PRODUCT LIFECYCLE MANAGEMENT

ProductCenter®

AT THE CENTER OF YOUR ENGINEERING SUCCESS

ProductCenter WebLink Toolkit Programmer Guide

RELEASE 9.6

January 2016

SofTech

Notice

Visit our Web site at **http://www.softech.com**

Comments? Write to **productcenter@softech.com**

ProductCenter

WebLink Toolkit Programmer Guide

CONTENTS

About this book

ProductCenter 7
- Who should read this book 7
- ProductCenter Documentation 8

Chapter 1: **Introduction**

Welcome to the ProductCenter Toolkits 12
Typical ProductCenter toolkit applications 12
Programming interfaces 12

Chapter 2: **Getting Started**

WebLink architecture 16
- How Web applications work 17

Weblink Processing Example 18
- Simple WML File 19
- Simple WML File web client “source” file 20

WebLink Simple Templates 20
Using the WebLink plugin 21
- Activating the plugin 21
- Supported platforms 22
- The components related to the plugin 23
- Special Weblink Variables 23

Weblink Plugin JavaScript functions 23
- ItemSave(wml_url)
  - ItemSaveToWorkingDir(wmd_url, workingdir) 23
- ItemRemove(filename)
  - ItemRemoveFromWorkingDir(filename, workingdir) 24
- ItemUpload(this.form) 25
- plugin.cgi 26

Creating applications from scratch 26

Storyboarding the application . . . 26
Creating the application pages . . . 28
Adding ProductCenter logic to the application . . . 30
Debugging an application . . . 38

Chapter 3: **Variables, Flow Control, and Attributes**

Variables . . . 40
Fundamental data types . . . 40
Compound data types . . . 40
Indirect lookups . . . 41
Other conventions . . . 41
Variable-related tags . . . 42
Control structures . . . 44
String operations . . . 46
File operations . . . 49
Handling of attributes . . . 51
Single-valued attributes . . . 51
Table-type attributes . . . 51

Chapter 4: **Connections, Status, and Settings**

Comment tag . . . 54
Connection tags . . . 54
General tags . . . 56

Chapter 5: **Users and Groups**

Getting group information . . . 62
Setting group information . . . 64
Getting user information . . . 65
Setting user information . . . 68
Item Column Layout Object . . . 72
Creating an item column layout object . . . 72
Constructing or Editing the layout . . . 72
Accessing the layout . . . 73
Layout storage functions . . . 74

Chapter 6: **Lists**

List tags . . . 78

Chapter 7: **Items**

Creating an item object . . . 84
Setting values . . . 85
Getting values . . . 87
Getting revision values . . . 90

Desktop tags 92
Inbound tags 92
Outbound tags 93
Manipulation tags 95
Release management tags 96
Identification tags 97
Examples 98
  Example 1: Add an item 99
  Example 2: Alter an item 100
  Example 3: Check in an item 101
  Example 4: Check out an item. 101
  Example 5: Undo checkout of an item 101
  Example 6: Get a copy of a file 102

Chapter 8: **Links**

Creating a link 104
Link object tags 104
Link attribute tags 105
Item object link tags 107
  Tags for adding, updating or removing links 107
  Tags for getting linked item information 109
  Tags for getting item link type information 113

Chapter 9: **Queries and Reports**

Creating a query object 116
Setting query object attributes 116
Getting query object attributes 120
Execute tags 121
Query storage tags 122
Workflow query tags 123
Query example 124
  Query Programming Tip 125
Reports 125

Chapter 10: **Workflow**

Getting workflow information 130
Process definition tags 131
Process instance tags 133
Activity instance tags 137
  Tags for acquiring activity instances, getting activity instance attributes or related objects. 137
  Activity instance assignment tags 139
  Tags for moving activity instances in a process. 141

Chapter 11: **BOMs**

BOMitem objects and item objects 146
- Loading BOMitem and item objects 146
- BOMitem object attribute tags 146
- BOMitem object linked item tags 147
- BOMitem object manipulation tags 149

Configuration and Iteration Objects 150
- Defining a config object 150
- Config object release management tags. 152
- Config object information and reporting tags 153
- Config iteration object tags 154

Item Object BOM tags 156

Chapter 12: **Forms**

Loading a form object 158
Form object tags 158
- Method One: Getting field objects from form objects 158
- Method Two: Getting field information from form objects 159

Field object tags 160
Examples 163

Appendix A: **Administration**

weblink.cfg 166
database.cfg 166
mimetype 166

Index 167

About this book

ProductCenter

This manual explains how to develop web-based application programs using the WebLink Markup Language (WML) to customize ProductCenter for your specific needs.

ProductCenter works with an Oracle database manager or Microsoft SQL Server to promote easy and accurate communication throughout a product development organization, ensuring that up-to-date information is available to the people who need it.

Who should read this book

This manual is for programmers with experience in HTML and Javascript, and who are familiar with ProductCenter.

ProductCenter Documentation

- *ProductCenter Installation Guide*— A guide for installing and configuring ProductCenter on Windows and UNIX systems using either Oracle or SQL Server RDBMS.
- *ProductCenter Administrator Guide* — A guide to all procedures involved in setting up and maintaining ProductCenter for use once it has been installed.
- *ProductCenter for Windows User Guide* — A guide for users who work with ProductCenter on a Microsoft Windows platform.
- *ProductCenter Web Client User Guide* — A guide for users who work with ProductCenter through a web browser.
- *ProductCenter Workflow Guide* — A guide to all procedures for setting up, maintaining and using ProductCenter Workflow.
- *ProductCenter Office Integrator User Guide* — A guide for MS-Office users who manage Microsoft Office® documents with ProductCenter.
- *ProductCenter Inventor Integrator User Guide* — A guide for Inventor Integrator users who manage Autodesk® Inventor® designs with ProductCenter.
- *ProductCenter AutoCAD Integrator User Guide* — A guide for AutoCAD Integrator users who manage Autodesk® AutoCAD® Mechanical and Electrical designs with ProductCenter.
- *ProductCenter SolidWorks Integrator User Guide* — A guide for SolidWorks Integrator users who manage SolidWorks® designs with ProductCenter.
- *ProductCenter CADRA Integrator User Guide* — A guide for CADRA Integrator users who manage CADRA® designs with ProductCenter.
- *ProductCenter Pro/ENGINEER Integrator User Guide* — A guide for Pro/ENGINEER Integrator users who manage Pro/ENGINEER™ designs with ProductCenter.
- *ProductCenter C/C++ and Perl Toolkits Programmer Guide* — A guide to the C/C++ and Perl Application Programming Interfaces to customize the ProductCenter environment.

- *ProductCenter WebLink Toolkit Programmer Guide* — A guide to the Web-based Application Programming Interface to customize the ProductCenter environment.
- *ProductCenter BatchLoader Guide* — A guide to the use of BatchLoader to automate the process of loading legacy data into ProductCenter.
- *ProductCenter BatchGetCopy Guide* — A guide to the use of BatchGetCopy to automate retrieving copies of multiple files from ProductCenter.
- *ProductCenter GenView*™ *Guide* — A guide to the use of GenView to automate the generation of viewable files.
- *ProductCenter Release Notes* — A description of all product changes for a specific ProductCenter release.

1

Chapter 1

Introduction

Just Ahead:

Welcome to the ProductCenter Toolkits 12
Typical ProductCenter toolkit applications 12
Programming interfaces. 12

Welcome to the ProductCenter Toolkits

The ProductCenter Toolkits (formerly known as *APIs*, or Application Programming Interfaces) are a set of software libraries for various languages that allow you to develop applications that exchange data directly with ProductCenter. The information you can store in ProductCenter includes *file* data, such as engineering drawings, and *record-oriented* data, such as part numbers and vendor information.

In most cases, you can do anything in a ProductCenter Toolkit that you can do in the ProductCenter user interface (UI), *except* administrative functions such as adding users.

With ProductCenter, you can add and delete data objects, manage revisions, grant access to various users and groups, execute queries on the database, manage processes, and create relationships among data elements through the use of links.

The ProductCenter Toolkits give you the freedom not only to obtain programmatic access to ProductCenter's most important features, but also to create your own client applications to enhance the capabilities of the system.

The toolkits are available for the following languages:

- C/C++
- Perl
- WML (WebLink Markup Language)

This document shows you the WebLink functions that you can use to tailor the product to your own purposes.

The C/C++ and Perl toolkits are documented separately in the *ProductCenter C/C++ and Perl Toolkit Programmer Guide*.

Typical ProductCenter toolkit applications

ProductCenter can be used in almost any industry and for almost any application that involves managing large amounts of data. Companies in many diverse industries have used previous versions of our Toolkit packages for a number of applications. Here are just a few of them.

- Import data from other sources.
- Export data to other sources.
- Monitor transactions for integrity.
- Automate creation or modification of items based on events.
- Develop simple "task-based" applications.

Programming interfaces

The ProductCenter Toolkits are available with the following interfaces:

C++ — This interface gives you a full object-oriented programming model and gives you access to the most commonly used functionality in ProductCenter.

C — The C interface gives you access to the same ProductCenter functionality as the C++ interface. All of the functions in the C++ objects have C language counterparts. Any program that can be written in C++ can be written easily using the C interface. Compile your C programs with a C+ compiler.

Perl — The Perl interface gives you access to the same ProductCenter functionality as the C and C++ interfaces. At present it can be used on Windows and Sun Solaris.

WebLink — The WebLink interface provides similar, but not identical, functionality as the C/C++ and Perl Toolkits. WebLink enables you to efficiently build web-based client interfaces using the proprietary WebLink Markup Language (WML).

Chapter 2

Getting Started

Just Ahead:

WebLink architecture 16
Weblink Processing Example 18
WebLink Simple Templates. 20
Using the WebLink plugin. 21
Weblink Plugin JavaScript functions 23
Creating applications from scratch. 26

The WebLink Markup Language (WML) allows a ProductCenter customer to create their own Web applications that interact with ProductCenter. This chapter describes the WebLink directory structure, explains how a Web application works, and discusses the process of designing an applications and adding ProductCenter logic to it with WebLink's WML tags. "WebLink Simple Templates" on page 20 describes the sample demo that is included with the Weblink Toolkit. Later chapters show how to use the variables, tags, and control structures that constitute the WML language.

NOTE: ProductCenter supports *only* the tags described in this document. Although other WML tags may be visible in the cgi-bin directory, customer use of those tags is not supported. Unsupported tags may be modified at any time by SofTech and can give unexpected results if used in an applications. Therefore, Customer Support will not respond issues regarding unsupported WML tags.

WebLink architecture

Before constructing a custom WebLink application, it may be useful to step back and understand how a Web application works. Understanding how a Web application works requires an understanding of the WebLink directory structure. This section reviews the directory structure, discusses which files are located in each subdirectory, and explains how these files are interrelated. These two subdirectories were identified when Weblink was installed from the ProductCenter installation media. It is important to know the locations of these directores and the server on which they are located.

- htdocs — This directory contains HTML files, which are generally static web pages.
- cgi-bin — The cgi-bin directory is scripts that generate dynamic content are generally located. This is directory where WML files will be placed. WML files are simply HTML files with additional WML tags that can interact with ProductCenter. This directory also contains:
  - three configuration files: database.cfg, weblink.cfg, and mimetype. These files are discussed in more detail in Appendix A, "Administration".
  - three binary files: weblink.cgi, webclient, and plugin.cgi. weblink.cgi is the *gateway* program. A gateway, or CGI, is a program that is triggered by a Web browser and that creates a link between a Web browser and some other client. In this case, weblink.cgi talks to the second binary file, webclient. This connection allows WebLink to extract data from ProductCenter and display that data in a browser.

The third binary file, plugin.cgi, allows users to upload and download files and directories.

- two message code files: msg_db.idx, which contains an index of all message codes, and msg_db.db, which contains a text message for every message code.

The web server launches weblink.cgi, which is launched every time a user tries to gain access to a WML page. Next, weblink.cgi talks to the web client to establish a link between ProductCenter and WebLink. Once this link is established, the webclient talks with the ProductCenter broker and server to obtain the information that the user requested from ProductCenter.

How Web applications work

This section describes the process by which WebLink transfers data from one Web page to another.

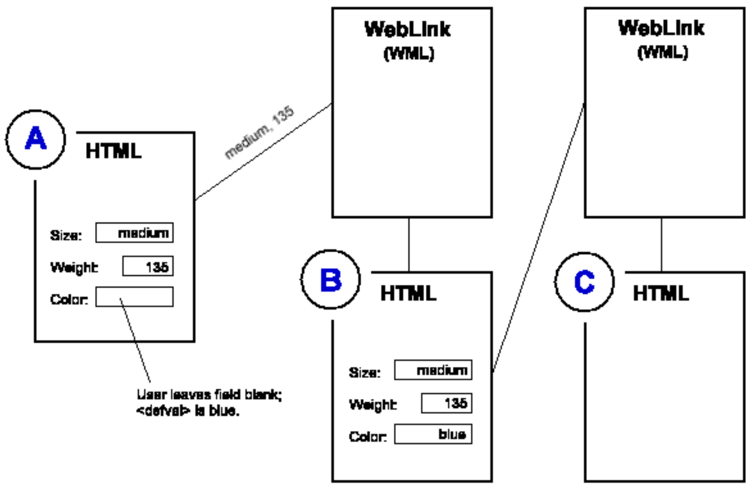


Figure 2-1: WebLink relationship between HTML and WML

The page labeled A is an HTML page that a user sees in a web browser application, such as Firefox. This page contains form information, which is a collection of variables. The user enters information into the fields and then presses an on-screen button to advance to the next page. The binary file weblink.cgi processes the variables on that form and, in turn, displays the new HTML page, labeled B. WebLink takes the information that the user entered into the form on page A, processes the form information, and generates another HTML page (B). If a user does not enter information for certain variables on page A, WebLink takes the <defval> settings for those variables *if those settings were placed explicitly in the Web page* and uses them in page B. If <defval> settings were not made, WebLink would generate a run time error.

For example, if page “A” asks the user to specify a color and the user does not do so, WebLink finds the <defval> setting for the variable color and uses that setting in page “B”.

Now suppose that the user did enter the color blue into page "A" and that value must be passed along to page "C". The method for doing this is to use an HTML form element which is hidden. When page "B" is submitted to Weblink for processing, the value of that hidden form element can be accessed by Weblink and then passed along to page "C" if necessary.

For the above scenario, the HTML code for page "B" might look like this:

```
<FORM>
....
<INPUT TYPE="Hidden" NAME="Color" VALUE="<printval color>">
....
</FORM>
```

In general, hidden variables are an excellent way to pass information back and forth among Web pages.

Weblink Processing Example

To understand how weblink processing works compare the contents of the following .wml file and how it appears when viewing the "source" in the web browser. Source file in this case is a misnomer, the source in the web client is the output from weblink.cgi process. The wml commands are no longer in the file, what is left is the html, javascript code, and the output from WML commands.

Simple WML File

```
<!-- Simple WML File -->
  <SetVal db   "pctr" >
  <SetVal user "cms" >
  <SetVal pass "cms" >
  <SetVal cnt "10">
  <SetVal login <Login USERNAME = user PASSWORD = pass DATABASE =
   db >>
  <html>
    <body>
      Logging In<br>
      <script>
        <If login ne "0" >
          alert( "<printval <Expr "Failed Login to " +
            <Expr db + <Expr " as " + <Expr user +
   <ErrorGetLastMessage>>>>>>");
        <Else>
          alert("Login Successful");
        </If>
      </script>
      <While cnt gt "0" ><PrintVal cnt>&nbsp;<SetVal cnt <Expr
   cnt - "1">></While>
      <If login eq "0" >
        <logout>
      </If>
    </body>
  </html>
```

Simple WML File web client “source” file

```
<HTML>
  <body>
    Logging In<br>
    <script>

        alert("Login Successful");

    </script>

 10&nbsp;9&nbsp;8&nbsp;7&nbsp;6&nbsp;5&nbsp;4&nbsp;3&nbsp;2&nb
 sp;1&nbsp;




  </body>
</html>
```

WebLink Simple Templates

ProductCenter provides templates for those users who want to begin learning WebLink by working with simple examples. These templates allow users to gain access to ProductCenter data by filling out simple, easy-to-use forms.

The simple templates can be accessed by entering the URL of the login page “http://<server>/weblink/simple/login.html” into a Web browser. This login page asks for a user ID, password, and the name of the database to which to connect. A simple templates menu is displayed, from which any of the template names can be selected.

The twelve templates are:

- Add a file
- Find items to view, check in, check out, undo checkout, or alter
- Find items to route, add to desktop
- View, remove items from desktop
- Find items to purge, delete, or rollback
- Find items to view/edit links
- Find items by custom attributes and save them as queries
- View, delete saved queries
- View, approve, disapprove, toggle worklist activities
- Retrieve claimable activities
- Class browser (Tree) * Simple Class browser

Using the WebLink plugin

The WebLink Toolkit includes a special plugin (an actual plugin for Firefox; an ActiveX control for Microsoft Internet Explorer) that provide support for the ProductCenter working directory and allows users to upload/download ProductCenter files and directories. After upload activities (Add, Checkin, and Return Unmodified), WebLink automatically removes files and directories from a user's working directory. After download activities (Checkout and GetCopy), WebLink deposits files and directories into the user's local working directory. For security reasons, the plugin allows applications to upload only from the user's local working directory.

A plugin is a separate code module (a dynamic link library or a shared library) that runs within the browser process space and extends the capability of the browser. ActiveX controls perform a similar function for Microsoft Internet Explorer. For more information about plugins and ActiveX controls, see::

http://developer.mozilla.org/en/docs/Plugins

http://www.microsoft.com/com/tech/activex.asp

Activating the plugin

Before the Weblink plugin can be used, a WebLink administrator must change the configuration file and appropriate WML pages, and each user must install either the Firefox plugin (a DLL or a shared library) or the Microsoft Internet Explorer (MSIE) ActiveX control, on each client machine for a particular browser.

Administrator tasks

To use the plugin, a WebLink administrator must:

1. Activate the plugin by modifying the weblink.cfg file (see “weblink.cfg” on page 166) and inserting the line below. This configuration parameter activates the plugin functionality on the server side; that is, this command tells Weblink that the browser client is using the plugin to upload or download files and directories.

   ```
   plugin_exec /cgi-bin
   ```

2. Modify WML pages where inbound/outbound files and directories occur by adding appropriate JavaScript functions from the “plugin.js” file. The JavaScript functions invoke the plugin with appropriate input parameters. See “Weblink Plugin JavaScript functions” on page 23 for more information on the “plugin.js” file and its JavaScript functions.

NOTE: Activating the plugin in weblink.cfg without modifying the relevant wml pages will break Weblink and produce incorrect results. If the WML pages are developed properly, the plugin can be deactivated for a site without modifying the WML pages.

User tasks

A user must load the control/plugin into their browser application (either Firefox or Internet Explorer). A user will be prompted to do so when a control/plugin is activated within a WML application but the control/plugin is not loaded into the user’s browser. If the control/plugin is already loaded, then a prompt to load the control/plugin will generally not appear to a user. Alternatively, users could be instructed to load the control/plugin before a plugin application is used. The file /htdocs/weblink/simple/download.html gives an example of such a download page.

To load the plugin into a browser application:

- **Firefox:** Place the DLL or shared library into the plugins directory for the browser:
  - For Firefox on Windows:
    C:\Program Files\Mozilla Firefox\plugins
  - For Firefox on UNIX:
    [*firefox_install_dir*]/plugins
- **Microsoft Internet Explorer:** Accept the ActiveX control by clicking YES in response to "Do you want to install and run ‘WTC file upload/download ActiveX control’

Supported platforms

Please refer to the latest appropriate ProductCenter Release Notes for supported web browsers, servers and operating systems.

The components related to the plugin

The following components provide the functionalities mentioned at the beginning of this section.

Special Weblink Variables

wml_plugin

If plugin is activated (via the entry "plugin_exec /cgi-bin" in the "weblink.cfg" file), wml_plugin becomes "1." Otherwise, this variable defaults to "0." In a WML page, an application can check this variable's value and take different actions based on whether the plugin is installed. If the plugin is installed, the application must include the ProductCenter-supplied JavaScript functions.

Also, an application can temporarily activate or deactivate the plugin through WML by setting the value of wml_plugin to "0" or "1."

The proper use of this variable is illustrated through the examples that appear later in this section.

wml_url

WebLink sets this special variable if the plugin is activated and the user is doing either an <ItemGetCopy> or an <ItemCheckout>. Applications should never reset this variable. This variable is an input parameter to the JavaScript function ItemSave( ).

The proper use of this variable is illustrated through the examples that appear later in this section.

Weblink Plugin JavaScript functions

The Weblink toolkit supplies JavaScript functions in the file /htdocs/weblink/javascript/plugin.js, that must be called with proper input parameters, at the points in WML pages where the WML code for file Upload/Download appears. These functions cannot be modified. The functions are:

- ItemSave(wml_url): Used with <ItemGetCopy> and <ItemCheckout>.
- ItemRemove(filename): Used with <ItemUndoCheckout>, <ItemAdd>, and <ItemCheckin>.
- ItemUpload(this.form): Used with <ItemAdd> and <ItemCheckin>.

ItemSave(wml_url)
ItemSaveToWorkingDir(wmd_url, workingdir)

<ItemGetCopy> and <ItemCheckout> obtain copies of a file from ProductCenter and deposit the file into the WebLink temporary working directory on the server side. If the plugin is not installed, the contents of the file are shipped immediately to the client Web

browser. If the plugin is installed, then the WML application must call the JavaScript function "ItemSave()" (with the input parameter "wml_url"), which invokes the plugin on the client side.

wml_url is a special variable that Weblink creates after a succesful GetCopy/CheckOut but is only created if the plugin is activated. This variable holds the absolute pathname of the outbound file in Weblink's temporary working directory. Applications should never modify this variable.

Once invoked in the browser process space, with the appropriate input, the plugin extracts the file from a working directory and saves the file in the user's local directory.

```
<if <ItemIsType item TYPE="FILE"> EQ "1">
    <setval error <ItemGetCopy item>>
</if>

<HTML>
<BODY>
<if wml_plugin EQ "1">
    <if error EQ "0">
        <if <ItemIsType item TYPE="FILE"> EQ "1">
          <SCRIPT SRC="/weblink/javascript/plugin.js"></SCRIPT>
          <SCRIPT>ItemSave('<printval wml_url>')</SCRIPT>
        </if>
    </if>
</if>
</BODY>
</HTML>
```

For a detail code (for <ItemCheckout> and <ItemGetCopy>) refer to modify.wml and view.wml pages for simple templates (located in the /cgi-bin/weblink/simple directory).

ItemRemove(filename)
ItemRemoveFromWorkingDir(filename, workingdir)

If the plugin is activated, then after a successful <ItemCheckin> or <ItemAdd>, ItemRemove(filename) invokes the plugin on the client side. The plugin then deletes the filename from the user's local working directory. The input parameter filename is the name of the file to be removed from user's working directory. Applications can also use

ItemRemoveFromWorkingDir() to remove items from folders other than the working directory.

```
<setval error <ItemUndoCheckout item>>
<if error EQ "0">
    <if wml_plugin eq "1">
      <!----- Include the Plugin JavaScript Functions -------->
      <SCRIPT SRC="/weblink/javascript/plugin.js"></SCRIPT>
      <!----Use the plugin to remove the file from the Working
   Directory ---->
      <SCRIPT>ItemRemove('<printval <ItemGetAttr item ATTR =
   "Name">>')</SCRIPT>
</if>
```

For a detailed code example, please refer to checkin.wml, file.wml, and undo.wml to view these simple templates (located in the /cgi-bin/weblink/simple directory).

ItemUpload(this.form)

If the plugin is not activated, the browser uploads a file directly to the action URL of the underlying form when the form is submitted. If the plugin is activated, then ItemUpload() function needs to be called when the form is to be submitted. ItemUpload() takes the form element, gathers its input, and uploads the file or directory (with the help of the plugin) to the URL specified in the form action attribute.

```
<HTML>

<HEAD>
<if wml_plugin eq "1">
    <!----- Include the Plugin JavaScript Functions -------->
    <SCRIPT SRC="/weblink/javascript/plugin.js"></SCRIPT>
</if>
</HEAD>

<BODY>

<if wml_plugin EQ "1">
1 <FORM NAME=addform ACTION="/cgi-
   bin/weblink.cgi/weblink/simple/file.wml">
<else>
    <FORM NAME="addform" METHOD="post" ENCTYPE="multipart/form-
   data"
           ACTION="/cgi-bin/weblink.cgi/weblink/simple/file.wml">
</if>
```

```
<if wml_plugin eq "1">
      <INPUT TYPE="button" VALUE="Add to ProductCenter"
   onClick="ItemUpload(this.form)"> &nbsp;
<else>
      <INPUT TYPE="button" VALUE="Add to ProductCenter"
   onClick="this.form.submit()"> &nbsp;
</if>
</FORM>

</BODY>
</HTML>
```

For a detailed code example, refer to the simple templates addfile.wml and modify.wml (located in the /cgi-bin/weblink/simple directory).

plugin.cgi

plugin.cgi delivers a file to the browser whenever a user tries to download a file and the plugin has been activated. When Weblink is installed, plugin.cgi appears in the cgi-bin directory. The line “plugin_exec /cgi-bin” tells Weblink where plugin.cgi and other Weblink executables are located.

Creating applications from scratch

WebLink provides all the tools needed to create a custom applications. This section provides suggested steps to follow when designing and implementing a Weblink application.

The process consists of the steps described in the following sections:

- “Storyboarding the application”
- “Creating the application pages”
- “Adding ProductCenter logic to the application”
- “Debugging an application”

These sections describe the steps in detail. To further illustrate the application building process, an example is illustrated throughout this section as one way to use the concepts presented here.

Storyboarding the application

A Web-based application is a series of Web pages that logically flow from one to another. When a user enters information into one page and clicks a button to proceed to the next page, the application passes the information entered from the first page to the second, and so on throughout the application.

Consequently, it is important that a developer plan the sequence of pages that each user will see before designing each individual page. The best way to do this is to storyboard the application.

A storyboard is a diagram that illustrates the flow of screens in a Web application. Just as motion picture directors storyboard their films to chart the flow of visual images from beginning to end, a good Web designer defines the screens that will constitute the application and depicts them in a diagram that clearly shows the path that users will follow.

The storyboards need not be elaborate. A developer can simply draw a sequence of boxes on a sheet of paper, with each box representing a screen that will appear to the users. Draw arrows from one box to the other to illustrate the sequence that the application follows.

Description of example

The sample application that we are going to create is a find-and-view application. The application searches for items in the ProductCenter database and then display those items on the screens we create.

These are the criteria for the items the application tries to find:

- All items are viewables from the class CMS:Files:Viewables.
- The file type for all viewables is Adobe Acrobat.
- The application searches for these viewables by the Name and Material attributes.

The application will consist of these pages. (For simplicity, we assume that the user already has logged on to the system, so there is no need to plan for a login screen.)

- A Search screen, which asks the user to enter a viewable name into a text field and to choose one or more materials from a pull-down list of all possible options.
- An Error screen, which the system displays if the user's query fails due to error.
- A No Results screen, which informs the user that the system was unable to locate the viewable that matched the specified criteria.
- A Search Results screen, which displays a table of all Adobe Acrobat viewables that matched the user's criteria. WebLink creates the Error, No Results, and Search Results screens as one file. We will add ProductCenter logic to the file, and this logic tells the interpreter which part of this file to display in the Web browser.
- A Details screen, which displays more detailed information about any of the located viewables. A user obtains this screen by clicking on one of the entries in the table from the Search Results screen.

Sample storyboard

The diagram below shows one way to storyboard this application.

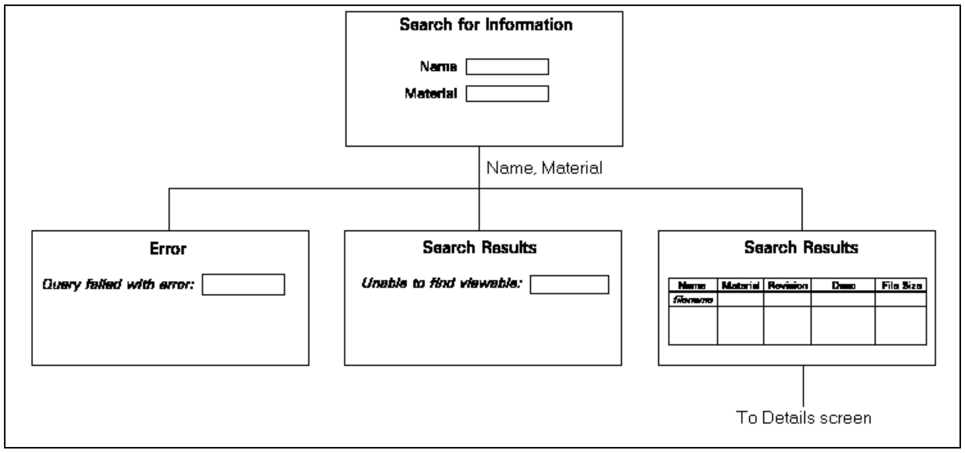


Figure 2-2: Storyboard example

Notice that once the user has entered the viewable name, has selected one or more materials from the pull-down menu, and has clicked on the Search button, one of three things happens. The user gets an error message, learns that no items matched the query criteria, or sees a table that displays the matching items. The three arrows emanating from the Search box illustrate these possibilities.

The use of the item ID

All Web applications need some mechanism by which to transfer data from one page to another. (Web architecture does not allow for data persistence between pages.) WebLink uses each item's ID to identify uniquely the objects that flow among the pages of the system. The relevance of this ID bcomes clear when we discuss the "Adding ProductCenter logic to the application" on page 30 and show how the <ItemLoad> WML tag uses this unique ID to load items into memory.

Creating the application pages

Once a developer has developed a storyboard and defined the pages of an application, the next step is to design each page.

A web authoring tool may assist a developer with creating the basic layout of each page. At this point in development, the developer should focus only on laying out the fields and other background details. Once each display has been finalized, then the developer can proceed to the next step where WML tags are added to the HTML code. (If a web authoring tool is being used for page layout, the authoring tool will generate the HTML code automatically as each page is created.)

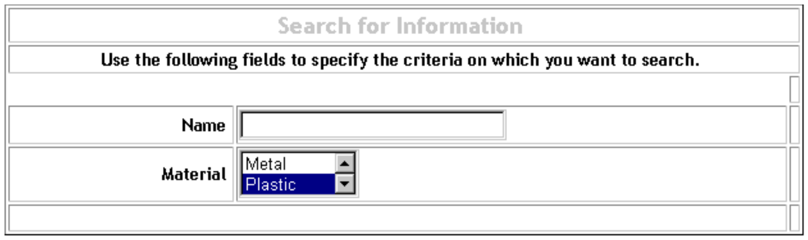
Example: Creating the Search for Information, Search Results, and Details pages

In our storyboard example, we defined three pages that we need to create:

- Search for Information page
- Search Results page, the layout of which depends on whether the query yielded an error, the query yielded no results, or the query was able to locate items
- Details page

The Search page

The Search page, shown below, asks the user to enter the name and material for which to search.



Search for Information

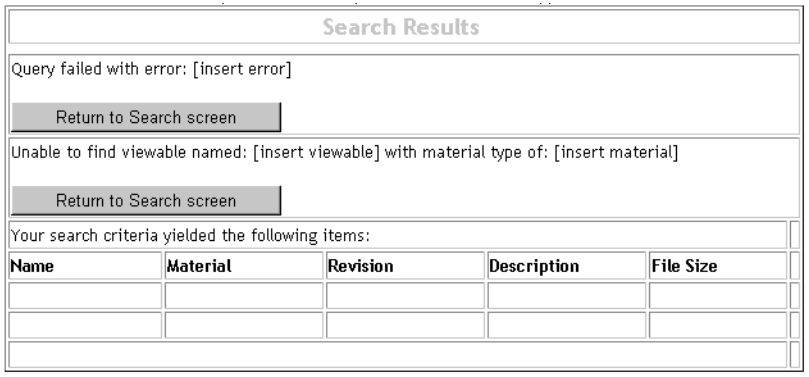
Use the following fields to specify the criteria on which you want to search.

Name	
Material	Metal / Plastic

Figure 2-3: Search page

The Results page

In our storyboard, we drew the Error and Search Results pages as three separate screens. When we created the pages with our WYSIWYG editor, however, we decided to combine all three into one page. We combined them because all three pages are possible results of the query entered into the Search screen. The ProductCenter logic that we enter into the HTML code in the next step determines the part of the screen that appears to the user.



Search Results

Query failed with error: [insert error]

Return to Search screen

Unable to find viewable named: [insert viewable] with material type of: [insert material]

Return to Search screen

Your search criteria yielded the following items:

Name	Material	Revision	Description	File Size

Figure 2-4: Search results

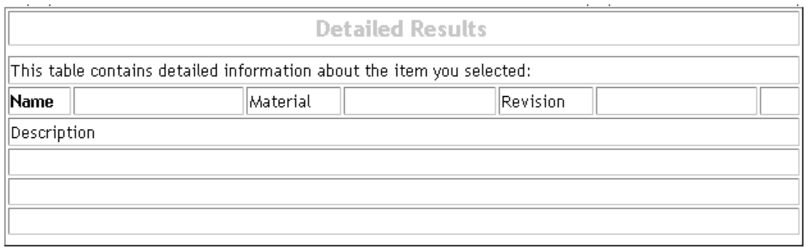
At the top of the page, text appears when the query results in an error. The next part of the page shows what appears on the screen if the query yields no results. The final part of the page shows the results of a successful query. Each row of the table contains information about one of the items found. For each item, the table displays its name, the material, the revision number, a description of the item, and its size.

(The text "insert error," "insert viewable," and "insert material" represents empty fields that will be populated by the results of the queries.)

The Detailed Results page

The final screen is the Detailed Results page, shown below. This page generates a table that is based on the attributes of the item the user chooses from the Results page.

In this example, users can click on entries in the Name column of the Search Results table. When they do, they see the Detailed Results page that applies to the selected item. For the chosen item, the Detailed Results page once again displays the Name, Material, and Revision number of the item, but it also displays a more detailed description.



Detailed Results

This table contains detailed information about the item you selected:

Name		Material		Revision		
Description						

Figure 2-5: Detailed results

Adding ProductCenter logic to the application

Perhaps the most important part of the application development process is the addition of ProductCenter logic to HTML code. If a web page editor is used to create web pages, the editor creates HTML code as each page is designed. To add the ProductCenter logic, WML tags are inserted, enclosed between special comment tags that were created for WebLink. Among other things, these tags extract data from ProductCenter. WebLink can can display data retrieved from ProductCenter and placed into the HTML.

Later chapters explain the WML tags in greater detail.

The remainder of this section shows the HTML code that was generated automatically when we created each of the pages in our example. We then show how to enter WML tags to add the ProductCenter logic to each screen.

The Search screen

The Search screen requires no ProductCenter logic. This screen simply accepts user input — the name and material by which the user wants to search — and then sends this information to the Results page. Two form variables, Name and Material, are sent to the Results screen from this Search screen.

The HTML code that our editor generated for the Search screen appears below.

```
<html>
<head>
<title>Search</title>
<meta name="GENERATOR" content="Microsoft FrontPage 3.0">
<meta name="Microsoft Theme" content="none, default">
<meta name="Microsoft Border" content="none, default">
</head>
<body>
<p>&nbsp;</p>
<p>&nbsp;</p>
<form method="POST"
action="http://info.softech.com/cgi-
   bin/weblink.cgi/result.html">
  <p>Locate all viewables whose Name <input type="text"
   name="name"
size="20"></p>
<p>&nbsp;&nbsp;&nbsp;&nbsp;
  and Material <select name="material" size="1">
    <option value="Steel">Steel</option>
    <option value="Aluminum">Aluminum</option>
    <option value="Plastic">Plastic</option>
  </select></p>
  <p>&nbsp;</p>
  <p>&nbsp;</p>
  <p><input type="submit" value="Execute Search"
   name="submit"><input
type="reset"
  value="Reset" name="B2"></p>
</form>
</body>
</html>
```

The Results screen

The Results screen takes the input from the Search page and tries to locate matching items in the ProductCenter database. WebLink displays one of three screens created in the Results page depending on the results of the query. Each of the two results sections of this page must contain ProductCenter logic that queries the database and retrieves items based on the attributes the user entered as search criteria.

The first thing the Results screen code must do is create a query based on the information the user entered into the Search screen. So the code must include ProductCenter logic that takes this information and creates a query.

The WML code that performs this activity appears below:

```
<!--wml
<QueryCreate newquery>
<QueryAddClause newquery VIEW = "Common Attributes" ATTR = "Name"
   CONDITION = "contains" VALUE = name>
<QueryAddClause newquery VIEW = "Common Attributes" ATTR = "File
   Type" CONDITION = "contains" VALUE = "acrobat_file">
<QueryAddClause newquery VIEW = "Common Attributes" ATTR =
   "Class" CONDITION = "one of" VALUE = "CMS:Files:Viewables">
<QueryAddClause newquery VIEW = "manufacturing" ATTR = "Material"
   CONDITION = "is exactly" VALUE = material>
<setval error <QueryExecute newquery THRESHOLD = 0>>
<setval count <QueryGetItemCount newquery>>
wml-->
```

Let's take a closer look at each line of code within the WML comment tags.

The <QueryCreate> tag indicates that the system is creating a new query object entitled newquery.

The four <QueryAddClause> tags allow a developer to define the query to perform.

- The first line indicates that the developer wants to find all items for which the common attribute Name (ATTR="Name") contains (CONDITION="contains") the name that the user entered into the Search screen. Notice that, in the argument VALUE=name, the word name does not appear in quotes. This indicates that VALUE takes on the value that the user entered into the Name form variable in the previous screen.
- The second line indicates that the matching items must have filetype attributes (ATTR="File Type") whose name contains (CONDITION="contains") the value acrobat_file (VALUE="acrobat_file").
- The third line specifies the class (ATTR="Class") to which the matching items must belong (CONDITION="one of"). The VALUE indicates that the items must belong to the Viewables subclass of the Files class in ProductCenter.
- The fourth and last <QueryAddClause> line indicates that, for all matching items, the name of the custom attribute Material (ATTR="Material") should be exactly equal to (CONDITION="is exactly") the name that the user chooses from the Material field of the Search screen (VALUE=material). Once again, notice that material does not appear in quotes, which means that VALUE assumes the value of the entry chosen from the Material form variable in the previous screen.

The final two command lines, both using <setval> tags, set the values of two variables, error and count.

- The error variable is set to the value of the <QueryExecute> command. This command runs the query we've just created (newquery) and sets the value of THRESHOLD equal to zero. Users can set the THRESHOLD variable equal to the maximum number of matches the user wants a query to return. By default, set THRESHOLD equal to zero to return all matching items. The error variable, therefore, equals zero if a query runs without error. If the query does not run correctly, the value of error is a non-zero error code.

- The count variable is set to the value generated by <QueryGetItemCount newquery>, which returns the number of items returned by the query. We use this variable again when we add ProductCenter logic to the Results screen.

The remainder of the code provides the logic that determines which of the three possible Results screens appears based on the results of the query. There are three possible scenarios:

- The query fails because of an error.
- The query yields no results.
- The query yields results.

We address each of these three possibilities by inserting <if ...elseif ...else> tags within WML comment tags.

Query fails due to error

The section of code that determines whether or not to display the error version of the Results screen appears below.

```
<!--wml <if error ne 0> wml-->
<p><big>Query failed with error: <!--wml <printval
   <ErrorGetMessage newquery>> wml-->
</big></p>
<p>Please press the <font color="#FF0000">Back</font> key on your
   Web Browser.</p>
```

The code within the <if> tag in the first line indicates the condition under which the query fails due to an error. This failure occurs when the value of the variable error does not equal zero (that is, the variable is equal to a non-zero error code). This <if> statement appears within the WML comment tag, and this comment tag precedes the HTML code that generates the Error screen.

The **Query failed with error** field of the Error screen must display the error generated. We insert another WML tag here, inside of which we include a <printval> statement. This statement tells the interpreter to display in the Error screen the error that occurred.

Query yields no results

The next section of the code indicates the display that appears in the Results screen if the query yielded no results.

```
<!--wml <elseif count lt 1> wml-->
<p><big>Unable to find viewables named: <!--wml <printval name>
wml-->&nbsp;&nbsp;&nbsp; with material type of : &nbsp; <!--wml
<printval material> wml--></big></p>
<p>Please press the <font color="#FF0000">Back</font> key on your
   Web
Browser.</p>
```

A query yields no results when the value of the variable count, into which we saved the number of items that the query found, is less than 1 (lt 1), or 0.

We use an <elseif> statement inside of a WML comment tag to specify this criterion.

This screen also displays the name and material of the items for which the query was searching. <printval> statements appearing within WML comment tags tell the interpreter to display the name and material in the proper places.

Query yields results

The remainder of the code is used when the query has found matching items and can display them in the table of the Results screen. We must add to this section of the code the ProductCenter logic that tells WebLink to get the located items and display them in this screen.

Let's look at the code that generates and fills in the Results screen.

```
<!--wml <else> wml-->
<p>Items Found</p>
<table border="1" width="100%">
  <tr>
    <td width="20%"><strong>Name</strong></td>
    <td width="20%"><strong>Material</strong></td>
    <td width="20%"><strong>Revision</strong></td>
    <td width="20%"><strong>Description</strong></td>
    <td width="20%"><strong>File Size</strong></td>
  </tr>
```

The first section of code generates the column headings of the Results page. The only ProductCenter logic that appears here is in the first line, where we have the <else> tag within the WML comment tags. When the query has yielded results, the program executes the code that follows the <else> tag.

The rows that appear in the **Results** table contain the name, material, revision number, description, and file size attributes of all of the items that the query found. The WML code, therefore, must include tags that fetch each item from the set of items found by the query, get the attributes from each item, and print the values of these items in the appropriate columns. The code must also provide a link on the name of the item. Users can click on this link to go to the Details screen and get detailed information about the item they choose.

The code that accomplishes these tasks appears below.

```
<!--wml
<setval index "0">
<while index lt count>
        <QueryGetItem newquery myitem INDEX = index>
        <setval name <ItemGetAttr myitem ATTR = "Name"> >
       <setval material <ItemGetAttr myitem ATTR = "Material" > >
       <setval revision <ItemGetAttr myitem ATTR = "Revision" > >
        <setval desc <ItemGetAttr myitem ATTR = "Description" > >
       <setval filesize <ItemGetAttr myitem ATTR = "File Size" >
   >
        <setval itemid <ItemGetAttr myitem ATTR = "CMS Id" > >
wml-->
```

The first line after the opening WML tag sets the value of an index variable to 0. The code increments this variable after the attributes for each item have been displayed on the screen until index equals count (<while index lt count>), which is equal to the number of items the query found. When these two variables are equal, the code has displayed the attributes for all of the items found by the query.

The <QueryGetItem> statement in the fourth line fetches the first item from the query's result set. The six <setval> statements that follow extract the desired attributes from the item, using the nested <ItemGetAttr> statements, and assign these values to properly named variables. For example, the first <setval> statement assigns a value to the variable name. This value is determined by means of the <ItemGetAttr> statement, which extracts the name attribute from the selected item.

The five <setval> statements that constitute the rest of this WML comment tag follow a similar process to set the values of the variables material, revision, desc, filesize, and itemid. The item ID, as we mentioned earlier, is a construct that we use to transfer data from one page to another. So that a user can click on the name of an item in the Name field and get detailed information in the Details page, WebLink needs a way to pass information about the item from the Results to the Details page. That is why the code extracts this attribute from the query set now.

The next section of the code fills the next row of the table with the values of the name, material, revision, desc, and filesize variables. (If it's the first time through the program, the code fills the first row immediately underneath the column headings.) This printing is accomplished through the series of <printval> statements shown below.

```
<tr>
    <td width="20%"><!--wml <a
href="/cgi-bin/weblink.cgi/detail.html?id=<printval
itemid>"><printval name></a>  wml--></td>
    <td width="20%"><!--wml <printval material> wml--></td>
    <td width="20%"><!--wml <printval revision> wml--></td>
    <td width="20%"><!--wml <printval desc> wml--></td>
    <td width="20%"><!--wml <printval filesize> wml--></td>
  </tr>
```

Notice the HREF link that precedes the <printval name> statement, which causes the item name to appear in the first column of the table. Each of the item names that appears in this column is a link. A <printval itemid> statement appears within this HREF. When a user clicks on a name, the program sends the item ID to the file detail.html, which is the name of the Details screen. That way, the program can identify the item for which the user wants to obtain detailed information.

Once a row has been filled with an item's attributes, the program then must get the next item from the query's result set. The program does this by incrementing the value of the index variable and then returning to the top of the <while> loop. If the value of index is still less than the value of count, there are still items to display, and the program once again goes through the procedure we have just described. If index and value are equal, then the program has displayed all of the items that the query found, and we have reached the end of the program.

```
<!--wml
<setval index <expr index + 1> >
</while>
wml-->
</table>
<!--wml </if> wml-->
</body>
</html>
```

The Details screen

As we mentioned earlier, users can click on the name of a file in the Results screen and go to the Details screen, which gives more complete information about that file. The code below generates and fills the Details screen.

```
<!--wml
<ItemLoad myitem ID=id>
<setval name <ItemGetAttr myitem ATTR = "Name">>
<setval material <ItemGetAttr myitem ATTR = "Material">>
```

```
<setval revision <ItemGetAttr myitem ATTR = "Revision">>
<setval desc <ItemGetAttr myitem ATTR = "Description">>
wml-->

<html>
<head>
<title>Name</title>
<meta name="GENERATOR" content="Microsoft FrontPage 3.0">
</head>
<body>
<table border="0" width="100%">
  <tr>
    <td width="100%"><table border="0" width="100%">
      <tr>
        <td width="9%"><strong>Name</strong></td>
        <td width="23%"><!--wml <printval name> wml--></td>
        <td width="13%"><strong>Material</strong></td>
        <td width="21%"><!--wml <printval material> wml--></td>
        <td width="13%"><strong>Revision</strong></td>
        <td width="21%"><!--wml <printval revision> wml--></td>
      </tr>
    </table>
    </td>
  </tr>
  <tr>
    <td width="100%"><strong>Description</strong></td>
  </tr>
  <tr>
    <td width="100%"><!--wml <printval description> wml--></td>
  </tr>
</table>
</body>
</html>
```

First, the <ItemLoad> statement loads the ID of the file the user clicked on in the Results screen. The Details screen needs this information to know which file it is going to display.

Next, the four <setval> statements set the name, material, revision, and desc variables equal to the respective valued passed over from the previous page.

The rest of the code creates tables that display the name, material, revision, and description information. The first three variables appear in one row, and the Description appears in a row of its own directly underneath. Note the four <printval> statements, which are used to print the appropriate variables on the screen.

Debugging an application

Although it is impossible in this format to give specific advice regarding the many types of problems that may be encountered, general guidelines can be considered when debugging WML code.

WebLink includes two binary files. The first is weblink.cgi, the gateway program. A gateway, or CGI, is a program that is triggered by a Web browser and that creates a link between a Web server and some other client. In this case, weblink.cgi talks to the other binary file, webclient, the ProductCenter client. It is this connection that allows WebLink to extract data from the ProductCenter database and display the data in the browser.

Be sure to set the $WEBLINK_HOME environment variable equal to the pathname where the webclient resides.

The binary file "webclient" can be started with several arguments. Use these arguments at different stages of the debugging process when it is time to debug WML code.

There are three sets of arguments that can be entered:

1. webclient
   webclient -?
   webclient -help

   This statement simply prints webclient usage. These three variations all perform the same function.
2. webclient -syntax <inputfile> <outputfile>

   This statement performs a syntax check of the <inputfile> and sends the results to <outputfile>.
3. webclient -interpret <inputfile> <outputfile>

   This statement executes the WML input file — that is, it interprets the file to make sure it works correctly — and then puts the results into the <outputfile> specified.

When debug WML code, do so one page at a time. That way, each page can be verified that it works as it was intended. From a page-by-page analysis, the source of the error can be exactly pinpointed.

To debug WML code:

1. Do a syntax check of each file using the -syntax argument.
2. Do a run time check using the -interpret argument.

In WebLink, each variable must be initialized before it can be used. The <defval> tag can be used to initialize all variables to a known state. After the run time check is performed, check the application in a browser to see if it works as expected.

Chapter 3

Variables, Flow Control, and Attributes

3

Just Ahead:

Variables 40
Variable-related tags 42
Control structures 44
String operations 46
File operations 49
Handling of attributes 51

WML (WebLink Markup Language) is an extended HTML language. This language allows the extraction of data from ProductCenter and the display of the data on Web pages. This chapter focuses on the basics of the WML language: variables, control structures, and processing attributes.

Variables

The WML scripting language does not support explicit data type declarations. The language is weakly typed, which means that strong type checking and enforcement are not built into the language. This is typical of most scripting languages.

The language supports only implicit data types, which can be classified broadly into two categories:

- Fundamental data types
- Compound data types

You must initialize variables before you use them. There are three ways to initialize variables:

- All HTML form variables (GET and POST) are initialized by WebLink before it starts processing a WML page.
- You can use the <SetVal> tag.
- You can use the <DefVal> tag.

Fundamental data types

WML supports two fundamental data types, strings and integers. WML does not support floating point numbers. Internally, integers are stored as strings and are converted as needed.

For example, consider the following construct:

```
<SetVal STRING "This is a string">
<SetVal INTEGER "1234">
```

This construct initializes a string value and an integer value to the variables STRING and INTEGER, respectively. STRING and INTEGER are variable names. (They could be any valid variable name.)

Compound data types

Compound data types are data types that represent the ProductCenter objects. You cannot do a <SetVal> on a compound data type.

For example, the following tag creates a query object.

```
<QueryCreate QUERY1>
<QueryCreate QUERY2>
```

But <SetVal QUERY1 QUERY2> is and would result in an error.

Indirect lookups

You can place a dollar sign ($) before variable names to perform indirect lookups. An indirect lookup on a variable returns the value of a second variable whose name matches the value of the first variable.

For example, consider the diagram below, which shows two variables and the contents of each. The first variable, TEMP, is set equal to the value RECORD1. That value matches the name of the second variable, whose value is set equal to TESTA.

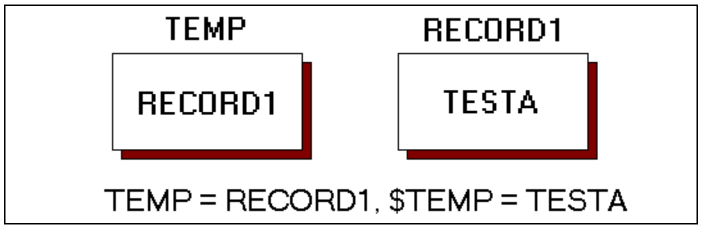


Figure 3-1: Indirect lookup of variables

Given these variables, the argument $TEMP in a WML statement returns TESTA — that is, the value of RECORD1, whose name matches the value of the first variable. In contrast, the argument TEMP returns the value RECORD1.

You can perform indirect lookups only on fundamental data types. For example, RECORD1 cannot be an item or query data type.

Other conventions

1. WML does not support floating point arithmetic; all floating point values are treated as strings.
2. You must surround all operators in WML, including assignment operators such as the equal sign, with a leading and trailing space. For example:
   - <expr A + B> is a valid statement.
   - <expr A+B> is not.
3. A variable name can be no more than 32 characters long, must begin with a letter, and should contain only alphanumeric characters. Although it is possible to start variable names with characters that are not alphanumeric -- such as leading dots (.), dollar signs ($), percent signs (%), question marks (?), forward slashes (/), and backward slashes (\) -- it is strongly discouraged. Finally, note that any of the relational or logical names shown in Table 3-1 on page 45 are reserved words and cannot be used for variable names.

Variable-related tags

SetVal

Function	Sets a variable to a value.
Usage	<SetVal *variable value*>
Arguments	*variable* is the name of the variable. *value* is the value to set the variable.
Return value	none.
Examples	<SetVal MYVAR VAR1> <SetVal MYVAR <ItemGetAttr ITEM1 ATTR = "NAME">> <SetVal PRINTDEBUG "0"> <SetVal PRINTDEBUG "1">

Sets a variable to a value.

The right hand side of the expression could be a string constant (such as "This is a string"), a numeric constant ("500"), the value of a variable (VAR1), or any WML construct that returns a value that is a fundamental data type. Notice the second example in the table above. The "ItemGetAttr" (see page 88) tag returns the value of the specified attribute (NAME). This example sets the variable MYVAR equal to that returned value.

DefVal

Usage	<defval *variable value*>
Arguments	*variable* is the name of the variable. *value* is the value to set the variable.
Return value	none.
Examples	<defval COUNT "10"> <defval COUNT2 "20">

Initializes a variable to a default value, assuming that the variable has not already been set to some value.

If the variable already exists, WebLink ignores the default value. You should initialize form variables to a known state in case users do not enter values for these variables into a form.

For example, suppose that a user is filling out a form that asks for a value for the variable count, and the user does not supply a value. The user clicks on a button to advance to the next form in an application. The first form cannot pass a value for count to the second form, since this value was not supplied. If the WML code for the second form includes the statement <defval COUNT "10">, then the system automatically initializes the value of count to 10 in the second form.

If the user does supply a value for count in the first form, however, then the <defval COUNT "10"> statement is ignored in the second form, since that variable already has been set. If count is used without having been initialized, you receive a run time error.

A later <SetVal> statement, however, can override a previous <DefVal> setting. If, for example, the system processes the <DefVal> statement above and then encounters the statement <SetVal COUNT "35">, then the value of count changes from 10 to 35.

Printval

Usage	<PrintVal *variable*>
Arguments	the variable to be printed. Only one variable can be printed at a time.
Return value	none.
Example	<PrintVal PARTNO1>

Displays the value of a variable.

For example, suppose the value of a ProductCenter attribute is in the variable MYVAR and you want to print that value in bold and italic type in a HTML page. Your HTML code should contain the following line:

```
<B> <I> <PrintVal MYVAR> </I> </B>
```

So, if the value of MYVAR were "20.0", the command above would display the value 20.0 in bold and italic type.

Expr

Usage	<expr *expression*>
Arguments	*expression* is the expression to be evaluated. The expression can include binary operators, and operands can be expressions.
Return value	the result of the expression evaluation.
Example	<expr VAR1 + VAR2> <SetVal MYVAR <expr VAR1 * VAR2>> <expr VAR1 / VAR2> <expr VAR1 % VAR> <expr VAR1 AND VAR2> <expr VAR1 OR VAR2>

Evaluates an arithmetic, logical, or relational expression, and performs string concatenation.

The operands of the binary operator can be anything that has a value, such as variable names.

Consider a statement in the form <expr VAR1 + VAR2>. VAR1 and VAR2 can be either integers or strings. If both of them are integers, then the statement adds the two integers together. If one or both of them is a string, then the statement performs string concatenation.

For example, consider the statement <expr x + y>, where x is equal to the integer "2" and y is equal to the integer "7". Since both of the arguments are integers, the statement returns the integer "9".

If the variables x and y were strings instead, then the two strings would be concatenated. So if x were equal to the string “Never” and y were equal to the string “more”, the statement would return the concatenated string “Nevermore”.

Integers and strings are valid operands in an <expr> statement. However, they must be enclosed in double quotes.

For example, consider the statement <expr x + “20”>. If x is equal to an integer, then the statement adds the integer 20 to the integer x and returns the result. So if x is equal to the integer “2”, then the statement returns the integer “22”.

If x were equal to a string, however, then the statement would convert the integer “20” to a string and concatenate the two arguments. So if x is equal to the string “Matchbox “, the statement would concatenate the two arguments and return the string “Matchbox 20”.

A non-zero value is true and zero is false for all logical operations and operands.

If a program encounters an error because an invalid data type was passed to the <Expr> tag, an error is generated.

Control structures

If-elseif-else

Usage	<If *condition*> (statements) <elseif *condition*> (statements) <else> (statements) </If>
Arguments	the condition(s) to be evaluated. Only relational operators are supported. See Table 3-1 on page 45 for the list of WML relational operators.
Return value	none.
End tag	</If>
Example	<If MYVAR EQ <expr VAR1 + VAR2>> (statements) </If> <SetVal class "CMS:Projects"> <If <expr class EQ "CMS:Projects"> OR <expr class EQ “CMS:Parts">> (statements) </If>

Provides flow control based on program logic.

The <If...elseif...else> tags are the familiar constructs that allows a block of tatements to be executed depending on which conditions hold true. Even if the if block is just one statement, as in the example in the table above, the closing </If> tag is required.

WML supports only simple conditions involving one binary operator with just two operands. Complex conditional expressions should be broken down into a series of simpler nested "Expr" (see page 43) tags.

The <If> tag does not support unary values as conditions. For example, <If VAR1> is an invalid construct. The statement should be written as <If VAR1 EQ "1"> instead.

In addition, the condition cannot be an arithmetic expression. <If A + B> is not a valid construct, whereas <If <expr A + B> eq "1"> is valid.

Finally, note that WML does not allow the bracing of conditions to indicate order of precedence.

Table 3-1: Relational and Logical Operators

Operator	Description
GT	Greater than
LT	Less than
GE	Greater than or equal to
LE	Less than or equal to
EQ	Equal to
NE	Not equal to
AND	Logical AND
OR	Logical inclusive OR

While

Usage	<While *condition*>
Arguments	the condition to be evaluated. Only relational operators are supported. See Table 3-1 for a list of relational operatos.
Return value	none.
End tag	</While>
Example	<While VAR1 LT <expr VAR2 + VAR3>> (statements) </While>

Provides a looping construct.

This construct checks for a specific condition and then executes a series of WML statements for as long as the condition holds returns TRUE.

The *condition* for the <While> construct supports only simple binary operators with two operands. To test for complex conditions, you must nest the "Expr" (see page 43) tag as one or both of the operands.

The <While> tag does not support unary values in a *condition*. For example, <While VAR2> is an invalid construct. You should write something like <While VAR2 EQ “1”> instead. In addition, the *condition* cannot be an arithmetic expression. <While A + B> is not a valid construct.

Remember that this looping construct must always end with a </While> tag. An error will be generated if this closing tag is missing.

Break

Usage	<Break>
Arguments	none.
Return value	none.

Example	<While A GT B>
	    (statements)
	    <If X GT Y>
	        <Break>
	    </If>
	</While>

Breaks the execution of a loop and transfers control of the program to the statement immediately at the end of the loop.

An error is generated if <Break> is used outside of a looping construct.

String operations

EncodeVal

Usage	<EncodeVal *string* TYPE = *type*>
Arguments	*string* is the value that will be encoded. *type* is "JS" “HTML”, "backslash" or “text
Return value	the encoded string
Example	<EncodeVal "R&D" TYPE="HTML"> returns "R&amp;D"

Encodes a string.

Encoding is performed as follows:

- “JS”: single quotes (“), double quotes (‘) and carriage returns charaters (\n and \r) are escaped with a backslash (\).
- “HTML”: less than signs (<) are replaced with “&lt;”, greater than symbols (>) are replaced with “&gt;”, carriage return characters (\n) are replaced with “<BR>”, double quotes (") are replaced with “&quot;” and ampersands (&) are replaced with “&amp;”.
- “backslash”: any backslash characters (\) are escaped with a backslash (\).
- “text”: *tring* is returned unmodified.

GetLength

Usage	<GetLength *inputString*>
Arguments	*inputString* is the variable or constamt.
Return value	the length of the variable or string.
Example	<SetVal strLen <GetLength "ABCDE">> sets strLen to 5

Computes the length of a string.

The argument you pass to this tag can be a variable, constant, or a value returned from another function. <GetLength> treats integers as strings, so the command <GetLength "375" returns the number 3.

If *inputString* is not a string, the command generates a runtime error.

GetSubString

Usage	<GetSubString *var1* SINDEX = *value1* EINDEX = *value2*>
Arguments	*var1* is a variable. *value1* is the index number that is the start of the extracted portion of var1. The first character of var1 has an index number of 0. *value2* is the index number that is the end of the portion you want to extract.
Return value	if successful, the segment of var1 that you specify. Otherwise, an empty string.
Example	<GetSubString "charlierose" SINDEX = "6" EINDEX = "9">

Extracts a portion of a string.

Each character in the input string has an index number associated with it. The first character has an index value of 0. The last has an index value of n-1, where n is the number of characters in the string. In the <GetSubString> statement, SINDEX is the index number of the character you want at the beginning of your extracted portion, and EINDEX is the index number of the character you want at the end.

Consider the example in the table above. This example takes the argument "charlierose," extracts the characters from index 6 (the first E) through index 9 (the S) and returns the segment "eros."

If you want to pass a string constant to <GetSubString>, be sure to enclose the constant in double quotes, as we have here. If you don't, <GetSubString> assumes that you have passed a variable name. When you pass a variable name, <GetSubString> takes the value to which you have set that variable and extracts the designated subset from that value.

The following rules apply to <GetSubString>:

1. *var1* has to be a fundamental data type.
2. If SINDEX is negative, it will default to 0.
3. If SINDEX is more than or equal to the string length, the return value will be an empty string.

4. You can set EINDEX equal to a negative number if you want to strip that number of characters from the end of the input string. For example, if you set *var1* equal to "sundance" and set SINDEX to 0 and EINDEX to -5, <GetSubString> will return "sun." If the absolute value of EINDEX exceeds the number of characters in *var1*, the return value will be an empty string.
5. If EINDEX is less than SINDEX and EINDEX is greater than or equal to 0, <GetSubString> returns an empty string.

SplitString

Usage	<SplitString *inputString* TOKEN = *token* VALUE = *outVariable*>
Arguments	*inputString* is the input string. *token* is the token delimiter appearing within the string var1. Text appearing before and after each instance of this delimiter will appear in a separate substring. *outVariable* is the prefix of the names of the newly created substring. New substrings will have the names *outVariable0*, *outVariable1*, *outVariable2*, and so on.
Return value	the number of substrings created if successful. "1" if the token was the first or last character (or both) in the string, or if the token did not appear in the string. "-1" on error.
Example	<SplitString "CMS:Files:Documentation" TOKEN = ":" VALUE = "class">

Divides a string of text into a series of substrings based on a token delimiter.

In the example shown above, the token delimiter is a colon (:). This delimiter appears twice in the argument "CMS:Files:Documentation." This statement divides the input string into three substrings: class0 contains the substring "CMS", class1 contains "Files", and class2 contains "Documentation".

If the command returns a -1, that result indicates an error. The probable reason for the error is that the token was a null string.

The token can be any valid character or the escape characters \n (new line)and \t (tab).

URLEncode

Usage	<URLEncode *stringToEncode*>
Arguments	*stringToEncode* is the string that will be encoded.
Return value	the encoded string
Example	<setval argEncoded <URLEncode filename>>

Encodes a string for safe usage as a URL value.

Te encode tag will interrogate each character of the supplied string and replace non-alpha numeric characters with a “%” followed by the character’s corresponding hexadecimal value.

URLDecode

Usage	<URLDecode *stringToDecode*>
Arguments	*stringToDecode* is the URL encoded string.
Return value	the decoded string.
Example	<setval filename <URLDecode argEncoded>>

Decodes a string with URL encoding.

File operations

FileAccess

Usage	<FileAccess FILE = *file* MODE = *mode*>
Arguments	*file* is the name of the file. *mode* is the access mode to check: “**r”** for read-only access or “**w”** for write access.
Return value	"1" if the file has the access privilege or "0" if not.
Example	<FileAccess FILE = "partslist" MODE = "r">

Returns the access mode of a file.

GetDatabaseEntries

Usage	<GetDatabaseEntries NAME = *javascriptObjectName* TYPE = *type*>
Arguments	*javascriptObjectName* is the javascript object array that will be defined. *type* can only be "JS".
Return value	the javascript code defining *javascriptObjectName.*
Example Code	var dbEntries = new Array(); <GetDatabaseEntries NAME = "dbEntries" TYPE = "JS">

Example Return	function dbEntries_object(dbname,hostname,portnumber)
	{
	    this.dbname = dbname;
	    this.hostname = hostname;
	    this.portnumber = portnumber;
	}
	
	    dbEntries[0] = new dbEntries_object("PCTR","lancaster",5400);
	    dbEntries[1] = new dbEntries_object("PCTRTEST","lowell",5400);

Generates javascript code to populate a javascript array with database entries in the Weblink “database.cfg” file.

GetStyleSheetListing

Usage	<GetStyleSheetListing NAME = *javascriptArrayName* TYPE = *type*>
Arguments	*javascriptArrayName* is the javascript object array that will be defined. *type* can only be "JS".
Return value	the javascript code defining *javascriptObjectName*.
Example Code	var myStyleSheets = new Array(); <GetStyleSheetListing NAME = "myStyleSheets" TYPE = "JS">

Example Return	function myStyleSheets_object(filename)
	{
	    this.filename = filename;
	}
	
	    myStyleSheets[0] = new myStyleSheets_object("action.xsl");
	    myStyleSheets[1] = new myStyleSheets_object("BOMMultiLevel.xsl");
	    myStyleSheets[2] = new myStyleSheets_object("compare.xsl");
	    myStyleSheets[3] = new myStyleSheets_object("filter.xsl");

Generates javascript code to populate a javascript array with a list of XSL files shown in the ProductCenter Webclient's XSL folder.

GetValueFromFile

Usage	<GetValueFromFile FILE = *file* NAME = *name*>
Arguments	*file* is the name of the file. *name* is a parameter in the file.
Return value	the value to which *name* is set, or a null string if the *name* does not exist in the FILE.
Example	<GetValueFromFile FILE = "mydoc.txt" NAME = "surname">

Searches a file for a parameter and returns the value of that parameter.

Parameters and values within *file* must be separated with "=" and only one parameter/value pair may exist per line.. If mydoc.txt contains a line such as "surname=Letterman", the example shown above returns the value "Letterman." If "surname" does not appear in mydoc.txt, the tag above returns a null string.

SaveToFile

Usage	<SaveToFile NAME = *data* FILE = *fileName* MODE = *fileMode*>
Arguments	*data* is the data to write to *fileName*. *fileName* is the name of the file. *fileMode* is "CREATE" or "APPEND".
Return value	"0" for success and a non-zero error code on error.
Example	<SAVETOFILE NAME = fileData FILE = "testData.txt" MODE = "CREATE">

Writes data to a file.

Creates a new file if *fileMode* is “CREATE” and appends if *fileMode* is “APPEND”.

UnfarcFile

Usage	<UnfarcFile FILE = *fileName*>
Arguments	*fileName* is the name of the compressed file.
Return value	"0" for success and a non-zero error code on error.

Uncompresses a “farc” file.

A “farc” file is generated using the Webclient plugin javascript function

```
WTCIEFileCtrl.CreateFarcFileFromList(filelist);
```

where filelist is a comma separated list of file names in the working directory. After calling this plugin function, a file named “PCTR_MULTIPLE_FILE_ADD” is generated. This file can then be decompressed in a Webclient application using the “UnfarcFile” tag.

Handling of attributes

Single-valued attributes

For single-valued attributes, the attribute can be set equal to the value to which you want it set. You do this through the use of the fundamental data types, integers or strings. Future chapters will describe tags for extraction of and setting of single-valued attributes.

Table-type attributes

Table-type attributes have a one-to-many relationship with a ProductCenter object. These attributes have one name, but many rows of data and each row may contain several pieces of data. To extract a row or rows of data from that attribute, use the convention “attribute_name[row].column_name”. If you want to cycle through several values of row, you must perform string concatenation with the <Expr> tag.

For example, consider the following <While> loop:

```
<SetVal index “0”>
<While index lt “3”>
   <SetVal loaderr <ITEMLOAD myitem ID = itemid>>
   <SetVal attrName <expr “CUSTOM:customAttribute[“
      + <expr index + “].integerPrompt”>>>
   <ItemSetAttr myitem ATTR = attrName VALUE = index>
   <SetVal index <expr index + “1”>>
</While>
```

The third <SetVal> command sets *attrName* equal to a table-type attribute, but the program needs to loop through several values of *index*. To do this, the programs concatenates the attribute name, (*customAttribute*), the attribute’s table column name (i*ntegerPrompt*) and the row index with the <Expr> tag. The <ItemSetAttr> tag then sets the attribute defined by *attrName* equal to *index*.

Chapter 4

Connections, Status, and Settings

4

Just Ahead:

Comment tag 54
Connection tags 54
General tags 56

This chapter describes the WML tags that can be used for comments, establishing a connection to the ProductCenter database, and other basic tasks.

Comment tag

WML honors the standard HTML comment tag. The HTML comment tag can be used whenever there is a need to add comments to HTML or WML code:

```
<!-- ...commented-out code... -->
```

When WML is embedded in HTML, it is recommended that WML is enclosed in WML comment tags:

```
<!--wml ...WML code... wml -->
```

Some web page editors modify non-standard tags, such as WML tags. Web page editors ignore anything that appears inside a WML comment tag, but WebLink processes whatever is inside of it, provided that “wml” appears immediately after the first two dashes of the opening tag and immediately preceding the two dashes at the ending tag.

Connection tags

SetClientType

Usage	<SetClientType Value=”client_type”>
Arguments	*client_type* is the client application.
Return value	"0" for success and a non-zero error code on error.

For customer specific WebLink applications, use the SetClientType WebLink tag to set the client_type or client application name just prior to the WebLink Login. This value will be recorded in the Transaction Log when tracking successful and/or failed logins.

If for example, client_type was set to a value of xCustApp then the Transaction Log for a successful login would be:

SUCCESS D745-02 (192.168.28.239) Client: D745-02.softech.com (192.168.28.239)=D745-02 (192.168.28.239)=D745-02 SYSTEM Web Browser: D745-02.softech.com (192.168.28.239) User Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022) xCustApp pcwebclient.exe - run D745-02 4386

Login

Usage	<Login USERNAME = *username* PASSWORD = *passwd* DATABASE = *database*>
Arguments	*username* is the user's login name. *passwd* is the user's password. *database* is the database name.
Return value	"0" for success and a non-zero error code on error.
Example	<Login NAME = "jtati" PASSWORD = "locate" DATABASE = "PCTR">

Logs a user in to the ProductCenter database.

The <Login> tag must appear before the <HTML> tag in WML pages.

Logout

Usage	<Logout>
Arguments	none.
Return value	"0" for success and a non-zero error code on error.

Terminates the current session.

The <Logout> tag must appear before the <HTML> tag in WML pages.

CreateHelperSession

Usage	<CreateHelperSession>
Arguments	none.
Return value	"0" for success and a non-zero error code on error.

Creates a helper session.

A helper session can be required by a ProductCenter integrator to do Weblink specific operations, for example imprting of a BOM from a CAD tool into ProductCenter. This tag will create a Weblink helper session where a license is not consumed for the session. Three Weblink variables must exist on the same page as "CreateHelperSession" in order for the tag to work: "user", "pass" and "db" where "user" is the login name of the user, "pass" is the decrypted user password of the user and "db" is the database name. No Weblink logout is required when connecting with this tag.

General tags

ErrorGetLastMessage

Usage	<ErrorGetLastMessage>
Arguments	none.
Return value	the error message text for the most recent WML tag that returns an error code. Blank if the last such WML tag processed successfully.
Examples	<ErrorGetLastMessage> <SetVal error <ItemAdd item>> <If error NE "0"> (The following error <B><I> <Printval <ErrorGetLastMessage>> </I></B> was encountered performing the Add operation. </if>

Returns the error message, if any, from the last executed WML tag.

Exec

Usage	<Exec *output* COMMAND = *command*>
Arguments	*output* is a variable into which the <Exec> tag places the output of the external application. *command* is the command for calling the external application.
Return value	none.
Example	<Exec output COMMAND=<Expr "mailx -s " +<Expr callnum + <Expr " " + <Expr assigned + "<messagefile>">>>>>

Executes another application and returns the output of that application into the variable *output*.

The example in the above table shows how to use the <Exec> tag to send email from within WebLink. This statement sends email to the *assigned* user. The subject of the message is in the variable *callnum*.

NOTE: The Exec tag will only work if the web server allows the weblink process to start the application. With the inherent security risks in allowing CGI applications (like WebLink) to start other applications, some web servers do not allow CGI applications. The Exec tag should be avoided.

GetMessage

Usage	<GetMessage MSGCODE = *msgCode*>
Arguments	*msgCode* is a message code returned by a Weblink tag.
Return value	the message description for *msgCode*.
Example	<SetVal ErrNum <ItemLoad myItem ID = "0">> ErrorMessage:<PrintVal <GetMessage MSGCODE = ErrNum>>

Returns the message description of a message or error code.

GetResourceVariable

Usage	<GetResourceVariable NAME = *resourcevariable*>
Arguments	*resourcevariable* is the name of the resource variable.
Return value	the value of the resource variable (in upper case) from the site file or a null string if the resource variable is not present.
Example	<GetResourceVariable NAME = "cms.action.print_desktop.ascii">

Returns the value of the resource variable from the ProductCenter site file.

GetSessionVal

Usage	<GetSessionVal *variableName*>
Arguments	*variableName* is the session variable to read.
Return value	the value of the variable.
Example	<SetVal rm_enabled <GetSessionVal site_relmgmt>>

Returns the value of a session variable.

IsValidSession

Usage	<IsValidSession>
Arguments	none.
Return value	0 = session is no longer active 1 = session is active but all the stored variables are lost 2 = session is active
Example	<SetVal sessionstatus <IsValidSession>> <If sessionstatus eq "2"> (statements) </if>

Determines if current session has timed out.

PrintFile

Usage	<PrintFile *filename*>
Arguments	*filename* is the name of the file, optionally including its path, that will be sent to the Web browser.
Return value	none.
Example	<PrintFile “/template/wml/weblink/control/edit_files/checkin.html”>

Sends a text file to the user’s browser.

The PrintFile tag is useful for sending common re-used components of a web application to the users browser. For example, a common re-used component in medium to large size web applications is HTML that creates a button or menu bar. Putting the code for that menu bar in one file, then referencing that file wherever needed with the “PrintFile” tag allows for cleaner code and easier maintenance.

These are the rules for the *filename* argument:

- If the *filename* includes an absolute path, as in the example above, then <PrintFile> sends *filename* to the user’s browser.
- If the *filename* includes a relative path (i.e., begins with “.” or “..”), <PrintFile> searches for *filename* relative to the location where the webclient is running (usually in the web server’s “cgi-bin” folder), then sends the file to the user’s browser.
- If *filename* doesnot include a path, WebLink searches for *filename* in the directory specified by the file_dir configuration variable in weblink.cfg (see “weblink.cfg” on page 166). When it finds the file, it ends the file to the user’s browser.

SessionGetClassId

Usage	<SessionGetClassId CLASS = *className*>
Arguments	*className* is the name of a class.
Return value	the ID of a class, or -1 if the class does not exist.
Example	<SessionGetClassId CLASS = "CMS:Files">

Returns the class ID of a class name.

SessionGetFile

Usage	<SessionGetFile FILE = *fileName*>
Arguments	*fileName* is the name of the file.
Return value	"0" for success and a non-zero error code on error.

Sends a binary file to the user’s browser.

SessionIsModulEnabled

Usage	<SessionIsModulEnabled NAME = *moduleName*>
Arguments	*moduleName* is "BOM_EDITOR" or "JVUE_INTEGRATOR".
Return value	“TRUE” is *moduleName* is enabled, “FALSE” if not.
Example	<SessionIsModulEnabled NAME = "BOM_EDITOR">

Determines if a module is enabled.

SetMimeType

Usage	<SetMimeType VALUE = *mimetype*>
Arguments	*mimetype* is the new mimetype to set.
Return value	none.
Example	<SetMimeType VALUE = "application/vnd.ms-excel">

Overwrites the default mimetype of output received from WebLink.

This tag must appear before the <HTML>, “ItemGetCopy” (see page 94), and “ItemCheckout” (see page 93) tags.

SetSessionVal

Usage	<SetSessionVal *variableName value*>
Arguments	*variableName* is the session variable to set. *value* is the value to set to *variableName*.
Return value	"0" for success and a non-zero error code on error.
Example	<SetsessionVal page_size page_size> <SetsessionVal threshold prefered_threshold> <SetsessionVal prefered_class prefered_class>

Sets a variable for the current session.

Chapter 5

Users and Groups

5

Just Ahead:

Getting group information 62
Setting group information 64
Getting user information 65
Setting user information 68
Item Column Layout Object 72
Creating an item column layout object 72
Constructing or Editing the layout 72
Accessing the layout 73
Layout storage functions 74

This chapter describes tags that enable management of ProductCenter users and groups. To get lists of users and groups, see “ListLoad” (see page 78). To get counts of these lists, see “ListGetCount” (see page 80).

Getting group information

GroupGetAttrCount

Usage	<GroupGetAttrCount *groupObject*>
Arguments	*groupObject* is the group to check.
Return value	the number of attributes associated with the group.
Example	<SetVal groupAttrCnt <GroupGetAttrCount myGroup>>

Returns the number of group attributes of *groupObject*.

GroupGetAttrNameByIndex

Usage	<GroupGetAttrNameByIndex *groupObject* INDEX = *index*>
Arguments	*groupObject* is the group to access. *index* is a value from 0 to the value returned by “GroupGetAttrCount”.
Return value	the group attribute name.
Example	<SetVal attrName <GroupGetAttrNameByIndex myGroup INDEX = "0" >>

Returns the group attribute name by index.

GroupGetAttr

Usage	<GroupGetAttr *groupObject* ATTR = *attrName*>
Arguments	*groupObject* is the group to access. *attrName* is the name of the attribute to return.
Return value	the attribute value.
Example	<GroupGetAttr myGroup ATTR = "Name">

Returns the group attribute value. Possible *attrName* values are "Group Name" and "Id".

GroupGetUserCount

Usage	<GroupGetUserCount *groupObject*>
Arguments	*groupObject* is the group to count.
Return value	the number of users in the group.
Example	<GroupGetUserCount myGroup>

Returns the number of users in the group.

GroupGetUser

Usage	<GroupGetUser *groupObject userObject* INDEX=*index*>
Arguments	*groupObject* is the group object to access. *userObject* is the user object. *index* is a value from 0 to the value returned by “GroupGetUserCount”.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupGetUser myGroup myUser INDEX = userPtr>

Loads the user object from the group object by the index.

GroupIsUserMember

Usage	<GroupIsUserMember *groupObject* TYPE = *loginName*>
Arguments	*groupObject* is the group to check. *loginName* is the user to check.
Return value	"TRUE" if the user is a member of the group and"FALSE" if the user is not.
Example	<GroupIsUserMember myGroup TYPE = loginName >

Determines whether the user named *loginName* is a member of the group represented by *groupObject*.

GroupLoadByName

Usage	<GroupLoadByName *groupObject* NAME = *groupName*>
Arguments	*groupObject* is the group object. *groupName* is the name of the group to load.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupLoadByName myGroup NAME = "Document Control">

Returns the group object based on the group name.

GroupLoad

Usage	<GroupLoad *groupObject* ID = *groupid*>
Arguments	*groupObject* is the group object. *groupid* is the id of the group to load.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupLoad myGroup ID = "100">

Returns the group object based on the group id.

Setting group information

GroupAddUser

Usage	<GroupAddUser *groupObject* NAME = *userName*>
Arguments	*groupObject* is the group to access. *userName* is the user name of the user to add to *groupObject*.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupAddUser myGroup NAME = "gmcohan">

Adds a user to a group. Must be followed by a call to GroupSave.

GroupCreate

Usage	<GroupCreate *groupObject*>
Arguments	*groupObject* is the group object that will be created.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupCreate myGroup>

Creates a group object.

GroupDelete

Usage	<GroupDelete *groupObject*>
Arguments	*groupObject* is the group to delete.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupDelete myGroup>

Deletes a group.

GroupRemoveUser

Usage	<GroupRemoveUser *groupObject* NAME = userName>
Arguments	*groupObject* is the group to access. *userName* is the user name of the user to remove from *groupObject*.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupRemoveUser myGroup NAME = "gmcohan">

Removes a user from a group. Must be followed by a GroupSave tag.

GroupSave

Usage	<GroupSave *groupObject*>
Arguments	*groupObject* is the group to save.
Return value	"0" for success and a non-zero error code on error.

Saves changes to group attributes or membership.

If *groupObject* does not exist in ProductCenter, the group is created.

You must set the Name attribute and add at least one user before saving a new group.

GroupSetAttr

Usage	<GroupGetAttr *groupObject* ATTR = *attrName* VALUE = *value*>
Arguments	*groupObject* is the group to access. *attrName* is the attribute to set. Possible values are "Group Name". *value* is the value to set to the attribute.
Return value	"0" for success and a non-zero error code on error.
Example	<GroupSetAttr myGroup ATTR = "Name" VALUE = "Checker">

Sets a group attribute.

Possible value for *attrName* is "Name".

Getting user information

GetScreenPermissions

Usage	<GetScreenPermissions NAME = *javascriptObjectName* TYPE = *type*>
Arguments	*javascriptObjectName* is the javascript object name that will be defined. *type* can only be "JS".
Return value	the javascript code defining *javascriptObjectName.*
Example	<GetScreenPermissions NAME = "userPerms" TYPE = "JS" >

Generates javascript code to populate an object with screen permission for the connected user.

SessionGetUser

Usage	<SessionGetUser *userObject*>
Arguments	*userObject* is the returned user object.
Return value	"0" for success and a non-zero error code on error.

Loads a user object for the connected user.

UserGetAttrCount

Usage	<UserGetAttrCount *userObject*>
Arguments	*userObject* is the user whose attributes will be counted
Return value	the number of attributes of the specified user.

Returns the number of user attributes.

UserGetAttrNameByIndex

Usage	<UserGetAttrNameByIndex *userObject* INDEX = *index*>
Arguments	*userObject* is the user object that will be accessed *index* is a value from 0 to the value returned by “UserGetAttrCount”.
Return value	the name of the attribute.

Returns a user attribute name.

The list of attributes names is shown in Figure 5-1 on page 70.

UserGetAttr

Usage	<UserGetAttr *userObject* ATTR = *attrName*>
Arguments	*userObject* is the user to access. *attrName* is the name of the attribute to return.
Return value	the attribute value.

Returns a user attribute value.

See Figure 5-1 on page 70 for the list of user attributes that can be retrieved.

UserGetGroupCount

Usage	<UserGetGroupCount *userObject*>
Arguments	*userObject* is the user whose group membership will be counted.
Return value	the number of groups to which the user belongs.

Returns the number of the groups to which a user is a member.

UserGetGroup

Usage	<UserGetGroup *userObject groupObject* INDEX = *index*>
Arguments	*userObject* is the user whose group membership ship you wish to access. *groupObject* is the returned group. *index* is a value from 0 to the value returned by “UserGetGroupCount”.
Return value	"0" for success and a non-zero error code on error.

Returns a group object to which a user is assigned.

UserGetScreenPermissionCount

Usage	<UserGetScreenPermissionCount *userObject*>
Arguments	*userObject* is the user whose screen permissions will be counted
Return value	the number of screen permissions that the user has been assigned.

Returns the number of screen permissions that a user has been assigned.

UserGetScreenPermissionNameByIndex

Usage	<UserGetScreenPermissionNameByIndex *userObject* INDEX = *index*>
Arguments	*userObject* is the user object that will be accessed. *index* is a value from 0 to the value returned by "UserGetScreenPermissionCount".
Return value	name of the screen permission.

Returns the name of a screen permission assigned to a user.

UserGetScreenPermission

Usage	<UserGetScreenPermission *userObject* ATTR = *perm* TYPE = *type*>
Arguments	*userObject* is the user object that will be accessed. *perm* is the name of the screen permission to access. *type* specifies the screen access level: "VIEW" or "EDIT".
Return value	value of the screen permission (TRUE or FALSE).

Determines whether a user has a specific screen permission.

UserIsMemberOf

Usage	<UserIsMemberOf *userObject* TYPE = *groupName*>
Arguments	*userObject* is the user object. *groupName* is the name of the group to check.
Return value	FALSE if *userObject* is not a member of the group *groupName* and TRUE if the user is.

Determines whether a user is a member of a group.

UserLoadByName

Usage	<UserLoadByName *userObject* NAME = *loginName*>
Arguments	*userObject* is the user object. *loginName* is the user's log-in name.
Return value	"0" for success and a non-zero error code on error.

Returns a user object based on a login name.

UserLoad

Usage	<UserLoad *userObject* ID = *id*>
Arguments	*userObject* is the user object. *id* is the id of the user to load.
Return value	"0" for success and a non-zero error code on error.

Returns a user object based on a user id.

VerifyPassword

Usage	<VerifyPassword *password*>
Arguments	*password* is the password you wish to check.
Return value	"1" if the password is correct, "0" if it is not.
Example	<VerifyPassword plainPass>

Determines whether a user's password is correct.

Setting user information

ChangePassword

Usage	<ChangePassword PASSWORD = *oldPassword* NEWPASSWORD = *newPassword*>
Arguments	*oldPassword* is the user's existing password *newPassword* is the password to which the user wishes to change
Return value	"0" for success and a non-zero error code on error.
Example	<if <ChangePassword password=oldPass newpassword=newPass> eq "0"> ...display success message... <else> ...display error message... </if>

Changes the current user's password.

SetNewPassword

Usage	<SetNewPassword USERNAME = *username* PASSWORD = *oldPassword* NEWPASSWORD = *newPassword*>
Arguments	*username* is the user name of the user whose password will be changed. *oldPassword* is *username*'s former password. *newPassword* is *username*'s new password.
Return value	"0" for success and a non-zero error code on error.
Example	<SetNewPassword USERNAME = "gmcohan" PASSWORD = "test" NEWPASSWORD = "music4all">

Changes the specified user's password.

UserSetAttr

Usage	<UserSetAttr *userObject* ATTR = *attrName* VALUE = *attrValue*>
Arguments	*userObject* is the user object that will be modified. *attrName* is the name of the attribute to set. *attrValue* is the value to set to the attribute.
Return value	"0" for success and a non-zero error code on error.
Example	<UserSetAttr myUser ATTR = "Check In Reminder" VALUE = "YES">

Sets a user attribute value.

See Figure 5-1 on page 70 for the list of user attributes that can be set..

UserSetScreenPermission

Function	Sets user screen permission.
Usage	<UserSetScreenPermission *userObject perm* TYPE = *type* ALLOW = *allow*>
Arguments	*userObject* is the user object that will be modified. *perm* is the screen permission name to be set. *type* is either "VIEW" or "EDIT". *allow* is either "TRUE" or "FALSE".
Return value	"0" for success and a non-zero error code on error.
Example	<UserSetScreenPermission myUser ATTR = "Submit" VALUE = "FALSE" TYPE = "VIEW" >

Sets a user screen permission.

See Figure 5-2 on page 71 for the list of permission names.

WmlDecrypt

Usage	<WmlDecrypt NAME = *encryptedString*>
Arguments	*encryptedString* is a encrypted password.
Return value	the decrypted password.
Example	<SetVal decryptedVal <WmlDecrypt NAME = encryptString>>

Decrypts a string previously encrypted with WmlEncrypt.

WmlEncrypt

Usage	<WmlEncrypt NAME = *string*>
Arguments	*string* is a string to encrypt.
Return value	an encrypted value for *string*.
Example	<SetVal encryptString <WmlEncrypt NAME = plainString>>

Encrypts a string.

The encryption algorithm used on *string* is the same one used to encrypt the current user's password.

Table 5-1: User Attributes

User Attribute	Can Get	Can Set
Can Change Password	YES	NO
Check In On Exit	YES	YES
Check In Reminder	YES	YES
Confirm On Exit	YES	YES
Decimal Length	YES	NO
Decimal Separator	YES	NO
Email Address	YES	YES
Extension	YES	YES
Group Length	YES	NO
Group Symbol	YES	NO
Id	YES	NO
Is Account Enabled	YES	NO
Is DBA Enabled	YES	YES
Items Per Page	NO	YES
Links Filter	YES	YES
Login Name	YES	NO
Open on Login	YES	YES
Password	YES	NO
Preferred Class	YES	YES
Preferred Window	YES	YES
Preview Threshold	YES	YES
Save Settings on Exit	YES	YES
Search Case Sensitive	YES	NO

Table 5-1: User Attributes

Search Threshold	NO	YES
Telephone	YES	YES
User Name	YES	NO
View Dir	YES	YES
Work Dir	YES	YES

Table 5-2: Screen Permission Names

Add File Main	Add File Perms	Add File Assoc
Add File Attrs	Add Project Main	Add Project Perms
Add Project Assoc	Add Project Attrs	Alter File Main
Alter File Perms	Alter File Assoc	Alter File Attrs
Alter File Browse	Alter Obsolete Items	Alter Project Main
Alter Project Perms	Alter Project Assoc	Alter Project Attrs
Alter Released Items	Alter Released Links	Alter Revision
Approve	Baseline Editor	BOM Export
BOM Import	Cancel Process Instance	Checkin Main
Checkin Perms	Checkin Assoc	Checkin Attrs
Checkin Others	Check Out	Choice List Admin
Class Admin	Collaboration	Column Layout Admin
Customize Column Layouts	Database Synchronization	Delete File
Delete Process Instance	Delete Project	Disapprove
Form Editor	Get Copy	Group Admin
GUI Extension 0	GUI Extension 1	GUI Extension 2
GUI Extension 3	GUI Extension 4	GUI Extension 5
GUI Extension 6	GUI Extension 7	GUI Extension 8
GUI Extension 9	Hold	Link Type Editor
Mark as Obsolete	Move File	Move Project
Move Released Items	Print	Process Editor
Purge File	Purge Project	Reinstate
Rename	Replication Admin	Report
Report Editor	Re-Release	Return Unmodified
Return Unmodified Others	Revision Editor	Rollback File
Rollback Project	Save As	Send Back
Send Back Admin	Send Forward	Send Forward Admin
Show Workflows	Start Workflow	SQL Reports
Submit	User Admin	Vault Admin
View File Main	View File Perms	View File Assoc
View File Attrs	View Project Main	View Project Perms
View Project Assoc	View Project Attrs	View Workflow

Item Column Layout Object

Creating an item column layout object

ItemColLayout Create

Usage	<ItemColLayoutCreate *itemlayout*>
Arguments	*itemlayout* is the item column layout object
Return value	"0" for success and a non-zero error code on error.

Creates a new, empty item column layout object.

Constructing or Editing the layout

ItemColLayoutSetColumn

Usage	<ItemColLayoutSetColumn *itemlayout field* INDEX=*index* WIDTH=*width* ALIGN=*align* >
Arguments	*itemlayout* is the item column layout object *field* is the attribute object *index* is the column index *width* is the column width *align* is the column text alignment
Return value	"0" for success and a non-zero error code on error.

Sets the definition of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The call may replace the current definition of a column in the layout, or it may add one more column to a layout (by specifying an index equal to the current number of columns in the layout).

The data that should be displayed in the column is specified by a particular attribute from a particular derived form, identified by vid and uda_id taken from the field object. The display will match the prompt for the column to the prompt for attributes to identify the data. The prompt specified in the field object should match the vid and uda_id, although it will not in fact be saved as part of the layout definition. The column width is specified in grid-width units, and the data alignment in the column as "left", "center", or "right".

ItemColLayoutRemoveColumn

Usage	<ItemColLayoutRemoveColumn *itemlayout* index=*index*>
Arguments	*itemlayout* is the item column layout object *index* is the column index
Return value	"0" for success and a non-zero error code on error.

Removes the definition of one column from the layout. Sort column numbers will be adjusted if necessary so that the same column remains specified for sorting.

ItemColLayoutRemoveAllColumns

Usage	<ItemColLayoutRemoveAllColumns *itemlayout*>
Arguments	*itemlayout* is the item column layout object
Return value	"0" for success and a non-zero error code on error.

Removes the definitions of all columns from the layout, making it again an empty layout object.

ItemColLayoutSetSort

Usage	<ItemColLayoutSetSort *itemlayout* SORT_INDEX=*sort_index* INDEX=*col_index* DIR=*direction* >
Arguments	*itemlayout* is the item column layout object *sort_index* is the sort priority index *col_index* is the column index *direction* is the direction of sort
Return value	"0" for success and a non-zero error code on error.

Sets the selection of one column in the layout for sorting. The sort_index specifies the priority for sorting, and must be 0. The col_index specifies the position in the layout (0 = left-most customizable column). The direction of sorting is specified by direction as "asc" (i.e. ascending order from top to bottom of the display grid) or "desc".

Accessing the layout

ItemColLayoutColumnGetCount

Usage	<ItemColLayoutColumnGetCount *itemlayout*>
Arguments	*itemlayout* is the item column layout object
Return value	The number of columns in the layout.

Gets the number of columns in the layout.

ItemColLayoutColumnGetAttr

Usage	<ItemColLayoutColumnGetAttr *itemlayout* INDEX=*index* ATTR=*colattr*>
Arguments	*itemlayout* is the item column layout object *index* is the column index *colattr* is the attribute of the column layout, one of "formid", "fieldid", "prompt", "width", "align", "attrname", "attrtype", "sort"
Return value	Value of the specified attribute.

Gets an attribute of the definition of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The colattr specifies which attribute of the column definition is to be returned.

For colattr "attrname": if the attribute for the column is a common attribute, then the return value will be the attribute name (i.e. the name of the column in the CMS_DFM database table). If the attribute is a custom attribute, then the return value will be the empty string.

For colattr "sort": the return value will be "0" if the layout does not sort by this column, "1" if it sorts by it in ascending order, and "2" if it sorts by it in descending order.

ItemColLayoutGetId

Usage	<ItemColLayoutGetId *itemlayout*>
Arguments	*itemlayout* is the item column layout object
Return value	The numeric ID of the layout.

Gets the id number of the layout.

ItemColLayoutGetSort

Usage	<ItemColLayoutGetSort *itemlayout* SORT_INDEX=*sort_index* ATTR=*sortattr* >
Arguments	*itemlayout* is the item column layout object *sort_index* is the sort priority index *sortattr* is the attribute of the sort, one of "index", "dir"
Return value	Value of the specified attribute.

Gets the information about one column in the layout used for sorting. The sort_index specifies the priority for sorting, and must be 0.

For sortattr "index": the return value will be the index of the column used for sorting (0 = left-most customizable column). If there is no sorting at this priority index, the value will be "-1".

For sortattr "dir": The direction of sorting is specified by direction: "1" if sorted in ascending order, and "2" if sorted in descending order. If there is no sorting at this priority index, the value will be "0".

Layout storage functions

ItemColLayoutSave

Usage	<ItemColLayoutSave *itemlayout* ID=*id*>
Arguments	*itemlayout* is the item column layout object *id* is the layout id value, either a number or one of "COL_LAY_ID_NEW", "COL_LAY_ID_USER_DEFAULT", or "COL_LAY_ID_SITE_DEFAULT"
Return value	"0" for success and a non-zero error code on error.

Saves the layout definition in the database and identifies it with a unique id value that can be used to load it. If the layout had previously been saved, then saving it again with the same id value will replace the previous definition in the database with the new one. Otherwise the id should be set to one of the following named values:

COL_LAY_ID_NEW = assign a new, unique id value and use it to save this definition. The id value in the layout object will be updated to the value that is assigned.

COL_LAY_ID_USER_DEFAULT = save the definition as the default item column layout for the current user.

COL_LAY_ID_SITE_DEFAULT = save the definition as the default item column layout for the ProductCenter site.

ItemColLayoutLoad

Usage	<ItemColLayoutLoad *itemlayout* ID=*id*>
Arguments	*itemlayout* is the item column layout object *id* is the layout id value, either a number or one of "COL_LAY_ID_CURRENT_DEFAULT", "COL_LAY_ID_CURRENT_SITE_DEFAULT", "COL_LAY_ID_USER_DEFAULT", "COL_LAY_ID_SITE_DEFAULT", "COL_LAY_ID_SYS_DEFAULT"
Return value	"0" for success and a non-zero error code on error.

Loads the specified layout definition from the database. If the layout had previously been saved as a non-default layout, then specify the id value that was assigned to it. Otherwise the id should be one of the following named values:

COL_LAY_ID_CURRENT_DEFAULT = If the user has specified a default item column layout, then load it. Otherwise, if a global default has been specified, then load it. If neither has been specified, then load the standard ProductCenter default layout.

COL_LAY_ID_CURRENT_SITE_DEFAULT = If a global default has been specified, then load it. Otherwise, load the standard ProductCenter default layout.

COL_LAY_ID_USER_DEFAULT = Load the default item column layout specified by the user. If he has not specified one, then return an error code.

COL_LAY_ID_SITE_DEFAULT = Load the global default item column layout. If the administrator has not specified one, then return an error code.

COL_LAY_ID_SYS_DEFAULT = Load the standard ProductCenter default layout.

ItemColLayoutDelete

Usage	<ItemColLayoutDelete *itemlayout* ID=*id*>
Arguments	*itemlayout* is the item column layout object *id* is the layout id value, either a number or one of "COL_LAY_ID_USER_DEFAULT" or "COL_LAY_ID_SITE_DEFAULT"
Return value	"0" for success and a non-zero error code on error.

Deletes the specified layout definition from the database. (This does not delete, clear, or destroy the definition in the layout object in the client.) If the layout had previously been saved as a non-default layout, then specify the id value that was assigned to it. Otherwise the id should be one of the following named values:

COL_LAY_ID_USER_DEFAULT = Delete the definition of the default item column layout specified by the user, so that there will not be a user-specific default. If he has not specified one, then return an error code.

COL_LAY_ID_SITE_DEFAULT = Delete the definition of the global default item column layout, so that there will not be a global default. If the administrator has not specified one, then return an error code.

Chapter 6

Lists

Just Ahead:

List tags 78

List tags

WebLink's list tags create and manipulate lists of ProductCenter items: users, groups, choice lists, class lists, and so on. Each element in a list contains two types of records, a Name and an ID. The Name is a human-readable name that can be displayed or used in WebLink applications. The ID is an internal identifier and cannot be used if the object associated with it is not accessible through WebLink.

ListLoad

Usage	<ListLoad *listObject* TYPE = *type* ATTR = *attrName* VALUE = *value*>
Arguments	*listObject* is the list object that will be created. *type* is the list type. *attrName* is a list name appropriate for *type*. *value* provides further granularity into the content of the list returned.
Return value	"0" for success and a non-zero error code on error.
Example	<ListLoad list TYPE = "Group" ATTR = "Users" VALUE = "329">

Creates a list object and populates it with the specified values.

This example creates a list object that consists of users in the group that has a group ID of 329.

Table 6-1 shows the combinations of parameter type, attribute, and value that can be entered as arguments into the <ListLoad> tag.

Table 6-1: <ListLoad> argument combinations

To create a list of...	TYPE	ATTR	VALUE
Choice lists in ProductCenter	System	ChoiceLists	null string
Active users in ProductCenter. (Does NOT include disabled user accounts.)	System	Users	null string
All users in ProductCenter.	System	Allusers	null string
Groups in ProductCenter	System	Groups	null string
Classes in ProductCenter (with full pathname)	System	Classes	null string
Forms in ProductCenter	System	Forms	null string
Linktypes in ProductCenter	System	Linktypes	null string
Choices in a choice list	ChoiceList	Choices	choiceListId
Users in a group	Group	Users	groupId
Short names of subclasses in a Class	Class	shortname	classId

Table 6-1: <ListLoad> argument combinations

To create a list of...	TYPE	ATTR	VALUE
Full names of subclasses in a Class	Class	fullname	classId
Saved item queries for the logged-in user	System	Queries	item
Saved process instance queries for the logged-in user.	System	Queries	process
Saved activity queries for the logged-in user.	System	Queries	activity
Process definitions in ProductCenter	System	ProcessDefs	route or form
Items checked out by the logged-in user	User	Workspace	null string
Items in the logged-in user's Desktop	User	Desktop	null string
Items on the logged-in user's Desktop with some attributes.	User	DesktopInfo	TRUE or FALSE
Activities in the logged-in user's work list	User	WorkList	TRUE or FALSE
Items that use the current item as a child in hierarchical links	Item	WhereUsed	CMS ID
Report names and id that are related to a class.	ItemSpecificReport	null string	classId
Report names and id that are related to a process name.	ProcessSpecificReport	null string	process Definition Name
Item query report names and ids.	ItemByQuery Report	folder ID, or null string for root folder	null string
Process or activity query based report names and ids.	ProcessByQueryReport	folder ID, or null string for root folder	null string
Report names and id that are query reports.	AllByQueryReport	null string	null string

Table 6-1: <ListLoad> argument combinations

To create a list of...	TYPE	ATTR	VALUE
Searchable forms.	SearchableForms	null string	null string
Used file type.	UsedFileType	null string	null string

ListGetCount

Usage	<ListGetCount *listObject*>
Arguments	*listObject* is the list object.
Return value	the number of items in the list object.
Example	<setval error <ListLoad list TYPE = "System" ATTR = "Classes" VALUE = "">> <setval count <ListGetCount list>> This example loads a list object and then sets a variable equal to the number of items in the object.

Returns the number of items in a list object.

ListGetDisplayName

Usage	<ListGetDisplayName *listObject* INDEX = *index*>
Arguments	*listObject* is the list object. *index* is the index number of the item in the list.
Return value	the name of the item at the index you specify.
Example	<ListGetDisplayName list INDEX = "7"> This example gets the eighth item in the list (the first item has an index number of 0) and returns its name.

Returns the name of the item in the list object at the index specified.

The name is not necessarily a unique name in ProductCenter.

ListGetSystemId

Usage	<ListGetSystemId *listObject* INDEX = *index*>
Arguments	*listObject* is the list object. *index* is the index number of the item in the list.
Return value	the ID of the item at the index specified.
Example	<ListGetSystemId list INDEX = "3"> This example gets the fourth item in the list (the first item has an index number of 0) and returns its ID. The ID can then be used to load an object.

Returns the ID of the item in the list object at the index specified.

Since workflow activities may be assigned to multiple users the ID returned from an activity list will include the ID followed by a pipe (|) character. If the activity is assigned the pipe character will be followed by the User name of the asignee

The expanded example below shows how to use “ListGetCount” (see page 80) to obtain the number of items in a list and “ListGetDisplayName” (see page 80) and <ListGetSystemId>, together with <printval>, to print the names and IDs of all of the items in the list.

```
<setval error
   <ListLoad list TYPE = "Class" ATTR = "Shortname" VALUE = "2">>
<setval items <ListGetCount list>>
<setval i "0">
<while i lt items>
   <printval <ListGetDisplayName list INDEX = i>>
   <printval <ListGetSystemId list INDEX = i>>
   <setval i <expr i + "1">>
</while>
```

ListGetActivityInstance

Usage	<ListGetActivityInstance *listObject activityInst* INDEX = *index*>
Arguments	*listObject* is the list object. *activityInst* is the returned activity instance object. *index* is the index number of the activity in the list.
Return value	"0" for success and a non-zero error code on error.
Example	<ListGetActivityInstance myList myActivity INDEX = listPtr>

Returns an activity instance from an activity instance list.

ListGetDesktopDisplayAttr

Usage	<ListGetDesktopDisplayAttr *desktopListObject* ATTR = *attrName* INDEX = *index*>
Arguments	*desktopListObject* is a list object generated by “ListLoad” containing a list of item objects on the user’s desktop. *attrName* is the name of the attribute of the item object at position *index* in *desktopListObject.* *index* is the position in *desktopListObject.*
Return value	the attribute value.
Example	<ListGetDesktopDisplayAttr myList attr = "Revision" index = listPtr>

6

Returns one of the following attribute value of an item object in a list object of desktop items.

Class	Is Checked Out	Last User	Title
CMS ID	Is File	Name	Version
File Type	Is Latest	Revision	
Is BOM	Is Part	Status	

The range of values for *index* can be determined by using the "ListGetCount" (see page 80) tag.

Chapter 7

Items

Just Ahead:

Creating an item object 84
Setting values 85
Getting values 87
Getting revision values 90
Desktop tags 92
Inbound tags 92
Outbound tags 93
Manipulation tags 95
Release management tags 96
Identification tags 97
Examples. 98

Creating an item object

ItemCreate

Usage	<ItemCreate *itemObject* CLASS = *className*>
Arguments	*itemObject* is the item object. *className* is the class to which the new item belongs.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemCreate item CLASS = "CMS:Files">

Creates an item by specifying the name of the item object and the class to which the item will belong.

Note that class names are case-sensitive.

ItemLoad

Usage	<ItemLoad *itemObject* ID = *id*>
Arguments	*itemObject* is the item object. *id* is set equal to the “CMS ID” of the item.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemLoad item ID = "1234">

Loads an existing item.

The arguments of the <ItemLoad> tag are the name of the item object and its “CMS ID” — the unique identification number for every object in ProductCenter.

ItemGetLatest

Usage	<ItemGetLatest *itemObject latestItemObject*>
Arguments	*itemObject* is an item object of which the latest version is being requested. *latestItemObject* is the returned item object which is the current, latest version of *itemObject*.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemGetLatest item itemNewest>

Loads the latest version of an item object into a new item object.

Setting values

ItemDeleteTableTypeAttrRow

Usage	<ItemDeleteTableTypeAttrRow *itemObject* ATTR = *attrName*>>
Arguments	*itemObject* is the item object. *attrName* is the name of the attribute in addition to the row to delete in aquare brackets.
Return value	returns "0" for success and a non-zero error code on error.
Example	<ItemDeleteTableTypeAttrRow item ATTR = "Material[2]"> <ItemDeleteTableTypeAttrRow item ATTR = "Material[2].density"> • Both of these examples delete row 3 of the “Material” table-type attribute.

Removes a row from a table-type attribute.

The argument *attrName* has the form “table[row]” or “table[row].column”, which deletes the row for the attribute table.

ItemSetAccessPerm

Usage	<ItemSetAccessPerm *itemObject* TYPE = *type* NAME = *permName* VALUE = *permVal*>
Arguments	*itemObject* is the item object. *type* is "USER" or "GROUP". *permName* is the name of the access permission to set. *permVal* is the access permission value to set.
Return value	"0" on success and a non-zero error code on failure.
Example	<ItemSetAccessPerm item TYPE=”GROUP” NAME=”ADMIN” VALUE = “RW”> <ItemSetAccessPerm item TYPE="USER" NAME = "Alan Smith" VALUE = "RWD" >

Sets a user or group access permission for an item.

The *permVal* value can contain one or more of the following characters:

- “R” for read access
- “W” for write access
- ”D” for delete access
- “N” for notify access

ItemSetAttr

Usage	<ItemSetAttr *itemObject* ATTR = *attrName* VALUE = *value*>
Arguments	*itemObject* is the item object. *attrName* is the attribute name. *value* is the value to which you want to set the attribute.
Return value	"0" on success and a non-zero error code on failure.
Example	ItemSetAttr myItem ATTR = "Comments" VALUE = "my comment">

Sets the value of an attribute.

The rules for *attrName* are the same as those described for "ItemGetAttr" (see page 88).

After updating an attribute's value, the itemObject must be saved using the appropriate function: "ItemAlter" (see page 96) to update data on an item that is not checked out, "ItemCheckin" (see page 93) for an item that has been checked out, or "ItemAdd" (see page 92) for a new item.

ItemSetFile

Usage	<ItemSetFile *itemObject* NAME = *filename*>
Arguments	*itemObject* is the item object. *filename* is the name you are assigning to the file.
Return value	returns "0" for success and a non-zero error code on error.
Example	<ItemSetFile item NAME = "st_rept">

Sets the name of a file to an item object.

When ProductCenter performs inbound operations on files, the file name and the item name may be different. Therefore, the name of the file must be set explicitly before any inbound operations can be performed. An error will be returned if the file name is not set before an inbound operation.

IItemSetTransactionLog

Usage	<ItemSetTransactionLog *itemObject* VALUE = *transactionText*>
Arguments	*itemObject* is the item object. *transactionText* is the text to add to the transaction log for *itemObject.*
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemSetTransactionLog myItem VALUE = "Updated via ml program">

Adds a transaction log entry for an item upon an item inbound or item manipulation operation. The logging will only take place if logging is specified in the cms_site file

Getting values

ItemGetAccessCount

Usage	<ItemGetAccessCount *itemObject* TYPE = *permType*>
Arguments	*itemObject* is the item object. *permType* is "USER" or "GROUP".
Return value	the number of access permissions.
Example	<SetVal permCnt <ItemGetAccessCount thisItem TYPE=permType>>

Returns the number of access permissions for a user or group.

ItemGetAccessNameByIndex

Usage	<ItemGetAccessNameByIndex *itemObject* TYPE = *permType* INDEX = *index*>
Arguments	*itemObject* is the item object. *permType* is "USER" or "GROUP". *index* is a value from 0 to the value returned by "ItemGetAccessCount".
Return value	the access permission name.
Example	<ItemGetAccessNameByIndex thisItem TYPE=*permType*" INDEX=*permIndex* >

Returns a user or group access name..

ItemGetAccessPermByIndex

Usage	<ItemGetAccessPermByIndex *itemObject* TYPE=*permType* INDEX = *index*>
Arguments	*itemObject* is the item object. *permType* is "USER" or "GROUP". *index* is a value from 0 to the value returned by "ItemGetAccessCount".
Return value	the access permission value.
Example	<ItemGetAccessPermByIndex thisItem TYPE=*permType* INDEX=*index*>

Returns a user or group access permission value.

The return value will contain one or more of the following characters:

- "R" for read access
- "W" for write access
- "D" for delete access
- "N" for notify access

ItemGetAttr

Usage	<ItemGetAttr *itemObject* ATTR = *attrName*>
Arguments	*itemObject* is the item object. *attrName* is the name of the attribute you want to obtain.
Return value	the value of the attribute for the item given by attrname.
Example	<ItemGetAttr item ATTR = "Date"> <setval title <ItemGetAttr item ATTR = "Title">>

Obtains the value of an attribute for an item.

The ProductCenter Form Editor allows an administrator to assign a prompt and a name to every item (and link) attribute. Prompt names for an item do not need to be unique, but attribute names in a form do need to be unique. Therefore, when identifying an attribute in a function such as “ItemGetAttr” (see page 88) or “ItemSetAttr” (see page 86), Weblink will search in this order for the attribute specified:

1. Common attributes column name.
2. Custom attributes column name.
3. Common attributes prompt name.
4. Custom attribute prompt name.

To explicity state whether an attribute is specified by name or prompt, “CUSTOM:*attrName*” instructs WebLink to search for the attribute by its prompt, while “CUSTOMCOL:*attrName*” instructs WebLink to search for the attribute by its name. Note that “CUSTOM” and “CUSTOMCOL” are case-sensitive tags.

For table-type attributes, the format is this:

```
<ItemGetAttr item ATTR = "CUSTOM:attrname[row#].columnname">
```

For example,

```
<ItemGetAttr item ATTR = "CUSTOM:PARTS[1].QUANTITY">
```

would retrieve the QUANTITY value in the second row of the PARTS attribute. (For the first row, the row number would be “[0]”.) To retrieve all of the rows in a table-type attribute, a “While” (see page 45) loop along with the “Expr” (see page 43) function can be used.

ItemGetEmbeddedObjectURL

Usage	<ItemGetEmbeddedObjectURL *itemObject* VALUE = *msgStr*>
Arguments	*itemObject* is the item object. *msgStr* is the embedded URL string.
Return value	the resolved string of an embedded URL.
Example	If an item’s form contains a URL field named “urlField” with the following definition src=pctr://getlinkedfile?linktype=child:tail and the item object has a child link to a file named “PrdCtr Install 8.6.0.pdf” then <ItemGetEmbeddedObjectURL itemObject VALUE = <itemGetAttr itemObject ATTR="urlField">> would yield this result src=pctr://webgetfile?PrdCtr Install 8.6.0.pdf

Returns the resolved string of an embedded URL; it will convert the URL from a dynamic string to specific item.

ItemGetRevisionFlag

Usage	<ItemGetRevisionFlag *itemObject* TYPE = *type*>
Arguments	*itemObject* is the item object. *type* is "Override".
Return value	“1” if the revision field is editable, “0” if not.
Example	<ItemGetRevisionFlag *thisItem* TYPE="Override"

Determines if an item’s revision attribute is editable.

ItemGetRowCount

Usage	<ItemGetRowCount *itemObject* ATTR = *attrName*>
Arguments	*itemObject* is the item object. *attrName* is name of the table-type attribute.
Return value	the number of rows of data in the specified table-type attribute. If the attribute specified is not table-type, the function returns “1”.
Example	<ItemGetRowCount item ATTR = "Material">

Returns the number of rows of data in the specified table-type attribute.

ItemHasVersionConflicts

Usage	<ItemHasVersionConflicts *itemObject* versionList>
Arguments	*itemObject* is the item object.
Return value	“1” if *itemObject* does have version conflicts, “0” if not.
Example	<ItemHasVersionConflicts item versionlist>

Returns a list of items that show multiple occurrences of the same item at different version or revision levels.

ItemLoadForm

Usage	<ItemLoadForm *itemObject formObject*>
Arguments	*itemObject* is the item object. *formObject* is a variable to hold the form object.
Return value	"0" for success and non-zero error code on failure.
Example	<ITEMLOADFORM *thisItem thisForm*>

Loads the form object associated with the specified item object.

After the form object is loaded, then tags in Chapter 12 can be used to query the item form.

Getting revision values

ItemGetNextRevision

Usage	<ItemGetNextRevision *itemObject*>
Arguments	*itemObject* is the item object.
Return value	the next revision for *itemObject*.
Example	<SetVal rev <ItemGetNextRevision thisItem>>

Returns the next revision defined in the revision sequence.

ItemGetNextRevisionCount

Usage	<ItemGetNextRevisionCount *itemObject* ACTION = *action*>
Arguments	*itemObject* is the item object. *action* is either "ADD", "CHECKIN" or "ALTER".
Return value	the number of next revisions for *itemObject*.
Example	<ItemGetNextRevisionCount thisItem ACTION = "CHECKIN">

Returns the number of the next revisions. This count will include all valid next revisions, including minor and legacy revisions.

ItemGetNextRev

Usage	<ItemGetNextRev *itemObject* INDEX = *index* ACTION = *action*>
Arguments	*itemObject* is the item object. *index* is the index of revisions available for *itemObject*. *action* is either "ADD", "CHECKIN" or "ALTER".
Return value	the revision sequence value at the index value for *itemObject*.
Example	<setVal nextRev <ItemGetNextRev thisItem INDEX=*index* ACTION=*action*>>

Returns the revision sequence based on the index.

ItemGetNextRevisionIsLegacy

Usage	<ItemGetNextRevisionIsLegacy *itemObject* INDEX = *index* ACTION = *action*>
Arguments	*itemObject* is the item object. *index* is the index of revisions available for *itemObject*. *action* is either "ADD", "CHECKIN" or "ALTER".
Return value	"1" if the revision is a legacy revision and "0" if not.
Example	<ItemGetNextRevisionIsLegacy thisItem INDEX=ptr ACTION="CHECKIN">

Returns a flag indicating if the revision has been defined as a legacy revision.

ItemGetNextRevisionIsMinor

Usage	<ItemGetNextRevisionIsMinor *itemObject* INDEX = *index* ACTION = *action*>
Arguments	*itemObject* is the item object. *index* is the index of revisions available for *itemObject*. *action* is either "ADD", "CHECKIN" or "ALTER".
Return value	"1" if the revision is a minor revision and "0" if not.
Example	<ItemGetNextRevisionIsMinor thisItem INDEX=ptr ACTION="CHECKIN">

Returns a flag indicating if the revision has been defined as a minor revision.

ItemGetNextRevisionGetParent

Usage	<ItemGetNextRevisionGetParent *itemObject* INDEX = *index* ACTION = *action*>
Arguments	*itemObject* is the item object. *index* is the index of revisions available for *itemObject*. *action* is either "ADD", "CHECKIN" or "ALTER".
Return value	the parent’s name for the minor revision.
Example	<ItemGetNextRevisionGetParent thisItem INDEX=ptr ACTION="CHECKIN">

Returns the parent's name for the minor revision.

Desktop tags

Note that the desktop state is only saved to the database on logout. If the logged in user has any other client also logged in at the same time, that client's desktop will not reflect any changes made from the Weblink program (or any other client). If the user then exits the other client, that client's desktop state will be saved to the database, overwriting any changes made by the Weblink program.

ItemAddToDesktop

Usage	<ItemAddToDesktop *itemObject*>
Arguments	*itemObject* is the item to be added.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemAddToDesktop thisItem>

Adds an item to the connected user's desktop.

ItemDeleteFromDesktop

Usage	<ItemDeleteFromDesktop *itemObject*>
Arguments	*itemObject* is the item to be deleted.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemDeleteFromDesktop thisItem>

Deletes an item from the connected user's desktop by specifying the item object.

ItemDeleteFromDesktopById

Usage	<ItemDeleteFromDesktopById ID = *id*>
Arguments	*id* is the ID of the item to remove.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemDeleteFromDesktopById ID = thisID>

Deletes an item from the connected user's desktop by specifying the ID of the item.

Inbound tags

ItemAdd

Usage	<ItemAdd *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and a non-zero error code on failure.

Adds an item.

Use <ItemAdd> after you have created an item and set attributes for that item. The item that you add to the ProductCenter database can be either a file or a project.

ItemAddSaveAs

Usage	<ItemAddSaveAs *itemObject origItemObject*>
Arguments	*itemObject* is the returned new item object. *origItemObject* is the original item object.
Return value	"0" for success and a non-zero error code on failure.

Performs a "Save As" operation from an item.

ItemCheckin

Usage	<ItemCheckin *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for successful checkin and a non-zero error code on failure.

Checks in an item.

The item must have been previously checked out by means of "ItemCheckout" (see page 93).

Before calling "ItemCheckin", the item must be loaded using tags such as "ItemLoad" (see page 84). Once loaded, the item's metadata or link relationships can be modified using tags described in this chapter. For a file item, "ItemSetFile" (see page 86) must be called to set the file name. After chnages to the item have been completed, then "ItemCheckin" (see page 93) can be called to check the updated item back in.

ItemUndoCheckout

Usage	<ItemUndoCheckout *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemUndoCheckout item>

Returns a checked-out item without storing any changes or creating a new version.

Outbound tags

ItemCheckout

Usage	<ItemCheckout *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and non-zero error code on failure.

Checks out an item.

Note that a user cannot checkout a file that has the same name as a file that has already been checked out by that user. This will return an error.

If the item being checked out is a file and the WML script in which this tag appears is not using the Weblink plugin,

- <ItemCheckout> must appear before the <HTML> tag. Otherwise, the wrong cookie or file type information may be sent to the browser.
- When the tag is encountered, the file is immediately sent to the user's browser. The rest of the script will be processed, but no output can be sent to the browser. However, if the tag returns an error, WebLink will execute the WML statements that follow the tag and send corresponding output to the browser.

If the item being checked out is a file and the Weblink plugin is activated in the script in which this tag appears, <ItemCheckout> will deliver the file to the user's working directory.

If the item being checked out is not a file, then the item is simply flagged as checked out in ProductCenter.

ItemCheckoutNoDeliver

Usage	<ItemCheckoutNoDeliver *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and non-zero error code on failure.

Checks out an item but does not deliver the file to the browser when the tag is processed.

ItemGetCopy

Usage	<ItemGetCopy *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and non-zero error code on failure.

Gets a copy of a file from a vault.

If the WML script in which this tag appears is not using the Weblink plugin:

- <ItemGetCopy> must appear before the <HTML> tag. Otherwise, the wrong cookie or file type information may be sent to the browser.
- When the tag is encountered, the file is immediately sent to the user's browser. The rest of the script will be processed, but no output can be sent to the browser. However, if the tag returns an error, WebLink will execute the WML statements that follow the tag and send corresponding output to the browser.

If a WML script in which this tag appears is using the Weblink plugin, then <ItemGetCopy> will deliver the file to the user's working directory.

Manipulation tags

ItemDelete

Usage	<ItemDelete *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and a non-zero error code on failure.

Deletes an item from the database.

ItemPurge

Usage	<ItemPurge *itemObject* VERSION = *verString*>
Arguments	*itemObject* is the item object. *verString* is a string that specifies the version or versions to be purged. For selective purge operations, *verString* can be a list of versions separated with commas, or a range separated with a dash.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemPurge thisItem VERSION = "1-3,4" >

Purges the specified version(s) of an item.

ItemPurgeLatest

Usage	<ItemPurgeLatest *itemObject* VERSION ="">
Arguments	*itemObject* is the item object.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemPurgeLatest thisItem VERSION="">

Purges the latest version of an item.

ItemCheckVersionsForRemoval

Usage	<ItemCheckVersionsForRemoval *itemObject outVersionListObject* Version=*inVersionListString*>
Arguments	*itemObject* is the item object. *outVersionListObject* is the return list containing the versions checked and corresponding warning messages. *inVersionListString* is the list of versions to check of *itemObject*.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemCheckVersionsForRemoval thisItem myVersionList Version="1-3,5">

Returns a list of revisions and warning messages in *outVersionListObject* to be used when deleting or purging items.

The *inVersionListString* is a list of versions to be checked. If it is NULL or empty it will check all versions up to the item versions. An example of the *inVersionListString* is “1-3,5,7,9-15”. There are three types of the warnings: the version has an attached workflow; the versions is not ‘In Progress’; removing the version would produce an invalid Release Management Configuration.

ItemAlter

Usage	<ItemAlter *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" for success and a non-zero error code on error.

Updates attribute information or replace a file for an existing item. Use the following sequence:

1. Load the item using a tag such as “ItemLoad” (see page 84).
2. Make changes to the item’s metadata with “ItemSetAttr” (see page 86)or a link-related tag such as “ItemAddLink” (see page 107). Or use “ItemSetFile” (see page 86) to replace a file associated with an item.
3. Commit your changes to the database with <ItemAlter>.

ItemRollback

Usage	<ItemRollback *itemObject* VERSION = *verString*>
Arguments	*itemObject* is the item object. *verString* specifies the version of the item to which to roll back. If *verString* is empty, then the item will roll back to the last version of the item.
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemRollback thisItem VERSION = "2">

Rolls back a previous version of an item to make it the latest version.

Release management tags

ItemPromote

Usage	<ItemPromote *itemObject* STAGE = *newStage* PROMOTECONFIG = *config*>
Arguments	*itemObject* is the item object. *newStage* is the state to promote the item: either “In Approval” or “Released”. *config* : either “YES” or “NO”
Return value	"0" for success and a non-zero error code on failure.
Example	<ItemPromote thisItem STAGE = "RELEASED" PROMOTECONFIG = "NO">

Promotes an item, and the configuration it heads, to a specified stage.

The argument *config* is used for negative testing and indicates whether the items linked to *itemObject* by release managed links are promoted along with *itemObject*.

Under ProductCenter's Release Management rules, an item cannot be promoted unless its release-managed linked items have been promoted as well. If *config* is equal to "NO" and WebLink does not return an error message, then that indicates that the release-managed linked items already have been promoted. If the tag does return an error message, then the release-managed linked items have not been promoted. These linked items must be promoted before the parent item can be promoted.

ItemDemote

Usage	<ItemDemote *itemObject*>
Arguments	*itemObject* is the item to demote.
Return value	"0" for success and a non-zero error code on failure.

Demotes an item from "In Approval" to "In Progress", but not those linked to it by releaase-managed links, to the previous stage.

ItemObsolete

Usage	<ItemObsolete *itemObject*>
Arguments	*itemObject* is the item to mark as obsolete.
Return value	"0" for success and a non-zero error code on failure.

Marks an item as Obsolete.

ItemReinstate

Usage	<ItemReinstate *itemObject*>
Arguments	*itemObject* is the item to reinstate from the Obsolete state.
Return value	"0" for success and a non-zero error code on failure.

Reverses the effect of marking an item as Obsolete.

Identification tags

ItemIsType

Usage	<ItemIsType *itemObject* TYPE = *itemType*>
Arguments	*itemObject* is the item object. *itemType* can be either "FILE" or "PART".
Return value	"0" if the item is not of the type specified by the TYPE argument, and returns "1" if it is. For example, setting TYPE="PART" where the item is a file, ItemIsType will return a "0".

Determines whether an item is a file or a part. If both types return 0 then the type is a Project.

ItemIsLatest

Usage	<ItemIsLatest *itemObject*>
Arguments	*itemObject* is the item object.
Return value	"0" if the item is not the latest version and "1" if it is.

Determines whether an item is the latest version.

ItemGetCheckoutStatus

Usage	<ItemGetCheckoutStatus *itemObject*>
Arguments	*itemObject* is the item that will be queried.
Return value	"0" if the item is not checked out; “1" if the user has checked out the item; “2" if someone else has checked out the item.

Determines whether an item is checked out.

ItemIsFile

Usage	<ItemIsFile *itemObject*>
Arguments	*itemObject* is the item that will be queried.
Return value	"1" if the item is a file and "0" if not.

Determines whether an item is a file.

ItemIsObsolete

Usage	<ItemIsObsolete *itemObject*>
Arguments	*itemObject* is the item which will be queried.
Return value	"1" for obsolete s and “0” for not obsolete.

Determines whether the item is in the Obsolete state.

Examples

The following brief examples give you an idea of how to perform basic item operations with WebLink. The first example includes complete files, subsequent examples snippets containing just the needed wml code. The first example assumes the user has already logged in to the weblink client.

Example 1: Add an item

To add a new project into ProductCenter:

- Use the HTML <FORM> tag and related tags to create a form in which the user selects the file he wants to add to ProductCenter, along with any attributes that need to be set.
- Use "ItemCreate" (see page 84) to create a new item in memory.
- Use "ItemSetAttr" (see page 86) once for each item attribute you want to set.
- Use "ItemAdd" (see page 92) to add the file from memory into the ProductCenter database.

The html file would be located in the htdocs directory or a subdirectory of htdocs. The wml file is located in a subdirectory of the cgi-bin directory (one directory below weblink.cgi. The example assumes that there are no required custom attributes. The example wml displays the ID of the added item, that item id could be passed on to another web page by using it to set a form attribute in the wml file and submitting the form.

addItemExample.html:

```
<html>
  <body>
    <FORM name="add_form" method="POST" action="/cgi-
   bin/weblink.cgi/custom/addItemExample.wml">
      <CENTER>Add Item</CENTER>
      <CENTER>Name <INPUT type="text" name="itemName"></CENTER>
      <CENTER>Title<INPUT type="text" name="itemTitle"></CENTER>
     <CENTER><B><FONT SIZE=+2><INPUT type="submit" name="submit"
   value="SUBMIT"></B></FONT></CENTER>
    </FORM>
  </body>
</html>
```

7

addItemExample.wml:

```
<html>
  <defval itemName "Name Not Passed">
  <defval itemTitle "Title Not Passed">
  <ItemCreate item CLASS = "CMS:Projects">
  <ItemSetAttr item ATTR = "Class" VALUE = "CMS:Class">
  <ItemSetAttr item ATTR = "Name" VALUE = itemName>
  <ItemSetAttr item ATTR = "Title" VALUE = itemTitle>
  <ItemSetAttr item ATTR = "Preparer" VALUE = "George
   Washington">
  <ItemSetAttr item ATTR = "Prepared on" VALUE = "JAN-03-2010">
  <setval error <ItemAdd item>>
  <if error GT "0">
    Error:<PrintVal <ErrorGetLastMessage>> <br>
  <else>
    <setval id <ItemGetAttr item ATTR = "Cms Id">>
    New Item ID: <PrintVal id> <br>
  </if>
</html>
```

For an example that adds a file see file.html, addfile.wml, and file.wml in the weblink/simple directory

NOTE: When you add files in interpreter mode, the file specified by <ItemSetFile> must be in the WebLink working directory. You find this directory by looking at the working_dir variable in the weblink.cfg file.

Example 2: Alter an item

To alter an item in ProductCenter:

- Use “ItemLoad” (see page 84) to load into memory an item stored in ProductCenter.
- Use “ItemSetAttr” (see page 86) once for every attribute you wish to alter.
- Use “ItemAlter” (see page 96) to store your changes in ProductCenter.

The sample code below alters the title of a previously saved item. Assume that the ID was passed from a previous page.

```
<ItemLoad item ID = id>
<ItemSetAttr item ATTR = "Title" VALUE = "New Title">
<setval error <ItemAlter item>>
```

Example 3: Check in an item

To check in an item using WebLink:

- Use "ItemLoad" (see page 84) to load into memory an item stored in ProductCenter.
- Use "ItemSetAttr" (see page 86) once for every attribute you wish to change.
- Use "ItemSetFile" (see page 86) if the item you are checking back in is a file instead of a project.
- Use "ItemCheckin" (see page 93) to store your changes in ProductCenter.

The sample code below illustrates the use of <ItemCheckin>.

```
<ItemLoad item ID = id>
<ItemSetAttr item ATTR = "Title" Value = "New Title">
<ItemSetFile item NAME = file>
<setval error <ItemCheckin item>>
```

Example 4: Check out an item

To check out an item using WebLink:

- Use "ItemLoad" (see page 84) to load into memory an item stored in ProductCenter.
- Use "ItemCheckout" (see page 93) to check out the item. If the item is a file, then the file is sent automatically to the Web browser. This means that a Web page cannot be sent when an item is checked out.

The sample code below illustrates the use of <ItemCheckout>. Assume the ID was passed from a previous page.

```
<ItemLoad item ID = id>
<setval error <ItemCheckout item>>
```

Example 5: Undo checkout of an item

To cancel the checking out of an item:

- Use "ItemLoad" (see page 84) to load into memory an item stored in ProductCenter.
- Use "ItemUndoCheckout" (see page 93) to cancel the checkout of an item.

The sample code below cancels the checkout of the item. Assume the ID was passed from a previous page.

```
<ItemLoad item ID = id>
<setval error <ItemUndoCheckout item>>
```

Example 6: Get a copy of a file

To extract a file out of the ProductCenter database:

- Use “ItemLoad” (see page 84) to load an item into memory.
- Use “ItemGetCopy” (see page 94) to get the file out of ProductCenter. You can use <ItemGetCopy> only with files, not projects. The file is sent automatically to the Web browser.

The sample code below gets a copy of a file stored in ProductCenter. Assume that the ID was passed from a previous page.

```
<ItemLoad item ID = id>
<setval error < ItemGetCopy item>>
```

Chapter 8

Links

Just Ahead:

Creating a link 104
Link object tags 104
Link attribute tags 105
Item object link tags 107

Creating a link

LinkCreate

Usage	<LinkCreate *linkObject* TYPE = *linkType*>
Arguments	*linkObject* is the returned link object.. *linkType* is the type of link to create. This value is case-sensitive, and must exactly match the assigned name of the link type.
Return value	"0" for success and a non-zero error code on error.
Example	<LinkCreate thisLink TYPE = "BomItem">

Creates a link of a specific link type.

Link object tags

LinkGetForm

Usage	<LINKGETFORM *linkObject* TYPE = *type*>
Arguments	*linkObject* is the link object. *type* is a either “VIEW” or “EDIT”.
Return value	"0" for success and non-zero error code on error.
Example	<LinkGetForm thisLink TYPE = "EDIT">

Generates HTML code for displaying or editing of a link.

The tag must appear between <TABLE> and </TABLE> tags in a WML page. If *type* equals “EDIT”, then <ItemGetForm> must appear between the <FORM> and </FORM> tags in addition to the <TABLE> tags. The tag “LinkSetForm” can be used as the action value to the FORM definition.

LinkGetHead

Usage	<LinkGetHead *linkObject itemObject*>
Arguments	*linkObject* is the link object. *itemObject* is the variable where the head of the link will be stored.
Return value	"0" for success and a non-zero error code on error.
Example	<LinkGetHead thisLink thisHeadItem>

Loads the item for the head of a link.

LinkGetTail

Usage	<LinkGetTail *linkObject itemObject*>
Arguments	*linkObject* is the link object. *itemObject* is the variable where the tail of the link will be stored.
Return value	"0" for success and a non-zero error code on error.
Example	<LinkGetTail thisLink thisTailItem>

Loads the item for the tail of a link.

LinkLoadForm

Usage	<LinkLoadForm *linkObject formObject*>
Arguments	*linkObject* is the link object. *formObject* is the variable where the form object will be stored.
Return value	"0" for success and non-zero error code on error.
Example	<LinkLoadForm thisLink thisLinkForm>

Loads the form object for a link.

Note that not all links have a form.

LinkSetForm

Usage	<LinkSetForm *linkObject*>
Arguments	*linkObject* is the link object whose form attributes will be saved.
Return value	"0" for success and a non-zero error code on failure.
Example	<LinkSetForm thisLink>

Saves a link's form attributes.

This tag can be used as the action command for the <FORM> tag which contains the HTML generated by the tag "LinkGetForm".

Link attribute tags

NOTE: After using any tag that modifies link attributes, an inbound operation on the item related to the link must be executed. See "Inbound tags" on page 92 for item inbound operations.

LinkDeleteTableTypeAttrRow

Usage	<LinkDeleteTableTypeAttrRow *linkObject* ATTR = *attrName*>
Arguments	*linkObject* is the link object. *attrName* is the table-type link attribute to modify.
Return value	"0" for success and a non-zero error code on error.
Example	<LinkDeleteTableTypeAttrRow thisLink ATTR = "Finish[2]">

Deletes a row in a tabletype link attribute.

The row is identified using the syntax of the tabletype attribute, for example attr_name[x], where x is the row to be deleted. Note that the row number is zero based.

LinkGetAttr

Usage	<LinkGetAttr *linkObject* ATTR = *attrName*>
Arguments	*linkObject* is the link object. *attrName* is the attribute name.
Return value	the link attribute value.
Example	<LinkGetAttr link ATTR = "Ref Des">

Returns a link attribute value.

The common attributes for links are:

- “VERSION”
- “TYPE”
- “CMS ID”
- “VERSION MODE” (The four valid version modes are “PCCLINK_NONE”, “PCCLINK_METADATA_VERSION”, “PCCLINK_FILEDATA_VERSION”, and “PCCLINK_ITEM_REVISION”.)
- “IS HIERARCHICAL“
- “IS LATEST”

LinkGetRowCount

Usage	<LinkGetRowCount *linkObject* ATTR = *attrName*>
Arguments	*linkObject* is the link object. *attrName* is the table-type attribute.
Return value	the number of rows. If *attrName* is a single-valued attribute, the tag returns "0".
Example	<LinkGetRowCount thisLink ATTR = "Finish">

Returns the number of rows in a table-type link attribute.

LinkSetAttr

Usage	<LinkSetAttr *linkObject* ATTR = *attrName* VALUE = *attrValue*>
Arguments	*linkObject* is the link object. *attrName* is the link attribute to set. *attrValue* is the attribute value.
Return value	"0" for success and a non-zero error code on error.
Example	LinkSetAttr thisLink ATTR = "Material" VALUE = "CRS">

Sets the value of a link attribute.

Item object link tags

Tags for adding, updating or removing links

NOTE: After using any tag that modify an item's links, an inbound operation on the item being affected must be executed. See "Inbound tags" on page 92 for item inbound operations.

ItemAddLink

Usage	<ItemAddLink *linkObject headItem tailItem*>
Arguments	*linkObject* is the link object. *headItem* and *tailItem* are the two items whose relationship will be established.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemAddLink thisLink thisHeadItem thisTailItem>

Creates a link between two items.

ItemAddHeadLink

Usage	<ItemAddHeadLink *headItemObject linkObject tailItemObject*>
Arguments	*headItemObject* is the head item that will have a tail item linked to it. *linkObject* is the link to add *tailItemObject* is the tail item on the link
Return value	"0" for success and a non-zero error code on error.
Example	<ItemAddLink thisItem thisLink thisTailItem>

Adds a tail item on a link to a head item.

ItemAddTailLink

Usage	<ItemAddTailLink *tailItemObject linkObject headItemObject*>
Arguments	*tailItemObject* is the tail item that will have a head item linked to it. *linkObject* is the link to add. *headItemObject* is the head item to link.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemAddLink thisItem thisLink thisHeadItem>

Adds a head item on a link to a tail item.

ItemAddLinkHeadToTail

Usage	<ItemAddLinkHeadToTail *linkObject headItemObject tailItemObject*>
Arguments	*linkObject* is the link to use between *headItemObject* and *tailItemObject*. *headItemObject* is the item that will have *tailItemObject* linked to it. *tailItemObject* is the tail item object.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemAddLinkHeadToTail thisLink headItem thisTailItem>

Adds a link from a head item to a tail item.

ItemAddLinkTailToHead

Usage	<ItemAddLinkTailToHead *linkObject headItemObject tailItemObject*>
Arguments	*linkObject* is the link to use between *headItemObject* and *tailItemObject*. *headItemObject* is the head item object. *tailItemObject* is is the item that will have *headItemObject* linked to it.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemAddLinkTailToHead thisLink headItem thisTailItem>

Adds a link from a tail item to a head item.

ItemRemoveLinkById

Usage	<ItemRemoveLinkById *headItem tailItem* ID = *id*>
Arguments	*headItem* is the input item to which the link belongs (the head item). *tailItem* is the tail item. *id* is the ID of the link as found in CMS_LINKS.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemRemoveLinkById headItem tailItem ID = thisID>

Deletes a link by ID.

If “0” is specified for *id*, this call removes the first link that matches the specified head and tail. (If there are more than one link between these two items, this removes only the first one encountered.) If a non-zero value is specified for *id*, this call removes the link with that ID (and ignores any specified head or tail item argument).

ItemUpdateLinkByID

Usage	<ItemUpdateLink *headItem tailItem linkObject* ID = *id*>
Arguments	*headItem* is the head item. *tailItem* is the tail item. *linkObject* is the link that contains the new properties. *id* is the ID of the item link whose properties will be replaced.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemUpdateLinkByID thisHeadItem thisTailItem thisLink ID = "0">

Changes the attributes of a link object.

If “0” is specified “0” for *id*, this call updates the first link that matches the specified *headItem* and *tailItem*. (If there are more than one link between these two items, this updates only the first one encountered.) If a non-zero value is specified for *id*, this call updates the link with that ID (and ignores any specified head or tail item argument).

Tags for getting linked item information

ItemGetHeadLinkCount

Usage	<ItemGetHeadLinkCount *itemObject* TYPE = *linkTypeName*>
Arguments	*itemObject* is the tail item object. *linkTypeName* is the link name.
Return value	the number of *linkType* links to *itemObject* where *itemObject* is the tail of the link.

Returns the number of the links of a specified type to an item where the item is the tail of the link.

ItemGetHeadLink

Usage	<ItemGetHeadLink *itemObject linkObject* TYPE = *linkTypeName* INDEX = *index*>
Arguments	*itemObject* is the tail item object. *linkObject* is the returned link object. *linkTypeName* is the link name. *index* is a value from 0 to the value returned by “ItemGetHeadLinkCount”.
Return value	"0" for success and a non-zero error code on error.

Loads a link from a tail item based on an index into the links of a specific type.

ItemGetLinkById

Usage	<ItemGetLinkById *headItem tailItem linkObject* ID = *id*>
Arguments	*headItem* is the item that owns the link (the head item). *tailItem* is the tail item. *linkObject* is the returned link object. *id* is the link's ID.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemGetLinkById item item2 link ID = "3625">

Loads a link between two items by link ID.

The arguments *headItem*, *tailItem* and *id* are all required and cannot be null or "0" in order for the tag to return useful results.

ItemGetLinkCount

Usage	<ItemGetLinkCount *itemObject* TYPE = *linkTypeName*>
Arguments	*itemObject* is the head item object. *linkTypeName* is the link type to search. This value is case-sensitive, and must exactly match the assigned name of the link type.
Return value	the number of linked tail items.
Example	<ItemGetLinkCount item TYPE = "Coupled">

Returns the number of items that are linked to an item object by a spcific link type.

ItemGetLink

Usage	<ItemGetLink *itemObject linkObject* TYPE = *linkTypeName* INDEX = *index*>
Arguments	*itemObject* is the item object. *linkObject* is the returned link object. *linkTypeName* is the type of link. This value is case-sensitive, and must exactly match the assigned name of the link type. *index* is a value from 0 to the value returned by "ItemGetLinkCount".
Return value	"0" for success and a non-zero error code on error.
Example	<ItemGetLink itemObj linkObj TYPE = "Coupled" INDEX = "4">

Returns a link of a specified type at a specified index.

The *itemObject* argument can be either the *head* of a hierarchical link, or *either the head or the tail* of a non-hierarchical link. An error will be returned if a tail item is specified for *itemObject* on a hierarchical link.

Before calling "ItemGetLink", execute "ItemGetLinkCount" (see page 110) to verify that the item has a valid link.

ItemGetLinkedItem

Usage	<ItemGetLinkedItem *itemObject* *linkedItemObject* TYPE = *linkTypeName* INDEX = i*ndex*>
Arguments	*itemObject* is the head item object. *linkedItemObject* is the returned linked tail item. *linkTypeName* is the type of link to load. This value is case-sensitive, and must exactly match the assigned name of the link type. *index* is a value from 0 to the value returned by “ItemGetLinkCount”.
Return value	"0" for success and a non-zero error code on error.
Example	<ItemGetLinkedItem item linked_item TYPE = "Child" INDEX = "0">

Loads the tail item at the specified index of items linked by a given type.

ItemGetLinkExists

Usage	<ItemGetLinkExists *itemObject* ISHEAD = *isHead* TYPE = *linkTypeName*>
Arguments	*itemObject* is the item object. *isHead* is either “0” or “1”. *linkTypeName* is the link name.
Return value	“1” if *itemObject* has a link of type *linkTypeName*, “0” if not.

Returns whether a specific link type exists for an item.

The value *isHead* determines whether *itemObject i*s the head of the link *linkTypeName* when determining of a link exisits: “1” means *itemObject* is the head, “0” means *itemObject* is the tail.

ItemGetLinkSideCount

Usage	<ItemGetLinkSideCount *itemObject* ISHEAD = *isHead* TYPE = *linkTypeName*>
Arguments	*itemObject* is the item object. *isHead* is either “0” or “1”. *linkTypeName* is the link name.
Return value	the number of links.

Returns the number of head or tail links from an item for a particular link type.

If *isHead* is set to “1”, then *itemObject* is the head of the link and the tag will return the number of tail links from *itemObject* that are of type *linkName*. Conversely, if *isHead* is set to “0”, then *itemObject* is the tail of the link and the tag will return the number of head links from *itemObject*.

ItemGetLinkSide

Usage	<ItemGetLinkSide *itemObject linkObject* ISHEAD = *isHead* TYPE = *linkTypeName* INDEX = *index*>
Arguments	*itemObject* is the item object. *linkObject* is the returned link object. *isHead* is either “0” or “1”. *linkTypeName* is the link name. *index* is a value from 0 to the value returned by “ItemGetLinkSideCount”.
Return value	"0" for success and a non-zero error code on error.

Returns a link object from an item.

See “ItemGetLinkSideCount” (see page 111) for a description of the *isHead* argument.

ItemGetObsoleteTailList

Usage	<ItemGetObsoleteTailList *itemObject obsoleteList*>
Arguments	*itemObject* is the item object. *obsoleteList* is the returned list object.
Return value	"0" for success and a non-zero error code on error.

Returns a list of obsolete items linked to an item.

ItemGetTailLinkCount

Usage	<ItemGetTailLinkCount *item* TYPE = *linkTypeName*>
Arguments	*item* is the tail item object. *linkTypeName* is the link name.
Return value	the number of *linkType* links to *itemObject* where *itemObject* is the head of the link.

Returns the number of the links of a specified type to an item where the item is the head of the link.

ItemGetTailLink

Usage	<ItemGetTailLink *itemObject linkObject* TYPE = *linkTypeName* INDEX = *index*>
Arguments	*itemObject* is the head item object. *linkObject* is the returned link object. *linkTypeName* is the link name. *index* is a value from 0 to the value returned by “ItemGetTailLinkCount”.
Return value	"0" for success and a non-zero error code on error.

Loads the link from a head item based on an index into the links of a specific type.

Tags for getting item link type information

ItemGetLinkDefSideCount

Usage	<ItemGetLinkDefSideCount *itemObject*>
Arguments	*itemObject* is the item object.
Return value	the number of link definitions.

Returns the number of both head and tail link definitions of an item.

ItemGetLinkDefSideProperty

Usage	<ItemGetLinkDefSideProperty *itemObject* PROPERTY = *property* INDEX = *index*>
Arguments	*itemObject* is the item object. *property* is "TYPE", "SIDELABEL" or "ISHEAD". *index* is a value from 0 to the value returned by "ItemGetLinkDefSideCount".
Return value	the property value.

Returns the link side property value of an item's link definition.

With *property* set to "TYPE", the tag will return the link type name; with *property* set to "SIDELABEL", the tag will return the link label; with *property* set to "ISHEAD", the tag will return whether *itemObject* would be the head ("1") or tail ("0") of the link.

ItemGetLinkTypeCount

Usage	<ItemGetLinkTypeCount *itemObject*>
Arguments	*itemObject* is the head or tail item object.
Return value	the number of available link types for the item.

Returns the number of available link types for an item.

A link type will add "2" to the count if both head-to-tail and tail-to-head are defined in the Link Type Editor, using the Items in these classes fields.

ItemGetLinkTypeName

Usage	<ItemGetLinkTypeName *itemObject* INDEX = *index*>
Arguments	*itemObject* is the head or tail item object. *index* is a value from 0 to the value returned by "ItemGetLinkTypeCount".
Return value	the name of the link type.

Returns the name of a link type at an index.

ItemGetLinkTypeLabel

Usage	<ItemGetLinkTypeLabel *itemObject* INDEX = *index*>
Arguments	*itemObject* is the head or tail item object. *index* is a value from 0 to the value returned by “ItemGetLinkTypeCount”.
Return value	the link type label.

Returns a link label based on an index.

ItemGetLinkTypeIsHead

Usage	<ItemGetLinkTypeIsHead *itemObject* INDEX = *index*>
Arguments	*itemObject* is the head or tail item object. *index* is a value from 0 to the value returned by “ItemGetLinkTypeCount”.
Return value	“1” if the link type at the specified index is the head.

Determines if an available link from an item is a head link type.

Chapter 9

Queries and Reports

Just Ahead:

Creating a query object 116
Setting query object attributes 116
Getting query object attributes 120
Execute tags 121
Query storage tags 122
Workflow query tags 123
Query example 124
Reports 125

Creating a query object

QueryCreate

Usage	<QueryCreate query>
Arguments	query is the query object.
Return value	"0" for success and a non-zero error code on error.
Example	<QueryCreate newquery>

Creates a query object.

Setting query object attributes

Precedence Order

When adding a query clause the attribute name will be resolved in the following precedence order:

1. Search the attribute name on the main form
2. Search the attribute name on the common form
3. Search the attribute name on the table type form
4. Search the attribute prompt on the main form
5. Search the attribute prompt on the common form
6. Search the attribute prompt on the table type form

Table 9-1 lists the conditions that you can use with different attribute types.

Table 9-1: Valid query conditions for various attributes

Attribute Type	Attribute Names	Valid Conditions	Comments
Text	Name Title Location Description Comments Process Name Activity Name Activity Work Notes Assigned Comments	BEGINS_WITH ENDS_WITH CONTAINS DOES_NOT_CONTAIN EXACTLY IS_EMPTY IS_NOT_EMPTY	

Table 9-1: Valid query conditions for various attributes

Attribute Type	Attribute Names	Valid Conditions	Comments
Text	Revision	BEGINS_WITH ENDS_WITH CONTAINS DOES_NOT_CONTAIN EXACTLY IS_EMPTY IS_NOT_EMPTY IS_CURRENT IS_NOT_CURRENT IS_LATEST IS_NOT_LATEST	Values for Revision must be in the format: [letter]:[number]
Text	Version	EQUAL LESS_THAN LESS_EQUAL GREATER_THAN GREATER_EQUAL NOT_EQUAL BETWEEN IS_CURRENT IS_NOT_CURRENT IS_LATEST IS_NOT_LATEST	
Text / Multichoice	Preparer Reviewer Issuer Status Last User Process Coordinator Process State Activity State Activity Assigned To Assigned Status File Type	ONE_OF NOT_ONE_OF BEGINS_WITH ENDS_WITH CONTAINS DOES_NOT_CONTAIN EXACTLY IS_EMPTY IS_NOT_EMPTY	
Text / Multichoice	Class	ONE_OF NOT_ONE_OF BEGINS_WITH ENDS_WITH CONTAINS DOES_NOT_CONTAIN EXACTLY IS_EMPTY IS_NOT_EMPTY	Values for Class must be in the format: CMS:Path

Table 9-1: Valid query conditions for various attributes

Attribute Type	Attribute Names	Valid Conditions	Comments
Number	Conditions Version File Size Activity Duration	EQUAL LESS_THAN LESS_EQUAL GREATER_THAN GREATER_EQUAL NOT_EQUAL BETWEEN	Value for Version can include CURRENT (for latest version) and LATEST in conjunction with a status code of Released to find the highest released version
Date	Prepared on Reviewed on Released on Last Modified Process Start Date Process Finish Date Activity Start Date Activity Finish Date	AFTER BEFORE BETWEEN ON PAST_24_HOURS PAST_7_DAYS PAST_30_DAYS PAST_YEAR IS_EMPTY IS_NOT_EMPTY	Values for Date must be in the default format specified by cms.date.format in the cms_site file (default is MMM-DD-YYYY). TODAY is also an allowed value.

QueryAddClause

Function	Adds a clause to the query object.
Usage	<QueryAddClause *queryObject* VIEW = *view* ATTR = *attrName* CONDITION = *condition* VALUE = *value*>
Arguments	*queryObject* is the query object. *view*, for item queries, is set equal to the view name in the ProductCenter database. For Workflow queries, set this to either "PROCESS" or "ACTIVITY". *attrname* is the name of the attribute in *view*. *condition* is the condition that is defined for the attribute. *value* is value of the attribute.
Return value	"0" for success and a non-zero error code on error.
Example	<QueryAddClause newquery VIEW = "Common Attributes" ATTR = "File Type" CONDITION = "contains" VALUE = "pdf_file">

Adds a clause to a query object.

QueryClearAllClauses

Usage	<QueryClearAllClauses *queryObject*>
Arguments	*queryObject* is the query object which will have its clauses cleared.
Return value	"0" for success and a non-zero error code on error.

Clears the search criteria from a query object.

QuerySetCaseSensitive

Usage	<QuerySetCaseSensitive *queryObject* MATCHCASE = *matchCase*>
Arguments	*queryObject* is the query object. *matchCase* is either “TRUE” or “FALSE”.
Return value	"0" for success and a non-zero error code on error.

Sets the case sensitivity for a query.

QuerySetOp

Usage	<QuerySetOp *queryObject* OP = *operator*>
Arguments	*queryObject* is the query object. *operator* is either “AND” or “OR”.
Return value	"0" for success and a non-zero error code on error.

Specifies whether to match all of the criteria in a query or any one of the criteria in a query.

The default matching criteria is “AND”. Note that if a query clause is set to “Version=Current” in an "OR" query, you will get the current version of all items in the database, plus any items matching other criteria.

QuerySetType

Usage	<QuerySetType *queryObject* TYPE = *type*>
Arguments	*queryObject* is the query object that will be set. *type is either* “ITEM”, “ACTIVITY”, or “PROCESS”
Return value	"0" for success and a non-zero error code on error.

Sets the type of a query that will be created.

The default query type is “ITEM”. Once clauses have been added to the query object, the query type cannot be changed.

QuerySetItemColLayout

Usage	<QuerySetItemColLayout *query* ID=*layout_id* >
Arguments	*query* is the query object *layout_id* is the item column layout id
Return value	"0" for success and a non-zero error code on error.

Every saved query has an item column layout associated with it - either a specific customized one or the COL_LAY_ID_CURRENT_DEFAULT layout. When you call the SaveQuery function <QuerySave> that association will be created or updated. This function sets the id of the item column layout definition that will be associated.

If this function has not been called since the query object was loaded, then the association would not be changed when you call <QuerySave>. If the query object has not been used in a LoadQuery function and this function has not been called, then the association will be set to COL_LAY_ID_CURRENT_DEFAULT.

Note that these query functions will not cause an item column layout definition to be created, modified, or deleted. That must be done separately.

Getting query object attributes

QueryGetCaseSensitive

Usage	<QueryGetCaseSensitive *queryObject*>
Arguments	*queryObject* is the query object.
Return value	the case sensitivity of *queryObject*.

Returns the case sensitivity of a query object.

The tag returns “TRUE” if the query is set for case sensitive matching, or “FALSE” if not.

QueryGetClauseCount

Usage	<QueryGetClauseCount *queryObject*>
Arguments	*queryObject* is the query object you want to save.
Return value	the number of clauses in *queryObject* or "-1" on error.

Returns the number of clauses in a query.

QueryGetClauseData

Usage	<QueryGetClauseData *queryObject* PROPERTY = *clauseproperty* INDEX = *index*>
Arguments	*queryObject* is the query object you want to save. *clauseproperty* can be “View”, “AttrName”, “AttrPrompt”, “AttrType”, “Condition”, “Value”, “View Prompt”, or “Main View”. *index* is the index value.
Return value	the clause property or a null string if not found.
Example	<QueryGetClauseData query PROPERTY = "view" INDEX = "1">

Returns the property of a query clause at an index.

QueryGetOp

Usage	<QueryGetOp *queryObject*>
Arguments	*queryObject* is the query object.
Return value	the query operator ("AND" or "OR") on success and an empty string on error.

Returns the logical operator of a query object.

The tag returns “AND” if all clauses in the query are joined with “AND” or it returns “OR” if all clauses in the query are set to be joined with a logical “OR”.

QueryGetType

Usage	<QueryGetType *queryObject*>
Arguments	*queryObject* is the query to check
Return value	"ITEM", "PROCESS", or "ACTIVITY"

Determines whether a query object is an item, process, or activity query.

QueryGetItemColLayout

Usage	<QueryGetItemColLayout *query itemlayout*
Arguments	*query* is the query object *itemlayout* is the item column layout object
Return value	"0" for success and a non-zero error code on error.

Every saved query has an item column layout associated with it - either a specific customized one or the COL_LAY_ID_CURRENT_DEFAULT layout. When you call the LoadQuery functions <QueryLoad> or <QueryLoadByID> the definition of the associated item column layout is fetched from the server together with the definition of the query. This function returns the pointer to that item column layout definition.

Execute tags

QueryExecute

Usage	<QueryExecute *queryObject* THRESHOLD = *threshhold*>
Arguments	*queryObject* is the query object. *threshhold* is the number of objects that *queryObject* will return.
Return value	"0" for success and a non-zero error code on error.
Example	<QueryExecute newquery THRESHOLD = "0">

Executes a query.

Set *threshhold* equal to "0" to get all objects found by the query.

QueryExecuteMatchCase

Usage	<QueryExecuteMatchCase *queryObject* THRESHOLD= *number* MATCHCASE= *matchCase*>
Arguments	*queryObject* is the query object. *number* is the number of objects that *queryObject* will return. Set equal to "0" to get all objects found by the query. *matchCase* is set to "TRUE" to make the query case-sensitive.
Return value	"0" for success and a non-zero error code on error.
Example	<QueryExecuteMatchCase newquery THRESHOLD = "0" MATCHCASE = "TRUE">

Executes a query with specification of case sensitivity.

QueryGetItemCount

Usage	<QueryGetItemCount *queryObject*>
Arguments	*queryObject* is the query object.
Return value	the number of objects returned. "-1" if error.

Returns the number of objects that a query returns upon execution.

QueryGetItem

Usage	<QueryGetItem *queryObject itemObject* INDEX = index>
Arguments	*queryObject* is the query object. *itemObject* is the object into which the specified item is loaded. *index* is the index number of the desired item.
Return value	"0" for success and a non-zero error code on error.
Example	<QueryGetItem newquery item INDEX = "1">

Loads a specific item from a query's result set.

<QueryGetItem> can only be called after "QueryGetItemCount" (see page 122) since that function will return the number of items found in the query execution.

Query storage tags

QueryDelete

Usage	<QueryDelete *queryObject*>
Arguments	*queryObject* is the query object that will be deleted.
Return value	"0" for success and a non-zero error code on error.

Deletes a stored query.

A stored query must be loaded before it can be deleted.

QueryLoad

Usage	<QueryLoad *queryObject* NAME = *queryName*>
Arguments	*queryObject* is the returned query object. *queryName* is the name of the stored query.
Return value	"0" for success and a non-zero error code on error.

Loads a stored query by name.

The query must belong to the connected user.

QueryLoadById

Usage	<QueryLoadByID *queryObject* ID = *queryId*>
Arguments	*queryObject* is the returned query object. *queryId* is the ID of the stored query.
Return value	"0" for success and a non-zero error code on error.

Loads a query object by its ID.

The query can belong to any user.

The ID of a query object can be retrieved by using the tag “ListLoad” (see page 78), then getting individual IDs from the returned list using “ListGetSystemId” (see page 80).

QuerySave

Usage	<QuerySave *queryObject* NAME = *queryName*>
Arguments	*queryObject* is the query object that will be saved. *queryName* is the name that will be assigned to the query.
Return value	"0" for success and a non-zero error code on error.
Example	<QuerySave query NAME = "getecos">

Saves a query to a stored query.

Workflow query tags

QueryGetActivityInstCount

Usage	<QueryGetActivityInstCount *queryObject*>
Arguments	*queryObject* is the query object.
Return value	the number of activity instances found in *queryObject* or "-1" on error.

Returns the number of activity instances found by a query.

QueryGetActivityInst

Usage	<QueryGetActivityInst *queryObject actinst* INDEX=*index*>
Arguments	*queryObject* is the query object. *acitivytInst* is the returned activity instance *index* is the index into the activity instance results to return.
Return value	"0" for success and a non-zero error code on error.

Returns an activity instance from a query.

QueryGetProcessInstCount

Usage	<QueryGetProcessInstCount *queryObject*>
Arguments	*queryObject* is the query object.
Return value	the number of process instances found in *queryObject* or "-1" on error.

Returns the number of process instances found by a query.

QueryGetProcessInst

Usage	<QueryGetProcessInst *queryObject processInst* INDEX = *index*>
Arguments	*queryObject* is the query object. *processInst* is the returned process instance. *index* is the index into the process instance results to return.
Return value	"0" for success and a non-zero error code on error.

Returns a process instance from a query.

Query example

The code example below illustrates the use of the query tags described in this section. This code sample locates the current version of all files in the database by name and then loops through each of the items.

```
<QueryCreate newquery>
<QueryAddClause newquery VIEW = "Common Attributes"
   ATTR = "Name" CONDITION = "contains" VALUE = name>
<QueryAddClause newquery VIEW = "Common Attributes"
   ATTR = "Class" CONDITION = "one of" VALUE = "CMS:Files">
<QueryAddClause newquery VIEW = "Common Attributes"
   ATTR = "Version" CONDITION = "=" VALUE = "current">
<QuerySetOp newquery OP = "AND">
```

```
<QueryExecute newquery THRESHOLD = "0">
<setval amount <QueryGetItemCount newquery> >
<setval index "0">
<while index lt amount>
<QueryGetItem newquery item INDEX = index>
<setval item_name <ItemGetAttr item ATTR = "Name">>
Item Name: <printval item_name>
<setval index <expr index + "1">>
</while>
```

Query Programming Tip

Generating a search with a ProductCenter web or Windows interface will aid in the development of a query. Use the **Search by attributes** option under the **File** menu to construct your query. You can then run the query to make sure it works, and incorporate each row into statements in your WML programs.

Reports

Use these tags for creating and running reports. Also see the report-related options in “ListLoad” (see page 78) .

ReportExecute

Usage	<ReportExecute *reportObject input* TYPE = *type*>
Arguments	*reportObject* is the report you want to execute. *input* is either the item or process object to run the report against, or the query object to run. *type* is one of: “ITEM_SPECIFIC”, “ITEM_BYQUERY”, “PROCESS_SPECIFIC”, “PROCESS_BYQUERY”.
Return value	"0" for success and a non-zero error code on error.

Executes a report.

ReportExecuteMatchCase

Usage	<ReportExecuteMatchCase *reportObject queryObject* TYPE = *type* THRESHOLD = *threshhold* MATCHCASE = *matchcase*>
Arguments	*reportObject* is the report object. *queryObject* is the query object to run the report against. *type* is "ITEM_BYQUERY" or "PROCESS_BYQUERY" *threshhold* is the number of objects that *queryObject* will return. *matchcase* is “TRUE” or “FALSE”.
Return value	"0" for success and a non-zero error code on error.

Executes a query based report.

Set *threshhold* equal to "0" to get all objects found by the query.

ReportGetAttrCount

Usage	<ReportGetAttrcount *reportObject* >
Arguments	*reportObject* is the report object.
Return value	the number of attributes of *reportObject*.

Returns the number of attributes of a report object.

ReportGetAttrNameByIndex

Usage	<ReportGetAttrNameByIndex *reportObject* INDEX = *index*>
Arguments	*reportObject* is the report object. *index* is the index into the attributes of *reportObject*.
Return value	the attribute name.

Returns the name of an attribute of a report object.

ReportGetAttr

Usage	<ReportGetAttr *reportObject* ATTR = attrN*ame*>
Arguments	*reportObject* is the report whose attribute value you wish to retrieve attrN*ame* is the name of the attribute whose value will be returned. Possible names are "Id", "Name", "Description", "Type", "Parent Id", "Export Location", "HTML Path", "XSL Filename", "Customizable", “Local File Path”, “Mime File Extension”, “Mime Type”.
Return value	the attribute value.

Returns the value of an attribute of a report object.

ReportGetQuery

Function	Returns the query object associated with the report.
Usage	<ReportGetQuery *reportObject queryObject*>
Arguments	*reportObject* is the report object. *queryObject* is the returned query object associated with *reportObject*.
Return value	"0" for success and a non-zero error code on error.

Returns the query object associated with a report.

ReportLoad

Usage	<ReportLoad *reportObject* ID = *reportID*>
Arguments	*reportObject* is the returned report object. *reportID* is the id of the report that will be loaded.
Return value	"0" for success and a non-zero error code on error.

Creates a report object from a report id.

ReportLoadByName

Usage	<ReportLoadByName *reportObject* NAME = report*Name*>
Arguments	*reportObject* is the returned report object. report*Name* is the name of the report that will be loaded.
Return value	"0" for success and a non-zero error code on error.

Creates a report object from a report name.

ReportSetAttr

Usage	<ReportSetAttr *reportObject* ATTR = attrN*ame* VALUE = attrV*alue*>
Arguments	*reportObject* is the report object. attrN*ame* is the name of the attribute whose value will be set. Possible names are "Export Location", "HTML Path", "Local File Path". attrV*alue* is the value to set
Return value	"0" for success and a non-zero error code on error.

Sets the value of an attribute of a report object.

ReportSetExport

Usage	<ReportSetExport *reportObject* NAME = *exportFile*>
Arguments	*reportObject* is the report object. *exportFile* is the name of the export file.
Return value	"0" for success and a non-zero error code on error.

Sets the name of an export report file.

ReportSetXslPath

Usage	<ReportSetXslPath *reportObject* PATH = *xslFile*>
Arguments	*reportObject* is the report object. *xslFile* is the file name of the XSL file.
Return value	"0" for success and a non-zero error code on error.

Sets the XSL (by file name) that will be used when a report is generated.

10

Chapter 10

Workflow

Just Ahead:

Getting workflow information 130
Process definition tags. 131
Process instance tags 133
Activity instance tags. 137

Getting workflow information

ProcessLoadForProcessList

Usage	<ProcessLoadForProcessList *processList* PROCESSTYPE = *processType*>
Arguments	*processList* is the lreturned process list. *processType* can be set to one of the following:: “INITIATED” “COMPLETED” “CANCELLED” “ACTIVE”
Return value	none.

Loads a process list with processes of a specific status.

Note that a typo in the *processType* argument (such as misspelling “CANCELLED” as “CANCELED”) will cause this tag to return *all* active processes.

ProcessGetProcessListCount

Usage	<ProcessGetProcessListCount *processList*>
Arguments	*processList* is the process list.
Return value	the number of processes in *processList*.
Example	<setval error <ItemLoad item ID=cmsId>> <if error eq "0"> <ITEMLOADATTACHEDPROCESSLIST item processlist> <setval count <ProcessGetProcessListCount processlist>> </if>

Returns the number of processes in a process list.

ProcessGetProcessListAttr

Usage	<ProcessGetProcessListAttr *processList* ATTR= *attrName* INDEX = *index*>
Arguments	*processList* is the process list. *attrName* is the attribute name. *index* is the index into *processlist*.
Return value	the value of the specified attibute.

Returns process attribute values in a process list.

The acceptable attrName values are:

- "Name"
- "Assignments"
- "Version"
- "State"
- "Start Date"
- "End Date"
- "Process Id"

Process definition tags

ProcessDefLoad

Usage	<ProcessDefLoad *processdef* ID = *id*>
Arguments	*processdef* is the process definition object. *id* is the CMS ID of the process definition.
Return value	"0" for success and a non-zero error code on error.

Loads a process definition by ID.

ProcessDefGetAttr

Usage	<ProcessDefGetAttr *processDef* ATTR = attrN*ame*>
Arguments	*processDef* is the process definition object. attrN*ame* is the name of the attribute whose value will be returned.
Return value	the value of the attribute or null string if error.

Returns a process definition attribute value.

The acceptable attrName values are:

- "Name": The name of the process definition.
- "Version": The version number of the process definition. When you obtain a list of process definitions, the workflow engine gives you the latest versions. You can retrieve previous versions with GetPastVersion().
- "Prepared on": The date on which the process definition was created.
- "Date Last Mod": The most recent date on which a user edited the process definition.
- "Prepared by": The user name of the person who created the process definition.
- "Description": An explanation of the process definition.
- "Coordinator": The name of the process definition’s coordinator. A coordinator is a special user who is responsible for resolving certain workflow issues.
- "CMS ID": The ID of the process definition.
- "Type": The process definition type. The value returned is "FORM_BASED" or "ROUTE_BASED".
- "Class": The class name associated with the process definition.

ProcessDefGetNextActivities

Usage	<ProcessDefGetNextActivities *processDef itemObject activityList* TYPE = “start”>
Arguments	*processDef* is the process definition object. *itemObject* is the item object which is attached to an instance of *processDef*. *activityList* is the return list of start activities for *itemObject* in *processDef*.
Return value	"0" on success or a non-zero error code on error.

Creates a list of activities for an item in a process.

Once the list is generated then tags such as “ListGetCount” (see page 80) and “ListGetActivityInstance” (see page 81) can be used to access the individual activity definitions in *activityList*.

Note that “start” is the only valid value for this tag’s “TYPE” argument.

ProcessDefGenerateUniqueName

Usage	<ProcessDefGenerateUniqueName *processDef*>
Arguments	*processDef* is the process definition object.
Return value	the name of the process instance, or returns a null string on error.

Creates a unique name for an instance of the process definition specified.

The returned value (process instance name) takes the form “*nameOfProcessDef number*” where *number* is incremented each time a new instance is created for the process definition named *nameOfProcessDef*.

ProcessDefRouteItem

Usage	<ProcessDefRouteItem *processDef itemObject processInst* NAME = *processName*>
Arguments	*processDef* is the process definition object from which the process instance will be created. *itemObject* is the item being routed. *processInst* is the returned process instance object. *processName* is the name to be assigned to the process instance.
Return value	"0" on success or a non-zero error code on error.

Creates an instance of the specified process definition, associates it with an existing item, and loads the instance into the variable process. The user must have been granted launch permission by the process definition.

Note that this tag is used for both route and form based workflows. For form based workflow, the form needs to exist before starting the workflow.

ProcessDefRouteItemWithFlags

Usage	<ProcessDefRouteItemWithFlags *processDef itemObject processInst activityList* NAME = *processName* TYPE = *type*>
Arguments	*processDef* is the process definition object from which the process instance will be created. *itemObject* is the item being routed. *processInst* is the returned process instance object. *activityList* is the list of activities to initiate. *processName* is the name to be assigned to the process instance. *type* is "START" or "START|WARN_UNFILLED".
Return value	"0" on success or a non-zero error code on error.

Routes a process based on flags. The user must have been granted launch permission by the process definition.

Process instance tags

This section describes the WebLink tags that operate on an existing process instance.

ProcessLoad

Usage	<ProcessLoad *processInst* ID = *id*>
Arguments	*processInst* is the process instance object. *id* is the CMS ID of the process instance.
Return value	"0" for success and a non-zero error code on error.

Loads a process instance by ID.

ProcessGetProcessDef

Usage	<ProcessGetProcessDef *processInst processDef*>
Arguments	*processInst* is the process instance object. *processDef* is the returned process definition object.
Return value	"0" for success and a non-zero error code on error.

Loads a process definition for a process instance.

ProcessGetAttr

Usage	<ProcessGetAttr *processInst* ATTR = *attrName*>
Arguments	*processInst* is the process instance object. *attrName* is the name of the attribute to retrieve.
Return value	the value of the attribute or a null string.

Returns the value of a process instance attribute.

The acceptable *attrName* values are:

- "Name": The name of the process instance. The system may have generated this name automatically when a user created the instance.
- "Prepared on": The date on which the item (not the process instance) was created.
- "Date Last Mod": The last date on which someone edited the item (not the process instance).
- "Prepared by": The user name that created the item (not the process instance).
- "Description": An explanation of the process instance. This text appears in the Description field of the General tab of the Workflow Properties window.
- "Coordinator": The name of the process instance's coordinator. A coordinator is a special user who is responsible for resolving certain workflow issues, such as processes that are disapproved or put on hold.
- "State": The state of the instance you choose. A process instance can have one of six states, but only four are accessible. The possible return values are:
  - "INITIATED"
  - "ACTIVATED"
  - "COMPLETED"
  - "CANCELLED"
  - Two other states, "NONE" and "DELETED", exist internally but are not accessible through this call.
- "CMS ID": The ID of the process instance.
- "Class": The class name associated with the process instance.

ProcessGetItem

Usage	<ProcessGetItem *processInst itemObject*>
Arguments	*processInst* is the process instance object. *itemObject* is the item being routed by *processInst*.
Return value	"0" for success and a non-zero error code on error.

Loads the item associated with a process instance.

ProcessGetCompletedActivityCount

Usage	<ProcessGetCompletedActivtyCount *processInst*>
Arguments	*processInst* is the process instance object.
Return value	the number of completed activities, -1 on error.

Returns the number of completed activities for a process instance.

ProcessGetCompletedActivty

Usage	<ProcessGetCompletedActitivy *processInst activityInst* INDEX = index>
Arguments	*processInst* is the process instance object. *activityInst* is the returned activity instance object. *index* is the index value.
Return value	"0" for success and a non-zero error code on error.

Returns an activity instance object from a list of completed activities.

ProcessGetAuditLogListCount

Usage	<ProcessGetAuditLogListCount *processInst*>
Arguments	*processInst* is the process instance object.
Return value	the number of audit log entries, “-1” if error.

Returns the number audit log entries for a process instance.

ProcessGetAuditLogAttr

Usage	<ProcessGetAuditLogAttr *processInst* PROPERTY = *attrName* INDEX = *index*>
Arguments	*processInst* is the process instance object. *attrName* is the name of the attribute. *index* is the index value.
Return value	the queried property, empty string "" on error.

Returns the attribute of an audit log entry.

The acceptable attrName values are:

- “Name”: Name of the audit log entry.
- “ActivityName”: Name of the activity instance to which the audit log entry is associated.
- “Comment”: The comment for the audit log entry.
- “UserName”: The name of the user who performed the action
- “DateTime”: Date and time during which the action was performed.

ProcessViewerGetListCount

Usage	<ProcessViewerGetListCount *processInst*>
Arguments	*processInst* is the process instance object.
Return value	number of activities in the proces instance.

Returns the number of activities in a process instance.

ProcessViewerGetAttr

Usage	<ProcessViewerGetAttr *processInst* ATTR = *attrName* INDEX = *index*>
Arguments	*processInst* is the process instance object. *attrName* is the name of the attribute whose value will be returned. *index* is the index into the activities of *processInst*.
Return value	the attribute value.

Returns an attribute value of an activity in a process.

The acceptable attrName values are:

- "Activity"
- "Assignments"
- "Status"
- "Initiated Date"
- "Completed Date"
- "Est. Completion"
- "Work Instructions"
- "Duration"
- "Done Threshold"
- "Back Threshold"
- "Travel Counter"

ProcessIsCoordinator

Usage	<ProcessIsCoordinator *processInst*>
Arguments	*processInst* is the process instance object.
Return value	"1" if connected user has coordinator privileges.
Example	<setval isCoordinator <ProcessIsCoordinator process>> <if isCoordinator eq "1"> *...do coordinator stuff...* <else> *...report error...* </if>

Returns “1” if user has coordinator privileges for a process instance.

ItemLoadAttachedProcessList

Usage	<ItemLoadAttachedProcessList *itemObject processList*>
Arguments	*itemObject* is the item object. *processList* is the returned process list.
Return value	"0" for success and a non-zero error code on error.
Example	<setval error <ItemLoad item ID=cmsId>> <if error eq "0"> <ItemLoadAttachedProcessList item processlist> <setval count <ProcessGetProcessListCount processlist>>

Loads a process list of processes attached to an item.

Once the list is loaded, then tags such as “ProcessGetProcessListCount” (see page 130) and “ProcessGetProcessListAttr” (see page 130) can be used to get information from the list.

Activity instance tags

Tags for acquiring activity instances, getting activity instance attributes or related objects.

ActivityLoad

Usage	<ActivityLoad *activityInst* ID = *id*>
Arguments	*activityInst* is the returned activity instance object. *id* is the CMS ID of the activity instance.
Return value	"0" for success and a non-zero error code on error.

Loads an activity instance by ID.

ActivityGetAttr

Usage	<ActivityGetAttr *activityInst* ATTR = *attrName*>
Arguments	*activityInst* is the activity instance object. attrN*ame* is the name of the attribute whose value will be returned.
Return value	the attribute value, or a null string on error.
Example	<ActivityGetAttr myActivity ATTR = "Time Start">

Returns an activity definition or instance attribute value.

The acceptable attrName values are:

- "Name": The name of the activity instance.
- "Work notes": A character string, up to 512 bytes long, that contains a description of the activity instance.
- "Time start": The actual start time for this instance.
- "Date Last Mod": The date on which the instance was most recently modified.
- "Duration": The expected amount of time in days that this instance will take to finish.
- "CMS ID": The ID of the activity instance. Note that Weblink returns cmsid with a pipe (|) appended, and you should specify this pipe when using cmsid as an argument to another call. This pipe was implemented as a convenience since calls often require an argument of the form cmsid|user.
- "State": The state of the instance you choose. An activity instance can have one of six states. The possible return values are: "None", "Initiated", "Activated", Claimed", "Suspended", "Cancelled". Two other states, "Deleted" and "On Hold", exist internally but are not accessible through this call.

ActivityGetItem

Usage	<ActivityGetItem *activityInst itemObject*>
Arguments	*activityInst* is the activity instance object. *itemObject* is the returned item associated with *activityInst*.
Return value	"0" for success and a non-zero error code on error.

Loads the item associated with an activity instance.

ActivityGetProcess

Usage	<ActivityGetProcess *activityInst processInst*>
Arguments	*activityInst* is the activity instance object. *processInst* is the returned process instance.
Return value	"0" for success and a non-zero error code on error.

Loads the process assocated with an activity instance.

ActivityInstGetNextActivities

Usage	<ActivityInstGetNextActivities *activityInst activityList* TYPE = *flag*>
Arguments	*activityInst* is the activity instance object. *activityList* is the return list of activities. *type* is the expected flow of the process.
Return value	"0" for success and a non-zero error code on error.

Returns a list of activities that might or will be activiated after a specified activity.

This is used before the tags SendForward, SendBack, Reassign, and Resume. The flag can have one of the following values "OP_SENDFORWARD", "OP_SENDBACK", "OP_SENDBACK_PREVIOUS", "OP_REASSIGN", and "OP_RESUME", and indicates which interface is about to be called. In the case of SendForward, Reassign or Resume, this list is used to assign any unfilled assigned roles (using the SetAssignment function). When an assignment is reassigned to a bypass user, it is possible that this will result in the complete approval of the activity instance (like SendForward). The same is true when a suspended activity is reassigned and than resumed. This interface returns the list of activities that would be activated if this happens. For SendBack when used with the OP_SENDBACK, the user will need to choose which activity to send back to from those in the returned list.

Activity instance assignment tags

ActivityGetAssignmentCount

Usage	<ActivityGetAssignmentCount *activityInst*>
Arguments	*activityInst* is the activity instance object.
Return value	the number of open role assignments, "-1" on error.

Returns the number of open role assignments for an activity.

ActivityGetAssignmentCountByType

Usage	<ActivityGetAssignmentCountByType *activityInst* TYPE = *type*>
Arguments	*activityInst* is the activity instance. *type* is "ASSIGNED" or "OPEN".
Return value	the number of assignments.

Returns the number of open or assigned assignments for an activity.

ActivityGetReassignCount

Usage	<ActivityGetReassignCount *activityInst*>
Arguments	*activityInst* is the activity instance object.
Return value	the number of assignments for *activityInst*.

Returns the number of users that need to be reassigned to an activity.

ActivityGetAssignmentActionType

Usage	<ActivityGetAssignmentActionType *activityInst* TYPE = *type* INDEX = *index*>
Arguments	*activityInst* is the activity instance object. *type* is the status of the assignment and can be either "OPEN" or "ASSIGNED". *index* is the index into the assignments to *activityInst*.
Return value	the action type of the assignment.

Returns the action type of an assignment.

Possible return values are "REVIEWER" or "APPROVER".

ActivityGetAssignmentChoiceID

Usage	<ActivityGetAssignmentChoiceID *activityInst* INDEX = *index*>
Arguments	*activityInst* is the activity instance. *index* is index into assignments for *activityInst*.
Return value	the choice list ID.

Returns the ID of the choice list for an activity assignment.

ActivityGetAssignmentName

Usage	<ActivityGetAssignmentName *activityInst* INDEX = *index*>
Arguments	*activityInst* is the activity instance object. *index* is the index into the assignments to *activityInst*.
Return value	the name of the assigned group or user or an empty string on error.

Returns the name of the assigned group or user to an activity instance.

ActivityGetAssignmentNameByType

Usage	<ActivityGetAssignmentNameByType *activityInst* TYPE = *type* INDEX = *index*>
Arguments	*activityInst* is the activity instance. *type* is "ASSIGNED" or "OPEN". *index* is index into assignments for *activityInst*.
Return value	the assignment name.

Returns the open or assigned name of an activity assignment.

ActivityGetAssignmentStatus

Usage	<ActivityGetAssignmentStatus *activityInst* TYPE = *type* INDEX = *index*>
Arguments	*activityInst* is the activity instance. *type* is "ASSIGNED" or “OPEN”. *index* is index into activity assignments.
Return value	the assignment status.

Returns the open or assigned status of an activity assignment.

ActivityGetAssignmentType

Usage	<ActivityGetAssignmentType *activityInst* INDEX = *index*>
Arguments	*activityInst* is the activity instance object. *index* is the index into the assignments to *activityInst*.
Return value	the type of assignment or an empty string on error.

Returns the type of an assignment.

Possible return values are “GROUP_ROLE” or “USER_ROLE”.

ActivitySetAssignment

Usage	<ActivitySetAssignment *activityInst* ROLE = *assignee* INDEX = *index*>
Arguments	*activityInst* is the activity instance object. *assignee* is the user name of the user or group name filling the role. *index* is the index into the assignments to *activityInst*.
Return value	"0" for success and a non-zero error code on error.

Assigns a specific user or group to a role assignment.

Tags for moving activity instances in a process.

ActivityClaim

Usage	<ActivityClaim *activityInst* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *comment* is descriptive text added to the audit log for the claiming of the activity.
Return value	"0" for success and a non-zero error code on error.
Example	<ActivityClaim activity COMMENT = "Will look at soon.">

Claims an activity for the connected user.

ActivityDisapprove

Usage	<ActivityDisapprove *activityInst* COMMENT = *comment* TYPE = *type*>
Arguments	*activityInst* is the activity instance object. *comment* is the comment associated with the action. *type* indicates the routing method for the rejected item and is either "PREVIOUS" or "ALL". "PREVIOUS" will send the activity to the previous step. For "ALL", no automatic routing will take place.
Return value	"0" for success and a non-zero error code on error.

Disapproves an activity.

ActivityPlaceOnHold

Usage	<ActivityPlaceOnHold *activityInst* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Places "On Hold" an activity instance's assignment to the connected user.

ActivityReassignFromUserToUser

Usage	<ActivityReassignFromUserToUser *activityInst* OLDUSER = *oldUserName* NEWUSER = *newUserName*>
Arguments	*activityInst* is the activity instance object. *oldUserName* is the name of the current assignee of *activityInst*. *newUserName* is the name of the user that will replace *oldUser* as an assignee to *activityInst*.
Return value	"0" for success and a non-zero error code on error.

Reassigns an activity assignment to a new user.

ActivityResume

Usage	<ActivityResume *activityInst* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Resumes a suspended activity.

ActivitySendBackward

Usage	<ActivitySendBackward *activityInst* IDLIST = *idList* TYPE = *type* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *idList* is a comma separated list of activity instance IDs to which *activityInst* will be sent back. *type* is either "backward" or "backward|Ignore_threshold". *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Sends an activity back in a process.

Only an administrator or process coordinator can include the phrase "Ignore_threshold" in the *type* argument.

ActivitySendForward

Usage	<ActivitySendForward *activityInst activityList* TYPE = *type* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *activityList* is the list of activities that with their role assignment to which *activityInst* will be sent forward. *type* is either "forward" or "forward|Ignore_threshold". *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Sends an activity forward in a process.

The *activityList* argument can be empty or null indicating that the activity will be sent forward according to the default process flow.

Only an administrator or process coordinator can include the phrase "Ignore_threshold" in the *type* argument.

ActivitySuspend

Usage	<ActivitySuspend *activityInst* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Places an activity instance in the suspended state.

ActivityToggleHoldForUser

Usage	<ActivityToggleHoldForUser *activityInst* USERNAME = *userName* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *userName* is the user name of the assignee whos hold status is toggled. *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Toggles the "on hold" status of an assignment to a specified user.

Only an administrator or a process coordinator may specify any user name for the *userName* argument. If *userName* is null, then the connected user is assumed.

ActivityTurnOffHold

Usage	<ActivityTurnOffHold *activityInst* COMMENT = *comment*>
Arguments	*activityInst* is the activity instance object. *comment* is the comment associated with the action.
Return value	"0" for success and a non-zero error code on error.

Turns off the hold an activity instance's assignment to the connected user.

Chapter 11

BOMs

11

Just Ahead:

BOMitem objects and item objects 146
Loading BOMitem and item objects 146
BOMitem object attribute tags. 146
BOMitem object linked item tags 147
BOMitem object manipulation tags 149
Configuration and Iteration Objects. 150
Defining a config object 150
Config object release management tags. 152
Config object information and reporting tags. . 153
Config iteration object tags 154
Item Object BOM tags 156

BOMitem objects and item objects

From a Weblink perspective, a ProductCenter BOMitem object differs from an item object in that the BOMitem object defines both the item in addition to any BOM links. A BOMitem object also has access to the tags below, whereas an item object does not.

Loading BOMitem and item objects

BomItemLoad

Usage	<BomItemLoad *bomItemObject itemObject*>
Arguments	*bomItemObject* is the returned BOM item object. *itemObject* is the item object.
Return value	"0" for success and a non-zero error code on error.

Loads a BOMitem object from an item object.

BomItemGetItem

Usage	<BomItemGetItem *bomItemObject itemObject*>
Arguments	*bomItemObject* is the BOMitem object. *itemObject* is the returned item object.
Return value	"0" for success and a non-zero error code on error.

Returns the item object associated with a BOMitem object.

BOMitem object attribute tags

BomItemGetBomItemAttr

Usage	<BomItemGetBomItemAttr *bomItemObject linkObject* ATTR = *attrName*>
Arguments	*bomItemObject* is the BOMitem object. *linkObject* is the link to query for a value for *attrName*. *attrName* is the name of the link attribute in *linkObject*.
Return value	the link attribute value.

Returns a link attribute value given a BOMitem object and link object.

BomItemGetBomItemAttrs

Usage	<BomItemGetBomItemAttrs *bomItemObject linkObject* TYPE = *type*>
Arguments	*bomItemObject* is the BOMitem object. *linkObject* is the link to query for attribute values. *type* is “BASIC” or "DEFAULT".
Return value	a pipe-delimited list of link attribute values.

Returns a pipe-delimited list of link attribute values given a BOMitem object and link object.

If type is set to "DEFAULT", the return value contains 7 pipe-delimited values in this order: "Find Number", "Type", "Quantity", "Unit of Measure", "Effectivity", "Tool" and "Reference Designators". If type is set to "BASIC", the return value contains the same information as "DEFAULT" except without the "Reference Designator" value.

BomItemSetBomItemAttr

Usage	<BomItemSetBomItemAttr *bomItemObject linkObject* ATTR = *attrName* VALUE = *attrValue*>
Arguments	*bomItemObject* is the BOMitem object. *linkObject* is the link object whose attribute will be set. *attrName* is the *linkObject* attribute name to set. *attrValue* is the value to set to *attrName*.
Return value	"0" for success and a non-zero error code on error.

Sets a link attribute value for a link to a BOMitem.

BOMitem object linked item tags

BomItemGetBomItemCount

Usage	<BomItemGetBomItemCount *bomItemObject*>
Arguments	*bomItemObject* is the BOMitem object.
Return value	the number of items linked to *bomItemObject*.

Returns the number of items linked to a BOMitem with a "BomItem" link.

BomItemGetBomItemByIndex

Usage	<BomItemGetBomItemByIndex *bomItemObject itemObject* INDEX = *index*>
Arguments	*bomItemObject* is the BOMitem object. *itemObject* is the returned item object. *index* is a value between 0 and the value returned by "BomItemGetBomItemCount".
Return value	"0" for success and a non-zero error code on error.

Returns an item linked to a BOMitem.

BomItemGetBomItemLinkByIndex

Usage	<BomItemGetBomItemLinkByIndex *bomItemObject linkObject* INDEX = *index*>
Arguments	*bomItemObject* is the BOMitem object. *linkObject* is the returned link object. *index* is a value between 0 and the value returned by "BomItemGetBomItemCount".
Return value	"0" for success and a non-zero error code on error.

Returns a link from a BOMitem given an index into the BOMitem.

BomItemGetLinkById

Usage	<BomItemGetLinkById *bomItemObject tailItemObject linkObject* ID = *linkId*>
Arguments	*bomItemObject* is the BOMitem object. *tailItemObject* is the tail item object link to *bomItemObject*. *linkObject* is the returned link object. *linkId* is the ID of *linkObject*.
Return value	"0" for success and a non-zero error code on error.

Returns a link to a BOMitem given a linkID and a tail item to a BOMitem.

BomItemGetReferenceCount

Usage	<BomItemGetReferenceCount *bomItemObject*>
Arguments	*bomItemObject* is the BOMitem object.
Return value	the number of reference links in *bomItemObject*.

Returns the number of reference links to a BOMitem.

BomItemGetReferenceItemByIndex

Usage	<BomItemGetReferenceItemByIndex *bomItemObject referenceItemObject* INDEX = *index*>
Arguments	*bomItemObject* is the BOMitem object. *referenceItemObject* is the returned reference item object. *index* is a value between 0 and the value returned by "BomItemGetReferenceCount".
Return value	"0" for success and a non-zero error code on error.

Returns a reference item linked to a BOMitem.

BOMitem object manipulation tags

BomItemAddBomItem

Usage	<BomitemAddBomitem *bomItemObject tailItemObject linkObject* FINDNUMBER = *findNumber*>
Arguments	*bomItemObject* is the BOMitem object. *tailItemObject* is the item object to be added to *bomItemObject*. *linkObject* is the link object to be used between *bomItemObject* and *tailItemObject*. *findNumber* is the Find Number for *tailItemObject* in *bomItemObject*.
Return value	"0" for success and a non-zero error code on error.

Adds an item to a BOMitem.

BomItemRemoveBomItemByLink

Usage	<BomItemRemoveBomItemByLink *bomItemObject linkObject*>
Arguments	*bomItemObject* is the BOMitem object. *linkObject* is link object that identifies the item within *bomItemObject* to remove.
Return value	"0" for success and a non-zero error code on error.

Removes an item linked to a BOMitem given a link object.

BomItemRemoveReferenceByIndex

Usage	<BomItemRemoveReferenceByIndex *bomItemObject* INDEX = *index*>
Arguments	*bomItemObject* is the BOMitem object. *index* is a value between 0 and the value returned by “BomItemGetReferenceCount”.
Return value	"0" for success and a non-zero error code on error.

Removes a reference item linked to a BOMitem.

BomItemRetrieveForMassChange

Usage	<BomItemRetrieveForMassChange *bomObject configIter*>
Arguments	*bomObject* is the BOMitem object. *configIter* is the returned config iteration object.
Return value	"0" for success and a non-zero error code on error.

Returns a config iterator object from a bomItem for a mass change operation.

Configuration and Iteration Objects

A Weblink configuration object ("config object") is designed to represent any ProductCenter item hierarchy and is defined by a root item or items and a baseline. The config object only includes information about the root(s) of the config object, its links and linked items. But there are no direct methods to get individual links or tail items from the config object. However, a configuration iteration object ("config iteration object") allows for stepping through a config object and getting individual links. From those links, the head or tail item of each link can be acquired using link object tags.

Config objects are useful for manipulating assemblies for release management operations, or for simply expanding an item for web interface needs.

Defining a config object

Defining a config object generally follows these steps: creating a config object, adding root items to the config object, setting any needed config object parameters, then loading the config object.

ConfigCreate

Usage	<ConfigCreate *configObject*>
Arguments	*configObject* is the created config object.
Return value	"0" for success and a non-zero error code on error.

Creates a config object.

ConfigAddRoot

Usage	<ConfigAddRoot *configObject itemObject*>
Arguments	*configObject* is the config object. *itemObject* is the itemObject to add as a root to *configObject*.
Return value	"0" for success and a non-zero error code on error.

Adds a root item to a config object.

ConfigSetBaseline

Usage	<ConfigSetBaseline *configObject* ID = *baseline*>
Arguments	*configObject* is the config object. *baseline* is the name of the baseline.
Return value	"0" for success and a non-zero error code on error.

Sets the baseline for a config object.

By default, without setting a baseline, then all links are loaded for a config object when the "ConfigLoad" tag is used. If the "ConfigLoadForReleaseManagement" tag is used without setting a baseline, then only release managed links (hierarchical and non-hierarchical) are loaded.

ConfigSetExport

Usage	<ConfigSetExport *configObject* NAME = *exportFile*>
Arguments	*configObject* is the config object. *exportFile* is the name of the file which will contain the XML/HTML representation of t*configObject*.
Return value	"0" for success and a non-zero error code on error.

Sets the name of the export file.

ConfigSetMaxDepth

Usage	<ConfigSetMaxDepth *configObject* DEPTH = *depth*>
Arguments	*configObject* is the config object. *depth* is the maximum depth of the linked items of *configObject* to load.
Return value	"0" for success and a non-zero error code on error.

Sets the maximum depth for a config object.

By default, without setting a depth value with this tag, the entire hierarchy of a config object is retrieved.

ConfigSetXsl

Usage	<ConfigSetXsl *configObject* NAME = *stylesheet*>
Arguments	*configObject* is the config object. *stylesheet* is name of the stylesheet.
Return value	"0" for success and a non-zero error code on error.

Sets the xsl style sheet to use when a config object is exported.

ConfigLoad

Usage	<ConfigLoad *configObject*>
Arguments	*configObject* is the config object.
Return value	"0" for success and a non-zero error code on error.

Loads a config object.

Before this tag can be used, the config object must first be created with “ConfigCreate”, then roots added with “ConfigAddRoot”.

ConfigLoadForReleaseManagement

Usage	<ConfigLoadForReleaseManagement *configObject*>
Arguments	*configObject* is the config object.
Return value	"0" for success and a non-zero error code on error.

Loads a config object for release management operations.

The difference between this tag and “ConfigLoad” is that this tag will load all release managed links, even if they are non-hierarchical.

Config object release management tags

ConfigApprove

Usage	<ConfigApprove *configObject*>
Arguments	*configObject* is the config object.
Return value	"0" for success and a non-zero error code on error.

Approves a release configuration.

ConfigDisapprove

Usage	<ConfigDisapprove *configObject* CYCLE = *cycle*>
Arguments	*configObject* is the config object. *cycle* is “TRUE or "FALSE".
Return value	"0" for success and a non-zero error code on error.

Disapproves a release configuration with or without a cyclical link check.

If *cycle* is “FALSE” then <ConfigDisapprove> fails if it detects a cyclical link in the config object. If *cycle* is “TRUE”, the <ConfigDisapprove> succeeds even if it detects a cyclical link but will not work if there are no cyclical links. Here is a way to use this tag:

```
<setval error <ConfigDisapprove config CYCLE="FALSE">>
<if error eq "12553">    <!-- cyclical link detected -->
  <setval error <ConfigDisapprove config CYCLE="TRUE">>
</if>
```

ConfigReplaceItem

Usage	<ConfigReplaceItem *configObject oldItemObject newItemObject*>
Arguments	*configObject* is the config object. *oldItemObject* is the item object to be replaced. *newItemObject* is the item to replace *oldItemObject*.
Return value	"0" for success and a non-zero error code on error.

Replaces an item with a new item in a config object.

During a submit or approve operation, this tag can be used to replace any item in a config object with a different one.

ConfigRerelease

Usage	<ConfigRerelease *configObject*>
Arguments	*configObject* is the config object.
Return value	"0" for success and a non-zero error code on error.

Re-releases a config object.

ConfigSetActionItem

Usage	<ConfigSetActionItem *configObject itemObject* ACTION = *action*>
Arguments	*configObject* is the config object. *itemObject* is the item object upon which to perform an action. *action* is "SUBMIT", "APPROVE", "RERELEASE", or "DISAPPROVE".
Return value	"0" for success and a non-zero error code on error.

Sets the action for an item in a config object before performing a release management operation.

ConfigSubmit

Usage	<ConfigSubmit *configObject*>
Arguments	*configObject* is the config object.
Return value	"0" for success and a non-zero error code on error.

Submits a configuration.

Config object information and reporting tags

ConfigExport

Usage	<ConfigExport *configObject*>
Arguments	*configObject* is the config object to export.
Return value	"0" for success and a non-zero error code on error.

Exports a config object in XML/HTML format.

ConfigGetRootCount

Usage	<ConfigGetRootCount *configObject*>
Arguments	*configObject* is config object.
Return value	the number of root items.

Returns the number of root items in a config object.

ConfigGetRoot

Usage	<ConfigGetRoot *configObject itemObject* INDEX = *index*>
Arguments	*configObject* is the config object. *itemObject* is the returned item object. *index* is the index into the number of root items of *configObject*.
Return value	"0" for success and a non-zero error code on error.

Returns a root item of a config object.

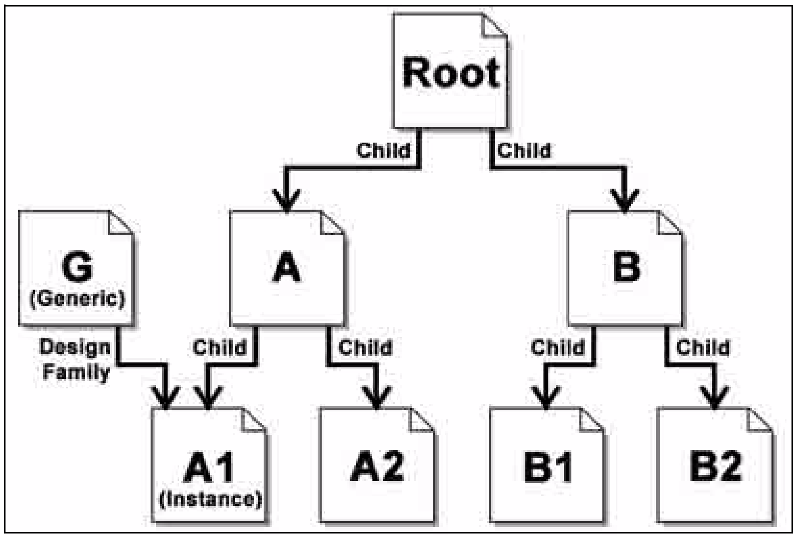
Config iteration object tags

ConfigIteratorCreate

Usage	<ConfigIteratorCreate *configObject itemObject configIter* TYPE = *type*>
Arguments	*configObject* is the config object on whuch the iteration will be generated. *itemObject* is the item object on which the iteration will be performed. *configIter* is the returned config iteration object. *type* is "PRE-ORDER" or "PRE-GENERIC", “POST-ORDER”, “WHEREUSED”
Return value	"0" for success and a non-zero error code on error.

Creates a config iterator object for a config object specific to the type specified.

The *type* argument impacts the order in which links are returned. To illustrate the different *type* argument options, consider the following tree structure:



Creating a config iterator on the “Root” item with *type* set to “PRE-ORDER” will return links in this order:

- Root -- Child --> B
- B-- Child --> B2
- B-- Child --> B1
- Root -- Child --> A
- A-- Child --> A2
- A-- Child --> A1

Creating a config iterator on the “Root” item with *type* set to “PRE-GENERIC” will include “Design Family” links and will return links in this order:

- Root -- Child --> B
- B-- Child --> B2
- B-- Child --> B1
- Root -- Child --> A
- A-- Child --> A2
- A-- Child --> A1
- G -- Design Family -> A1

Creating a config iterator on the “Root” item with *type* set to “POST-ORDER” will return links in this order:

- B -- Child --> B2
- B -- Child --> B1
- Root -- Child --> B
- A -- Child --> A2
- A -- Child --> A1
- Root --Child --> A

Creating a config iterator on the “B1” item with *type* set to “WHERE-USED” will return links in this order:

- B -- Child --> B1
- Root -- Child --> B

ConfigIteratorGetNext

Usage	<ConfigIteratorGetNext *configIter linkObject*>
Arguments	*configIter* is the config iteration object. *linkObject* is the returned link at the new position of *configIter*.
Return value	the depth level of the new config iteration position.

Moves a config iterator object to its next position and returns the link object at that new position and the depth level of the item to which the link points at that new position.

ConfigIteratorIsValid

Usage	<ConfigIteratorIsValid *configIter*>
Arguments	*configIter* is the config iteration object.
Return value	“1” if *configIter* is still valid, “0” if it is not.

Determines if a config iteration object is still valid and can be moved to its next postion.

Item Object BOM tags

ItemHasBom

Usage	<ItemHasBom *itemObject*>
Arguments	*itemObject* is the item object.
Return value	“1” if *itemObject* has linked BOM items, “0” if *itemObject* does not.

Determines whether an item object has items linked as BOM items.

ItemIsBom

Usage	<ItemIsBom *itemObject*>
Arguments	*itemObject* is the item object.
Return value	“1” if *itemObject* is in a “CMS:Parts” class, “0” if *itemObject* is not.

Determines whether an item object is in a “CMS:Parts” class and therefore a BOM.

Chapter 12

Forms

12

Just Ahead:

Loading a form object 158
Form object tags 158
Field object tags 160
Examples. 163

Loading a form object

FormLoad

Usage	<FormLoad *form* ID = *id*>
Arguments	*form* is the returned form object. *id* is the ID of the form.
Return value	"0" on success and a non-zero error code on error.

Loads a form object by its ID.

The tag “ListLoad” (see page 78) can be used to get the IDs of forms.

FormLoadByName

Usage	<FormLoadByName *formObject* NAME = *formName*>
Arguments	*form* is the returned form object. *formName* is the name of the form that will be loaded.
Return value	"0" on success and a non-zero error code on error.

Loads a form object by its name.

FormGetMasterForm

Usage	<FormGetMasterForm *formObject masterFormObject*>
Arguments	*form* is the derived item or link form object. *masterFormObject* is the returned master form of the derived form.
Return value	"0" on success and a non-zero error code on error.

Loads the master form of a derived form.

Form object tags

There are two methods for retrieving information about fields within a form. The first method loads a field object from a form object. Once a field object is obtained, then “Field object tags” (see page 160) can be used to get field information. The second method gets field information directly from the form object.

Method One: Getting field objects from form objects

FormGetFieldCount

Usage	<FormGetFieldCount *formObject* TYPE = *type*>
Arguments	*formObject* is the form object. *type* is the type of attributes to count and can be set to “CUSTOM”, “COMMON” or “ALL”.
Return value	the number of attributes of the specified type.

Returns the number of fields in a form.

FormGetFieldByIndex

Usage	<FormGetFieldByIndex *formObject fieldObject* INDEX = *index*>
Arguments	*formObject* is the form object. *fieldObject* is the returned field object. *index* is the index into the fields available in *formObject*.
Return value	"0" on success and a non-zero error code on error.

Returns a field object from a form object by index.

FormGetField

Usage	<FormGetField *formObject fieldObject* NAME = *fieldName*>
Arguments	*formObject* is the form object. *fieldObject* is the returned field object. *fieldName* is the name of the field object.
Return value	"0" on success and a non-zero error code on error.

Returns a field object from a form object by field name.

Method Two: Getting field information from form objects

FormGetAttrCount

Usage	<FormGetAttrCount *formObject*>
Arguments	*formObject* is the form object.
Return value	the number of custom attributes.

Returns the number of custom attributes in a form.

FormGetAttr

Usage	<FormGetAttr *formObject* PROPERTY = *propertyName* INDEX = *index*>
Arguments	*formObject* is the form object. *propertyName* is the name of the attribute property. Possible values for the property names are: “Name”, “Prompt”, “Type”, “Default Value”, “Is Required”, “Row”, “Column”, “XPos”, “YPos”, “Width”, “Height”, “Prompt Width”, “Stored Len”, “Format”, “Tabletype Id”, “Tabletype Name”, “Choicelist Id”, “Choicelist Name”, “Editable”, “Is Choice”, “Is Table Type”, “Is Searchable” and “CMS ID”. *index* is the index of the attribute.
Return value	the value requested, or a null string.

Returns a property value of a custom attribute.

See table “Properties of Field Objects” on page 161 for the list of acceptable values for *property*.

FormGetFormAttr

Usage	<FormGetFormAttr form Property=*propertyName* INDEX=index>
Arguments	*form* is the form. *propertyName* is the name of the attribute property. Possible values for the property name are: “Form Id”, “Form Name”, “Table Name”, “Is Searchable”. *index* is the index of the attribute.
Return value	the value requested, or a null string.

Returns information about the form.

FormGetCommonAttrCount

Usage	<FormGetCommonAttrCount *formObject*>
Arguments	*formObject* is the form object.
Return value	the number of common attributes.

Returns the number of common attributes in a form.

FormGetCommonAttr

Usage	<FormGetCommonAttr *formObject* PROPERTY = *propertyName* INDEX = *index*>
Arguments	*formObject* is the form object. *propertyName* is the name of the attribute property. *index* is the index of the attribute.
Return value	the property value requested, or a null string.

Returns the property value of a common attribute.

See table “Properties of Field Objects” on page 161 for the list of acceptable values for *property*.

Field object tags

FieldGetAttr

Usage	<FieldGetAttr *fieldObject* ATTR = *propertyName*>
Arguments	*fieldObject* is the field object. *propertyName* is the name of the attribute property.
Return value	the property value requested, or a null string.

Returns the property value of a field object.

The acceptable *propertyName* values are:

Table 12-1: Properties of Field Objects

"is searchable"
"choicelistid"
"colposition"
"CMS ID"
"defaultvalue"
"displaylength"
"editable"
"format"
"height"
"isrequired"
"maxlength"
"name"
"prompt"
"rowposition"
"type"
"width"
"xpos"
"ypos"

FieldGetChoiceTypeChoiceCount

Usage	<FieldGetChoiceTypeChoiceCount *fieldObject*>
Arguments	*fieldObject* is the choice list field object.
Return value	the number of choices.

Returns the number of choices in a choice list field object.

FieldGetChoiceTypeChoiceName

Usage	<FieldGetChoiceTypeChoiceName *fieldObject* INDEX = *index*>
Arguments	*fieldObject* is the choice list field object. *index* is the index of the choice.
Return value	the name of the choice at the specified index.

Returns the name of a choice in a choice list field object.

FieldGetChoiceTypeChoiceDesc

Usage	<FieldGetChoiceTypeChoiceDesc *fieldObject* INDEX = *index*>
Arguments	*fieldObject* is the choice list field object. *index* is the index of the choice.
Return value	the description of the choice at the specified *index*.

Returns the description of a choice in a choice list field object.

FieldGetTableTypeColumnCount

Usage	<FieldGetTableTypeColumnCount *fieldObject*>
Arguments	*fieldObject* is the table-type field object.
Return value	the number of columns in the table-type *fieldObject*.

Returns the number of columns in a table-type field object.

FieldGetTableTypeColumnName

Usage	<FieldGetTableTypeColumnName *fieldObject* INDEX = *index*>
Arguments	*fieldObject* is the table-type field object. *index* is the index of the column.
Return value	the name of the column at the specified index.

Returns the name of a column in a table-type field object.

FieldGetTableTypeColumnWidth

Usage	<FieldGetTableTypeColumnWidth *fieldObject* INDEX = *index*>
Arguments	*fieldObject* is the table-type field object. *index* is the index of the column.
Return value	the width of the column at the specified index.

Returns the width of a column in a table-type field object.

Examples

The followng example shows how to retrieve information on all of the forms in ProductCenter

```
<html>
<body>
    <script>
      <ListLoad formsList TYPE = "System" ATTR = "Forms" VALUE =
   "">
      <SetVal formCnt <ListGetCount formsList>>
      <SetVal formPtr "0">
      var tableData = "<table border=1><tr><th>Form
   Name</th><th>Fields</th><th>Attribute</th><th>Prompt</th><th>
   Type</th></tr>";
      <while formPtr lt formCnt>
        <setVal formName <ListGetDisplayName formsList INDEX =
   formPtr>>
        <FormLoadByName thisForm NAME = formName>
        <FormGetMasterForm thisForm thisMasterForm>
        <SetVal fieldCnt <FormGetAttrCount thisForm>>
        tableData += "<tr><td><bold><PrintVal
   formName></bold></td>";
        tableData += "<td align=right><PrintVal fieldCnt></td><td
   colspan=2></td></tr>";
        <SetVal fieldPtr "0">
        <while fieldPtr lt fieldCnt>
          tableData += "<tr><td colspan=2></td>";
          tableData += "<td><PrintVal <FormGetAttr thisForm
   PROPERTY="NAME"   INDEX = fieldPtr>></td>";
          tableData += "<td><PrintVal <FormGetAttr thisForm
   PROPERTY="PROMPT" INDEX = fieldPtr>></td>";
          tableData += "<td><PrintVal <FormGetAttr thisForm
   PROPERTY="TYPE"   INDEX = fieldPtr>></td>";
          tableData += "</tr>";
          <SetVal fieldPtr <expr fieldPtr + "1">>
        </while>
        <SetVal formPtr <expr formPtr + "1">>
      </while>
      document.write( tableData );
    </script>
  </body>
</html>
```

Appendix A

Administration

Just Ahead:

weblink.cfg 166
database.cfg 166
mimetype 166

This chapter describes the three configuration files that come with WebLink.

weblink.cfg

The weblink.cfg file contains general configuration variables. You can set the following variables in this file:

- file_dir — This variable is for use with the “PrintFile” (see page 58) tag. A path relative to file_dir is entered as an argument to a “PrintFile” statement. WebLink checks file_dir in weblink.cfg to determine an absolute path. The file_dir directory must be within the Web server environment, but it need not be in the CGI directory.
- working_dir — This variable is set equal to the working directory, a temporary directory in which WebLink stores files while it is processing.

Note that there may be obsolete variables, license_timeout, user_timeout and login_page in older weblink.cfg files. The timeout is now fixed at 60 minutes.

In addition, be sure to add the line plugin_exec /cgi-bin to weblink.cfg. This line activates the WebLink plugin on the server side. The command tells WebLink that the browser client is using the plugin to upload or download files and directories. See “Using the WebLink plugin” on page 21 for more detailed information about this plugin.

database.cfg

The database.cfg is generated automatically when you install WebLink, but users can update this file at any time thereafter. This file contains login information that WebLink needs when it logs in to ProductCenter.

Each line of this file pertains to a single database. Each row has four fields:

- The database instance name, as found in the cms_site file.
- The host name where the broker is running.
- The port number where the broker is running.
- The locale, for international purposes. For now, this field should be set to ASCII.

mimetype

The mimetype file maps ProductCenter filetypes to appropriate mime filetypes. This file contains two columns: The first column lists the ProductCenter filetypes and the second column lists the the corresponding mime filetypes. Refer to the *ProductCenter Administrator Guide* for more information on ProductCenter filetypes.

Index

A

ActiveX control 21
ActivityClaim 141
ActivityDisapprove 142
ActivityGetAssignmentActionType 140
ActivityGetAssignmentChoiceID 140
ActivityGetAssignmentCount 139
ActivityGetAssignmentCountByType 139
ActivityGetAssignmentName 140
ActivityGetAssignmentNameByType 140
ActivityGetAssignmentStatus 141
ActivityGetAssignmentType 141
ActivityGetAttr 138
ActivityGetItem 138
ActivityGetProcess 138
ActivityGetReassignCount 139
ActivityInstGetNextActivities 139
ActivityLoad 137
ActivityPlaceOnHold 142
ActivityReassignFromUserToUser 142
ActivityResume 142
ActivitySendBackward 143
ActivitySendForward 143
ActivitySetAssignment 141
ActivitySuspend 143
ActivityToggleHoldForUser 144
ActivityTurnOffHold 144
architecture 16

B

BomItemAddBomItem 149
BomItemGetBomItemAttr 146
BomItemGetBomItemAttrs 146
BomItemGetBomItemByIndex 147
BomItemGetBomItemCount 147
BomItemGetBomItemLinkByIndex 148
BomItemGetItem 146
BomItemGetLinkById 148
BomItemGetReferenceCount 148
BomItemGetReferenceItemByIndex 148
BomItemLoad 146
BomItemRemoveBomItemByLink 149
BomItemRemoveReferenceByIndex 149
BomItemRetrieveForMassChange 149
BomItemSetBomItemAttr 147
Break 46

C

cgi-bin 16
ChangePassword 68
comments 54
ConfigAddRoot 150
ConfigApprove 152
ConfigCreate 150
ConfigDisapprove 152
ConfigExport 153
ConfigGetRoot 153
ConfigGetRootCount 153
ConfigIteratorCreate 154
ConfigIteratorGetNext 155
ConfigIteratorIsValid 155
ConfigLoad 151
ConfigLoadForReleaseManagement 151
ConfigReplaceItem 152
ConfigRerelease 152
ConfigSetActionItem 153
ConfigSetBaseline 150
ConfigSetExport 151
ConfigSetMaxDepth 151
ConfigSetXsl 151
ConfigSubmit 153
CreateHelperSession 55

D

database.cfg 16, 166
DefVal 42

directory structure 16

E

EncodeVal 46
ErrorGetLastMessage 56
Exec 56
Expr 43

F

FieldGetAttr 160
FieldGetChoiceTypeChoiceCount 161
FieldGetChoiceTypeChoiceDesc 162
FieldGetChoiceTypeChoiceName 161
FieldGetTableTypeColumnCount 162
FieldGetTableTypeColumnName 162
FieldGetTableTypeColumnWidth 162
FileAccess 49
file_dir 166
files
    database.cfg 16, 166
    mimetype 16, 166
    msg_db.db 17
    msg_db.idx 17
    plugin.cgi 16, 26
    webclient 16
    weblink.cfg 16, 166
    weblink.cgi 16
FormGetAttr 159
FormGetAttrCount 159
FormGetCommonAttr 160
FormGetCommonAttrCount 160
FormGetField 159
FormGetFieldByIndex 159
FormGetFieldCount 158
FormGetFormAttr 160
FormGetMasterForm 158
FormLoad 158
FormLoadByName 158

G

GetAttr 160
GetDatabaseEntries 49
GetLength 47
GetMessage 57
GetResourceVariable 57
GetScreenPermissions 65
GetSessionVal 57
GetStyleSheetListing 50
GetSubString 47
GetValueFromFile 50
GroupAddUser 64
GroupCreate 64
GroupDelete 64
GroupGetAttr 62
GroupGetAttrCount 62
GroupGetAttrNameByIndex 62
GroupGetUser 63
GroupGetUserCount 62
GroupIsUserMember 63
GroupLoad 63
GroupLoadByName 63
GroupRemoveUser 64
GroupSave 64
GroupSetAttr 65

H

htdocs 16

I

If-elseif-else 44
interface
    C 13
    C++ 13
    Perl 13
    WebLink 13
IsValidSession 57
ItemAdd 92
ItemAddHeadLink 107
ItemAddLink 107
ItemAddLinkHeadTotail 108
ItemAddLinkTailToHead 108
ItemAddSaveAs 93
ItemAddTailLink 108
ItemAddToDesktop 92
ItemAlter 96
ItemCheckin 93
ItemCheckout 93
ItemCheckoutNoDeliver 94
ItemCheckVersionsForRemoval 95
ItemCreate 84
ItemDelete 95
ItemDeleteFromDesktop 92
ItemDeleteFromDesktopById 92
ItemDeleteTableTypeAttrRow 85
ItemDemote 97
ItemGetAccessCount 87

ItemGetAccessNameByIndex 87
ItemGetAccessPermByIndex 87
ItemGetAttr 88
ItemGetCheckoutStatus 98
ItemGetCopy 94
ItemGetEmbeddedObjectURL 89
ItemGetHeadLink 112
ItemGetHeadLinkCount 109
ItemGetLatest 84
ItemGetLink 110
ItemGetLinkById 110
ItemGetLinkCount 110
ItemGetLinkDefSideCount 113
ItemGetLinkDefSideProperty 113
ItemGetLinkedItem 111
ItemGetLinkExists 111
ItemGetLinkSide 112
ItemGetLinkSideCount 111
ItemGetLinkTypeCount 113
ItemGetLinkTypeIsHead 114
ItemGetLinkTypeLabel 114
ItemGetLinkTypeName 113
ItemGetNextRev 90
ItemGetNextRevision 90
ItemGetNextRevisionCount 90
ItemGetNextRevisionGetParent 91
ItemGetNextRevisionIsLegacy 91
ItemGetNextRevisionIsMinor 91
ItemGetObsoleteTailList 112
ItemGetRevisionFlag 89
ItemGetRowCount 89
ItemGetTailLink 112
ItemGetTailLinkCount 112
ItemHasBom 156
ItemHasVersionConflicts 89
ItemIsBom 156
ItemIsFile 98
ItemIsLatest 98
ItemIsObsolete 98
ItemIsType 97
ItemLoad 84
ItemLoadAttachedProcessList 137
ItemLoadForm 90
ItemObsolete 97
ItemPromote 96
ItemPurge 95
ItemPurgeLatest 95
ItemReinstate 97
ItemRemoveLinkById 108
ItemRollback 96
ItemSetAccessPerm 85
ItemSetAttr 86
ItemSetFile 86
ItemSetForm 86
ItemSetTransactionLog 86
ItemUndoCheckout 93
ItemUpdateLinkById 109

L

LinkCreate 104
LinkDeleteTableTypeAttrRow 106
LinkGetAttr 106
LinkGetForm 104
LinkGetHead 104
LinkGetRowCount 106
LinkGetTail 105
LinkLoadForm 105
LinkSetAttr 107
LinkSetForm 105
ListGetCount 80
ListGetDisplayName 80
ListGetSystemId 80
ListLoad 78
Login 54, 55
Logout 55

M

mimetype 16, 166
msg_db.db 17
msg_db.idx 17

P

plugin 21
  administrator tasks 21
  components related to 23
  ItemRemove() 24
  ItemRemoveFromWorkingDir() 24
  ItemSave() 23
  ItemSaveToWorkingDir() 23
  ItemUpload() 25
  plugin.js 23
plugin.cgi 16, 26
plugin_exec 22, 166
PrintFile 58
Printval 43
ProcessDefGenerateUniqueName 132
ProcessDefGetAttr 131
ProcessDefGetNextActivities 132
ProcessDefLoad 131
ProcessDefRouteItem 133
ProcessDefRouteItemWithFlags 133

ProcessGetAttr 134
ProcessGetAuditLogAttr 135
ProcessGetAuditLogListCount 135
ProcessGetCompletedActivityCount 135
ProcessGetCompletedActivty 135
ProcessGetItem 135
ProcessGetProcessDef 134
ProcessGetProcessListAttr 130
ProcessGetProcessListCount 130
ProcessIsCoordinator 137
ProcessLoad 133
ProcessLoadForProcessList 130
ProcessViewerGetAttr 136
ProcessViewerGetListCount 136

Q

QueryAddClause 118
QueryClearAllClauses 118
QueryCreate 116
QueryDelete 122
QueryExecute 121
QueryExecuteMatchCase 122
QueryGetActivityInst 124
QueryGetActivityInstCount 123
QueryGetCaseSensitive 120
QueryGetClauseCount 120
QueryGetClauseData 120
QueryGetItem 122
QueryGetItemCount 122
QueryGetOp 120
QueryGetProcessInst 124
QueryGetProcessInstCount 124
QueryGetType 121
QueryLoad 123
QueryLoadById 123
QuerySave 123
QuerySetCaseSensitive 119
QuerySetOp 119
QuerySetType 119

R

ReportExecute 125
ReportExecuteMatchCase 125
ReportGetAttr 126
ReportGetAttrcount 126
ReportGetAttrNameByIndex 126
ReportGetQuery 126
ReportLoad 126
ReportLoadByName 127
ReportSetAttr 127
ReportSetExport 127
ReportSetXslPath 127

S

SaveToFile 51
SessionGetUser 65
SetClientType 54
SetMimeType 59
SetNewPassword 68
SetSessionVal 59
SetVal 42
SplitString 48

U

UnfarcFile 51
URLDecode 49
URLEncode 48
UserGetAttr 66
UserGetAttrCount 65
UserGetAttrNameByIndex 66
UserGetGroup 66
UserGetGroupCount 66
UserGetScreenPermission 67
UserGetScreenPermissionCount 66
UserGetScreenPermissionNameByIndex 67
UserIsMemberOf 67
UserLoad 67
UserLoadByName 67
UserSetAttr 69
UserSetScreenPermission 69

V

VerifyPassword 68

W

webclient 16
weblink.cfg 16, 166
weblink.cgi 16
While 45
WmlDecrypt 69
WmlEncrypt 69
wml_plugin 23
wml_url 23
working_dir 166

SofTech Inc.

CORPORATE HEADQUARTERS:

650 Suffolk St., Suite 415
Lowell, MA 01854 - USA

CUSTOMER SUPPORT:

Telephone:	(978) 513-2698
E-Mail:	productcenter@softech.com
Online Support:	http://softech.com/productcenter-support