is processed. If the RpNextPage keyword is not specified, the page that the form is on will be served again. The pgid parameter for the RpNextPage keyword specifies the internal PBuilder generated name of the next page to be served. This name is of the form Pg*xxxxx* where *xxxxx* is the name of the page.

Normally the RomPager engine uses HTTP redirection techniques (using the 302 Moved Temporarily header) to serve the page after a form is processed. The redirection techniques allow the browsers to maintain more accurate history lists. The optional RpDirect keyword specifies that the next page is to be served directly and not redirected.

The RpAccess keyword specifies the security realm for both page and form objects. The allowed values are: Unprotected, Realm1, Realm2, … Realm8. The RpAccess keyword can be specified multiple times to indicate the page is allowed in multiple realms. If RpAccess is not specified, a value of Unprotected is used.

The RpFunctionPtr keyword specifies a pre-processing function to be called before the page items are displayed or a post-processing function to be called after the form items are processed.

Object structure definitions and prototypes

```
/* Required for Page and Form objects */

typedef struct rpObjectDescription {
   char *         fURL;
   rpItem *       fItemsArrayPtr;
   rpObjectExtensionPtr fExtensionPtr;
   Unsigned32        fLength;
   rpAccess       fObjectAccess;
   rpDataType        fMimeDataType;
   rpObjectType      fCacheObjectType;
}  rpObjectDescription, *rpObjectDescriptionPtr;

/* Required for Form objects, optional for Page Objects */

typedef struct rpObjectExtension {
   rpProcessDataFuncPtr fProcessDataFuncPtr;
   rpObjectDescription *   fPagePtr;
   rpObjectDescription *   fRefreshPagePtr;
   Unsigned16        fRefreshSeconds;
   rpObjectFlags     fFlags;
}  rpObjectExtension, *rpObjectExtensionPtr;

/* Page object pre-processing, Form object post-processing */

typedef void (*rpProcessDataFuncPtr)(void *theServerDataPtr,
      Signed16Ptr theIndexValuesPtr);
```

Sample

Empty Page and Form

Commented HTML

```
<!-- RpPageHeader RpRefreshTime=17 RpFunctionPtr=SamplePagePreProcessing
   RpAccess=Realm1 RpAccess=Realm2 -->
<!-- RpDataZeroTerminated -->
   <HTML>
   <HEAD>
   <TITLE>Sample Page</TITLE>
   </HEAD>
      <BODY>
<!-- RpEnd -->
<!-- RpFormHeader RpNextPage=PgSample2 RpFunctionPtr=SampleFormPostProcessing
   RpAccess=Realm1 -->
   <FORM METHOD=“POST” ACTION=“/Forms/Sample”>
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
   <P><CENTER>Empty Page and Form</CENTER></P>
   </FORM>
   </BODY>
   </HTML>
   <!-- RpEnd -->
```

Internal structures

```
rpObjectDescription PgSample = {
   “/Sample”,
   PgSample_Items,
   &PgSample_ObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Realm1 | kRpPageAccess_Realm2,
   eRpDataTypeHtml,
   eRpObjectTypeDynamic
};
static rpObjectExtension PgSample_ObjectExtension = {
   SamplePagePreProcessing,
   (rpObjectDescriptionPtr) 0,
   (rpObjectDescriptionPtr) 0,
   17,
   kRpObjFlags_None
};
static rpItem PgSample_Items[] = {
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_FormHeader,     &PgSample_Form },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_2 },
   { eRpItemType_LastItemInList }
};
rpObjectDescription PgSample_Form = {
   “/Forms/Sample”,
   PgSample_FormItems,
   &PgSample_FormObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Realm1,
   eRpDataTypeForm,
   eRpObjectTypeDynamic
};

rpObjectExtension PgSample_FormObjectExtension = {
   SampleFormPostProcessing,
   &PgSample2,
   (rpObjectDescriptionPtr) 0,
   0,
   kRpObjFlags_None
};

static rpItem PgSample_FormItems[] = {
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_1[] =
   C_oHTML_oHEAD_oTITLE “Sample Page”
   C_xTITLE_xHEAD_oBODY;

static char PgSample_Item_2[] =
   C_oP C_oCENTER “Empty Page and Form” C_xCENTER C_xP C_xFORM
   C_xBODY_xHTML;

/* Variables and Functions */

void SamplePagePreProcessing(void *theServerDataPtr,
               Signed16Ptr theIndexValuesPtr) {
}
void SampleFormPostProcessing(void *theServerDataPtr,
               Signed16Ptr theIndexValuesPtr) {
}
```

Sample generated HTML

```
<HTML>
<HEAD>
<TITLE>Sample Page</TITLE>
</HEAD>
<BODY>
<FORM METHOD=“POST” ACTION=“/Forms/Sample”>
<P><CENTER>Empty Page and Form</CENTER></P>
</FORM></BODY></HTML>
```

Server-side image maps

Image maps are used to support selection of URL links from an image. With Client-side image maps, the coordinate processing is done on the browser, and the browser determines which URL to request next. Client-side image maps can be specified with standard HTML, so no special RomPager tags are needed.

With Server-Side image maps, when the user selects a point by clicking on the image, the browser sends the coordinates of the point to the server, which uses an internal map to determine which part of the image has been selected and which URL link to serve. The following tags are used to describe the image map and the selection coordinates.

HTML comment tags

```
<!-- RpFormImageMap RpUrl=u RpImgKeywords=ik RpDefaultPage=pgid
   RpSetType=st RpSetPtr=sp -->

<!-- RpImageMapCircle RpItemPage=pgid RpItemNumber=i
   RpCenter=(X,Y) RpRadius=r -->

<!-- RpImageMapRectangle RpItemPage=pgid RpItemNumber=i
   RpCorners=(X1,Y1),(X2,Y2) -->

<!-- RpImageMapPolygon RpItemPage=pgid RpItemNumber=i
   RpPoints=(X1,Y1),(X2,Y2),…, (Xn,Yn) -->

<!-- RpEndImageMapAreas -->
```

The `RpFormImageMap` tag specifies the information used to generate the image map HTML and the processing of the server-side image map. The `RpUrl` keyword specifies the URL that is the `ACTION` parameter used to process the map. If the `RpUrl` keyword is not specified, the PBuilder compiler creates the URL in the form /`Forms/`*`xxxxx`* where *xxxxx* is the name of the page (***`xxxxx`*****`.html`**). When the RomPager engine builds an image map, it generates an `<IMG ISMAP>` tag. The `RpImgKeywords` keyword specifies a string that contains the keyword attributes to associate with the `<IMG>` tag. The string must contain the URL of the image and can contain other attributes such as `WIDTH` or `HEIGHT`. Individual selection areas within the map are specified by the `RpImageMapCircle`, `RpImageMapRectangle`, and `RpImageMapPolygon` tags.

The `RpDefaultPage` keyword specifies the page to serve if a point is clicked outside the specified selection areas. If the `RpDefaultPage` keyword is not specified, the default page will be set to the page that the image map is contained on.

The Set function specified by the `RpSetType` and `RpSetPtr` keywords is used to store a value associated with the selected map area. PBuilder creates a special RomPager form object that is used to process the map.

The image area tags specify the page to serve with the `RpItemPage` keyword and use the `RpItemNumber` keyword to specify the value to be passed to the Set function when the area is selected. If the `RpItemNumber` keyword is not specified, PBuilder generates a value for each map area starting at 1.

The `RpImageMapCircle` tag specifies the information for processing an image map circle. The `RpCenter` keyword specifies the pixel coordinates of the center of the circle. The `RpRadius` keyword specifies the radius of the circle.

The `RpImageMapRectangle` tag specifies the information for processing an image map rectangle. The `RpCorners` keyword specifies the left, top, right, and bottom coordinates of the rectangle.

The RpImageMapPolygon tag specifies the information for processing an image map polygon. The RpPoints keyword specifies the coordinates of the points of the polygon.

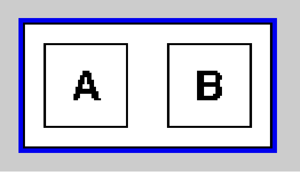
The list of image map area tags that is associated with an RpFormImageMap tag is terminated when the RpEndImageMapAreas tag is encountered.

The pgid parameter for the RpDefaultPage and RpItemPage keywords specifies the internal PBuilder-generated name of the page to be served in response to an image map query. This name is of the form Pg*xxxxx* where *xxxxx* is the name of the page.

Function prototypes

```
SetType = Function

typedef void (*rpStoreLocationPtr)(rpMapLocation theValue);

SetType = Complex

typedef void   (*rpStoreLocationComplexPtr)(void *theServerDataPtr,
                     rpMapLocation theValue,
                     char *theNamePtr,
                     Signed16Ptr theIndexValuesPtr);
```

Sample



Commented HTML

```
<!-- RpFormImageMap RpDefaultPage=PgSample
   RpSetType=Direct RpSetPtr=ImageMapValue
   RpImgKeywords="SRC=\"/Images/ABImageMap\" WIDTH=109 HEIGHT=55" -->
<A HREF="/Forms/ImageMapValidation">
<IMG SRC="/Images/ABImageMap" WIDTH=109 HEIGHT=55 ISMAP></A>
<!-- RpEnd -->
<!-- RpImageMapRectangle RpItemPage=PgSample RpItemNumber=1
   RpCorners=(9, 9),(45,45) -->
<!-- RpImageMapRectangle RpItemPage=PgSample RpItemNumber=2
   RpCorners=(63,9),(96,45) -->
<!-- RpEndImageMapAreas -->
```

Internal structures

```
/* Structures */

rpObjectDescription PgSample = {
   “/Sample”,
   PgSample_Items,
   (rpObjectExtensionPtr) 0,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeHtml,
   eRpObjectTypeDynamic
};

static rpItem PgSample_Items[] = {
   { eRpItemType_DataZeroTerminated, &PgSample_Item_1 },
   { eRpItemType_LastItemInList }

rpObjectDescription PgSample_Form = {
   “/Forms/Sample”,
   PgSample_FormItems,
   &PgSample_FormObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeMap,
   eRpObjectTypeDynamic
};

rpObjectExtension PgSample_FormObjectExtension = {
   (rpProcessDataFuncPtr) 0,
   &PgSample,
   (rpObjectDescriptionPtr) 0,
   0,
   kRpObjFlags_None
};

static rpItem PgSample_FormItems[] = {
   { eRpItemType_FormImageMap, &PgSample_Item_2 },
   { eRpItemType_LastItemInList }
};

static char PgSample_Item_1[] =
   C_oANCHOR_HREF “/Forms/Sample\”>“ C_oIMG_SRC “/Images/ABImageMap\”“
   C_WIDTH “109” C_HEIGHT “55 ISMAP>“
   C_xANCHOR;
static rpImageMapFormItem PgSample_Item_2 = {
   &PgSample,
   &PgSample_Item_2_Option_1,
   &ImageMapValue,
   eRpVarType_Direct
};
```

```
rpImageMapLocation PgSample_Item_2_Option_1 = {
   &PgSample_Item_2_Option_2,
   &PgSample,
   eRpLocationType_Rectangle,
   (rpPointPtr) 0,
   9,
   45,
   9,
   45,
   1
};

rpImageMapLocation PgSample_Item_2_Option_2 = {
   (rpImageMapLocationPtr) 0,
   &PgSample,
   eRpLocationType_Rectangle,
   (rpPointPtr) 0,
   9,
   45,
   63,
   96,
   2
};

/* Variables and Functions */

rpImageMapLocation ImageMapValue;
```

Sample generated HTML

```
<A HREF=“/Forms/Sample”><IMG SRC=“/Images/ABImageMap”
      WIDTH=109 HEIGHT=55 ISMAP></A><BR>
```

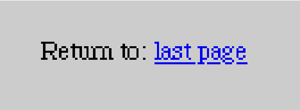
HTML referer tag

HTML comment tag

```
<!–– RpHtmlReferer RpText=t -->
```

When a browser sends in a request, it usually sends a HTTP header (Referer) that indicates the page the request came from. The `RpHtmlReferer` tag is used to generate a dynamic HTML link to the referring page. PBuilder generates the link using the format `<A HREF=“`*`xxxxxx`*`”>Link_Text</A>.` The `RpText` keyword defines the text to be used to replace Link_Text portion of the format. The *xxxxxx* portion of the format is replaced at runtime by the value of the HTTP Referer header.

Sample



Return to: last page

Commented HTML

```
   <!-- RpDataZeroTerminated -->
      <BR>Return to:
   <!-- RpEnd -->
   <!-- RpHtmlReferer RpText=“last page”-->
      <A HREF=“http://www.device.com/Sample”>last page</A>
   <!-- RpEnd -->
   <!-- RpDataZeroTerminated --><BR><!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_HtmlReferer,    &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },
/* Structures */

static char PgSample_Item_1[] =
   C_oBR “Return to: “

static char PgSample_Item_2[] =
   “last page”;

static char PgSample_Item_3[] =
   C_oBR;
```

Sample generated HTML

```
<BR>Return to: <A HREF=“http://www.device.com/Sample”>last page</A><BR>
```

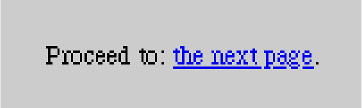
URL state tag

HTML comment tag

```
<!–– RpUrlState RpText=t ––>
```

The `RpUrlState` tag is used to generate a link with the current URL state. The URL state is appended to the text defined by the `RpText` keyword. See the State Management callback routines for more information.

Sample



Proceed to: the next page.

Commented HTML

```
<!-- RpDataZeroTerminated -->
   <BR>Proceed to:
<!-- RpEnd -->
<!-- RpUrlState RpText=“<A HREF=\”“ -->
   <A HREF=“/US/12345
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->/NextPage”>the next page</A>.<BR><!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_UrlState,       &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },

/* Structures */

static char PgSample_Item_1[] =
   C_oBR “Proceed to: “;

static char PgSample_Item_2[] =
   C_oANCHOR_HREF;

static char PgSample_Item_3[] =
   “/NextPage\”>the next page” C_xANCHOR “.”
   C_oBR;
```

Sample generated HTML

```
<BR>Proceed to: <A HREF=“/US/12345/NextPage”>the next page</A>.<BR>
```

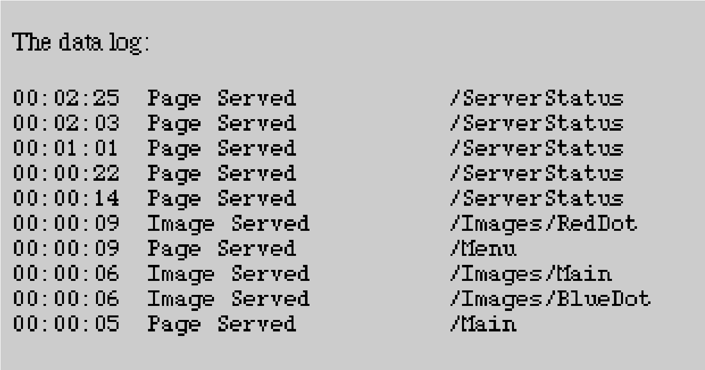
File insert HTML tag

HTML comment tag

```
<!–– RpFile RpFileName=fn ––>
```

The `RpFile` tag is used to specify the name of a file whose contents will be dynamically inserted into the HTML stream. This can be useful for creating a page to display contents that are dynamically created elsewhere in the device, such as logging streams.

Sample



```
The data log:

00:02:25  Page Served           /ServerStatus
00:02:03  Page Served           /ServerStatus
00:01:01  Page Served           /ServerStatus
00:00:22  Page Served           /ServerStatus
00:00:14  Page Served           /ServerStatus
00:00:09  Image Served          /Images/RedDot
00:00:09  Page Served           /Menu
00:00:06  Image Served          /Images/Main
00:00:06  Image Served          /Images/BlueDot
00:00:05  Page Served           /Main
```

Commented HTML

```
<!-- RpDataZeroTerminated -->
   <BR>The data log:<BR><PRE>
<!-- RpEnd -->
<!-- RpFile RpFileName=“DataLog” -->
   00:00:05  Page Served           /Main
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
   </PRE><BR>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_1 },
   { eRpItemType_File,        &PgSample_Item_2 },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_3 },

/* Structures */

static char PgSample_Item_1[] =
   C_oBR “The data log:” C_oBR “<PRE>“;

static char PgSample_Item_2[] =
   “DataLog”;

static char PgSample_Item_3[] =
   “</PRE>“ C_oBR;
```

Sample generated HTML

```
<BR>The data log:<BR><PRE>
00:02:25  Page Served          /ServerStatus
00:02:03  Page Served          /ServerStatus
00:01:01  Page Served          /ServerStatus
00:00:22  Page Served          /ServerStatus
00:00:14  Page Served          /ServerStatus
00:00:09  Image Served         /Images/RedDot
00:00:09  Page Served          /Menu
00:00:06  Image Served         /Images/Main
00:00:06  Image Served         /Images/BlueDot
00:00:05  Page Served          /Main
</PRE><BR>
```

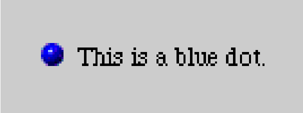
Image source tag

HTML comment tag

```
<!–– RpImageSource RpObjectPtr=op ––>
```

The RpImageSource tag is used to create the beginning portion of the HTML <IMG> tag. The RpObjectPtr keyword provides the name of a RomPager internal image object created with PBuilder. When the RomPager engine creates the HTML, it creates <IMG SRC=“*objecturl*” where *objecturl* is the URL of the image object. This tag can provide some compression of an HTML page by eliminating the need to store the URL name of the image multiple times.

Sample



This is a blue dot.

Commented HTML

```
<!-- RpImageSource RpObjectPtr=PgBlueDot -->
   <IMG SRC=“/Images/BlueDot”>
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
   &nbsp;This is a blue dot.<BR>
<!-- RpEnd -->
```

Internal structures

```
/* Element List Items */
   { eRpItemType_ImageSource,    &PgBlueDot },
   { eRpItemType_DataZeroTerminated,   &PgSample_Item_2 },

/* Structures */

static char PgSample_Item_2[] =
      C_NBSP “This is a blue dot.” C_oBR;
```

Sample generated HTML

```
<IMG SRC="/Images/BlueDot">&nbsp;This is a blue dot.<BR>
```

Structure sharing tags

The PBuilder compiler takes standard HTML pages with special RomPager comment tags to create the internal C structures for the ROM-based pages. There are a few tags used to share structures within a page and across pages.

HTML comment tag

```
<!-- RpUseIdentifier RpIdentifier=i RpItemType=t -->
<!-- RpUseId RpId=i RpItemType=t -->
<!-- RpFormItem RpIdentifier=i RpItemType=t -->
```

When PBuilder compiles an HTML page, it generates a set of C structures and creates default identifiers for the structures based upon the name of the page. Since substantial memory savings can be achieved by sharing common structures, it is necessary to specify an identifier for the structures in order to share them. The `RpIdentifier` keyword is used in the RomPager comment tags so that PBuilder uses the supplied identifier instead of creating one.

The `RpUseIdentifier` tag indicates that an item with the given `RpIdentifier` and `RpItemType` is defined elsewhere, but should be used here. `RpUseId` is an alias for `RpUseIdentifier` and `RpId` is an alias for `RpIdentifier`.

Normally, when PBuilder encounters a comment tag describing a form item, it adds an item to the current form item list as well as to the current page item list. If a `RpUseIdentifier` tag is used that specifies a group, PBuilder cannot determine the types of items contained within the group. Therefore, if there are form items in the group, PBuilder will not have enough information to add the form item pointers to the current form item list. The `RpFormItem` tag is used to tell PBuilder to add a form item with the given `RpIdentifier` and `RpItemType` to the current form item list.

The `RpItemType` keyword specifies the internal object type used in the element list. The keyword value is formed by removing the leading constant from the RomPager internal item type. So, for example, the `eRpItemType_DisplayText` item type is specified with a keyword value of `DisplayText`. The possible keyword values are:

```
RpItemType = [DataZeroTerminated, DZT, DataLengthEncoded, DisplayText,
   DynamicDisplay, ExtendedAscii, File, FormAsciiText, FormButton, FormCheckbox,
   FormCheckboxDyn, FormFile, FormFixedMultiDyn, FormFixedMultiSelect,
   FormFixedSingleDyn, FormFixedSingleSelect, FormHeader, FormHiddenText,
   FormHiddenDyn, FormImageMap, FormNamedSubmit, FormPasswordText, FormPasswordDyn,
   FormRadioGroupDyn, FormRadioButton, FormRadioButtonDyn, FormReset, FormSubmit,
   FormTextArea, FormTextAreaBuf, FormTextDyn, FormVariableMultiDyn,
   FormVariableMultiSelect, FormVariableSingleDyn, FormVariableSingleSelect,
   FormVarValueMultiDyn, FormVarValueMultiSelect, FormVarValueSingleDyn,
   FormVarValueSingleSelect, HtmlReferer, ImageSource, IndexDisplay_0,
   IndexDisplay_1, IndexDisplay_2, IndexDisplay_3, IndexDisplay_4, IndexDisplay_5,
   ItemGroup, NamedDisplayText, QueryDisplay, RepeatGroup, RepeatGroupDynamic,
   RepeatGroupWhile, SlaveIdentity, UrlState]
```

Sample

The third example in Appendix A (Printer Status Page) shows an example of the `RpUseIdentifier` tag, as does the sample in the Repeat Groups Dynamic Form Items section.

Miscellaneous formatting tags

The PBuilder compiler takes standard HTML pages with special RomPager comment tags to create the internal C structures for the ROM-based pages. In order for the pages to be properly viewed with browsers or page editors, there are a few tags used to help with formatting.

HTML comment tag

```
<!-- RpHiddenDataZeroTerminated RpText=h -->
<!-- RpHDZT RpText=h -->
<!-- RpSample -->
```

The `RpHiddenDataZeroTerminated` tag is used when there is HTML text is to be included in the internal C structures, but that a browser is to ignore when displaying the sample page. Usually this tag is used with the `RpDynamicDisplay` tag, when there are multiple choices that could be displayed, but only one will be shown on the sample page. The `RpText` keyword supplies the HTML to process. `RpHDZT` is an alias for `RpHiddenDataZeroTerminated`.

The `RpSample` tag is used when there is some HTML that a browser or page editor should show during the page design process, but that PBuilder is to ignore since there will be dynamically created HTML by other tags. The sample HTML follows this tag, and is terminated by the `<!-- RpEnd -->` tag.

Sample

The third example in Appendix A (Printer Status Page) shows examples of these tags.

Using the PBuilder Compiler

Once you have built your HTML pages, you need to turn them into compilable source code. The internal format of a page consists of an Object Header structure, a list of Object elements, and the actual object elements with the variable access structures. PBuilder will convert HTML pages, as well as GIF, JPEG, PICT, PNG images and Java applets into C source code files. The PBuilder program is distributed as a Microsoft Windows 95 application and as a Macintosh application. Copy the appropriate version of the program to the directory containing your HTML pages. Start the PBuilder program and type the name of a file to be converted. PBuilder will find the file and create the internal format C source file with a **.c** extension. PBuilder will also create a C source file with a **_v.c** extension. This file contains stub routines for your Set/Get routines.

When you specify a file, if there is no path, Pbuilder assumes the file is in the current directory or folder. If a path is specified, it must use a slash (/) as the path separator character. If you specify a file without specifying a file extension, PBuilder tries to open files with extensions that it knows about. The first extension it tries is **.pbb** or **.txt**, for a batch file. It then tries extensions for HTML, image, and applet files.

If a file has an HTML extension (**.htm**, **.html**, **.js**, or **.css**), PBuilder requires a user dictionary file, **RpUsrDct.txt,** which contains the list of items that are to be in the RomPager user dictionary. PBuilder uses the contents of **RpUsrDct.txt** to compress the output of the converted HTML. PBuilder also creates two files from **RpUsrDct.txt** that must be part of the RomPager project: **RpUsrDct.h** and **RpUsrDct.c**.

The file extensions for image files can be:

Image File	File Extension
GIF	**.gif**
JPEG	**.jpeg** or **.jpg**
TIFF	**.tiff** or **.tff**
PICT	**.pict** or **.pct**
PNG	**.png**

The file extensions for Java files can be:

.class
.jar
.cla or **.cls**

To process a batch of files in Pbuilder, set up a batch file with the **.pbb** or **.txt** extension. Each line of the batch file should contain a separate name of a file to parse. When you run PBuilder, specify that file as the filename to process. For example, to process the files **Main.html**, **Logo1.gif**, and **Logo2.gif** in one pass, the batch file would contain:

```
Main
Logo1
Logo2
```

If a batch file is processed, the `CreateSingleSourceFile` flag in **PbSetUp.txt** controls whether the batch output goes to one combined C source files or separate files. Additionally, PBuilder creates a master object list in a file called **RpPages.c**. This object list is the list that RomPager requires to find the objects that are stored in ROM. On startup, PBuilder looks for a default batch file named **PBuilder.pbb**. If this file exists, PBuilder does not show a prompt dialog and proceeds to process the contents of the file.

Controlling the PBuilder compiler

There are two input files that modify the behavior of the PBuilder compiler. The first file is named **`PbSetUp.txt`** and is used to modify the PBuilder output for various requirements. Some C compilers have different requirements for the format of the Page items. Other modifications may be desirable for different editors. The PBuilder output can be modified by placing the **`PbSetUp.txt`** file in the same directory as PBuilder. If this file does not exist, the default behaviors will be used. Each line of the **`PbSetUp.txt`** file is either empty, a comment line (the first non-whitespace character is a slash (/), or a command line. Command lines are used to modify the behaviour of Page Builder. A command line contains a keyword and a value. The format of a command line is:

```
xxxxx = yyyyy
```

where *`xxxxx`* is the keyword and *yyyyy* is the value. A value is a string, integer or flag depending on the keyword. The value is placed in quotes if it is a string. The following is a list of keywords, their types and default values, and a description of their use.

PbSetup.txt

Keyword: `AllowedCharactersPerLine`

Type: integer

Default value: 80

Description: Determines the length of lines for `eRpItemType_ExtendedAscii and eRpItemType_DataZeroTerminated` items.

Keyword: `AllowMacroNameForNumbers`

Type: flag

Default value: false

Description: If the flag is specified, no numeric checks are performed on numeric constants. This allows macro names to be passed through PBuilder to the C compilation that follows.

Keyword: `BufferSizeMultiplie`r

Type: integer

Default value: 1

Description: Determines the amount of memory to allocate for working buffers. Large files or files with many form items may need a larger value.

Keyword: `CreateSingleSourceFile`

Type: flag

Default value: false

Description: If the flag specified, the output from all conversions is written to a single file; otherwise a file is created for each conversion of a file.

Keyword: `DestinationPath`

Type: string

Default value: “”

Description: If the string is specified the output from the conversions will be written to the directory specified. Otherwise, the conversions will be written to the current directory. The URL path separator (/) is used in the path name to separate directories.

Keyword: `DontCreateVariablesFile`

Type: flag

Default value: false

Description: If the flag is specified, the ***`xxx`*****`_v.c`** file with stub or structured access routines is not created.

Keyword: `ExtendedAsciiEscapeChar`

Type: integer

Default value: 251 (‘\373’)

Description: This value is used to quote extended characters that are not in an `eRpItemType_ExtendedAscii` item.

Keyword: `ExtendedAsciiGroupSize`

Type: integer

Default value: 10

Description: Determines the number of standard ASCII characters between extended ASCII characters before terminating an `eRpItemType_ExtendedAscii` item.

Keyword: `IgnoreWhitespaceAfterEol`

Type: flag

Default value: false

Description: If the flag is specified, white space is ignored after an end of line is detected.

Keyword: `JustGenerateObjectList`

Type: flag

Default value: false

Description: If the flag is specified and a batch file is input, only the **`RpPages.c`** file with the object list will be created. This flag can be useful in an automated makefile environment.

Keyword: `LeadInChar`

Type: string

Default value: “char”

Description: The string value is the type of `eRpItemType_ExtendedAscii` and `eRpItemType_DataZeroTerminated` items. It may be useful to change this to “const char” in some environments.

Keyword: `LeadInUnsignedChar`

Type: string

Default value: “unsigned char”

Description: The string value is the type of `eRpItemType_DataLengthEncoded` items. It may be useful to change this to “const unsigned char” in some environments.

Keyword: `MaxBytesInItem`

Type: integer

Default value: 498

Description: Determines the maximum number of bytes in the initializer for `eRpItemType_ExtendedAscii`, `eRpItemType_DataZeroTerminated`, and `eRpItemType_DataLengthEncoded` items.

Keyword: `MaxBytesPerLine`

Type: integer

Default value: 16

Description: Determines the number of bytes to generate on a line for `eRpItemType_DataLengthEncoded` items.

Keyword: `MinimizeWhiteSpace`

Type: flag

Default value: false

Description: If the flag is specified, white space that will be provided by the system dictionary is removed.

Keyword: `TabLength`

Type: integer

Default value: 4

Description: Should be set to the number of characters a tab replaces in your source code editor.

Keyword: `UseFileNameForUrl`

Type: flag

Default value: false

Description: If the flag is specified, the URL for an object includes the file name extension of the source file. See the section called *URL Names Created by PBuilder* for more information.

Keyword: `UseFilePathForUrl`

Type: flag

Default value: false

Description: If the flag is specified, the URL for an object includes the path of the file. See the section called *URL Names Created by PBuilder* for more information.

Keyword: `UseJavaScript`

Type: flag

Default value: false

Description: Specifies that structures are to be generated with pointers to JavaScript.

RpUsrDct.txt

The user dictionary file contains frequently used phrases that are in the HTML but not in the system dictionary. The format of the user dictionary file is similar to that of the **`PBSetUp.txt`** file: Each line of the file is a user-dictionary element, a comment line (indicated by a slash as the the first non-whitespace character), a blank line. A user-dictionary element contains a macro name and a phrase. The format of a user-dictionary element is:

```
C_S_RomPager = “RomPager”
```

where *C_S_RomPager* is the macro name and “RomPager” is the phrase. PBuilder searches for the phrases in the HTML and replaces them with the macro names. It also generates two files: **`RpUsrDct.h`** and **`RpUsrDct.c`**.

The **`RpUsrDct.h`** file is required to compile the pages that PBuilder builds and contains lines like:

```
#define C_S_RomPager “\377\000”     /* “RomPager” */
```

The **`RpUsrDct.c`** file must be compiled and linked with the RomPager engine. It contains a structure like this:

```
const char *gUserPhrases[] = {
   /* \377\000 -- C_S_RomPager */ “RomPager”,
         .
         .
         .
};
```

URL names created by PBuilder

When PBuilder creates the URL that is stored with an object, it uses a variety of ways to determine the URL name. First, if the object is a page defined with HTML comments, a URL specified with the `RpUrl` keyword in the `RpPageHeader` tag is used if it exists. If the source file is not an HTML file or does not use `RpPageHeader` with `RpUrl`, the URL will be created from the filename of the source. The URL that is created by PBuilder will take different forms depending on the settings of the `UseFileNameForUrl` and `UseFilePathForUrl` flags in the **`PbSetup.txt`** file.

The following table shows how PBuilder creates the URL name for an HTML file with an input name of **`/MyDirectory/MyPage.htm`**:

		UseFilePathForUrl	
		True	**False**
UseFileNameForUrl	**True**	**`/MyDirectory/MyPage.htm`**	**`/MyPage.htm`**
	False	**`/MyDirectory/MyPage`**	**`/MyPage`**

For an image object with an input file name of **`/MyDirectory/MyGif.gif`**, the generated URL is shown in the following table:

		UseFilePathForUrl	
		True	**False**
UseFileNameForUrl	**True**	**`/MyDirectory/MyGif.gif`**	**`/Images/MyGif.gif`**
	False	**`/MyDirectory/MyGif`**	**`/Images/MyGif`**

Java applets — file extension **`.cla`**, **`.cls`**, or **`.class`** — are treated the same as image objects with the exception that if the `UseFileNameForUrl` flag is not set, the URL will have a **`.class`** extension.

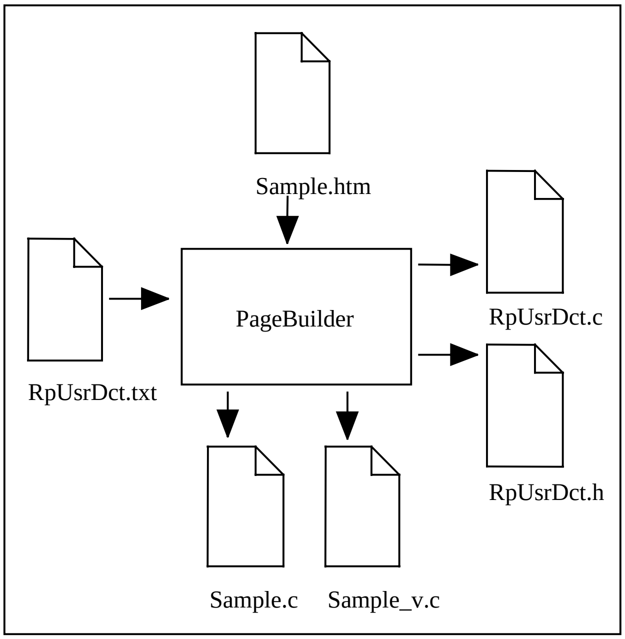
For form objects, the `Action` keyword in the `RpFormHeader` tag specifies the URL that is used to find the form object. If `RpFormHeader` tag or the `Action` keyword is not specified, the URL is created from the source filename without using the `UseFileNameForUrl` and `UseFilePathForUrl` flags.

If the HTML file above had a single form, the created URL would be `/Forms/MyPage`. If the HTML file had two forms in it, the created URLs would be `/Forms/MyPage_1` and `/Forms/MyPage_2`.

Note that special characters (non-alphanumeric) should be avoided in URL names (and in input file names) since different browsers handle these names differently when presenting a request to the Advanced Web Server, which can result in a URL object not being found correctly.

Phrase Dictionaries

The RomPager engine uses a phrase dictionary technique to provide compression for static ASCII text strings. The PBuilder compiler searches the HTML input files for strings that match the phrase dictionaries and replaces the strings with tokens as it is storing the static text elements.



Compression

Standard HTML phrases are recognized by the PBuilder compiler and stored as single byte tokens in the compressed text. User-specified phrases can also be recognized and are stored as two-byte tokens in the compressed text.

The input phrase of:

```
<TITLE>The AS Group Ltd. C3PO Control Page</TITLE>
```

would be stored internally as:

```
static char PgTitle[] = C_oTITLE “The AS Group Ltd. C3PO Control Page” C_xTITLE;
```

The **`RpUsrDct.txt`** file specifies the user-defined strings that the compiler should look for when compiling HTML pages. Each entry is of the form:

```
MacroName = “Search Phase”
```

which would tell PBuilder to search for phrases matching the specified search phase and replace them with the specified macro definition, which would represent a two-byte token.

User dictionary definitions can contain system dictionary definitions by specifying the system dictionary macro name in the user dictionary definition. For example, in the provided **`RpUsrDct.txt`**, the definition —

```
C_S_AllegroLogo = C_oHR C_oP C_oCENTER C_oIMG_SRC “/Images/Main\”“ C_WIDTH “162”
C_HEIGHT “83>“ C_xCENTER
```

will tell the PBuilder compiler to look for all phrases of the form —

```
<HR><P><CENTER><IMG SRC=/Images/Main WIDTH=162 HEIGHT=83></CENTER>
```

and replace them with the two byte token represented with the name `C_S_AllegroLogo`.

If company name and device phrases were defined, the input phrase of:

```
<TITLE>The AS Group Ltd. C3PO Control Page</TITLE>
```

would be stored internally as:

```
static char PgTitle[] = C_oTITLE "The " C_S_Company " " C_S_Device
                          " Control Page" C_xTITLE;
```

The compiler generates two C source files — **`RpUsrDct.c`** and **`RpUsrDct.h`** — for the user dictionary using the **`RpUsrDct.txt`** file as input. The individual entries in the dictionary will be created in the order they are specified in **`RpUsrDct.txt`**. The definitions of the user dictionary are in **`RpUsrDct.h`** with the data entries in **`RpUsrDct.c`**. These files must be compiled and linked with your project along with the RomPager engine and the page files.

The **`RpUsrDct.txt`** file provided with RomPager contains definitions for two sets of entries. The first 64 definitions are used to define error message strings that are used by the RomPager engine. These entries are reserved. The definitions that follow the error message definitions are used in the demo pages that come with the RomPager development environment and can be replaced in the production environment.

International support with dynamic user dictionaries

User dictionaries can be dynamically selected at runtime using the `RpSetUserPhraseDictionary` callback routine. One use of this feature is to conveniently support international HTML pages. By defining all the phrases on an HTML page that are to be internationalized as user dictionary phrases, the display language of the page can be changed by selecting a different user dictionary while maintaining the graphical layout and access to device variables. This technique means that for an application with six languages, the HTML page structure would be stored only once, with a different user-phrase dictionary for each language. The different user-phrase dictionaries can be stored in ROM or in a file system depending on product needs, since the `RpSetUserPhraseDictionary` specifies a pointer to the table of string pointers at run-time.

The development process typically is to define the HTML pages and the phrases to be internationalized in a single language. For example, the English phrases to be translated are defined as user dictionary phrases in the **`RpUsrDct.txt`** file and the HTML that defines the page layout and variable access using the English phrases are defined in Sample.htm.

After running the PBuilder compiler, there will be four files created:

- **`Sample.c`** will contain the compressed and tokenized HTML
- **`Sample_v.c`** will contain the stub variable access routines
- **`RpUsrDct.h`** will contain the index to the user dictionary
- **`RpUsrDct.c`** will contain the English version of the phrases (without the HTML)

  It can then be given to translation teams to create alternate language versions of the phrases, without the translation teams needing to worry about HTML.

The demonstration pages that come with RomPager include alternate language support for the internal error messages. Currently, the included languages are French, German, Italian, Portuguese and Spanish as well as English. The international dictionary files are named **`RpFreDct.c`**, **`RpGerDct.c`**, **`RpItaDct.c`**, **`RpPorDct.c`**, and **`RpSpaDct.c`**, respectively.

Expansion

As the static text elements of type `eRpItemType_DataZeroTerminated` are served, they are examined for characters with the high-bit set and replacement phrases are substituted. Standard ASCII characters are in the range `<000>` to `<177>` and pass through the dictionary expansion process unchanged. For international pages that use the high-bit characters (such as the SJIS form of Japanese), the international extended characters are stored by PBuilder using the `eRpItemType_ExtendedAscii` element type, and are not passed through the dictionary expansion process.

There are two phrase dictionaries, the system dictionary, which uses single byte tokens, and the user specified dictionary, which uses double byte tokens. The Extended ASCII characters of `<200>` to `<372>` are used by the built-in system dictionary. Each of these characters is a direct index to a commonly used HTML tag or other phrase. There are 123 possible system dictionary entries of which 94 are currently used.

By default, there are 1024 available two byte tokens that can be used for the user specified phrase dictionary. These tokens are stored in four sets of 256 entries of two byte tokens. The first 256 entries are selected with characters in the range `<377><000>` to `<377><377>`, the next 256 entries are selected with characters in the range `<376><000>` to `<376><377>`, and the fourth set of 256 entries are selected are selected with characters in the range `<374><000>` to `<374><377>`. The first 64 entries (`<377><000>` to `<377><077>`) are reserved by the RomPager engine for error messages.

The Extended ASCII character defined by `kRpCompressionEscape` is used to signal that the character following should not be passed through the dictionary expansion process. This is useful for storing a single high-bit international character in a phrase that is otherwise all standard ASCII.

If a phrase dictionary will contain many phrases that use high-bit characters such as a dictionary with many Japanese or Chinese double-byte phrases, then the `htheCompressionFlag` variable in the `RpSetUserPhraseDictionary` call should be set to False, so that no further expansion will be performed on phrases in the dictionary being set.

Stub Routines

The PBuilder compiler produces two files for a Web page input file — a file containing the page data and variable access structures, and a file containing stub routines for the variable access structures. For an input file name, the PBuilder output files will be named **`MyPage.c`** for the page data and **`MyPage_v.c`** for the stub routines.

The stub routines are the Get and Set routines necessary to access the device variables. The engineer building the Web application needs to complete the stub routines so that they store and retrieve the device variables. For each variable on a page, there will be a Get routine and a Set routine if the variable is part of a form. The examples in Appendix A show the stub routines that are generated by PBuilder.

After the stub routines have been completed, the `DontCreateVariablesFile` flag can be set in **`PbSetup.txt`** to prevent PBuilder from regenerating the stub routines file and overwriting the completed stub routines.

Structured access

In some cases, it may be desirable to Get and Set all of the variables for a page in one routine. For example, if each variable requires another task to retrieve the information, it can be more efficient to retrieve or store the variables with one routine rather than to access the dynamic information one variable at a time. The RomPager environment provides structured access routines to support this capability.

When a page is displayed, the structured access pre-processing routine is called by the RomPager engine (page processing – step 5). This routine accesses all the variables for a page and stores them in a temporary structure defined by PBuilder. The structure is stored in the user data area of the request control block. A stub version of the structured access routine is created by PBuilder and needs to be completed by the device engineer.

When the RomPager engine executes the Get functions (page processing – step 10), it will call the routines generated by PBuilder to retrieve the variables from the structure. The device engineer does not need to modify these routines, since they are created by PBuilder specifically for the individual page structure.

When a form is processed, the engine uses the PBuilder generated Set routines (form processing – step 8) to store the variables in the page structure. When the form post-processing structured access routine is called (form processing – step 10), this routine takes the information from the structure and stores them in the device variables. The stub version of this routine is created by PBuilder and must be completed by the device engineer. The *Configure Network Info Page* (Example 2 in Appendix A) shows an example of using structured access routines as well as normal variable access routines.

```
typedef  Signed16 (*rpFetchSigned16FuncPtr)(void);
typedef  Signed16 (*rpFetchSigned16ComplexPtr)(void *theServerDataPtr,
                          char *theNamePtr, Signed16Ptr theIndexValuesPtr);
typedef  void (*rpStoreSigned16FuncPtr)(Signed16 theValue);
typedef  void (*rpStoreSigned16ComplexPtr)(void *theServerDataPtr,
                     Signed16 theValue, char *theNamePtr,
                     Signed16Ptr theIndexValuesPtr);
```

The RomPager engine makes calls to the device code at various points to build pages or process forms. Each form item on a page contains a variable access structure that provides the ability to exit to the device for getting or setting the parameter values. These prototypes show sample Set and Get functions for a signed 16-bit variable with simple and complex functions. The simple functions provide a simple way to pass back a variable value for a Get, or to receive a value for a Set. The complex functions are passed additional information that may be useful in the function. A void pointer, theServerDataPtr points to the engine internal data structures, and is passed back to the engine in the callback routines described in the next section. Where applicable, the HTML name of a form item is passed in a complex function using a char pointer (theNamePtr). The current state of the repeat group index values is passed as a pointer (theIndexValuesPtr) to an array of signed 16-bit values. The first item of the array is index level 0, the next array item is index level 1, and so on.

```
typedef  void (*rpResetCheckboxArrayPtr) (void *theServerDataPtr,
                   char *theNamePtr, Signed16Ptr theIndexValuesPtr);
```

If a repeating checkbox array (eRpItemType_FormCheckboxDyn) is used, an exit function will be called at the beginning of the form processing. This function should be used to reset all of the checkboxes in the array, since the Set function will be called only for the checkboxes that have been checked on the form. The function of type rpResetCheckboxArrayPtr is called with the server data pointer, a pointer to the name of the form item and a pointer to the index variables array. This function is also called if the eRpItemType_FormFixedMultiDyn item is used, so that the list of menu selections can be reset.

```
typedef void (*rpResetVarSelectPtr)(void *theServerDataPtr);
```

If a <SELECT MULTIPLE> HTML form item with a variable list of menu items is used (eRpItemType_FormVariableMultiSelect, eRpItemType_FormVariableMultiDyn, eRpItemType_FormVarValueMultiSelect, or eRpItemType_FormVarValueMultiDyn), an exit function will be called at the beginning of the form processing. This function should be used to deselect all the items in the list, since the Set function will be called for only those items that have been selected in the list.

```
typedef  void (*rpDynamicRepeatFuncPtr)(void *theServerDataPtr,
                       Signed16Ptr theStart,
                       Signed16Ptr theLimit,
                       Signed16Ptr theIncrement);
```

The repeat group dynamic page element (eRpItemType_RepeatGroupDynamic) allows the device to set up an HTML table display based on current internal values. The device will be called with a function (rpDynamicRepeatFuncPtr) that passes in the server data pointer and pointers to the starting value, limiting value, and increment value of the table.

```
typedef  void (*rpRepeatWhileFuncPtr)(void *theServerDataPtr,
               Signed16Ptr theIndexPtr, void ** theRepeatGroupValuePtr);
```

The repeat group while page element (eRpItemType_RepeatGroupWhile) also allows the device to set up an HTML table display based on current internal values. The device will be called with a function (rpRepeatWhileFuncPtr) that passes in the server data pointer and a pointer to the index variables array. The function also passes a pointer to an arbitrary value that the repeat while function may use. Initially, the pointer points to the value (void *) 0. On subsequent calls, the pointer points to whatever the device stored on the previous call. The repeat while function will continue building the table until the device set the pointer so that it points to a value of (void *) 0.

Besides exits for each individual form element, the engine provides optional exits for page pre-processing and form post-processing. If this exit is used the address of the function of type rpProcessDataFuncPtr is placed in the fProcessDataFuncPtr field of the rpObjectExtension structure. This function will be passed the server data pointer and the index values pointer. These optional functions are used for setting up a collection of data before the page is served, or for validating form elements after all the individual form items have been processed.

```
typedef void (*rpProcessCloseFuncPtr)(void *theServerDataPtr);
```

If the RpSetRequestCloseFunction routine is used to set up a close function, a function of type rpProcessCloseFuncPtr will be called after the TCP connection has been closed.

```
extern rpPasswordState  SpwdGetExternalPassword(void *theServerDataPtr,
                     Unsigned8 theConnectionId,
                     char *theRealmNamePtr,
                     char *theUsernamePtr,
                     char *thePasswordPtr,
                        Unsigned32 theIpAddress);

typedef  void (*rpSessionCloseFuncPtr) (void *theServerDataPtr,
               void *theUserCookie);
```

If the Web application will provide the URL authentication, the SpwdGetExternalPassword routine will be called when the engine has a URL access to authenticate. For external password session timeouts, a function of type rpSessionCloseFuncPtr will be called.

Engine Callback Routines from the User Application

There are a variety of callback routines that are used to control various internal states of the Web server engine. These routines can be called from the device Get/Set routines or the optional page/form processing routines. The callback routines pass back to the engine the pointer to the engine global data and other variables as appropriate.

Page flow and connection control

```
extern char *  RpGetSubmitButtonValue(void *theServerDataPtr);
extern char *  RpGetCurrentUrl(void *theServerDataPtr);
extern void    RpSetNextPage(void *theServerDataPtr,
rpObjectDescriptionPtr theNextPagePtr);
extern void    RpSetNextFilePage(void *theServerDataPtr, char *theNextPagePtr);
extern void    RpSetRedirect(void *theServerDataPtr);
extern void    RpSetRefreshTime(void *theServerDataPtr,
Unsigned16 theRefreshSeconds);
extern void    RpSetRefreshPage(void *theServerDataPtr,
rpObjectDescriptionPtr theRefreshPagePtr);
extern void    RpSetRequestCloseFunction(void *theServerDataPtr,
rpProcessCloseFuncPtr theFunctionPtr);
extern void    RpSetConnectionClose(void *theServerDataPtr);
```

The page flow callback routines are used to control the page state. The callback function `RpGetSubmitButtonValue` returns a character pointer to the ASCII value of the Submit button. This routine is useful with forms that have more than one Submit button so that the management application can determine which button was pressed. The `RpGetCurrentUrl` routine returns a character pointer to the URL that invoked the page or form. This can be useful for processing routines that are common to multiple pages or forms.

The `RpSetNextPage` routine passes in an object pointer to the ROM-based page to be served next after redirection. This routine is typically called in the optional form post-processing routine to override the value in the `fPagePtr` of the `rpObjectExtension` structure. The `RpSetNextFilePage` callback works the same way as `RpSetNextPage`, but passes a pointer to a null-terminated string that contains the next URL to be served. In this way, the next page to be served can be located in the file system, or on a remote host, as well as in ROM. If the `RpSetNextPage` or `RpSetNextFilePage` functions are called in the page pre-processing routine, normally the page items pointed to by the “Next” page will be served directly. If it is desirable to have the page served by browser redirection, then the `RpSetRedirect` call should be used.

The `RpSetRefreshTime` and `RpSetRefreshPage` callback routines can be used to dynamically set up the time in seconds and the page pointer to the next page. Setting the refresh time to 0 will end the page refresh cycle. The information may also be set up statically in the object extension fields. If these callback routine are used, they will only be effective if called during page setup pre-processing. They are not effective if called during page item processing, since the HTTP headers have already been sent. They also are not effective during form processing, since forms handling provides redirection to a new page that will have its own refresh values.

The `RpSetRequestCloseFunction` callback is used to set up a function that will be called after the HTTP request has been completely processed and the TCP connection has been closed. The primary use for this function is to trigger device reset functions after being assured that a page with a “Reset about to start” message has been completely delivered.

The `RpSetConnectionClose` routine is used to force the TCP connection being used for the current HTTP request to close after the current HTTP request is completed. Both Netscape “keep alive” support and HTTP 1.1

persistent connections attempt to keep the TCP connection open for multiple HTTP requests from a single browser. In some cases, it may be useful for the server to free up a TCP connection for other users or for other TCP applications.

ROM master object list

```
extern void    RpSetRomObjectList(void *theServerDataPtr,
                  rpObjectDescPtrPtr *theMasterObjectListPtr);
```

The ROM object master list consists of a list of pointers to object lists each of which point to a set of ROM objects. The lists are searched linearly, so there may be some small gains achieved by putting the popular pages towards the front. The home or root page is the first object in the first list (index offset 0). Other than that, there are no order dependencies in the lists.

The `RpSetRomObjectList` callback routine can be used to tell the RomPager engine to use an alternative ROM object master list. Since the object master list is a set of pointers to individual ROM object lists, the entire search list may be controlled at runtime by varying the contents of the object master list. This can be useful for dynamically enabling/disabling feature sets.

Browser request

```
extern char *  RpGetUserAgent(void *theServerDataPtr);
extern char *  RpGetAcceptLanguage(void *theServerDataPtr);
extern char *  RpGetHostName(void *theServerDataPtr);
```

The `RpGetUserAgent` routine is used to return the information from the User-Agent HTTP header that the browser sends in with the request. This can be useful in distinguishing between browser types to decide which kind of HTML to serve. The `RpGetAcceptLanguage` routine is used to return the information from the Accept-Language HTTP header. This may be useful for internationalization support. The `RpGetHostName` routine passes in the host name used by the browser to access the page. This may be useful for constructing fully qualified URL references.

User data

```
extern void    RpInitUserData(void *theServerDataPtr);
extern void    RpSetCookie(void *theServerDataPtr, void *theCookie);
extern void *  RpGetCookie(void *theServerDataPtr);
extern void    RpSetRequestCookie(void *theServerDataPtr, void *theCookie);
extern void *  RpGetRequestCookie(void *theServerDataPtr);
extern void *  RpGetRepeatWhileValue(void *theServerDataPtr);
```

The `RpInitUserData` routine is called by the RomPager engine at startup time if global data is assigned dynamically. This allows the device to perform any Web server specific data storage initialization.

The `RpSetCookie` routine is used to save a pointer to arbitrary user data in the engine data structure for retrieval later with the `RpGetCookie` routine. The `RpSetRequestCookie` routine saves a pointer to arbitrary user data in the current request for retrieval later with the `RpGetRequestCookie` routine.

The RpGetRepeatWhileValue routine can be used in individual Get functions that are called during processing of an eRpItemType_RepeatGroupWhile item. The routine retrieves the current value that was returned by the Repeat While function. This allows the individual Get functions to use the current repeat value as an index or in any other way to modify the value the Get function returns.

Time access

```
extern Unsigned32 RpGetMonthDayYearInSeconds(Unsigned32 theMonth,
         Unsigned32 theDay,Unsigned32 theYear);
extern Unsigned32 RpGetSysTimeInSeconds(void);
```

The RpGetSysTimeInSeconds routine is used internally by the RomPager engine and can also be used by the device page or form routines. It returns the system time in internal format, which is seconds since 1/1/1901. The RpGetMonthDayYearInSeconds call translates an external date into internal format.

Query index

```
extern Signed16   RpPopQueryIndex(void *theServerDataPtr);
extern void       RpPushQueryIndex(void *theServerDataPtr,
                     Signed16 theQueryValue);
extern Signed8    RpGetQueryIndexLevel(void *theServerDataPtr);
```

The query index callback routines are used to modify the query index state. The callback function RpPushQueryIndex places a new value on the query index stack and the RpPopQueryIndex routine pops a value off the stack and returns it to the caller. These routines can be used to control the index values that are passed to other pages. The values used in these routines are internal index values and therefore are 0-relative.

The RpGetQueryIndexLevel routine reports back the current index level in use starting at 0. If there are no index levels in use, the value returned will be -1.

Phrase dictionary

```
extern void    RpSetUserPhraseDictionary(void *theServerDataPtr,
                        char **theUserDictionaryPtr,
                        Boolean theCompressionFlag);
extern void    RpSetRequestUserPhraseDictionary(void *theServerDataPtr,
                     char **theUserDictionaryPtr,
                     Boolean theCompressionFlag);
```

The RpSetUserPhraseDictionary routine allows the user to change the current user phrase dictionary. A phrase dictionary is an array of char pointers indexed by characters used in the HTML text. Using alternate user phrase dictionaries can be helpful in setting up pages that will have alternate appearances depending on the dictionary.

The theCompressionFlag variable is used to signal whether the phrases in the dictionary can contain other compressed phrases. If theCompressionFlag is True, then the phrases in the dictionary being set may contain additional compressed phrases, from either the system or user dictionaries. This can reduce the size of a phrase dictionary if there are many common phrase fragments.

If `theCompressionFlag` is False, no further expansion will be performed on phrases in the dictionary being set. This means that any characters with high bits set will be passed directly to the browser. If a phrase dictionary is being set up with many double-byte phrases (for example, Japanese or Chinese), setting `theCompressionFlag` to False will reduce the overall size of the dictionary by eliminating the need for `kRpCompressionEscape` characters.

To reset the current phrase dictionary to the default user phrase dictionary, use the following call:

```
RpSetUserPhraseDictionary(theServerDataPtr, &gUserPhrases, True);
```

This routine changes default user dictionary for the entire system. The `RpSetRequestUserPhraseDictionary` routine changes the user dictionary for a single page or form request. This routine can be called from within the page pre-processing routine or from within the first item on a page or form to set the dictionary that will be used for processing the rest of the items of that page or form.

HTTP event logging

```
extern Signed16   RpGetHttpLogItemCount (void * theTaskDataPtr);
extern void       RpBuildHttpEventStrings (void * theTaskDataPtr,
                           Signed16 theIndex,char * theEventTimeString,
                           char * theEventTypeString,
                           char * theEventObjectString);
```

The RomPager engine stores an event in a memory ring buffer for every page that is served, every form that is processed, or other HTTP events. Two callback routines are available to query the log.

The `RpGetHttpLogItemCount` routine returns the number of items currently in the HTTP event log.

Given an item index, `RpBuildHttpEventStrings` builds the event time, type, and object strings in the string buffers specified by the caller. The string buffers contain short messages about the event and are 32 characters long. If the index is invalid, the strings will contain only a null terminator. If the index is valid, the event time and type strings will be created but the event object string may be empty because some tyeps do not have an associated object.

Form item handling

```
extern char *  RpGetFormBufferPtr(void *theServerDataPtr);
extern void    RpGetFormItem(char ** theBufferPtr,
                  char * theNamePtr,
                  char * theValuePtr);
extern void    RpSetFormObject(void *theServerDataPtr,
               rpObjectDescriptionPtr theObjectPtr);
extern void    RpReceiveItem(void *theServerDataPtr, char *theNamePtr,
               char *theValuePtr);
```

Normally, RomPager handles forms for the user application by processing items against a form item list pointed to by a form object. In some cases, an application may want to handle its own form items, or may want to use some of the internal RomPager routines to do partial form processing.

The RpGetFormBufferPtr call returns a pointer to the form arguments passed in from the browser with a GET Query command or a POST command. Request arguments from HTML forms are passed as a group of name/value pairs encoded in a format called Form URL Encoding. The RpGetFormItem routine is used to decode and retrieve a name/value pair from the form buffer. The buffer pointer will be updated to point to the next name/value pair and the values of the current name/value pair will be copied into the buffers passed as input to the call. The RpReceiveItem routine passes the name/value pair against the current ROM-based form object to use the standard RomPager processing to drive calls to the Set functions. The current form object may be set up by the RpSetFormObject call.

Number to string conversion

```
extern Unsigned16 RpConvertSigned32ToAscii(Signed32 theNumber,
                  char *theBufferPtr);
extern Unsigned16 RpConvertUnsigned32ToAscii(Unsigned32 theNumber,
                     char *theBufferPtr);
extern Unsigned16 RpConvertSigned64ToAscii(Signed64 theNumber,
                  char *theBufferPtr);
extern Unsigned16 RpConvertUnsigned64ToAscii(Unsigned64 theNumber,
                     char *theBufferPtr);
```

The RpConvertUnsigned32ToAscii and RpConvertSigned32ToAscii routines are used internally by the RomPager engine and can also be used by the device page display routines. These routines take parameters of a signed or unsigned 32 bit number and a char pointer to the output buffer. The routines also return the number of characters in the converted string. Similarly, the RpConvertUnsigned64ToAscii and RpConvertSigned64ToAscii routines are used internally by the RomPager engine and can also be used by the device page display routines if 64 bit integers are used.

String to number conversion

```
extern void    RpConvertHexString(void *theServerDataPtr,
               char *theHexPtr, char *theValuePtr,
               Unsigned8 theOutputCharCount, char theSeparator);
extern void RpConvertDotFormString(void *theServerDataPtr,
               char *theDotFormPtr, char *theValuePtr,
               Unsigned8 theOutputCharCount);
extern Signed8 RpConvertStringToSigned8(void *theServerDataPtr,
               char *theValuePtr);
extern Signed16   RpConvertStringToSigned16(void *theServerDataPtr,
               char *theValuePtr);
extern Signed32   RpConvertStringToSigned32(void *theServerDataPtr,
               char *theValuePtr);
extern Signed64   RpConvertStringToSigned64(void *theServerDataPtr,
               char *theValuePtr);
extern Unsigned8  RpConvertStringToUnsigned8(void *theServerDataPtr,
               char *theValuePtr);
extern Unsigned16    RpConvertStringToUnsigned16(void *theServerDataPtr,
               char *theValuePtr);
extern Unsigned32    RpConvertStringToUnsigned32(void *theServerDataPtr,
               char *theValuePtr);
extern Unsigned64    RpConvertStringToUnsigned64(void *theServerDataPtr,
               char *theValuePtr);
```

```
extern rpItemError   RpGetConversionErrorCode(void *theServerDataPtr);
extern void    RpSetUserErrorMessage(void *theServerDataPtr,
char *theMessagePtr);
```

The string to number routines are used internally by the RomPager engine during the forms item handling and can also be used by the device form item routines. These routines (RpConvertHexString, RpConvertDotFormString, RpConvertStringToSigned8, RpConvertStringToSigned16, RpConvertStringToSigned32, RpConvertStringToUnsigned8, RpConvertStringToUnsigned16, and RpConvertStringToUnsigned32) are normally used by the engine to perform the conversion according to the numeric type set in the fTextType field.

In the case where the device form item handling routine wishes to examine the string before the conversion, or look at the number after the conversion, the fTextType field should be set to eRpTextType_ASCII and the function pointed to by the fSetPtr field can call the conversion routine. The RpGetConversionErrorCode callback can be used to check the results of the conversion routine. The RpSetUserErrorMessage callback can be used to trigger an error page display with a custom message.

Delayed functions — external tasks

```
extern unsigned char    RpInitiateDelayedFunction(void *theServerDataPtr);
extern void             RpCompleteDelayedFunction(void *theServerDataPtr,
                     unsigned char theConnection);
extern unsigned char    RpGetCurrentConnection(void *theServerDataPtr);
extern unsigned char    RpSetCurrentConnection(void *theServerDataPtr,
                     unsigned char theConnection);
```

The process of preparing a page or processing form results may require access to an external task that can incur a delay. Such a task might be a read or write from a database, or a request for information over an internal network. To allow communications with these external tasks without suspending processing for other HTTP requests, the RomPager engine provides these callback routines for controlling the state of a RomPager internal task or connection.

The RpInitiateDelayedFunction routine is called when a delayed function is started and returns the connection number of the internal RomPager task that will be suspended.

The connection number should be saved away for use by the completion routine of the delayed function. When the completion routine of the delayed function is ready to allow the RomPager internal task to resume processing, it should issue the RpCompleteDelayedFunction call with the saved connection number.

If the completion routine of the delayed function needs to issue any other RomPager callback routines such as RpSetNextPage, it needs to issue an RpSetCurrentConnection call to set up RomPager for the correct RomPager internal connection. The RpSetCurrentConnection call returns the old current connection, which will need to be restored with another RpSetCurrentConnection call before the completion routine finishes.

The RpGetCurrentConnection call can be used to determine which connection is being used by the internal RomPager task.

The RpInitiateDelayedFunction function can be called from within the optional page pre-processing routine to set up local variables that will be accessed when the page is served or from the optional form post-processing routine to store away variables that were entered.

The following is an example of the processing flow:

RomPager task

```
/* begin page pre-processing */
/* spawn host OS task to do delayed processing */
/* suspend RomPager processing for the current connection */

theSuspendedConnectionId=RpInitiateDelayedFunction(theDataPtr);

/* allow RomPager task to process other connections */

return;
```

Spawned external task

```
/* begin external task execution */
/* do external tasks */
/* wait for external task completion */
/* finish external tasks */
/* set the RomPager current connection to the suspended connection */

theCurrentConnectionId=RpSetCurrentConnection(theDataPtr,
theSuspendedConnectionId);

/* issue engine control calls for the suspended connection */

RpSetNextPage(theDataPtr, &PgResultsPage);

/* the suspended connection */

RpCompleteDelayedFunction(theDataPtr, theSuspendedConnectionId);

/* restore the RomPager current connection */

theCurrentConnectionId=RpSetCurrentConnection(theDataPtr,
theCurrentConnectionId);

/* end external task execution */

return;
```

RomPager task

```
   /* finish page pre-processing */
   /* start page display */
```

The Web Application Toolkit supports the client pull method developed by Netscape. (The method is fully supported by Microsoft Internet Explorer.) With the client pull method, when an object is served to the browser, it is sent with an HTTP Refresh header that tells the browser when to ask for the next page to be served. The browser maintains a timer. If the user has not clicked on another link, the browser will automatically request the next page after the timer has expired. The next page is either the same as the previous page, or the Refresh header can optionally specify a different page to be served when the timer expires. In a Web-based management application, this can be useful to set up a series of pages that each point to the next one. For example, status page A would be displayed for 15 seconds, then status page B would be displayed for 10 seconds, followed by status page C, which would be displayed for 30 seconds. After this, status page A would be displayed again and the whole process would repeat.

To use the client pull method, a refresh time in seconds must be specified for the page. This can be specified when the page is created by specifying the `RpRefreshTime` keyword in the `RpPageHeader` tag. A page refresh cycle can also be started at runtime by using the `RpSetRefreshTime` callback routine. If a page other than the current page is to be refreshed, the new page can be specified at page creation by using the `RpRefreshPage` keyword or at runtime by using the `RpSetRefreshPage` callback routine. The page refresh cycle can be ended by serving a page that has a refresh time of 0, or by calling the `RpSetRefreshTime` routine with a value of 0.

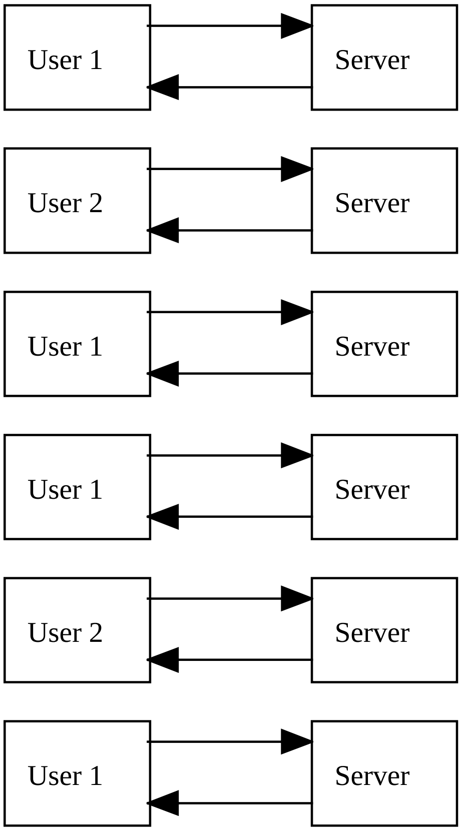
An alternative method for using the client pull refresh technique is to use the `<META HTTP-EQUIV>` tag in the page that is sent to the browser. The purpose of this tag is to send a header inside the HTML page. This header is ignored by the server but treated by the browser as if it received this header with the rest of the HTTP headers from the server.

To use this technique, `<META HTTP-EQUIV>` is placed in the `<HEAD>` section of the page as shown in the following examples.

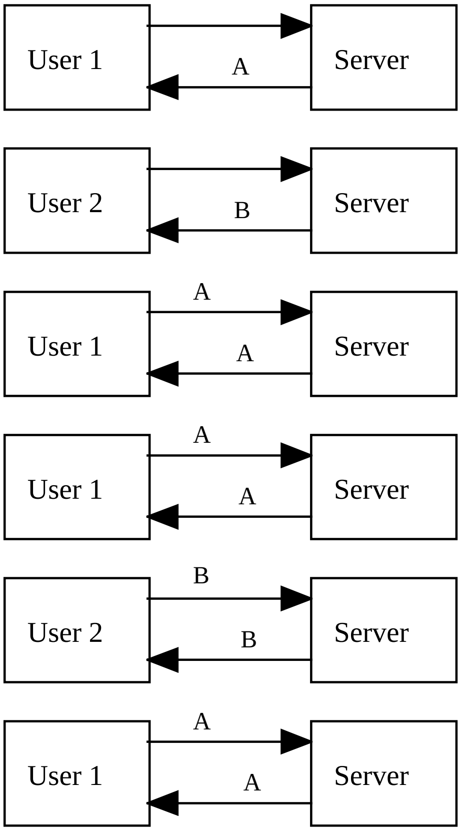
```
<HTML>
<HEAD>
<TITLE>Refresh this page in 10 seconds</TITLE>
<META HTTP-EQUIV=“Refresh” CONTENT=“10”>
</HEAD>
<BODY>
Refresh this page in 10 seconds
</BODY>
</HTML>

<HTML>
<HEAD>
<TITLE>Refresh another page in 5 seconds</TITLE>
<META HTTP-EQUIV=“Refresh” CONTENT=“5;URL=http://www.allegrosoft.com/index.html”>
</HEAD>
<BODY>
Refresh another page in 5 seconds
</BODY>
</HTML>
```

State Management

The HTTP protocol is a stateless protocol. Each request from a client browser to a Web server is an individual request from the point of view of the protocol. From the point of view of a Web application written on the server, it may be desirable to identify multiple requests from the same user and group them into a session. The server creates a session by assigning a “State” or a “Session ID” and sending it with the response. If the browser supports the technique the server is using, then the browser will return the “State” on subsequent requests, allowing the server to recognize subsequent requests from the same user.



Normal Requests with no States



Requests with States

The Advanced Web Server supports two techniques for state management. The first technique is to hide the state information in the URL, and the second technique is to use HTTP cookies.

Hiding information in the URL is a low-overhead technique that works with any browser but is limited in the amount of state information that can be passed. The HTTP cookies technique was invented by Netscape and is supported by most recent browsers. HTTP cookies have somewhat higher overhead than URL hiding, but allow greater amounts of state information to be passed back and forth.

The RomPager URL hiding technique works by prepending the URL in a link with a signal sequence and a state value. When the Web application wants to set a state for a user, it uses the `RpSetUrlState` call to store a string into the Web server. When a link or pathname is generated with `RpUrlState`, it will have the form `http://`*`deviceaddress`*`/US/`*`SSSSS`*`/`*`pathname`* where *SSSSS* is the value of the state string. On subsequent requests, the URL is examined to see if it is in this format. If it is, the state information is stripped off from the URL and stored where the application can retrieve it using `RpGetUrlState`. The rest of the URL pathname is passed to the engine for action. The URL state can be cleared at the server by using the `RpSetUrlState` call to set the state to a null string. When the browser quits, it will stop using the special URLs and the session will be terminated.

The HTTP cookies technique works by sending state information in special HTTP headers. While these headers are non-standard, most recent browsers will support them. The HTTP cookie headers allow multiple states to be passed between the server and the browser. In the Advanced Web Server, if HTTP cookies are enabled, an array of character strings is available for storing and retrieving state information. The size of the array is 6 and the cookie size is 64. The `RpSetHttpCookie` call is used to set a cookie value that will be passed to the browser and the `RpGetHttpCookie` call is used to retrieve the cookie that the browser has sent in. These calls use theIndex as a parameter to choose which string in the array to reference. Once a cookie has been sent to a browser, it will send it in with each subsequent request. When the user quits the browser, it deletes the stored cookies, and the session is completed.

With both techniques, a session is a series of independent requests that the server has asked the browser to tag with state information. There is no notification to the server by the browser when the session completes. The server can deduce that a session has completed by observing that no requests from a tagged user have come in for a while. Application timers can be used to determine session ends so that session resources can be freed up on the server side.

Security Control

Overview

The internal Web server security environment consists of a user database, a set of security realms, and the individual pages and other resources to be protected. A security realm is a collection of objects (pages, forms, images, and so on) that are grouped together with a common security level. An individual page may belong to one or more realms, and an individual user may have access to one or more realms.

The first time a browser request (GET, POST, and so on.) is made for an object in a given realm, the user is challenged to provide their known username and password. When a user successfully logs on, a security session is established for that user. Subsequent requests are negotiated by the browser on behalf of the user, so that the user doesn't have to re-enter the username-password for each request, but the request is still authenticated, since the browser will carry the user's authentication credentials for each subsequent request.

The HTTP protocol currently supports two authentication methods — basic and digest.

The basic method encodes the username-password using the Base64 encoding scheme and sends it over the wire to the server that decodes it and compares it with the known username-password. The basic method is supported by all browsers, and provides some protection, in that another browser user can not access the protected realm without knowing the username-password. Since a determined attacker can use a packet sniffer to gain the encoded username-password string and since the Base64 algorithm is a widely known reversible encoding scheme, this method provides minimal protection, but is better than plain text passwords.

The digest method provides stronger protection, since it uses one-way encoded hash numbers and never sends the username-password over the wire. With the digest method, the server creates a challenge code and sends it to the browser. The browser uses the MD5 one-way hash algorithm to create a response based on the challenge plus the user provided username-password. The server calculates what the expected response should be using its copy of the username-password and challenge and compares it with the actual response. If they match, access to the object is granted. Since the challenge can change with each transaction, and the username-password is never sent over the wire, this method provides fairly strong protection. The challenge is usually made unique by incorporating such things as the device IP address and the time, so that each device will send out unique challenges. The digest method provides strong authentication and is appropriate for an embedded environment, but at present has not been widely implemented by most browsers. Internet Explorer 5.0 and above supports part of the digest authentication specification.

The Advanced Web Server supports both basic and digest authentication. Pages and forms may be assigned to eight different realms. The realms may be overlapping, thus allowing the creation of supersets and sidesets. You may have one realm for regular users, another for supervisory users, and a third for field engineering users, for example. The supervisory and field engineering users might each have access to all the pages and forms of the regular users but no access to each other's unique pages and forms.

An additional security method uses the Secure Socket Layer (SSL) protocol or a later version, Transport Layer Security (TLS) to provide an encryption layer between the TCP protocol and the HTTP protocol. If SSL/TLS encryption is used, HTTP basic authentication is usually sufficient for identifying the authorized user. The RomPager Secure Web server provides SSL/TLS capabilities as well as HTTP basic and digest authentication.

Implementation

The security database contains a user data base and a realm data base. Each user has a username, password, an access code that indicates the realms that the user has access to and a session timer that indicates how long the user has access to the realm without being rechallenged. Each username is unique in the database. The number of users to allow in the security database is 12.

There are a maximum of 9 realms that each have a realm name, and a security level. The realm name is passed to the browser so that it can be displayed in security dialogs to the user. Realms are accessed by an index of 0 to 8 in the security callback access routines. The security level indicates what services the Advanced Web Server should provide.

The individual pages and forms have an access code that indicates which security realms that the object belongs to. The access codes for an object are specified in the HTML source using the RomPager comment tags. The access code for a page is set using the `RpAccess` parameter of the `RpPageHeader` comment tag, and the access code for a form is set using `RpAccess` of the `RpFormHeader` comment tag. The internal storage of the object access codes is in the `fObjectAccess` field of the `rpObjectDescription` structure. The default for each object is `kRpPageAccessUnprotected`. An object may belong to multiple realms by setting multiple flags. The User Settings (`PgUsers`) page in the sample pages is an example page that belongs to the first and second realms. Using overlapping realms allows the creation of superuser realms where one realm has complete access and another realm has only partial access.

When an object that belongs to a protected given realm is accessed, the Advanced Web Server will send a challenge to the browser the first time an object in the realm is accessed. The browser will display a request to the user that shows the realm name and asks for a username and password to be entered. Different browsers will use different dialog formats for the password request, but all display the username in clear text and the password in obscured form. With HTTP basic authentication, the username-password combination is returned to the server in the Base64 encoding scheme. When the server receives a valid username-password, it establishes a security session for that user. As long as the session is active, the user will have access to the objects in all the realms for which that user is authorized.

If an object belongs to more than one realm, RomPager uses the security method of the least secure realm to determine which realm to issue a challenge for. If an object belongs to multiple realms that all have the same security method, the lowest numbered realm is used in the challenge. If the application is going to use different security levels for different realms, a good way to organize the realms is to assign the highest level of security to the highest numbered realm.

As the user makes additional requests, the browser will send along the encoded username-password combination that was last entered. If the user has requested another object in the same realm, the server will use the received username-password combination to authorize the request. If the user requests an object in another realm, the server will issue another challenge to the browser, which will in turn ask the user for a new username-password combination.

The RomPager engine provides a series of callback routines that can be used to dynamically administer the security database. The intialization of the security tables is performed using these callback routines by the `RpInitializeBox` routine (in `RpUser.c`). The security callback routines are used for realm database manipulation, user database manipulation, session control and feature access control. These routines may be called at initialization, at various exit points in the Web application, or from other device management functions such as SNMP or Telnet environments.

Realm manipulation routines

```
extern char *     RpGetRealmName(void *theTaskDataPtr, unsIgned chartheIndex);
extern void    RpSetRealmName(void *theTaskDataPtr, unsIgned chartheIndex,
               char * theRealmNamePtr);
extern rpSecurityLevel  RpGetSecurityLevel(void*theTaskDataPtr,
                        unsigned char theIndex);
extern void          RpSetSecurityLevel(void*theTaskDataPtr,
                     unsigned char theIndex,
                     rpSecurityLevel theSecurityLevel);
extern void          RpRealmLocking (void *theTaskDataPtr,
                     Boolean theLockState);
```

The RpGetRealmName routine is passed an index (0–8) of the realm and returns a pointer to an ASCII string containing the current realm name. The RpSetRealmName routine is passed an index (0-8) of the realm and a pointer to an ASCII string containing the new realm name. The realm name is sent to the browser when a protected object is accessed, so that the name can be displayed in security dialogs to the user.

The RpGetSecurityLevel routine is passed an index (0–8) of the realm and returns a security level code. The RpSetSecurityLevel routine is passed an index (0–8) of the realm and a security level code indicating the new security level for the realm. The available security level codes are:

```
eRpSecurity_Disabled
eRpSecurity_PasswordOnly
eRpSecurity_PasswordAndIpAddress
eRpSecurity_DigestPasswordOnly
eRpSecurity_DigestAndIpAddress
eRpSecurity_StrictDigestPasswordOnly
eRpSecurity_StrictDigestAndlpAddress
eRpSecurity_SecureSocketOnly
eRpSecurity_SecureSocketPasswordOnly
eRpSecurity_SecureSocketAndIpAddress
```

The eRpSecurity_Disable security level turns off security for a particular realm. Pages and objects belonging to a realm with this level are unprotected.

The eRpSecurity_PasswordOnly security level allows a user with knowledge of the username and password access to the realm. The HTTP basic authentication mechanism is used for this security level.

The eRpSecurity_PasswordAndIpAddress security level requires that the user provide the username and password and be running a browser from a specified IP address in order to gain access to the realm. The HTTP basic authentication mechanism is used for this security level. The eRpSecurity_DigestPasswordOnly security level allows a user with knowledge of the username and password access to the realm. The HTTP digest authentication mechanism is used for this security level. A new challenge is created once per session. This method is supported by Internet Explorer 5.0 and above.

The eRpSecurity_DigestAndIpAddress security level requires that the user provide the username and password and be running a browser from a specified IP address in order to gain access to the realm. The HTTP digest authentication mechanism is used for this security level. A new challenge is created once per session. This method is supported by Internet Explorer 5.0 and above.

The eRpSecurity_StrictDigestPasswordOnly security level allows a user with knowledge of the username and password access to the realm. The HTTP Digest Authentication mechanism is used for this security level. A new challenge is created for every request. Currently there are no known browsers supporting this capability.

The eRpSecurity_StrictDigestAndIpAddress security level requires that the user provide the username and password and be running a browser from a specified IP address in order to gain access to a realm. The HTTP digest authentication mechanism is used for this security level. A new challenge is created for every request. Currently there are no known browsers supporting this capability.

The last three security level codes are available only when you are using the RomPager Secure Web server, which includes SSL/TLS support.

The eRpSecurity_SecureSocketOnly security level sets up an encrypted connection between the server and the browser for objects in a realm with this security level. It does not require any other additional user authentication. The SSL protocol is used for this security level.

The eRpSecurity_SecureSocketPasswordOnly security level allows a user with knowledge of the username and password access to the realm. The HTTP basic authentication mechanism within an SSL connection is used for this security level.

The eRpSecurity_SecureSocketAndIpAddress security level requires that the user provide the username and password and be running a browser from a specified IP address in order to gain access to the realm. The HTTP basic authentication mechanism within an SSL connection is used for this security level.

If multiple users have access to the same realm, or if multiple users are using the same username/password identifier, they will be able to access the same pages and resources from the Web server. For pages that only display information, multiple access is fine, but for pages that have forms to control parameter values this can cause problems. If two users each access the same control page at the same time, and each makes changes based on the information they see and then submit a form, the first set of changes will be lost, since the second user will overide them.

To prevent this possibility and allow control of critical resources, the security system offers a concept called realm locking, which is controlled by the RpSetRealmLocking callback routine. The RpSetRealmLocking routine is called to enable and disable realm locking. If realm locking is enabled, when a user logs on successfully, all the pages in the realms the user has access to become reserved for that user until the user completes the session. Other users who log on (even if they normally have access to a given realm) will be blocked from accessing the pages and forms in that realm until the first user logs off or realm locking is disabled. This capability is useful to protect device information from being changed by multiple users at the same time.

User database manipulation routines

```
extern   Boolean RpSetUserAttributes(void *theTaskDataPtr, char *theUsernamePtr,
                                     char *thePasswordPtr, rpAccess theAccessCode
                                     Unsigned32 thelpAddress,
                                     Unsigned16 theTimeoutSeconds);
extern void  RpGetUserAttributes(void *theTaskDataPtr, Unslgned16 theUserlndex,
                                 char **theUsernamePtr, char **thePasswordPtr,
                                 rpAccess *theAccessCodePtr,
                                 Unsigned32 *theIpAddressptr,
                                 Unsigned16 *theTlmeoutSecondsPtr);
extern void     RpDeleteUser(void*theTaskDataPtr, cchar *theUsernamePtr);
extern char *   RpGetCurrentUserName(void *theTaskDataptr);
extern void     RpSetPasswordCookie(void *theTaskDtrPtr, char *theUsernamePtr,
                void *theCookie);
extern void*    RpGetPasswordCookie(Void *thetaskDataPtr, char *theUsernamePtr);
```

The RpSetUserAttributes routine is called to create a user entry for a new user or modify the attributes of an existing user. Each username in the user database must be unique. The first time RpSetUserAttributes is called for a specific username, a new entry will be set up in the user database. Additional calls with the same username can be used to change the attributes for a specific user. RpSetUserAttributes return False if there is no more room in the user database. If the value of the IpAddress parameter is 0, any IP address will be acceptable. If an IP address is specified, then objects that belong to a realm with an appropriate security level code (such as eRpSecurity_PasswordAndIpAddress) will require the user to be running a browser at the specified address in order to access the object. If the value of the theTimeoutSeconds parameter is 0, the server's master security session timeout value will be used.

The RpGetUserAttributes routine is passed an index (0-based)into the user routine and returns the username, password, realm access code, IP address, and session value for the given user entry. If there is no user entry for the requested index, the return value of theUsernamePtr will be NULL. This routine is useful for sequentially accessing the RomPager security database in order to store the information somewhere else. The user database entries are returned sequentially, so once an empty user entry is returned, no more user entries with a higher index.

The RpDeleteUser routine is passed a pointer to an ASCII string containing the name of the user to be deleted. This routine is used to remove an active entry from the user database. If the user being deleted has an active security session, it will be terminated and the user entry will be deleted.

The RpGetCurrentUserName routine can be called from within a page to determine the name of the user that has authenticated access. It returns a pointer to an ASCII string containing the name of the current authenticated user.

The internal user security database can store an opaque variable that can be used for communicating with external password servers such as Radius servers. The opaque variable (or cookie) is set using the RpSetPasswordCookie routine and retrieved using the RpGetPasswordCookie routine.

Session manipulation routines

```
extern void    RpSetCurrentSessionCloseFunction(void *theTaskDataPtr,
      rpSessionGloseFuncPtr theFunctionPtr, void *theUserCookie);
typedef void (*rpSes sionCloseFuncPtr) (void *theTaskDataPtr,
              void *theUserCookie);
extern void    RpSetServerPasswordTImeout(voId*theTaskDataPtr,
               Signed16 theTimeoutSeconds);
extern Boolean RpCheckSession(void *theTaskDataPtr,rpAccess theAccessCode);
exter void     RpResetCurrentSession(void*theTaskDataPtr,
              Signed16Ptr thelndexValuesPtr);
extern void RpResEtUserSession(Void *therask9ataPtr, char*theUsernamePtr);
```

Use RpSetCurrentSessionCloseFunction to set up a routine that will be called when the current user security session completes. Typically, this happens when the user's security session timer expires. The value of the security session timer is set when the user entry is created or modified with RpSetUserAttributes. An opaque variable (theUserCookie) can be passed with RpSetCurrentSessionCloseFunction. This variable will be returned by a completion function (of type rpSessionCloseFuncPtr) when the session times out.

The RpSetServerPasswordTimeout routine is used to change the server's master security session timeout value. The master security session timeout value is used as the default timeout value for calls to the RpSetUserAttributes routine that have the theTimeoutSeconds parameter set to 0. The RomPager engine maintains timeout counters for each user session. The timeout counter is initialized to the timeout value for the user each time a protected object is accessed. If the counter hits 0, the server will force a challenge to the browser, even if the correct username-password combination was sent in with the request. This technique can be used to protect objects from access by an unauthorized user who uses an unattended browser at an authorized user's desk.

The RpCheckSession routine is used to find out the current access state for a realm or realms, The call passes in an access code containing the realms to be checked and receives back a Boolean indicating whether access is currently authorized for the realms.The RpResetCurrentSession routine can be called from within a page to reset the security session for the current authenticated user. The user security session will be reset and any locked realms will be released. Further access by the user will result in a re-challenge. This call uses the rpProcessDataFuncPtr format so that it may be issued directly from from a page or form to terminate a security session for the current user.

The RpResetUserSession routine may be called from any point within the device and will force the reset of a specific user security session.

Access control routines

```
   extern void       RpSetPutAccess(void *theTaskDataPtr, rpAccess theAccessCode};
```

The security access codes specifies a realm or realms that a given object or function belongs to. The format of the access code uses flags such as kRpPageAccessRealml, kRpPageAccessRealm2, and so on, to identify the realms. Each realm in the realm database in turn contains the security level code used to control access to the object or function.

Access codes for pages and forms that are stored in ROM are assigned by using HTML comment tags in the pages that are compiled with PageBuilder. For objects stored in the file system, the access codes are returned when the file is opened.

The RpSetPutAccess routine is used to assign the security realm or realms that control access by the HTTP PUT command. Since the PUT command can directly load a file into the device, only users that are authorized for the realms specified by the theAccessCode parameter will be able to upload files to the device.

External security validation

```
   extern rpPasswordState  SpwdGetExternalPassword(void*theTaskDataPtr,
                                   Unsigned8 theConnectibnld;
                                   char *theUsernamePtr,
                                   char *thePasswordPtr,
                                   Unsigned32 *thelpAddressPtr,
                                   rpAccess *theRealmAccessPtr);
```

Normally, the RomPager engine maintains internal security databases for handling HTTP user authentication requests. Another approach to user authentication is to have the RomPager engine handle the HTTP security protocols and pass the actual authentication request to a process elsewhere in the device.

The `SpwdGetExternalPassword` routine is called to obtain the external information that the server needs to validate the information supplied by the browser. `SpwdGetExternalPassword` provides a pointer to the username that needs to be validated, and pointers for the routine to return validation information. The external validation routine is also provided with a connection number that may be used to aid in processing multiple simultaneous validation requests.

The external validation routine needs to return three pieces of information so that the Advanced Web Server can finish the validation process. The valid password must be copied to the buffer pointed to by the `thePasswordPtr` parameter. If the user should be restricted to making accesses from a particular IP address, this address should be placed in the area pointed to by the `theIpAddressPtr` parameter. If no restrictions are placed on the IP address, the value of the address should be set to 0. Lastly, the validation application should provide the realm access code that specifies which realms the user has access to.

The external validation routine is called the first time that the browser provides user credentials after being challenged. For subsequent requests for the same user session, the Advanced Web Server will use the information provided by the external validation routine to validate these requests. If the user session is terminated, or timed out for inactivity, or if a different user attempts access, the external password function will be called again to provide the validation information.

From the point of view of the host operating system, the RomPager engine is a single task. The engine contains its own scheduler and control blocks for supporting multiple HTTP requests. The call it makes to the external password routine must be asynchronous for any activity that will incur delay in order to allow the engine to service other requests. The call return state is used to determine whether the external password function has been completed. If the username is to be validated on an external password server, any verification operation that can incur delay needs to create a separate operating system task that can block on call completion.

The possible return states from theSpwdGetExternal Password call are:

`eRpPasswordPending` — The external authorization process is not yet complete. The RomPager engine will call the `SpwdGetExternalPassword` routine again to retrieve the validation information.

`eRpPasswordNotAuthorized` — The external process is complete, the user is not authorized because the username was not recognized. The Advanced Web Server will reject the request.

`eRpPasswordDone` — The external process is complete. The user is partially authorized. The string pointed to by `thePasswordPtr` has been filled in with the authorized password from the external database. The realm access flag has been set. The IP address (if any) has been set. The Advanced Web Server will complete the rest of the authorizaton process.

ROM object list

The ROM object list is used by the RomPager engine to find the stored ROM objects (pages, forms, graphics, applets, etc.). The ROM object list consists of a master object list that points to individual object lists, each of which point to a set of ROM objects. The lists are searched linearly, so there may be some small gains achieved by putting the popular pages towards the front. The home or root page is the first object in the first list (Index offset 0). Other than that, there are no order dependencies in the lists.

The ROM object list for a single page application is shown in **`RpPage1.c`** and looks like this:

```
rpObjectDescPtrPtr   gRpMasterObjectList[] = {
                  gRpObjectList,
                  0
};
rpObjectDescriptionPtr  gRpObjectList[] = {
                     &PgFirstPage,
                     0
};
```

The ROM object list for the multi-page demo application is shown in **`RpPages.c`** and looks like this:

```
extern rpObjectDescriptionPtr gRpValidationObjectList[];
extern rpObjectDescriptionPtr gRpChassisObjectList[];
extern rpObjectDescriptionPtr gRpJavaDemoObjectList[];
extern rpObjectDescriptionPtr gRpRomPopObjectList[];

rpObjectDescPtrPtr   gRpMasterObjectList[] = {
                  gRpObjectList,
                  gRpValidationObjectList,
                  gRpChassisObjectList,
                  gRpJavaDemoObjectList,
                  gRpRomPopObjectList,
                  0
};
rpObjectDescriptionPtr gRpObjectList[] = {
                  &PgMain,
                    ...
                  0
};
```

Object headers

The object header is used by the RomPager engine to analyze the HTTP request and prepare the necessary HTTP headers in the response. The source definitions of the object header structures can be found in the **RpPages.c** file. The structure of the rpObjectDescription is as shown as follows:

```
typedef struct rpObjectDescription {
      char *                 fURL;
      rpItem *               fItemsArrayPtr;
      rpObjectExtensionPtr fExtensionPtr;
      Unsigned32             fLength;
      rpAccess               fObjectAccess;
      rpDataType             fMimeDataType;
      rpObjectType           fCacheObjectType;
   }  rpObjectDescription, *rpObjectDescriptionPtr;
```

The fURL field contains the path used to find the object.

The fItemsArrayPtr points to the list of elements used to process the object.

The fExtensionPtr points to an optional extension structure, which is discussed below.

The fLength field contains the length of the object if known. Images and applets are usually fixed length items whose length will be filled in by PBuilder. The length of HTML pages with dynamic text or form items is calculated at run time.

The fObjectAccess field contains the security realms for the object. An unprotected object has the value kRpPageAccess_Unprotected in this field, while an object belonging to both realms 1 and 4 has the value kRpPageAccess_Realm1+kRpPageAccess_Realm4 in this field.

The fMimeDataTypefield contains the MIME type of the HTTP object. Typical values are eRpDataTypeHtml and eRpDataTypeImageGif.

The fCacheObjectType field contains info used to control browser caching. Static objects are allowed to be cached by the browser, while dynamic objects are forced to be updated, as the information on the page may have changed. The values stored in this field are eRpObjectTypeStatic and eRpObjectTypeDynamic.

The optional rpObjectExtension structure is used by all form objects and by other objects that need additional capabilities. The structure looks like this:

```
typedef  struct rpObjectExtension {
      rpProcessDataFuncPtr fProcessDataFuncPtr;
      rpObjectDescription *   fPagePtr;
      rpObjectDescription *   fRefreshPagePtr;
      Unsigned16              fRefreshSeconds;
      rpObjectFlags           fFlags;
      char *                  fJavaScriptPtr;
   }  rpObjectExtension, *rpObjectExtensionPtr;
```

The fProcessDataFuncPtr points to an optional processing routine that can be called for the object. If the object is a page, the routine is called before any element processing occurs. If the object is a form, the routine is called after all the individual form elements are processed.

The fPagePtr points to the page that will be served after the form is processed. This field can be overridden using the RpSetNextPage or RpSetNextFilePage callback routine.

The `fRefreshSeconds` field is used to specify the length in seconds of the refresh time. The `fRefreshPagePtr` points to the object to be served after the refresh time expires. For a fuller discussion of refreshing pages with the client pull method, see the Dynamic Pages section.

The `fFlags` field contains optional flags that control special behavior for the object.

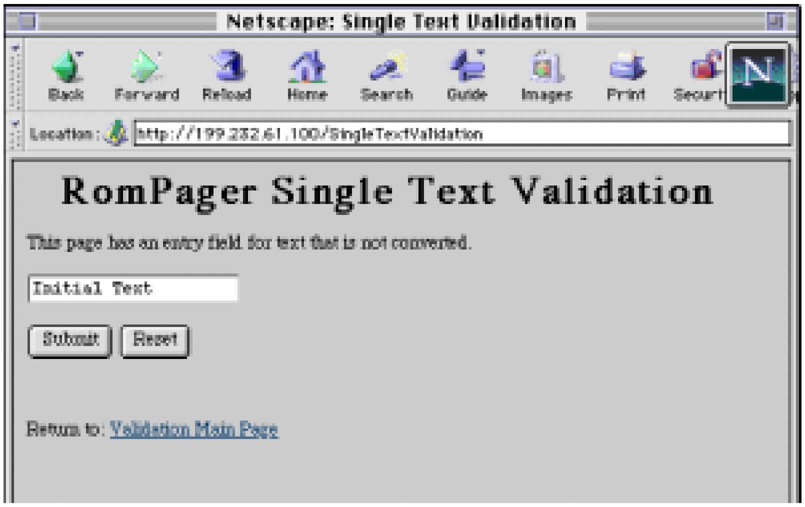
- If the `kRpObjFlag_Direct` flag is set on a form object, the HTML for the following page object will be sent directly. Normally, the HTML for the following page object is sent using HTTP redirect commands to preserve the browser cache and history information.
- The `kRpObjFlag_Aggregate` flag is used to identify a page that will be sent by RomMailer with embedded images.
- The `kRpObjFlag_Disposition` flag is used to identify a page that is sent to the browser with an HTTP header of Content-Disposition: Attachment. Some browsers use this header to store a page directly as a file rather than interpreting it.

The `fJavaScriptPtr` field contains a pointer to the Java Script that is inserted into the `<FORM>` tag if JavaScript is being used.

Appendix A
PBuilder HTML Comment Tag Examples

These example pages are taken from the demonstration pages shipped with the Advanced Web Server. To see the pages in action, compile the demo pages with your device and use a standard browser to view the pages.

Example 1: Single text validation page



http://199.232.61.100/SingleTextValidation

RomPager Single Text Validation

This page has an entry field for text that is not converted.

Initial Text

Return to: Validation Main Page

HTML with comment tags

```
<!-- RpPageHeader RpUrl=/SingleTextValidation -->
<HTML>
<HEAD>
<TITLE>Single Text Validation</TITLE>
</HEAD>
<BODY>
<H1 ALIGN=CENTER>RomPager Single Text Validation</H1>
<P>This page has an entry field for text that is not converted.</P>
<!-- RpFormHeader ACTION=“/Forms/SingleTextValidation” -->
<FORM METHOD=“POST” ACTION=“/Forms/SingleTextValidation”>
<!-- RpEnd -->
<!-- RpFormInput TYPE=“TEXT” NAME=“TextInput” SIZE=“20” MAXLENGTH=“20”
   RpGetType=Direct RpGetPtr=theTextValue
   RpSetType=Direct RpSetPtr=theTextValue RpTextType=ASCII -->
<INPUT TYPE=“TEXT” NAME=“TextInput” SIZE=“20” MAXLENGTH=“20” VALUE=“Initial
Text”>
<!-- RpEnd -->
<BR>
<BR>
<!-- RpFormInput TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit” -->
<INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit”>
<!-- RpEnd -->
```

```
<!-- RpFormInput TYPE=“RESET” VALUE=“Reset” -->
<INPUT TYPE=“RESET” VALUE=“Reset”>
<!-- RpEnd -->
<P>&nbsp;</P>
Return to: <A HREF=“ValidationSuite”>Validation Main Page</A>
</FORM>
</BODY>
</HTML>
```

C source file generated by PBuilder

```
#include “RpExtern.h”

extern rpObjectDescription PgSingleText;

static char PgSingleText_Item_1[] =
   C_oHTML_oHEAD_oTITLE “Single Text Validation” C_xTITLE_xHEAD_oBODY “<H1”
   C_ALIGN_CENTER “>“ C_S_RomPager “ Single Text Validation” C_xH1 C_oP
   “This page has an entry field for text that is not converted.”
   C_xP;

extern char *theTextValue;
static rpTextFormItem PgSingleText_Item_3 = {
   “Text”,
   &theTextValue,
   &theTextValue,
   eRpVarType_Direct,
   eRpVarType_Direct,
   eRpTextType_ASCII,
   20,
   20
};
static char PgSingleText_Item_4[] =
   C_oBR
   C_oBR;
static rpButtonFormItem PgSingleText_Item_5 = {
   “Submit”
};
static rpButtonFormItem PgSingleText_Item_6 = {
   “Reset”
};
static char PgSingleText_Item_7[] =
   C_oP_NBSP_xP “\n”
   “Return to: “ C_oANCHOR_HREF “ValidationSuite\”>Validation Main Page”
   C_xANCHOR C_xFORM
   C_xBODY_xHTML;
static rpItem PgSingleText_FormItems[] = {
   { eRpItemType_FormAsciiText, &PgSingleText_Item_3 },
   { eRpItemType_FormSubmit, &PgSingleText_Item_5 },
   { eRpItemType_FormReset, &PgSingleText_Item_6 },
   { eRpItemType_LastItemInList }
};
```

```
static rpObjectExtension PgSingleText_FormObjectExtension = {
   (rpProcessDataFuncPtr) 0,
   &PgSingleText,
   (rpObjectDescriptionPtr) 0,
   0,
   kRpObjFlags_None
};
rpObjectDescription PgSingleText_Form = {
   “/Forms/SingleTextValidation”,
   PgSingleText_FormItems,
   &PgSingleText_FormObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeForm,
   eRpObjectTypeDynamic
};
static rpItem PgSingleText_Items[] = {
   { eRpItemType_DataZeroTerminated, &PgSingleText_Item_1 },
   { eRpItemType_FormHeader, &PgSingleText_Form },
   { eRpItemType_FormAsciiText, &PgSingleText_Item_3 },
   { eRpItemType_DataZeroTerminated, &PgSingleText_Item_4 },
   { eRpItemType_FormSubmit, &PgSingleText_Item_5 },
   { eRpItemType_FormReset, &PgSingleText_Item_6 },
   { eRpItemType_DataZeroTerminated, &PgSingleText_Item_7 },
   { eRpItemType_LastItemInList }
};
rpObjectDescription PgSingleText = {
   “/SingleTextValidation”,
   PgSingleText_Items,
   (rpObjectExtensionPtr) 0,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeHtml,
   eRpObjectTypeDynamic
};
```

C stub routines file generated by PBuilder

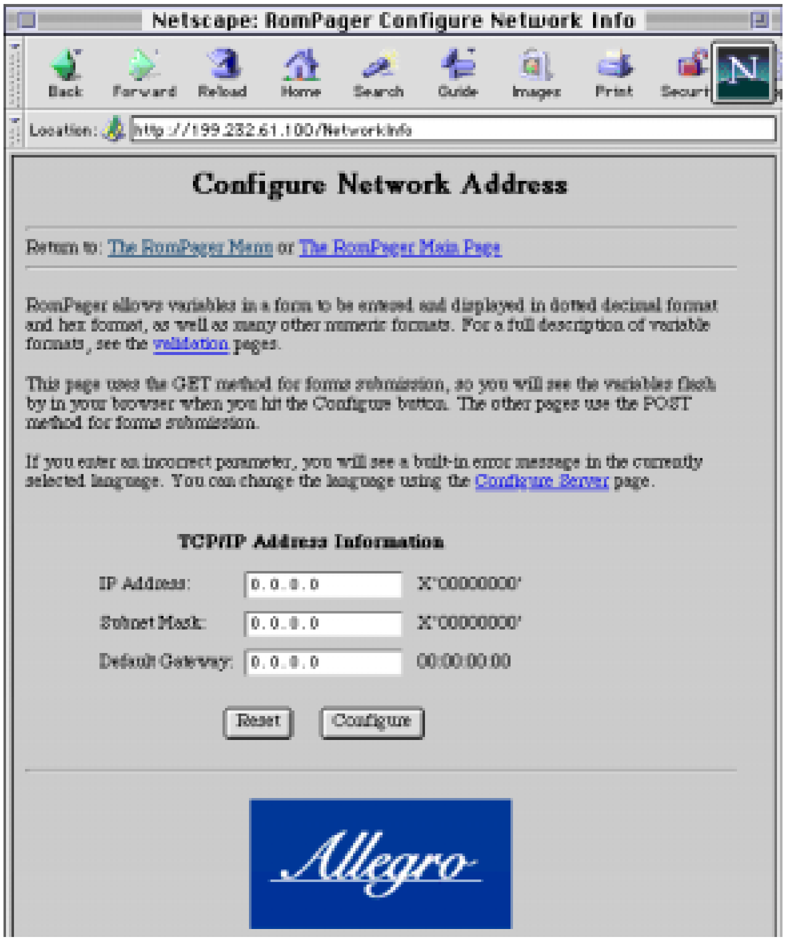
```
#include “RpExtern.h”
char *theTextValue = “ “;
```

HTML without comment tags as served by the Advanced Web Server

```
<HTML>
<HEAD>
<TITLE>Single Text Validation</TITLE>
</HEAD>
<BODY>
<H1 ALIGN=center>RomPager Single Text Validation</H1>
<P>
This page has an entry field for text that is not converted.</P>
<FORM METHOD=“POST” ACTION=“/Forms/SingleTextValidation”>
<INPUT TYPE=“TEXT” NAME=“Text” SIZE=“20” MAXLENGTH=“20” VALUE=“Initial Text”>
<BR><BR>
<INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Submit”>
<INPUT TYPE=“RESET” VALUE=“Reset”>
```

```
<P>&nbsp;</P>
Return to: <A HREF="ValidationSuite">Validation Main Page</A>
</FORM>
</BODY>
</HTML>
```

Example 2: Configure network info page



http://199.232.61.100/NetworkInfo

Configure Network Address

Return to: The RomPager Menu or The RomPager Main Page

RomPager allows variables in a form to be entered and displayed in dotted decimal format and hex format, as well as many other numeric formats. For a full description of variable formats, see the validation pages.

This page uses the GET method for forms submission, so you will see the variables flash by in your browser when you hit the Configure button. The other pages use the POST method for forms submission.

If you enter an incorrect parameter, you will see a built-in error message in the currently selected language. You can change the language using the Configure Server page.

TCP/IP Address Information

IP Address:	0.0.0.0	X'00000000'
Subnet Mask:	0.0.0.0	X'00000000'
Default Gateway:	0.0.0.0	00:00:00:00

HTML with comment tags

```
<HTML>
<HEAD>
<TITLE>RomPager Configure Network Info</TITLE>
</HEAD>
<BODY>
<P>
<H2><CENTER>Configure Network Address</CENTER>
</H2>
<!-- RpFormHeader ACTION="/Forms/NetworkInfo" -->
<FORM METHOD="GET" ACTION="/Forms/NetworkInfo">
<!-- RpEnd -->
<P>
```

```
<HR>
Return to:&nbsp;<A HREF=“/Menu”>The RomPager Menu</A>&nbsp;or&nbsp;
<A HREF=“/Main”>The RomPager Main Page</A><HR>
<P>
Variables in a form can be entered and displayed in dotted decimal format and hex
format, as well as other numeric formats.  This page is formatted using table
commands, and will look best with Netscape 1.1 or other browsers that support
tables.
<BLOCKQUOTE>
<BR>
<TABLE><CAPTION><B>TCP/IP Address Information</B></CAPTION><TR><TD>IP
Address:</TD>
<TD>
<!-- RpFormInput TYPE=“TEXT” NAME=“IpAddress” SIZE=“15” MAXLENGTH=“15”
   RpGetType=Direct RpGetPtr=theIpAddress
   RpSetType=Direct RpSetPtr=theIpAddress RpTextType=“DotForm” -->
<INPUT TYPE=“TEXT” NAME=“IpAddress” SIZE=“15” MAXLENGTH=“15” VALUE=“0.0.0.0”>
<!-- RpEnd -->
</TD>
<TD>X’
<!-- RpDisplayText RpGetType=Direct RpGetPtr=theIpAddress
   RpTextType=Hex RpSize=8 -->
00000000
<!-- RpEnd -->
‘</TD>
<P>
</TR>
<TR><TD>Subnet Mask:</TD>
<TD>
<!-- RpFormInput TYPE=“TEXT” NAME=“SubnetMask” SIZE=“15” MAXLENGTH=“15”
   RpGetType=Function RpGetPtr=GetSubnetMask
   RpSetType=Function RpSetPtr=SetSubnetMask RpTextType=“DotForm” -->
<INPUT TYPE=“TEXT” NAME=“SubnetMask” SIZE=“15” MAXLENGTH=“15” VALUE=“0.0.0.0”>
<!-- RpEnd -->
</TD>
<TD>X’
<!-- RpDisplayText RpGetType=Function RpGetPtr=GetSubnetMask
   RpTextType=Hex RpSize=8 -->00000000
<!-- RpEnd -->‘</TD>
<P>
</TR>
<TR><TD>Default Gateway:</TD>
<TD>
<!-- RpFormInput TYPE=“TEXT” NAME=“DefaultGateway” SIZE=“15” MAXLENGTH=“15”
   RpGetType=Complex RpGetPtr=GetDefaultGateway
   RpSetType=Complex RpSetPtr=SetDefaultGateway RpTextType=“DotForm” -->
<INPUT TYPE=“TEXT” NAME=“DefaultGateway” SIZE=“15” MAXLENGTH=“15”
VALUE=“0.0.0.0”>
<!-- RpEnd -->
</TD>
<TD>
```

```
<!-- RpDisplayText RpGetType=Complex RpGetPtr=GetDefaultGateway
RpTextType=HexColonForm
RpSize=11 -->
00:00:00:00
<!-- RpEnd --></TD>
<P>
</TR>
</TABLE>
<P>
</BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<!-- RpFormInput TYPE="RESET" VALUE="Reset" -->
<INPUT TYPE="RESET" VALUE="Reset">
<!-- RpEnd -->
&nbsp;&nbsp;&nbsp;&nbsp;
<!-- RpFormInput TYPE="SUBMIT" NAME="Submit" VALUE="Configure" -->
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Configure">
<!-- RpEnd -->
</BLOCKQUOTE>
</BLOCKQUOTE>
</BLOCKQUOTE>
<HR>
<P>
<CENTER><IMG SRC="/Images/Main" WIDTH=160 HEIGHT=80></CENTER>
</FORM>
</BODY>
</HTML>
```

C source file generated by PBuilder

```
#include "RpExtern.h"

extern rpObjectDescription PgConfigureNetwork;

static char PgConfigureNetwork_Item_1[] =
   C_oHTML_oHEAD_oTITLE C_S_RomPager " Configure Network Info"
   C_xTITLE_xHEAD_oBODY C_oP C_oH2 C_oCENTER "Configure Network Address"
   C_xCENTER
   C_xH2;

static char PgConfigureNetwork_Item_3[] =
   C_oP C_oHR "Return to:" C_NBSP C_oANCHOR_HREF "/Menu\">The "
   C_S_RomPager " Menu" C_xANCHOR C_NBSP "or" C_NBSP "\n"
   C_oANCHOR_HREF "/Main\">The " C_S_RomPager " Main Page" C_xANCHOR
   C_oHR C_oP "Variables in a form can be entered and displayed in dotted "
   "decimal\n"
   "format and hex format, as well as other numeric formats. This page is\n"
   "formatted using\n"
   "table commands, and will look best with Netscape 1.1 or other browsers\n"
   "that support tables.\n"
   C_oBLOCKQUOTE C_oBR C_oTABLE “<CAPTION>“ C_oB “TCP/IP Address “
   “Information” C_xB “</CAPTION>“ C_oTR C_oTD “IP Address:” C_xTD
   C_oTD;

extern char *theIpAddress;

static rpTextFormItem PgConfigureNetwork_Item_4 = {
   “IpAddress”,
   &theIpAddress,
   &theIpAddress,
   eRpVarType_Direct,
   eRpVarType_Direct,
   eRpTextType_DotForm,
   15,
   15
};
static char PgConfigureNetwork_Item_5[] =
   C_xTD C_oTD “X\’\n”;

extern char *theIpAddress;

static rpTextDisplayItem PgConfigureNetwork_Item_6 = {
   &theIpAddress,
   eRpVarType_Direct,
   eRpTextType_Hex,
   8
};
static char PgConfigureNetwork_Item_7[] =
   “\’” C_xTD_oP_xTR C_oTR C_oTD “Subnet Mask:” C_xTD
   C_oTD;

extern char *GetSubnetMask(void);
extern void SetSubnetMask(char *theValuePtr);

static rpTextFormItem PgConfigureNetwork_Item_8 = {
   “SubnetMask”,
   GetSubnetMask,
   SetSubnetMask,
   eRpVarType_Function,
   eRpVarType_Function,
   eRpTextType_DotForm,
   15,
   15
};
static char PgConfigureNetwork_Item_9[] =
   C_xTD C_oTD “X\’\n”;

extern char *GetSubnetMask(void);
```

```
static rpTextDisplayItem PgConfigureNetwork_Item_10 = {
   GetSubnetMask,
   eRpVarType_Function,
   eRpTextType_Hex,
   8
};
static char PgConfigureNetwork_Item_11[] =
   “\’” C_xTD_oP_xTR C_oTR C_oTD “Default Gateway:” C_xTD
   C_oTD;

extern char *GetDefaultGateway(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
extern void SetDefaultGateway(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);

static rpTextFormItem PgConfigureNetwork_Item_12 = {
   “DefaultGateway”,
   GetDefaultGateway,
   SetDefaultGateway,
   eRpVarType_Complex,
   eRpVarType_Complex,
   eRpTextType_DotForm,
   15,
   15
};
static char PgConfigureNetwork_Item_13[] =
   C_xTD
   C_oTD;

extern char *GetDefaultGateway(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

static rpTextDisplayItem PgConfigureNetwork_Item_14 = {
   GetDefaultGateway,
   eRpVarType_Complex,
   eRpTextType_HexColonForm,
   11
};
static char PgConfigureNetwork_Item_15[] =
   C_xTD_oP_xTR_xTABLE C_oP C_xBLOCKQUOTE C_oBLOCKQUOTE C_oBLOCKQUOTE
   C_oBLOCKQUOTE;

static rpButtonFormItem PgConfigureNetwork_Item_16 = {
   “Reset”
};
static char PgConfigureNetwork_Item_17[] =
   C_S_NBSP4 “\n”;

static rpButtonFormItem PgConfigureNetwork_Item_18 = {
   “Configure”
};
```

```
static char PgConfigureNetwork_Item_19[] =
   C_xBLOCKQUOTE C_xBLOCKQUOTE C_xBLOCKQUOTE C_S_AllegroLogo C_xFORM
   C_xBODY_xHTML;

static rpItem PgConfigureNetwork_FormItems[] = {
   { eRpItemType_FormAsciiText, &PgConfigureNetwork_Item_4 },
   { eRpItemType_FormAsciiText, &PgConfigureNetwork_Item_8 },
   { eRpItemType_FormAsciiText, &PgConfigureNetwork_Item_12 },
   { eRpItemType_FormReset, &PgConfigureNetwork_Item_16 },
   { eRpItemType_FormSubmit, &PgConfigureNetwork_Item_18 },
   { eRpItemType_LastItemInList }
};

static rpObjectExtension PgConfigureNetwork_FormObjectExtension = {
   (rpProcessDataFuncPtr) 0,
   &PgConfigureNetwork,
   (rpObjectDescriptionPtr) 0,
   0,
   kRpObjFlags_None
};

rpObjectDescription PgConfigureNetwork_Form = {
   “/Forms/NetworkInfo”,
   PgConfigureNetwork_FormItems,
   &PgConfigureNetwork_FormObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeForm,
   eRpObjectTypeDynamic
};

static rpItem PgConfigureNetwork_Items[] = {
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_1 },
   { eRpItemType_FormHeader, &PgConfigureNetwork_Form },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_3 },
   { eRpItemType_FormAsciiText, &PgConfigureNetwork_Item_4 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_5 },
   { eRpItemType_DisplayText, &PgConfigureNetwork_Item_6 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_7 },
   { eRpItemType_FormAsciiText, &PgConfigureNetwork_Item_8 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_9 },
   { eRpItemType_DisplayText, &PgConfigureNetwork_Item_10 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_11 },
   { eRpItemType_FormAsciiText, &PgConfigureNetwork_Item_12 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_13 },
   { eRpItemType_DisplayText, &PgConfigureNetwork_Item_14 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_15 },
   { eRpItemType_FormReset, &PgConfigureNetwork_Item_16 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_17 },
   { eRpItemType_FormSubmit, &PgConfigureNetwork_Item_18 },
   { eRpItemType_DataZeroTerminated, &PgConfigureNetwork_Item_19 },
   { eRpItemType_LastItemInList }
};
```

```
rpObjectDescription PgConfigureNetwork = {
   “/NetworkInfo”,
   PgConfigureNetwork_Items,
   (rpObjectExtensionPtr) 0,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeHtml,
   eRpObjectTypeDynamic
};
```

C stub routines file generated by PBuilder

```
#include “RpExtern.h”

char *theIpAddress = “ “;

extern char *GetSubnetMask(void);

extern char *GetSubnetMask(void) {
       char * theResult;

   return theResult;
}

extern void SetSubnetMask(char *theValuePtr);

extern void SetSubnetMask(char *theValuePtr) {

   return;
}
extern char *GetDefaultGateway(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

   char *GetDefaultGateway(void *theServerDataPtr, char *theNamePtr,
         Signed16Ptr theIndexValuesPtr) {
   char * theResult;
   return theResult;
}
extern void SetDefaultGateway(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);

void SetDefaultGateway(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr)
{
   return;
}
```

If the HTML file includes the following comment tag, PBuilder will use structured access techniques to build the Source and Stub Routines files:

```
<!–– RpPageHeader RpUrl=/NetworkInfo RpStructuredAccess RpMaxItems=20 ––>
```

C source file generated by PBuilder (structured access)

This file is the same as the source file without structured access except for the object header structures.

```
extern void RpStorePgConfigureNetwork_Form_Data(void *theServerDataPtr,
            Signed16Ptr theIndexValuesPtr);

static rpObjectExtension PgConfigureNetwork_FormObjectExtension = {
   RpStorePgConfigureNetwork_Form_Data,
   &PgConfigureNetwork,
   (rpObjectDescriptionPtr) 0,
   0,
   kRpObjFlags_None
};

rpObjectDescription PgConfigureNetwork_Form = {
   “/Forms/NetworkInfo”,
   PgConfigureNetwork_FormItems,
   &PgConfigureNetwork_FormObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeForm,
   eRpObjectTypeDynamic
};

extern void RpFetchConfigureNetworkData(void *theServerDataPtr,
            Signed16Ptr theIndexValuesPtr);

static rpObjectExtension PgConfigureNetwork_ObjectExtension = {
   RpFetchConfigureNetworkData,
   (rpObjectDescriptionPtr) 0,
   (rpObjectDescriptionPtr) 0,
   0,
   kRpObjFlags_None
};

rpObjectDescription PgConfigureNetwork = {
   “/NetworkInfo”,
   PgConfigureNetwork_Items,
   &PgConfigureNetwork_ObjectExtension,
   (Unsigned32) 0,
   kRpPageAccess_Unprotected,
   eRpDataTypeHtml,
   eRpObjectTypeDynamic
};
```

C stub routines file generated by PBuilder (structured access)

```
#include “RpExtern.h”

/*    Built from “ConfigureNetwork.html”    */

typedef struct {
   char *         fIpAddress;
   char *         fDisplay_1;
   char *         fSubnetMask;
   char *         fDisplay_2;
   char *         fDefaultGateway;
   char *         fDisplay_3;
} ConfigureNetworkData, *ConfigureNetworkDataPtr;

/*    Structured Access Form Storage routine (to be fleshed out)    */

void RpStorePgConfigureNetwork_Form_Data(void *theServerDataPtr,
                  Signed16Ptr theIndexValuesPtr);

void RpStorePgConfigureNetwork_Form_Data(void *theServerDataPtr,
                  Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;
   char *         theEmptyCharPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   theEmptyCharPtr = theDataPtr->fIpAddress;
   theEmptyCharPtr = theDataPtr->fSubnetMask;
   theEmptyCharPtr = theDataPtr->fDefaultGateway;
   return;
}

/*     Structured Access Page Access routine (to be fleshed out)       */

void RpFetchConfigureNetworkData(void *theServerDataPtr,
                  Signed16Ptr theIndexValuesPtr);

void RpFetchConfigureNetworkData(void *theServerDataPtr,
                  Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   theDataPtr->fIpAddress = ““;
   theDataPtr->fDisplay_1 = ““;
   theDataPtr->fSubnetMask = ““;
   theDataPtr->fDisplay_2 = ““;
   theDataPtr->fDefaultGateway = ““;
   theDataPtr->fDisplay_3 = ““;
return;
}
```

```
/*    Automatic Structured Access Page and Form routines    */

extern char *RpGet_IpAddress(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

char *RpGet_IpAddress(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   return theDataPtr->fIpAddress;
}

extern void RpSet_IpAddress(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);

void RpSet_IpAddress(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   theDataPtr->fIpAddress = theValuePtr;
   return;
}

extern char *RpGet_Display_1(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

char *RpGet_Display_1(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   return theDataPtr->fDisplay_1;
}

extern char *RpGet_SubnetMask(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

char *RpGet_SubnetMask(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   return theDataPtr->fSubnetMask;
}

extern void RpSet_SubnetMask(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);
```

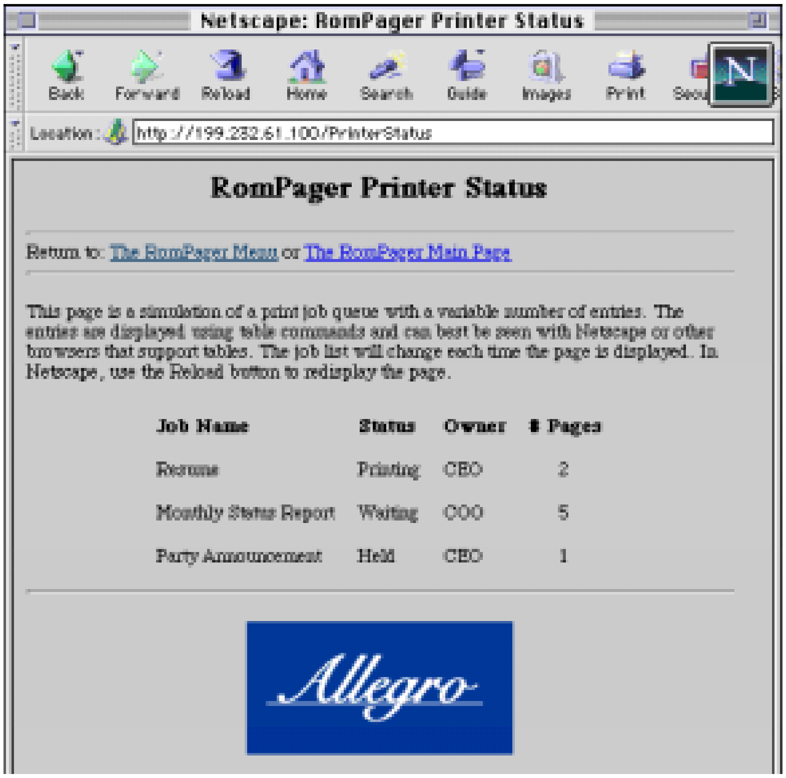
```
void RpSet_SubnetMask(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   theDataPtr->fSubnetMask = theValuePtr;
   return;
}
extern char *RpGet_Display_2(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
char *RpGet_Display_2(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   return theDataPtr->fDisplay_2;
}

extern char *RpGet_DefaultGateway(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
char *RpGet_DefaultGateway(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   return theDataPtr->fDefaultGateway;
}

extern void RpSet_DefaultGateway(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr);
void RpSet_DefaultGateway(void *theServerDataPtr, char *theValuePtr,
      char *theNamePtr, Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   theDataPtr->fDefaultGateway = theValuePtr;
   return;
}

extern char *RpGet_Display_3(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
char *RpGet_Display_3(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   ConfigureNetworkDataPtr theDataPtr;

   theDataPtr = (ConfigureNetworkDataPtr) RpGetRequestCookie(theServerDataPtr);
   return theDataPtr->fDisplay_3;
}
```

HTML without comment tags as served by the Advanced Web Server

```
<HTML>
<HEAD>
<TITLE>RomPager Configure Network Info</TITLE>
</HEAD>
<BODY>
<P>
<H2><CENTER>Configure Network Address</CENTER>
</H2>
<FORM METHOD=“POST” ACTION=“/Forms/NetworkInfo”>
<P>
<HR>
Return to:&nbsp;<A HREF=“/Menu”>The RomPager Menu</A>
&nbsp;or&nbsp;<A HREF=“/Main”>The RomPager Main Page</A><HR>
<P>
Variables in a form can be entered and displayed in dotted decimal format and hex
format, as well as other numeric formats. This page is formatted using table
commands, and will look best with Netscape 1.1 or other browsers that support
tables.
<BLOCKQUOTE>
<BR>
<TABLE><CAPTION><B>TCP/IP Address Information</B>
</CAPTION><TR><TD>IP Address:</TD>
<TD><INPUT TYPE=“TEXT” NAME=“gIpAddress” SIZE=“15”
MAXLENGTH=“15” VALUE=“0.0.0.0”>
</TD>
<TD>X’00000000’</TD>
<P>
</TR>
<TR><TD>Subnet Mask:</TD>
<TD><INPUT TYPE=“TEXT” NAME=“gSubnetMask” SIZE=“15”
MAXLENGTH=“15” VALUE=“0.0.0.0”>
</TD>
<TD>X’00000000’</TD>
<P>
</TR>
<TR><TD>Default Gateway:</TD>
<TD><INPUT TYPE=“TEXT” NAME=“gDefaultGateway” SIZE=“15”
MAXLENGTH=“15” VALUE=“0.0.0.0”>
</TD>
<TD>00:00:00:00</TD>
<P>
</TR>
</TABLE>
<P>
</BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<BLOCKQUOTE>
<INPUT TYPE=“RESET” VALUE=“Reset”>
&nbsp;&nbsp;&nbsp;&nbsp;
<INPUT TYPE=“SUBMIT” NAME=“Submit” VALUE=“Configure”>
```

```
</BLOCKQUOTE>
</BLOCKQUOTE>
</BLOCKQUOTE>
<HR>
<P>
<CENTER><IMG SRC="/Images/Main" WIDTH=160 HEIGHT=80></CENTER>
</FORM>
</BODY>
</HTML>
```

Example 3: Printer status page



Netscape: RomPager Printer Status

http://199.232.61.100/PrinterStatus

RomPager Printer Status

Return to: The RomPager Menu or The RomPager Main Page

This page is a simulation of a print job queue with a variable number of entries. The entries are displayed using table commands and can best be seen with Netscape or other browsers that support tables. The job list will change each time the page is displayed. In Netscape, use the Reload button to redisplay the page.

Job Name	Status	Owner	# Pages
Resume	Printing	CEO	2
Monthly Status Report	Waiting	COO	5
Party Announcement	Held	CEO	1

HTML with comment tags

```
<HTML>
<HEAD>
<TITLE>RomPager Printer Status</TITLE>
</HEAD>
<BODY>
<P>
<H2><CENTER>RomPager Printer Status</CENTER>
</H2>
<P>
<HR>
Return to:&nbsp;<A HREF="/Menu">The RomPager Menu</A>&nbsp;or&nbsp;<A
HREF="/Main">The RomPager Main Page</A><HR>
<P>
```

```
This page is a simulation of a print job queue with a variable number of entries.
The entries are displayed using table commands and can best be seen with Netscape
or other browsers that support tables. The job list will change each time the
page is displayed. In Netscape, use the Reload button to redisplay the
page.<CENTER>
<!-- RpDynamicDisplay RpGetType=function RpGetPtr=gTestPrintJobs RpItemCount=2 --
>
   <!-- RpHiddenDataZeroTerminated RpText=“<P>&nbsp;</P>There are no jobs in the
print queue<P>&nbsp;</P>“ -->
   <!-- RpItemGroup -->
      <!-- RpDataZeroTerminated -->
      <TABLE CELLPADDING=5>
         <TR>
            <TH ALIGN=left>Job Name</TH>
            <TH ALIGN=left>Status</TH>
            <TH ALIGN=left>Owner</TH>
            <TH># Pages</TH>
            <P>
         </TR>
      <!-- RpEnd -->
      <!-- RpRepeatGroupDynamic RpFunctionPtr=gGetPrinterStatusLimits -->
         <!-- RpDZT --><TR><TD><!-- RpEnd -->
         <!-- RpDisplayText RpGetType=complex RpGetPtr=gGetPrintJobName -->
            Resume
         <!-- RpEnd -->
         <!-- RpDataZeroTerminated RpIdentifier=gRpEndCellStartCell -->
            </TD><TD><!-- RpEnd -->
         <!-- RpDisplayText RpGetType=complex RpGetPtr=gGetPrintJobStatus -->
            Printing
         <!-- RpEnd -->
         <!-- RpUseIdentifier RpIdentifier=gRpEndCellStartCell
                  RpItemType=DataZeroTerminated -->
            </TD><TD>
         <!-- RpEnd -->
         <!-- RpDisplayText RpGetType=complex RpGetPtr=gGetPrintJobOwner -->
            CEO
         <!-- RpEnd -->
         <!-- RpDZT --></TD><TD ALIGN=center><!-- RpEnd -->
         <!-- RpDisplayText RpGetType=complex RpGetPtr=gGetPrintJobPages
                  RpTextType=Unsigned8 -->
            2
         <!-- RpEnd -->
         <!-- RpDZT --></TD><P></TR><!-- RpEnd -->
      <!-- RpLastItemInGroup -->
      <!-- RpSample -->
         <TR><TD>Monthly Status Report</TD>
         <TD>Waiting</TD>
         <TD>COO</TD>
         <TD ALIGN=center>5</TD>
         <P>
         </TR>
         <TR><TD>Party Announcement</TD>
         <TD>Held</TD>
         <TD>CEO</TD>
         <TD ALIGN=center>1</TD>
         <P>
         </TR>
      <!-- RpEnd -->
      <!-- RpDZT --></TABLE><!-- RpEnd -->
   <!-- RpLastItemInGroup -->
<!-- RpLastItemInGroup -->
</CENTER>
<HR>
<P>
<CENTER><IMG SRC=“/Images/Main” WIDTH=160 HEIGHT=80></CENTER>
</BODY>
</HTML>
```

C source file generated by PBuilder

```
#include “RpExtern.h”
extern rpObjectDescription PgPrinter_Status;
static char PgPrinter_Status_Item_1[] =
   C_oHTML_oHEAD_oTITLE C_S_RomPager “ Printer Status”
   C_xTITLE_xHEAD_oBODY C_oP C_oH2 C_oCENTER C_S_RomPager “ Printer Status”
   C_xCENTER C_xH2 C_oP C_oHR “Return to:” C_NBSP C_oANCHOR_HREF “/Menu\”>“
   “The “ C_S_RomPager “ Menu” C_xANCHOR C_NBSP “or” C_NBSP
   C_oANCHOR_HREF “/Main\”>The “ C_S_RomPager “ Main Page” C_xANCHOR
   C_oHR C_oP “This page is a simulation of a print job queue with a “
   “variable number of\n”
   “entries. The entries are displayed using table commands and can best be\n”
   “seen with Netscape or other browsers that support tables. The job list\n”
   “will change each time the page is displayed. In Netscape, use the\n”
   “Reload button to redisplay the page.” C_oCENTER “\n”;

static char PgPrinter_Status_Item_3[] =
   C_oP_NBSP_xP “There are no jobs in the print queue”
   C_oP_NBSP_xP;

static char PgPrinter_Status_Item_5[] =
   C_oTABLE_CELLPADDING “5>\n”
   C_oTR “\n”
   C_oTH_ALIGN_LEFT “>Job Name” C_xTH C_oTH_ALIGN_LEFT “>Status” C_xTH
   C_oTH_ALIGN_LEFT “>Owner” C_xTH C_oTH “# Pages” C_xTH C_oP C_xTR;

static char PgPrinter_Status_Item_7[] =
   C_oTR
   C_oTD;

extern char *gGetPrintJobName(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

static rpTextDisplayItem PgPrinter_Status_Item_8[] = {
   gGetPrintJobName,
   eRpVarType_Complex,
   eRpTextType_ASCII,
   20
};

extern char gRpEndCellStartCell[] =
   C_xTD
   C_oTD;

extern char *gGetPrintJobStatus(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
static rpTextDisplayItem PgPrinter_Status_Item_10[] = {
   gGetPrintJobStatus,
   eRpVarType_Complex,
   eRpTextType_ASCII,
   20
};
extern char gRpEndCellStartCell[];
extern char *gGetPrintJobOwner(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

static rpTextDisplayItem PgPrinter_Status_Item_12[] = {
   gGetPrintJobOwner,
   eRpVarType_Complex,
   eRpTextType_ASCII,
   20
};

static char PgPrinter_Status_Item_13[] =
   C_xTD C_oTD_ALIGN_CENTER “>“;

extern Unsigned8 gGetPrintJobPages(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

static rpTextDisplayItem PgPrinter_Status_Item_14[] = {
   gGetPrintJobPages,
   eRpVarType_Complex,
   eRpTextType_Unsigned8,
   20
};

static char PgPrinter_Status_Item_15[] =
   C_xTD_oP_xTR;

rpItem PgPrinter_Status_Item_6_Group[] = {
   { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_7 },
   { eRpItemType_DisplayText, PgPrinter_Status_Item_8 },
   { eRpItemType_DataZeroTerminated, gRpEndCellStartCell },
   { eRpItemType_DisplayText, PgPrinter_Status_Item_10 },
   { eRpItemType_DataZeroTerminated, gRpEndCellStartCell },
   { eRpItemType_DisplayText, PgPrinter_Status_Item_12 },
   { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_13 },
   { eRpItemType_DisplayText, PgPrinter_Status_Item_14 },
   { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_15 },
   { eRpItemType_LastItemInList }
};
```

```
extern void gGetPrinterStatusLimits(void *theServerDataPtr, Signed16Ptr theStart,
      Signed16Ptr theLimit, Signed16Ptr theIncrement);

extern rpRepeatGroupDynItem PgPrinter_Status_Item_6[] = {
   gGetPrinterStatusLimits,
   PgPrinter_Status_Item_6_Group
};

static char PgPrinter_Status_Item_16[] =
   C_xTABLE;

rpItem PgPrinter_Status_Item_4[] = {
   { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_5 },
   { eRpItemType_RepeatGroupDynamic, PgPrinter_Status_Item_6 },
   { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_16 },
   { eRpItemType_LastItemInList }
};

rpItem PgPrinter_Status_Item_2_Group[] = {
   { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_3 },
   { eRpItemType_ItemGroup, PgPrinter_Status_Item_4 },
   { eRpItemType_LastItemInList }
};

extern Unsigned8 gTestPrintJobs(void);

extern rpDynamicDisplayItem PgPrinter_Status_Item_2[] = {
   gTestPrintJobs,
   eRpVarType_Function,
   2,
   PgPrinter_Status_Item_2_Group
};

static char PgPrinter_Status_Item_17[] =
   C_xCENTER C_S_AllegroLogo
   C_xBODY_xHTML;

      static rpItem PgPrinter_Status_Items[] = {
         { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_1 },
         { eRpItemType_DynamicDisplay, PgPrinter_Status_Item_2 },
         { eRpItemType_DataZeroTerminated, PgPrinter_Status_Item_17 },
         { eRpItemType_LastItemInList }
      };

      rpObjectDescription PgPrinter_Status = {
         "/Printer_Status",
         PgPrinter_Status_Items,
         (rpObjectExtensionPtr) 0,
         (Unsigned32) 0,
         kRpPageAccess_Unprotected,
         eRpDataTypeHtml,
         eRpObjectTypeDynamic
};
```

C stub routines file generated by PBuilder

```
#include “RpExtern.h”

extern char *gGetPrintJobName(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

char *gGetPrintJobName(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   char * theResult;
   return theResult;
}

extern char *gGetPrintJobStatus(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);
char *gGetPrintJobStatus(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   char * theResult;
   return theResult;
}
extern char *gGetPrintJobOwner(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

char *gGetPrintJobOwner(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   char * theResult;

   return theResult;
}

extern Unsigned8 gGetPrintJobPages(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr);

Unsigned8 gGetPrintJobPages(void *theServerDataPtr, char *theNamePtr,
      Signed16Ptr theIndexValuesPtr) {
   Unsigned8  theResult;

   return theResult;
}

extern void gGetPrinterStatusLimits(void *theServerDataPtr, Signed16Ptr theStart,
      Signed16Ptr theLimit, Signed16Ptr theIncrement);

void gGetPrinterStatusLimits(void *theServerDataPtr, Signed16Ptr theStart,
      Signed16Ptr theLimit, Signed16Ptr theIncrement) {

   *theStart = 1;
   *theLimit = 10;
   *theIncrement = 1;
   return;
}
extern Unsigned8 gTestPrintJobs(void);

extern Unsigned8 gTestPrintJobs(void) {
   Unsigned8  theResult;
   return theResult;
}
```

HTML without comment tags as served by the Advanced Web Server

```
<HTML>
<HEAD>
<TITLE>RomPager Printer Status</TITLE>
</HEAD>
<BODY>
<P>
<H2><CENTER>RomPager Printer Status</CENTER>
</H2>
<P>
<HR>
Return to:&nbsp;<A HREF=“/Menu”>The RomPager Menu</A>&nbsp;or&nbsp;
<A HREF=“/Main”>The RomPager Main Page</A><HR>
<P>
This page is a simulation of a print job queue with a variable number of entries.
The entries are displayed using table commands and can best be seen with Netscape
or other browsers that support tables. The job list will change each time the
page is displayed. In Netscape, use the Reload button to redisplay the page.
<CENTER>
<TABLE CELLPADDING=5>
<TR>
<TH ALIGN=left>Job Name</TH>
<TH ALIGN=left>Status</TH>
<TH ALIGN=left>Owner</TH>
<TH># Pages</TH>
<P>
</TR>
<TR><TD>Resume</TD>
<TD>Printing</TD>
<TD>CEO</TD>
<TD ALIGN=center>2</TD>
<P>
</TR>
<TR><TD>Monthly Status Report</TD>
<TD>Waiting</TD>
<TD>COO</TD>
<TD ALIGN=center>5</TD>
<P>
</TR>
<TR><TD>Party Announcement</TD>
<TD>Held</TD>
<TD>CEO</TD>
<TD ALIGN=center>1</TD>
<P>
</TR>
</TABLE>
</CENTER>
<HR>
<P>
<CENTER><IMG SRC=“/Images/Main” WIDTH=160 HEIGHT=80></CENTER>
</BODY>
</HTML>
```

Index

NetSilicon

A Digi International® Company