PRODUCT LIFECYCLE MANAGEMENT

ProductCenter®

AT THE CENTER OF YOUR ENGINEERING SUCCESS

ProductCenter C/C++ & Perl 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 C/C++ & Perl Toolkits Programmer Guide

About this book

ProductCenter

This manual explains how to develop application programs in C, C++, or Perl 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 C, C++, or Perl, 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.

Chapter 1

Introduction

Just Ahead:

Welcome to the ProductCenter Toolkits

The ProductCenter Toolkits (formerly known as the *API*, or Application Programming Interface) 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).

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 C, C++, and Perl functions that you can use to tailor the product to your own purposes.

WebLink is documented separately in the *ProductCenter WebLink 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 the ProductCenter Toolkits 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.

NOTE: ProductCenter code examples are typically written using C syntax, although they are written for the C++ compiler.

Perl — The Perl interface gives you access to the same ProductCenter functionality as the C and C++ interfaces.

NOTE: The Perl Toolkit also provides a *convenience layer*, which is a set of scripts that combine basic Perl Toolkit functions into efficient calls to save programming time. See Appendix A, "Convenience Layer Functions" for more information.

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). These calls are documented in the *ProductCenter WebLink Toolkit Programmer Guide*.

Operating systems and compilers for C/C++

The ProductCenter C/C++ Toolkit has been built on several platforms. Table 1-1 lists the supported operating systems, as well as the compiler used to build the Toolkit on each platform. You should check the release notes for the most up-to-date information concerning supported platforms.

Table 1-1: Supported environments for ProductCenter C/C++ Toolkit

Operating System	Supported Compiler Version	Notes
Windows 7 x64 Windows 8.1 x64 Windows 10 x64 Windows Server 2008 R2 Windows Server 2012 R2	Microsoft Visual Studio 2008	
Solaris 10	Sun WorkShop 6 Update 1 C++ 5.2	If upgrading from ProductCenter 7.x, you must use the new compiler when updating applications previously built with g++.

NOTE: You may program the ProductCenter C/C++ Toolkit in either C or C++, but you must compile with a C++ compiler as there are no API libraries available for the C compiler.

Perl support

Below is a table with Perl compatibility support for each operating system:

Table 1-2: Perl Toolkit Information

Operating System	Supported PERL Version
Windows 7 x64 Windows 8.1 x64 Windows 10 x64 Windows Server 2008 R2 Windows Server 2012 R2	PERL 5.20
Solaris 10	PERL 5.14

ProductCenter Toolkit architecture

Figure 1-1 shows the main behind-the-scenes activities in the ProductCenter Toolkit. At the far left is the Toolkit library. You use the library file and its related header files to create your own programs. The library allows you to create a link across the network and connect to the ProductCenter server. This link is the *connection object*, which we discuss in greater detail in Chapter 4, “Connections, Status, and Settings”.

When you create a Toolkit program, the program talks to a broker. The broker then starts a ProductCenter server, which talks to a ProductCenter database. A ProductCenter broker and server *must* be running on a Windows or UNIX workstation before you can begin creating programs.

To connect to a database, you also must know:

- the server database name
- the server host name
- the server port number
- a Username
- a Password

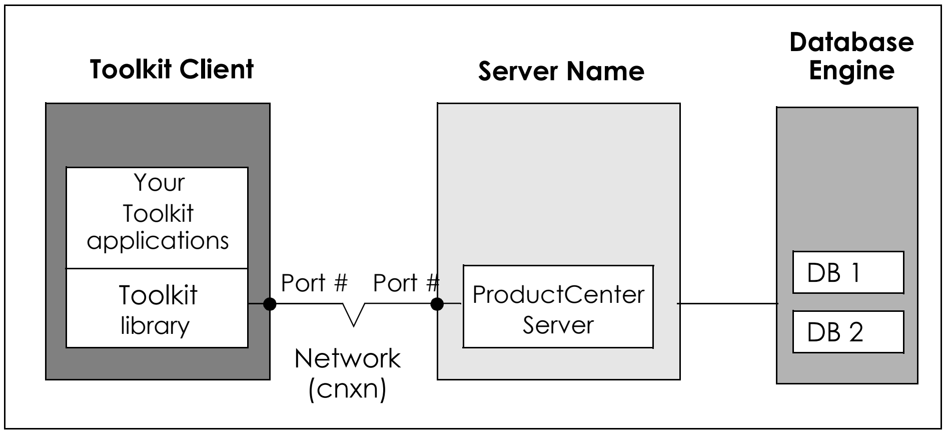


Figure 1-1: ProductCenter Toolkit architecture

So what does your code need to do? Every ProductCenter Toolkit program you write should:

- Create a connection object with the specified server name, server port number, and database name
- Log in with the appropriate user name and password
- Perform the tasks
- Disconnect from the server once the main part of the program has completed
- Delete the connection to reclaim memory

Perl convenience layer functions

The calls documented in the following chapters are the basic Perl and C/C++ Toolkit calls.

However, the Perl Toolkit also provides a number of *convenience layer* functions. These are “shortcuts”, implemented by the ProductCenter Perl Toolkit itself, and documented in Appendix A, “Convenience Layer Functions”.

If you see a call used in an example that does not appear in Chapters 2 -12, it is probably a convenience layer function listed in Appendix A.

Chapter 2

Getting Started

2

Just Ahead:

This chapter describes how to use the supplied example programs to quickly become familiar with the toolkit environment.

Information about the C/C++ Toolkit starts below. Information about the Perl Toolkit starts on page 32.

Getting started with the C/C++ on Windows

You must use Microsoft Visual Studio 2008 to build your C/C++ Toolkit programs.

The ProductCenter C/C++ Toolkit comes with two demo programs to get you started:

- example demo
- monitor demo

Both of these come in both C and C++ versions. This chapter describes how to use the example demo. Use similar procedures for working with the monitor demo.

The example demo comes complete with a Visual Studio Project/Solution (.sln) file, and is an excellent example of how to write an application across multiple source files, and then compile and link them using Microsoft Visual Studio. Use this demo to acquaint yourself with the Toolkit build environment and procedures on Windows.

The example demo exercises most of the functionality available through the ProductCenter Toolkits, and is an excellent source for seeing how the various function calls can be woven together into a useful application.

Once you succeed in running example, begin writing your own applications, using example as a guide.

To build the example demo

The Toolkit comes with two versions of the example demo.

The C version is in:

```
ProductCenter_home\toolkits\c\Example
```

The C++ version is in:

```
ProductCenter_home\toolkits\c++\Example
```

To build and test this demo (C++ version shown):

1. Open Windows Explorer and navigate to

   ```
   ProductCenter_home/toolkits/c++
   ```

2. It is always good to make a backup of an example directory before changing anything. In this exercise we will not be suggesting you change the code. If you are planning making changes to the example programs you should make a backup of the directory now.
3. In folder example, double-click **CPP_Toolkit_Example.sln**. This invokes Microsoft Visual Studio.
4. You should now see a Microsoft Visual Studio window similar to Figure 2-2. Go to the **Build** menu and click **Build CPP_Toolkit_Example**.

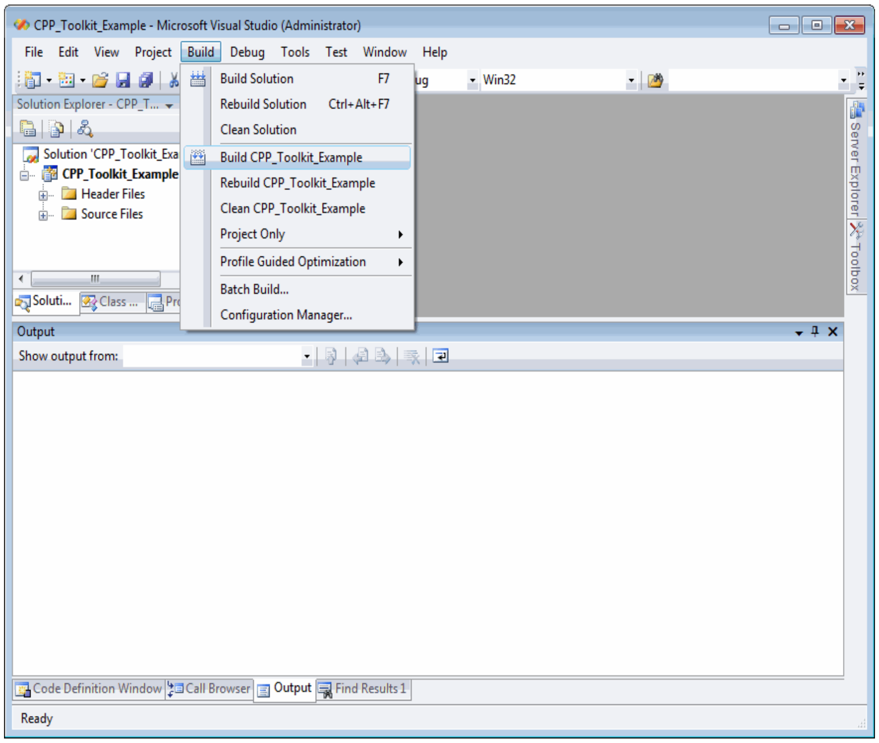
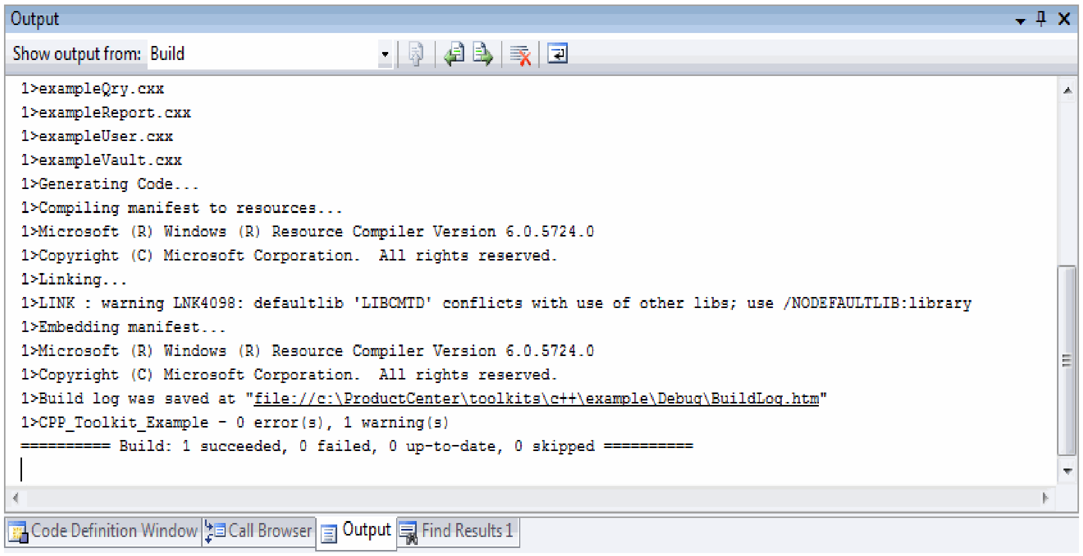

CPP_Toolkit_Example - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Tools Test Window Help
Build Solution F7
Rebuild Solution Ctrl+Alt+F7
Clean Solution
Build CPP_Toolkit_Example
Rebuild CPP_Toolkit_Example
Clean CPP_Toolkit_Example
Project Only
Profile Guided Optimization
Batch Build...
Configuration Manager...
Win32
Solution Explorer - CPP_T...
Solution 'CPP_Toolkit_Exa
CPP_Toolkit_Example
Header Files
Source Files
Output
Show output from:
Code Definition Window
Call Browser
Output
Find Results 1
Ready
Server Explorer
Toolbox

Figure 2-2: Visual C++ window

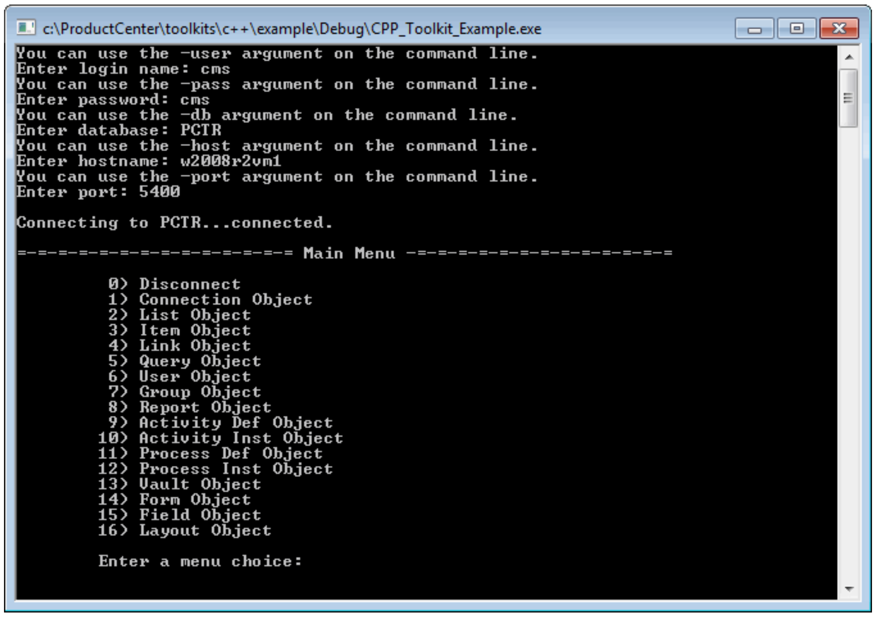
5. You should now see messages reporting the results of the compilation and link operations, as shown in Figure 2-3. One or two warnings may appear; these can usually be ignored. If you see any errors, check to make sure that Visual Studio and the ProductCenter Toolkit are installed correctly.



```
1>exampleQry.cxx
1>exampleReport.cxx
1>exampleUser.cxx
1>exampleVault.cxx
1>Generating Code...
1>Compiling manifest to resources...
1>Microsoft (R) Windows (R) Resource Compiler Version 6.0.5724.0
1>Copyright (C) Microsoft Corporation.  All rights reserved.
1>Linking...
1>LINK : warning LNK4098: defaultlib 'LIBCMTD' conflicts with use of other libs; use /NODEFAULTLIB:library
1>Embedding manifest...
1>Microsoft (R) Windows (R) Resource Compiler Version 6.0.5724.0
1>Copyright (C) Microsoft Corporation.  All rights reserved.
1>Build log was saved at "file://c:\ProductCenter\toolkits\c++\example\Debug\BuildLog.htm"
1>CPP_Toolkit_Example - 0 error(s), 1 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Figure 2-3: Visual C++ compilation messages

6. To run the example in the MS Visual Studio, press F5.
7. You should see the example menu appear. At the prompts, enter values for broker host, port, database name, user name, and password (the program displays reminders that you can enter these values at the command line). After a few seconds, the demo program displays a menu for exercising various functions. See Figure 2-4 for details.


c:\ProductCenter\toolkits\c++\example\Debug\CPP_Toolkit_Example.exe

```
You can use the -user argument on the command line.
Enter login name: cms
You can use the -pass argument on the command line.
Enter password: cms
You can use the -db argument on the command line.
Enter database: PCTR
You can use the -host argument on the command line.
Enter hostname: w2008r2vm1
You can use the -port argument on the command line.
Enter port: 5400

Connecting to PCTR...connected.

=-=-=-=-=-=-=-=-=-=-=-=-=-= Main Menu -=-=-=-=-=-=-=-=-=-=-=-=-=-=

         0) Disconnect
         1) Connection Object
         2) List Object
         3) Item Object
         4) Link Object
         5) Query Object
         6) User Object
         7) Group Object
         8) Report Object
         9) Activity Def Object
        10) Activity Inst Object
        11) Process Def Object
        12) Process Inst Object
        13) Vault Object
        14) Form Object
        15) Field Object
        16) Layout Object

        Enter a menu choice:
```

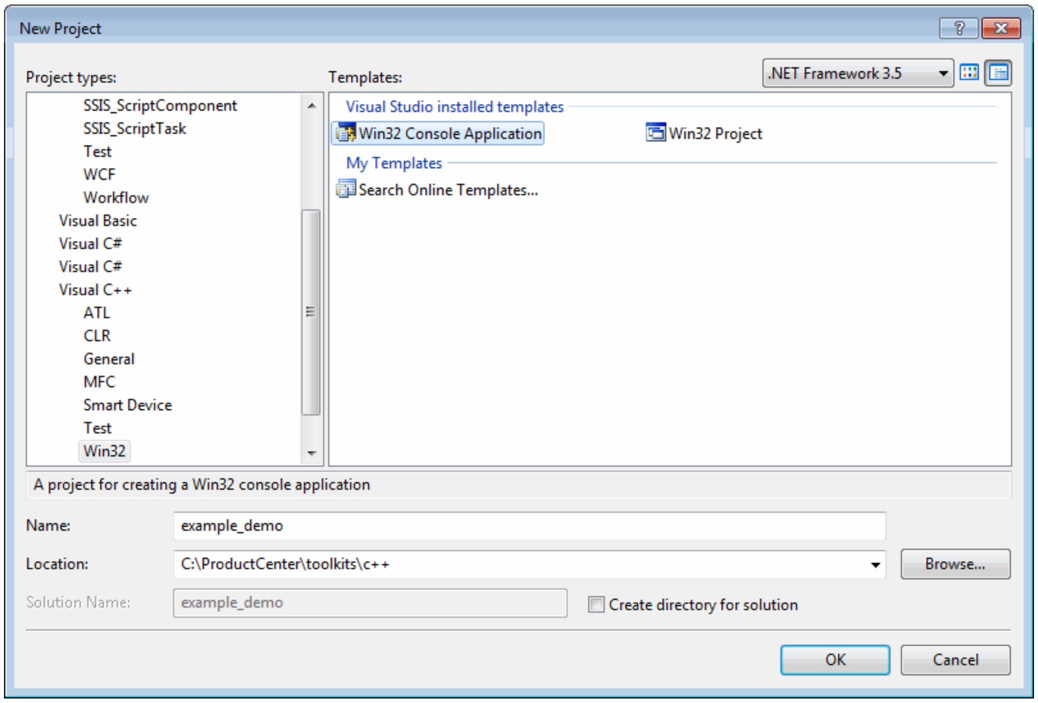
Figure 2-4: Example menu

8. Explore this example by trying different menu options. Entering “0” from the main menu will disconnect and terminate the application. Entering “0” from any other menu will return you to the previous menu.

To build your own example project

Once you have built and run the example project, you are ready to build your own project.

1. Open Microsoft Visual Studio 2008 on your machine and choose **File** → **New** → **Project**.
2. In **Visual C++** → **Win32** project type of the New Project window, click Win32 Console Application. Enter a name for your project into the Name field. Visual Studio automatically uses the project name as the name of the directory it will create for your files. Note that you can change to a different directory by entering its name into the Location field. We strongly suggest that you choose a location which is new; please do not use the example location or a location with another project. Uncheck the **Create directory for solution** box as shown in Figure 2-5, then click OK.



A project for creating a Win32 console application

Name: example_demo

Location: C:\ProductCenter\toolkits\c++

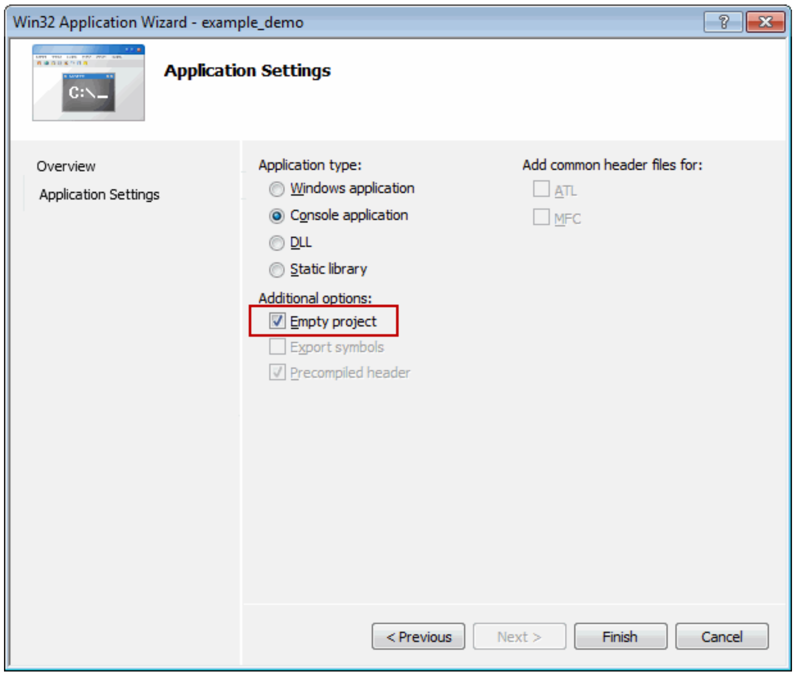
Solution Name: example_demo

☐ Create directory for solution

Figure 2-5: Windows 32 Console Application - New Project

3. Select **Next** in the Welcome to the Win32 Application Wizard window.

4. In the Application Settings window as shown in Figure 2-6, ensure that **Empty project** is selected, then click Finish.



Application type:
- ( ) Windows application
- (●) Console application
- ( ) DLL
- ( ) Static library

Additional options:
- [✓] Empty project
- [ ] Export symbols
- [✓] Precompiled header

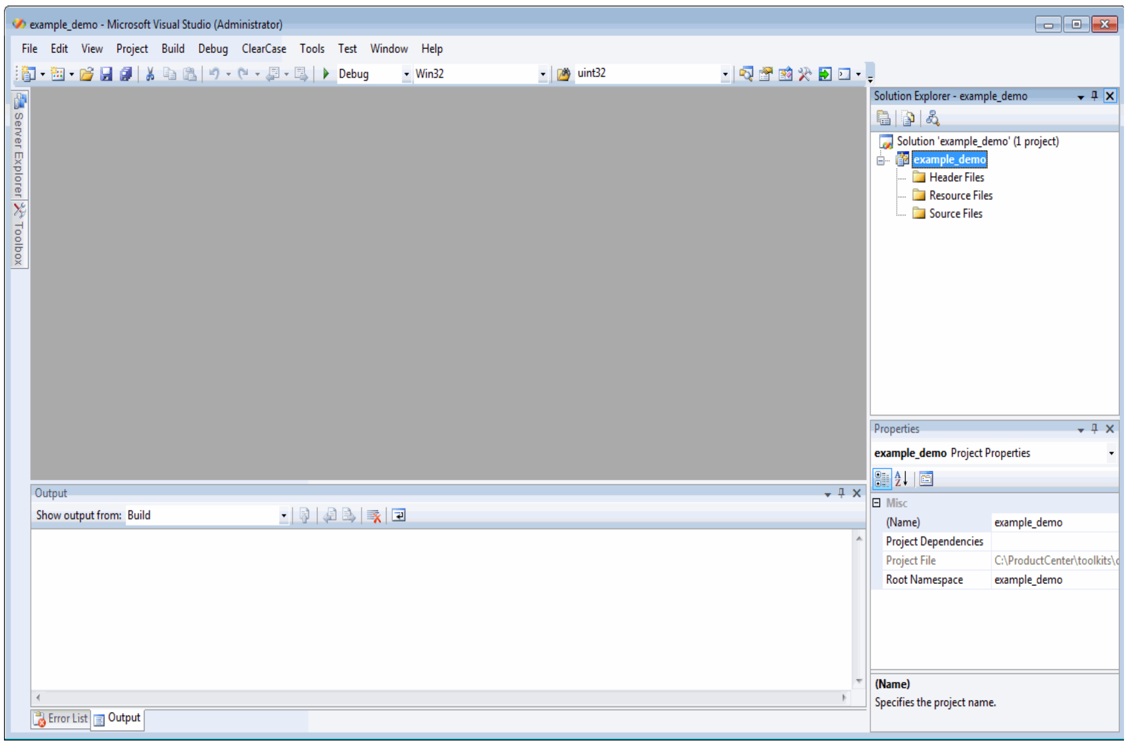
Add common header files for:
- [ ] ATL
- [ ] MFC

Figure 2-6: Application Settings

5. You should now have the following four files in your example_demo directory:

- *project_name*.suo - The solution user options file.
- *project_name*.sln - The solution file.
- *project_name*.ncb - The browser.
- *project_name*.vcproj - The project file contains settings on how to compile your code.

Your Visual Studio file view window should resemble Figure 2-7.


example_demo - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug ClearCase Tools Test Window Help
Debug
Win32
uint32
Server Explorer
Toolbox
Solution Explorer - example_demo
Solution 'example_demo' (1 project)
example_demo
Header Files
Resource Files
Source Files
Output
Show output from: Build
Properties
example_demo Project Properties
Misc
Error List
Output

(Name) example_demo
Project Dependencies
Project File C:\ProductCenter\toolkits\c
Root Namespace example_demo

(Name)
Specifies the project name.

Figure 2-7: Developer Studio window with new, unpopulated project

6. Next, add to your project the C++ version of the example files. From the Solution Explorer pane, RMB on Source Files and select **Add → Existing Item** as shown in Figure 2-8. When the Add Existing Item window appears, navigate to:

   ProductCenter_home\toolkits\c++\example

   Select all .cxx files and click OK.

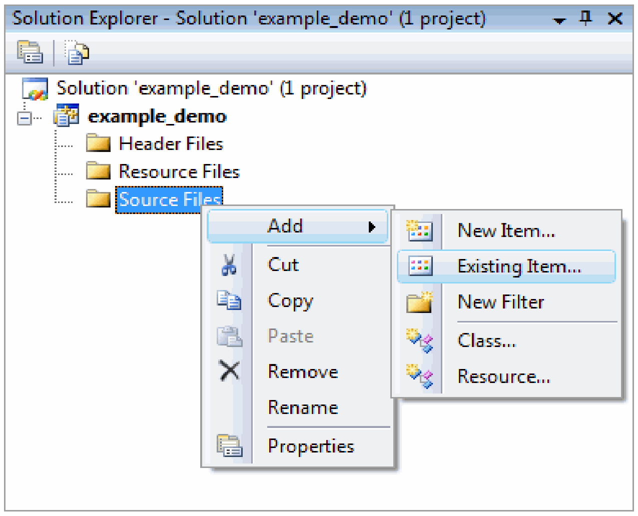
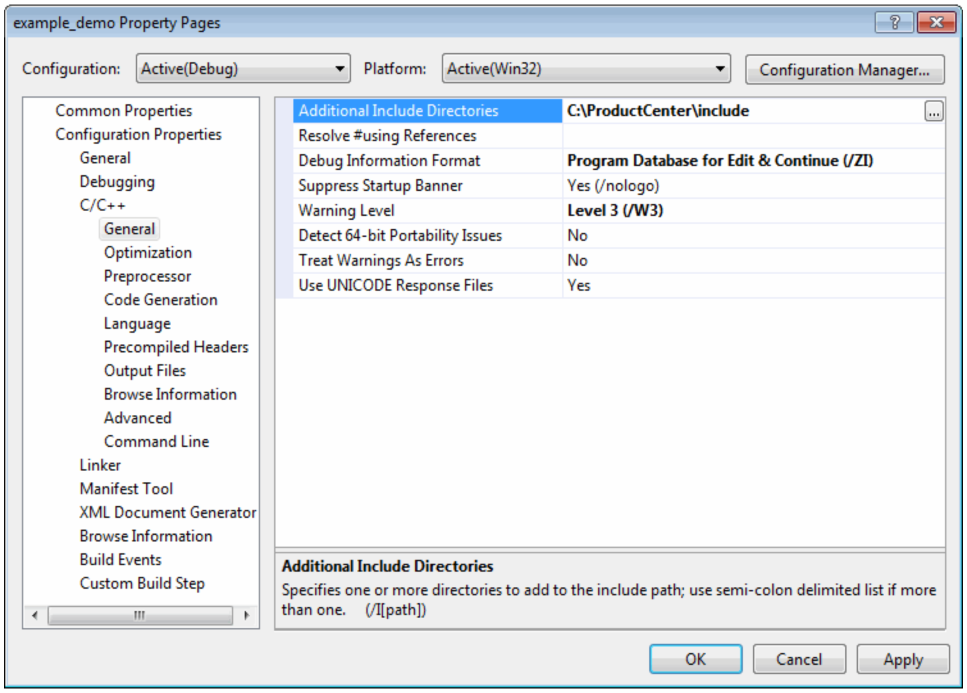


Figure 2-8: Solution Explorer window, C/C++

7. Next, From the Project menu select "example_demo Properties", then select **C/C++→ General** and add C:\ProductCenter\include to the Additional Include Directories as shown in Figure 2-9.



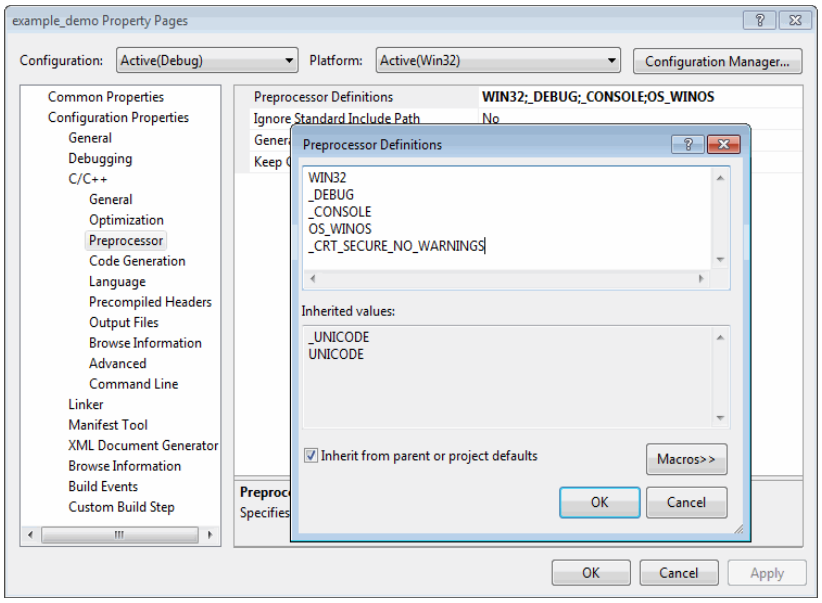
Configuration: Active(Debug) Platform: Active(Win32)

Additional Include Directories	C:\ProductCenter\include
Resolve #using References	
Debug Information Format	Program Database for Edit & Continue (/ZI)
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level 3 (/W3)
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

Figure 2-9: Property Pages window

8. Next, select **C/C++ → Preprocessor** and append OS_WINOS and _CRT_SECURE_NO_WARNINGS as shown in Figure 2-10.



Configuration: Active(Debug) Platform: Active(Win32)

Preprocessor Definitions	WIN32;_DEBUG;_CONSOLE;OS_WINOS
Ignore Standard Include Path	No

```
WIN32
_DEBUG
_CONSOLE
OS_WINOS
_CRT_SECURE_NO_WARNINGS
```

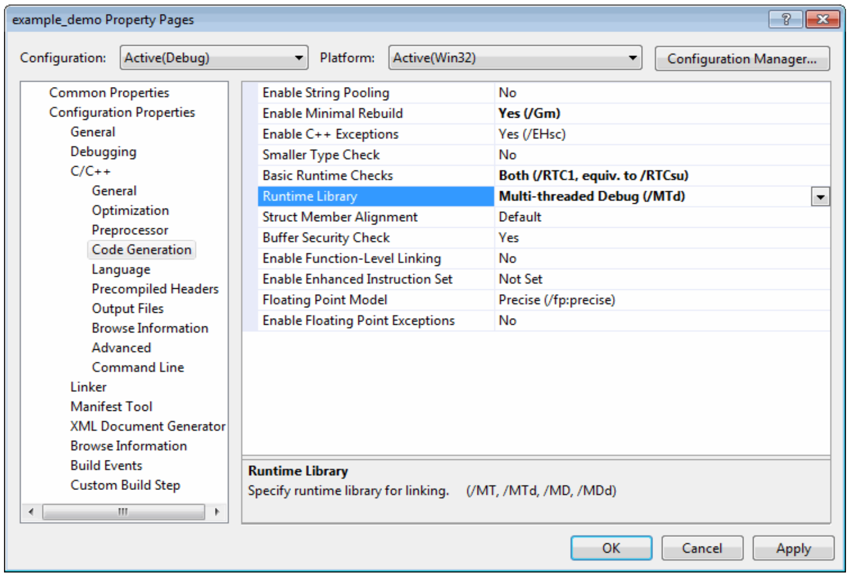
Inherited values:

```
_UNICODE
UNICODE
```

☑ Inherit from parent or project defaults

Figure 2-10: Property Pages, Preprocessor Definitions

9. Next, select **C/C++** → **Code Generation** and change the Runtime Library to Multi-threaded Debug (/MTd) as shown in Figure 2-11.


example_demo Property Pages

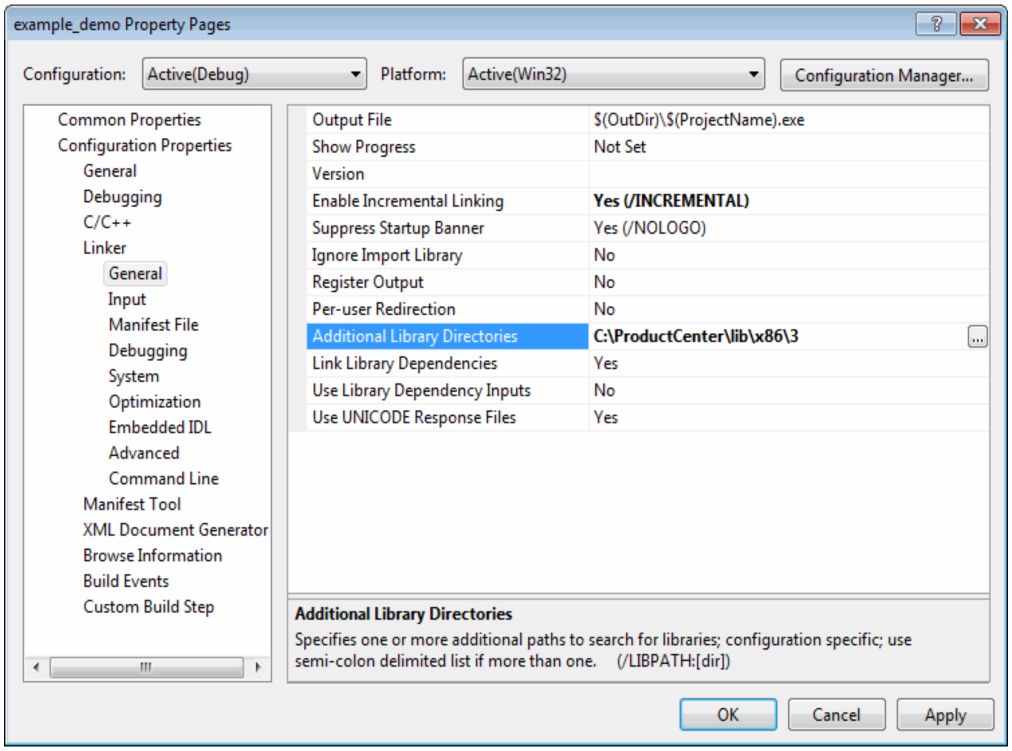
Configuration: Active(Debug) Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	**Yes (/Gm)**
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	**Both (/RTC1, equiv. to /RTCsu)**
Runtime Library	**Multi-threaded Debug (/MTd)**
Struct Member Alignment	Default
Buffer Security Check	Yes
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd)

Figure 2-11: Property Pages, Runtime Library

10. Next, select **Linker** → **General** and add C:\ProductCenter\lib\x86\3 to Additional Library Directories as shown in Figure 2-12.


example_demo Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

Output File	$(OutDir)\$(ProjectName).exe
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**C:\ProductCenter\lib\x86\3**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Use UNICODE Response Files	Yes

Additional Library Directories
Specifies one or more additional paths to search for libraries; configuration specific; use semi-colon delimited list if more than one. (/LIBPATH:[dir])

Figure 2-12: Property Pages, Additional Library Directories

11. Next, select **Linker** → **Input** and add the following libraries as shown in Figure 2-13:

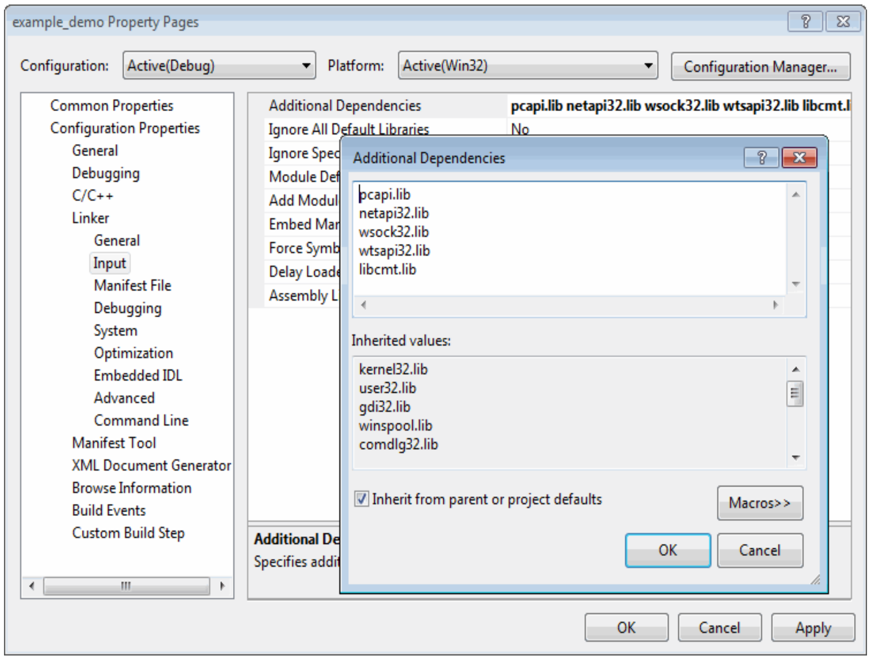
    pcapi.lib

    wsock32.lib

    netapi32.lib

    wtsapi32.lib

    libcmt.lib.



Additional Dependencies: pcapi.lib netapi32.lib wsock32.lib wtsapi32.lib libcmt.l

Ignore All Default Libraries: No

Additional Dependencies

```
pcapi.lib
netapi32.lib
wsock32.lib
wtsapi32.lib
libcmt.lib
```

Inherited values:

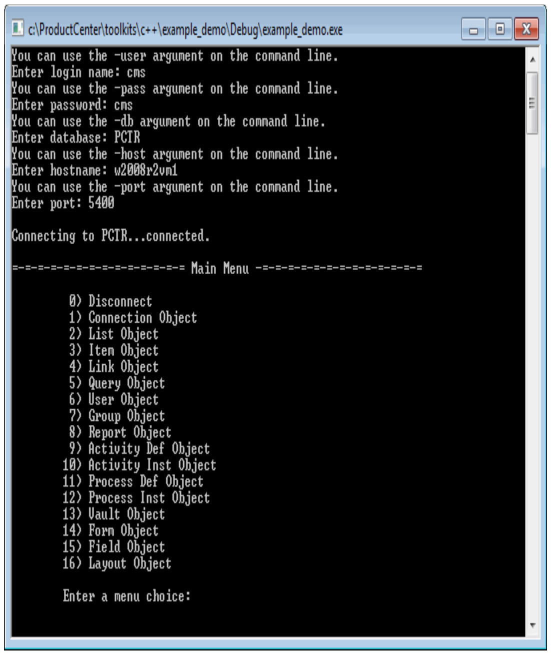
```
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
```

☑ Inherit from parent or project defaults

Figure 2-13: Property Pages, Additional Dependencies

12. Now compile the program by selecting Build Solution from the Build menu.
13. You can run the example demo from within Microsoft Visual Studio 2008 with the F5 or CTRL-F5 keys, or from the Debug menu:

    Debug → Start Debugging
14. In the resulting command window, plug in values for login name, password, database name, host name, and port, as shown in Figure 2-14 (the application reminds you these values can be entered at the command line when invoking the example). Explore the demo by entering various menu choices.



c:\ProductCenter\toolkits\c++\example_demo\Debug\example_demo.exe

```
You can use the -user argument on the command line.
Enter login name: cms
You can use the -pass argument on the command line.
Enter password: cms
You can use the -db argument on the command line.
Enter database: PCTR
You can use the -host argument on the command line.
Enter hostname: w2008r2vm1
You can use the -port argument on the command line.
Enter port: 5400

Connecting to PCTR...connected.

=-=-=-=-=-=-=-=-=-=-=-=-=-= Main Menu -=-=-=-=-=-=-=-=-=-=-=-=-=-=

         0) Disconnect
         1) Connection Object
         2) List Object
         3) Item Object
         4) Link Object
         5) Query Object
         6) User Object
         7) Group Object
         8) Report Object
         9) Activity Def Object
        10) Activity Inst Object
        11) Process Def Object
        12) Process Inst Object
        13) Vault Object
        14) Form Object
        15) Field Object
        16) Layout Object

        Enter a menu choice:
```

Figure 2-14: Exercising the demo in a command window.

We strongly encourage you to examine the source code in the example files and use it as a starting point for your own custom applications.

Getting started with the C/C++ on UNIX

You may program the ProductCenter C/C++ Toolkit in either C or C++, but you must compile with a C++ compiler as there are no API libraries available for the C compiler.

Typical compilers for the supported UNIX platforms are:

- Sun Solaris: Sun Workshop 6, update 1, C++ 5.2

See Table 1-1 on page 16 for details about supported UNIX platforms; check the Release Notes for any recent changes to this information.

Example programs

The ProductCenter C/C++ Toolkit comes with two demo programs to get you started:

- “example” demo
- “monitor” demo

These demo programs come in both C and C++ versions. Both come with a Makefile.

The example demo exercises most of the functionality available through the ProductCenter Toolkits, and is an excellent example of how to write an application across multiple source files, and then compile and link them using a Makefile. Use this demo to acquaint yourself with Toolkit build environments and procedures.

The monitor demo is much smaller and is of use to those who wish to write an Application Queue Manager (AQM) monitor application. This type of application allows you to trigger execution from a ProductCenter event. AQM is described in Chapter 11.

Once you succeed in running the example demo, begin writing your own applications, using example as a guide.

To use the example demo

The Toolkit comes with two versions of the example demo the examples use the csh shell.

The C version is in:

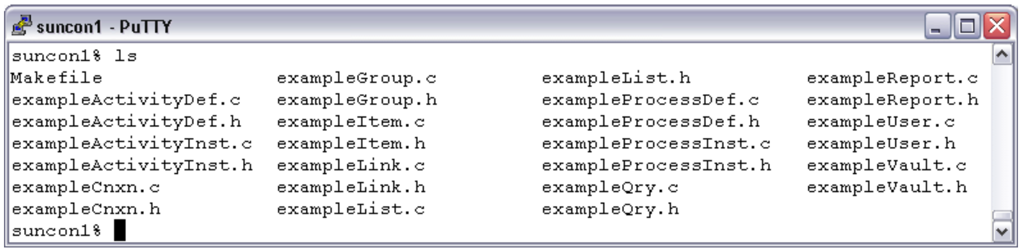
`$CMS_HOME/toolkits/c/Example`

The C++ version is in:

`$CMS_HOME/toolkits/c++/Example`

To build and test this demo:

1. **cd** to the appropriate directory.
2. Use the **ls** command display the contents of this directory. If you are in the C version of the demo directory path, as shown in Figure 2-15.Figure 2-15



```
suncon1% ls
Makefile                exampleGroup.c          exampleList.h           exampleReport.c
exampleActivityDef.c    exampleGroup.h          exampleProcessDef.c     exampleReport.h
exampleActivityDef.h    exampleItem.c           exampleProcessDef.h     exampleUser.c
exampleActivityInst.c   exampleItem.h           exampleProcessInst.c    exampleUser.h
exampleActivityInst.h   exampleLink.c           exampleProcessInst.h    exampleVault.c
exampleCnxn.c           exampleLink.h           exampleQry.c            exampleVault.h
exampleCnxn.h           exampleList.c           exampleQry.h
suncon1% 
```

Figure 2-15: Listing of Example Directory

The C++ version of the demo directory contains similar source files, but with ".cxx" and ".hxx" extensions.

3. Type **make** with no arguments to display a short help message describing the options available to you:

```
PC API Example Program Makefile
```

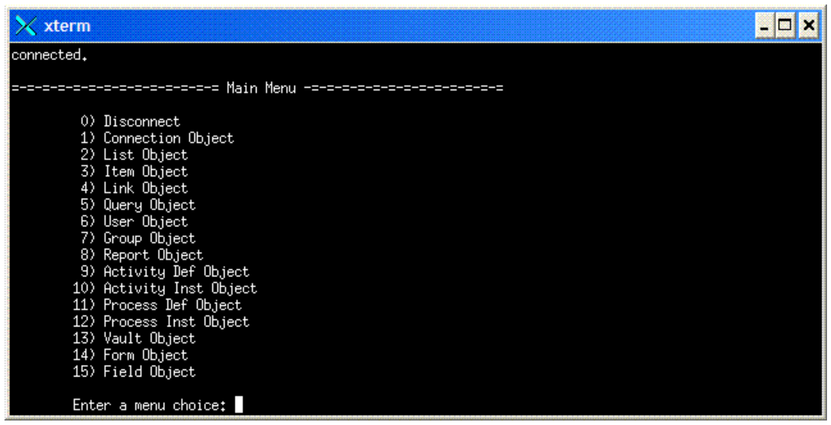
```
   make setup   - Sets up build area
   make solaris - Builds a Solaris 2.x
```
example program using the Sun WorkShop C++ compiler"
```
   make clean   - Cleans the build area"
```

NOTE: You must have CMS_HOME set to your ProductCenter installation area prior to building your example program."

4. Type **make setup** to prepare the directory. This creates symbolic links for the ProductCenter message files (msg_db.db and msg_db.idx) from $CMS_HOME/resource.
5. Type **make** *platform_name* to build the demo. You should see the following messages (Solaris version shown):
   ```
   Building: example.sun4_5
   Compiling source.
   Linking executable.
   Done.
   ```
6. Type the name of the newly created executable to run the demo. The executable should be called **example.***platform*
7. Enter values as shown in Figure 2-16. (The program displays reminders that you can enter these values at the command line when you invoke the demo.):
   - Broker host name — The name of a computer where a ProductCenter broker process is running.
   - Broker port number — The default value of '5400' is good for most installations. You will need to consult with the ProductCenter administrator if the value is different for your site.
   - Database name — A common default name is 'pctr'. You will need to consult with the ProductCenter administrator if the value is different for your site.
   - Login name — You should have a ProductCenter user account that you can use for Toolkit applications. You can also use the 'CMS' account.
   - Password — Enter the password for the user account entered in the previous step. If you are using the CMS account, your administrator has undoubtedly changed the default password.

   The program attempts to connect using the specified information and, if successful, displays a menu.

8. From the text menu (Figure 2-16), enter options **q**, **i**, or **w** to display further menus for specifying ProductCenter operations. When done, enter **e** from the main menu to exit.



```
connected.

=-=-=-=-=-=-=-=-=-=-=-= Main Menu -=-=-=-=-=-=-=-=-=-=-=-=

         0) Disconnect
         1) Connection Object
         2) List Object
         3) Item Object
         4) Link Object
         5) Query Object
         6) User Object
         7) Group Object
         8) Report Object
         9) Activity Def Object
        10) Activity Inst Object
        11) Process Def Object
        12) Process Inst Object
        13) Vault Object
        14) Form Object
        15) Field Object

        Enter a menu choice: █
```

Figure 2-16: example menu

9. Explore the demo by entering various values at the menu. Enter “0” to disconnect when done.
10. Make copies of the demo files and use them as a starting point for your own custom application.

Perl Basics

The general procedures for building ProductCenter Perl Toolkit programs are similar whether you are working in a UNIX or Windows environment, although specifics vary.

To use the ProductCenter Perl Toolkit, you perform the following steps:

- Ensure that a supported version of Perl is installed on your ProductCenter server. See the ProductCenter Release Notes for more information.
- Install the ProductCenter Perl Toolkit module on your ProductCenter server. See the ProductCenter Installation Guide for more information.
- Use the provided example Perl scripts to get a feel for how the Perl Toolkit works. These example files are installed in \productcenter\toolkits\example. Make copies of the examples and use the copies as the starting

point for your own Perl Toolkit application(s). (The Perl examples are all accessed from exampleCnxn.pl)

- Check for syntax errors without running the script by using the "-c" option to the Perl interpreter:

```
perl -c exampleCnxn.pl
```

- Test the script and print warnings about possible errors using the "-w" option:

```
perl -w exampleCnxn.pl
```

- Execute the perl script:

```
perl exampleCnxn.pl
```

- Debug your program using the "-d" option:

```
perl -d exampleCnxn.pl
```

NOTE: Before running any Perl script, you must ensure that it can access the message database files (msg_db.db and msg_db.idx). You generally do this by setting the environment variable "MSG_DB_HOME" to the directory where you installed the message database files, or by copying the files from the ProductCenter bin directory to the folder where your Perl script resides. See "Message files" on page 44 for details.

Toolkit syntax

All Perl syntax given as examples have the following format

```
$return_value = $obj->Function($arg1, $arg2, ...);
```

where:

Function is the name of the ProductCenter Toolkit object member function.

obj is an object instance of one of the classes described in "Objects" on page 45.

The print statements in the examples are based both on the native Perl format as well as the 'C' format (since Perl supports C format print statements)

Getting Started with Perl on Windows

The ProductCenter Perl Toolkit includes a series of sample Perl scripts that exercise much of the functionality in the Toolkit. These example files are installed in \productcenter\toolkits\example. You can use these scripts as the starting point for your own programs. See the release notes for information about which version of Perl is supported.

To use the Perl example in Windows

To use the example scripts for your installation:

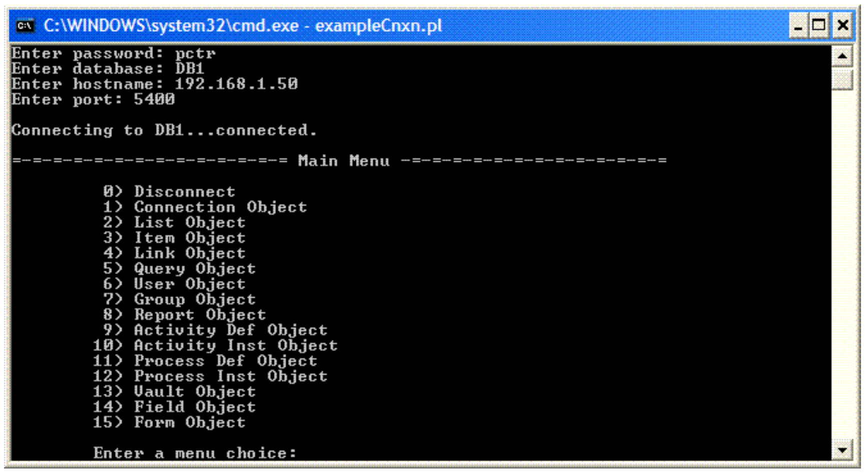
1. Make backup copies of the example files in the example directory.
2. Open a command window. If necessary, change to the drive where the ProductCenter Perl Toolkit directory is located.
3. Navigate to the ProductCenter Perl folder:

   ```
   cd \productcenter\toolkits\perl\example
   ```

4. Execute the exampleCnxn.pl script:

   ```
   perl exampleCnxn.pl
   ```

5. When prompted, enter username, password, database (often “pctr”), host name, and port (typically “5400”). After a few seconds, the example script displays a menu as show in Figure 2-17.



```
Enter password: pctr
Enter database: DB1
Enter hostname: 192.168.1.50
Enter port: 5400

Connecting to DB1...connected.

=-=-=-=-=-=-=-=-=-=-=-=-=-= Main Menu -=-=-=-=-=-=-=-=-=-=-=-=-=-=

         0) Disconnect
         1) Connection Object
         2) List Object
         3) Item Object
         4) Link Object
         5) Query Object
         6) User Object
         7) Group Object
         8) Report Object
         9) Activity Def Object
        10) Activity Inst Object
        11) Process Def Object
        12) Process Inst Object
        13) Vault Object
        14) Field Object
        15) Form Object

        Enter a menu choice:
```

Figure 2-17: Running the example.pl demo with the -w switch

Troubleshooting If the scripts do not run at all, check that you have copied the msg_db files to the example folder.

If the script fails with an error, make sure that:

- You can successfully access the same database with a Windows or Web ProductCenter client.
- You are running a supported version of Perl.
- The version of the Perl Toolkit matches the version of the ProductCenter server.

Also, if you updated Perl or the ProductCenter Perl Toolkit, check the filenames and date stamps to ensure that the files installed correctly.

6. Explore the example by trying different options. Enter **0** when you are done.
7. Try editing the example scripts and testing your changes. You can debug Perl scripts by invoking the script with "perl -d". See the Perl documentation for details about using the Perl debugger.
8. If you will be working with the AQM Event Monitor, go through a similar process with the Monitor example found in the same example directory.

Getting Started with Perl on UNIX

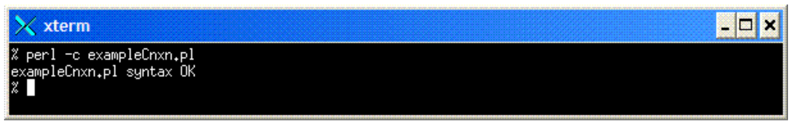
The ProductCenter Perl Toolkit includes example Perl scripts that exercise much of the functionality in the Toolkit. You can use these scripts as a starting point for your own programs.

To use the example scripts in UNIX

To use the example scripts:

1. Make a backup copy of the example files that are found in $CMS_HOME/toolkits/perl/example
2. In an xterm window, run the perl interpreter on exampleCnxn.pl, using the -c switch to check for syntax errors without actually executing the script.

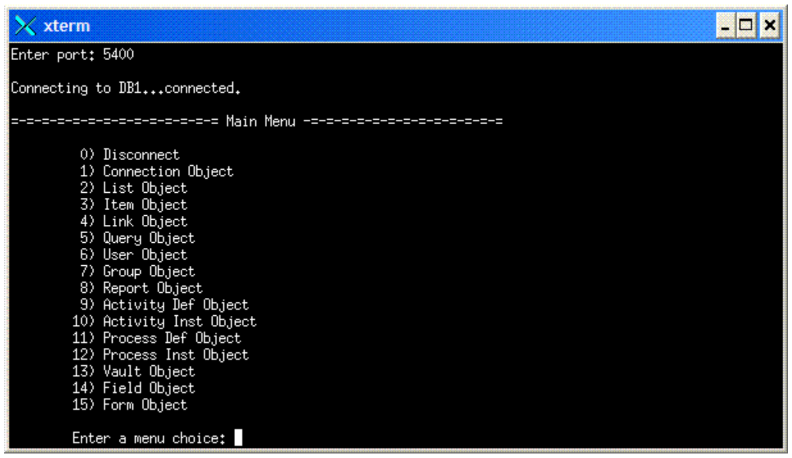
   This step should complete without any messages.



```
% perl -c exampleCnxn.pl
exampleCnxn.pl syntax OK
% 
```

Figure 2-18: Test the script with perl -c.

3. Now try executing the script with the -w switch, to see if it generates any warning messages. Execute the command line changing **-c** to **-w**. You might see one or two warnings, which can be ignored. You should then see the text menu as shown in Figure 2-19



```
Enter port: 5400

Connecting to DB1...connected.

=-=-=-=-=-=-=-=-=-=-=-=-=-= Main Menu -=-=-=-=-=-=-=-=-=-=-=-=-=-=

         0) Disconnect
         1) Connection Object
         2) List Object
         3) Item Object
         4) Link Object
         5) Query Object
         6) User Object
         7) Group Object
         8) Report Object
         9) Activity Def Object
        10) Activity Inst Object
        11) Process Def Object
        12) Process Inst Object
        13) Vault Object
        14) Field Object
        15) Form Object

        Enter a menu choice:
```

Figure 2-19: Testing the script with the -w switch

4. Explore the example by trying different options. Enter **0** when you are done. When you wish to run the example in the future, you can leave off the -w switch, or just type **"perl exampleCnxn.pl"** from an xterm window.
5. Try editing the example scripts and testing your changes. You can debug Perl scripts by invoking the script with "perl -d". Refer to Perl documentation for details about using the Perl debugger.
6. If you will be working with the AQM Event Monitor, go through a similar process with the Monitor example found in the same example directory.

Chapter 3

ProductCenter Toolkits: The Basics

3

Just Ahead:

The previous chapter provided you with a quick introduction to the Toolkit environment by having you build and run the supplied demo programs. This chapter goes into more detail about basic concepts and terminology including:

- guidelines for writing Toolkit application programs
- for C/C++ programmers:
    - comparisons of C++ functions and C structures
    - predefined types
    - message files
    - file locations
- descriptions of the basic Toolkit objects

Guidelines for writing Toolkit programs

Every program that you write should consist of three basic components:

- Startup
- Program body
- Exit

You can study the examples provided with the installation when reading the following sections.

Startup

The first step in writing any Toolkit program is to establish a connection to a server and the database. Processing cannot occur unless you make this connection. The connection remains open until you disconnect (i.e., logout) or unless an error occurs. A connection is often referred to as a *session*.

Perl considerations Perl Toolkit programmers—before establishing a connection—must include the required modules of the ProductCenter Perl Toolkit. The different modules can be included or in Perl syntax "used", as given below.

If the ProductCenter Perl Toolkit is installed in the standard Perl directories, then the following statements can be used:

```
use ProductCenter
```

If you have installed the ProductCenter Perl Toolkit in any other directory other than the standard Perl directory then you must include the following statement or statements in the program before any “use” statement:

```
use lib 'installation_dir\arch';

use lib 'installation_dir\lib';
```

Depending upon your installation, you may need to specify this path instead (if the above cause compilation errors):

```
use lib 'installation_dir/lib/site_perl';
```

where *installation_dir* is the absolute path of the directory where you have installed the ProductCenter Perl Toolkit module.

The command line option **-I** also can be used instead of the above **use** statement if you have installed the Perl Toolkit in any other directory other than the standard Perl directory. An example Windows command line is given below where C:\MyPerlLib is the root directory where ProductCenter Perl Toolkit has been installed and **test.pl** is the Perl script.

```
Command prompt > perl -IC:\MyPerlLib\lib
   -IC:\MyPerlLib\arch test.pl
```

For more details on using modules, refer to any standard Perl book or Perl manual pages, or visit the following URLs:

http://www.perl.com

http://www.cpan.org

Program body

There is not one set way to construct the body of your program, Perl allows you to place your subroutines at the beginning or end of your program. You will need to start with your login code and end with your logout code but the flow of the rest of the program will be dictated by the type of application you are creating. The chapters in this book explain the Toolkit functionality in greater detail and show you how to use all of the member functions. But the way in which you use these functions depends on the type of action you want to perform.

For example, suppose you want to perform a query through the ProductCenter Toolkit. Your program might follow this basic template:

- Create a connection object, configure it, then call a ConnectAsUser function (see page 58) to make the connection. You must establish a connection in *all* of your programs.

- Pass the connection object to the query object to construct the query; build clauses by specifying attributes, comparators, and their desired values; then execute the query by using the Execute function (see page 139).
- Ask the query for the number of items returned using GetItemCount (see page 140). Retrieve one from the list; this retrieval creates a new item for you. You then can make modifications to the item you chose and call Alter (see page 115) when you have finished.

Exit

Use the Disconnect function (see page 58) from the connection object when you want to log out and terminate your program. C/C++ Toolkit programmers should be sure to destroy the connection object when finished. (Perl Toolkit programmers should read “Destructors and Perl” on page 43 before using a destructor.) If the connection stays open, the server remains actively idle and consumes CPU time on the server/host.

Special considerations: C/C++

Differences between C++ functions and related C structures

The following chapters present the C and C++ versions of all of the functions you can use to write your programs. Almost all of the C++ functions are related to their C counterparts in the exact same way. Constructors, destructors, and copy functions are the only functions that differ.

All C++ commands have the following format:

```
obj->Function(p1, p2, ...)
```

where *Function* is the name of the C++ member function.

The corresponding C code is:

```
pcObjFunction(<Obj *>, p1, p2, ...)
```

where *Obj* is a string designating one of the Toolkit object described in “Objects” on page 45.

Special predefined types

ProductCenter Toolkit functions use several predefined types of which you should be aware:

UINT32—An unsigned integer that is 32 bits long. UINT32 is defined in pc_types.hxx and pc_types.h.

Int—A 32-bit signed integer. Note that Int always begins with a capital letter.

MSG_CODE—A number that corresponds to a message that appears on a user's screen when a particular event occurs. For example, suppose a program tries to connect to a nonexistent host. That event has a corresponding error code. The code, in turn, corresponds to a message ("Error (2000) occurred—database not opened.") that is stored in an Toolkit message file. That message appears on the user's screen.

File locations

The tables that follow describe the build environment, header files, and libraries that constitute the ProductCenter Toolkit.

To write programs using the ProductCenter Toolkit, you need the header and library files. In addition, you may find it helpful to refer to the code examples that appear throughout this document.

Path specifications

Some standard C and C++ functions convert '\' into a special char. For example, in the literal string "c:\temp", C converts the '\t' into a tab, while when inputting the string "c:\temp" through the standard C function gets() does not do the conversion. When using a method that does the conversion of the special char '\', use the character sequence '\\' for '\'.

Header files

You received 38 header files with the ProductCenter C/C++ Toolkit. The header files contain the declarations that you use to construct your own C and C++ programs. Table 3-1 lists the header files and describes their contents.

Table 3-1: ProductCenter C/C++ Toolkit header files

Name	Contents
pc_types.hxx pc_types.h	Typedefs and forward class declarations.
pccnxn.hxx pccnxn.h	Functions for establishing a connection to the ProductCenter server and setting certain parameters.
pcitem.hxx pcitem.h	Functions for manipulating projects and files in the ProductCenter database.
pclink.hxx pclink.h	Functions for manipulating links as objects.
pclist.hxx pclist.h	Functions for working with lists.
pcqry.hxx pcqry.h	Functions for performing queries.
pcevent.hxx pcevent.h	Functions for working with events in the Application Queue Manager ("event monitor").
pcmonitor.hxx pcmonitor.h	Functions for monitoring the events that appear in the Application Queue Manager ("event monitor").
pcprocessdef.hxx pcprocessdef.h	Functions for working with Workflow process instances.
pcprocessinst.hxx pcprocessinst.h	Functions for working with instance of Workflow process definitions.
pcactivitydef.hxx pcactivitydef.h	Functions for working with Workflow activity definitions.
pcactivityinst.hxx pcactivityinst.h	Functions for working with Workflow process instances.
pcreport.hxx pcreport.h	Functions for working with reports.
pcuser.hxx pcuser.h	Functions for working with users.
pcvault.hxx pcvault.h	Functions for working with vaults.
pcgroup.hxx pcgroup.h	Functions for working with groups.
pcform.hxx pcform.h	Functions for working with forms.

Table 3-1: *ProductCenter C/C++ Toolkit header files*

Name	Contents
pcfield.hxx pcfield.h	Functions for working with form fields.
pcitemcollayout.h pcitemcollayout.hxx	Functions for working with item column layouts.

Libraries

The ProductCenter C/C++ Toolkit has one library file. For supported Windows operating systems, the library file name is pcapi.lib. For supported Unix operating systems, the library file name is pcapi.a.

Each file contains all of the objects you need to work with the Toolkit on the respective platform.

Special considerations: Perl

Destructors and Perl

Perl Toolkit programmers should avoid making explicit destructor calls, and allow automatic garbage collection to occur as objects go out of scope. A variable declared in a subroutine goes out of scope when that subroutine exits *unless you create a reference to the variable and return the reference rather than contents of the variable to the calling program.*. Making an explicit destructor call after a call has already gone out of scope may result in a segmentation fault. Perl destructors are documented in this book, but should not be used.

The automatic garbage collection works by storing a count to the references to a variable. When a variable is declared in a subroutine the reference count is set to one, when you exit the routine it is decremented and since the reference count is now zero it is out of scope and garbage collected. If you create a reference to that variable in the subroutine that is stored in a variable that was created before you entered the subroutine then reference count will not be decremented to zero.

If you are having a memory usage issue with a Perl program the first place to look for problems is a circular reference. If you are using references to variables in and out of subroutines it is possible confuse Perl's reference counter.

PERL_BADFREE environment variable

On some UNIX platforms, Perl may generate the following internal error message when it tries to free memory that wasn't allocated:

```
Bad free() ignored at line nnnn
```

You can suppress this message by defining the following environment variable:

```
setenv PERL_BADFREE 0
```

Perl Toolkit Booleans

Throughout this documentation we make reference to the values TRUE and FALSE. In the ProductCenter Perl Toolkit, TRUE equates to "1" and FALSE equates to "0".

Message files

The ProductCenter Toolkits come with two message files

- msg_db.idx — An index of all message codes.
- msg_db.db — The file that contains all of the possible messages.

NOTE: You should set the MSG_DB_HOME environment variable before running your Toolkit applications. This environment variable specifies the location of the msg_db.db and msg_db.idx files in the ProductCenter server installation.

In a typical Microsoft Windows installation is set in the registry, if you need to override the registry value the syntax would be:

```
set MSG_DB_HOME=C:\ProductCenter\resource
```

You can make this setting permanent through the **Environment** tab on the System Properties window available through the Control Panel.

In UNIX, the csh syntax is:

```
setenv MSG_DB_HOME $CMS_HOME/resource
```

The Toolkit observes the following rules when searching for these two files:

- The Toolkit looks in your current working directory first.
- If the files are not in the current working directory, the Toolkit checks to see if the environment variable MSG_DB_HOME is set. If set, the value of MSG_DB_HOME is an absolute path to the directory containing msg_db.idx and msg_db.db. The Toolkit checks this directory for the message files. If the path is invalid or the files are not located in that directory, you receive an error.
- If the environment variable CMS_HOME is set, then the Toolkit looks for the files at the end of the path *CMS_HOME*\resource.If all of the above fail, the Toolkit will be unable to open the message database. You will still receive error message codes, but you do not see the text strings that define their meanings.

Objects

The ProductCenter Toolkits are object oriented. You write Toolkit programs with C++ member functions, Perl member functions, or C structures that act upon these objects, which are described in the following sections. Note that four of the following objects are devoted just to Workflow functionality, and another two are provided for Event Monitor (AQM) functionality.

For the Perl Toolkit, all these objects are encompassed by a single object called “ProductCenter”. Therefore, the Perl syntax always includes the parent object ProductCenter, as in *ProductCenter*::Cnxn.

In the C/C++ Toolkit, these objects are defined in header files. For example, the header files pccnxn.hxx and pccnxn.h contain the definition of the connection object. This object consists of the class definition pcCnxn, which contains the member functions you use in your programs.

Connection object: *pcCnxn*

The connection object contains member functions that allow you to configure, connect to, or disconnect from a ProductCenter server. Other functions in this object allow you to read or alter the working environment of the Toolkit program. You also can retrieve data specific to the database to which you connect, such as user names.

All programs that you write with the ProductCenter Toolkit should:

- Create a connection object
- Configure a connection object by setting a server name, server port number, and database name
- Log in with an appropriate user name and password
- Disconnect from the server once the main part of the program has completed
- Delete the connection to reclaim memory

The member functions in pcCnxn allow you to create and destroy connection objects, set and get member variables, establish or break your server connections, print errors and other messages, and print lists of session related data.

Most connection functions return message codes that contain the status of each function's performance. If a function did not complete successfully, it returns a code other than 0 (zero). Some functions return NULL if unsuccessful, and the connection object's "GetStatus" function returns the actual message code. If "SetAutoPrintError" is set, then all messages are displayed to standard output.

All of the Set functions are optional. If you do not use them, default values are used for the variables.

NOTE: Before you establish a connection to a server, make sure a broker is running on the host machine. The broker automatically creates and assigns a server for each connection.

Basic pcCnxn operations are discussed in Chapter 4, "Connections, Status, and Settings", but you will find other pcCnxn functions discussed throughout the other chapters of this book as well.

List object: *pcList*

The list object allows your ProductCenter Toolkit programs to return specific lists of data, as supported by the Toolkit. The Toolkit creates these lists, which are usually owned by the calling processes.

Lists may return:

- Names of lists and values in a list
- Process definitions
- Process instances
- Activity definitions
- Activity instances
- System IDs (*System IDs* are unique for an object type. You can use system IDs to create an object of that type.)

The member functions in the list object allow your ProductCenter Toolkit programs to return specific lists of data that are supported by the Toolkit. With these functions, you can get the list type, number of items in a list, the system ID of a list object, and more.

The lists and items are read-only; users cannot add objects to them. Consequently, no information is ever imparted to the Toolkit through a list from a user.

You will find discussions of pcList functions in Chapter 6, “Lists” and also in Chapter 10, “Workflow”. Note that many list-related functions are also implemented through the pcCnxn object.

A note about display names and system IDs

In general you use system IDs to return more information. For example, if you have a class and wish to get a list of its subclasses, you need to provide the system ID, not the display name of the class.

The member functions in the list object allow your ProductCenter Toolkit programs to return specific lists of data that are supported by the Toolkit. With these functions, you can get the list type, number of items in a list, the system ID of a list object, and more.

Item object: *pcItem*

The item object contains the functionality and items you need to perform operations on the data files in ProductCenter. Item objects are the main constructs that you use to transfer data into and out of the system. The functions in the item object allow you to retrieve specific attributes, determine attribute types, and obtain the number of values for a given attribute. Other functions allow you to set notification parameters for individual users or groups and to set access permission levels.

The pcItem object provides functions similar to those that users can perform through the ProductCenter user interface.

ProductCenter uses *item objects* to transfer data in and out of the system.

You will find most pcItem functions discussed in Chapter 7, “Items”, with some also discussed in Chapter 8, “Links”.

Query object: *pcQry*

The query object contains the functionality you need to perform queries through your ProductCenter Toolkit programs. You use queries to locate item objects. A *query* is a set of search criteria. When you run a query, ProductCenter retrieves all database items that match the criteria. You can search the database for entries by their names or by other attributes.

To perform a query using the Toolkit:

- Establish the way in which you want to query. You do this by using the query attribute functions that appear in Chapter 9, “Queries and Reports”.
- Execute the query using “Execute”, defined on page 139.
- Extract item objects, the entities that transfer data in and out of ProductCenter, from the query record once the Toolkit has completed the query. These objects contain the items that matched your query specifications. To extract these records, use the Get functions that appear beginning on page 141. (Chapter 7, “Items” discusses the item object in greater detail.)

You can create queries from scratch from within the ProductCenter Toolkit, as well as save queries. You can also use the Toolkit to modify queries that were built through the ProductCenter user interface.

All pcQry functions are discussed in Chapter 9, “Queries and Reports”.

Link object: *pcLink*

In ProductCenter, constructs such as links are objects that you can manipulate. Consequently, they too have attributes, and you can create your own kinds of link through the ProductCenter user interface.

You cannot define links through the ProductCenter Toolkit, but you can assign values to the links that have been defined.

The link object contains a suite of functions that allow you to manipulate links as objects. These member functions allow you to manipulate a link's attributes and obtain a link's head and tail objects.

Links vs. link attributes

It is important to understand the difference between a *link* and a *link attribute*. A *link* is a physical object that connects two items. A *link attribute* contains information about the relationship between the linked items.

For example, the names of two employees may be joined by a link, perhaps indicating that one employee reports to the other. One of the link's attributes might be LastMet, which indicates the last time that the two employees held a meeting together. The link attribute does not describe a characteristic of either item. Instead, it describes the relationship between the two items.

The definition of a link object

You do not create link objects directly. The pcItem object creates pcLink objects automatically when you link two items together. The pcItem object returns these links, which then gives you access to the link's parts. You update links by making changes with the functions in this chapter and then performing the "Alter" function on the item.

Please see "Inbound functions" on page 111 for descriptions of the inbound operations available through the ProductCenter Toolkit.

All pcLink functions are discussed in Chapter 8, "Links". Note that some link-related functions are implemented through the pcItem object.

Process Definition object: *pcProcessDef*

Four objects are devoted to Workflow, which is designed around *activities* and *processes*, which in turn are both implemented with *definitions* and *instances*. The first Workflow object is based on process definitions: pcProcessDef.

Functions associated with the process definition object allow you to manipulate process definitions (also known generically as *workflows*. See the *ProductCenter Workflow Guide* for more information about Workflow terminology and concepts).

These functions allow you to get the name and ID of a process definition; determine the number of the process's activities you can retrieve; create instances of the process; and more.

The Get functions that return const char * return NULL on error. Get functions for scalar values return message codes (MSG_CODE). All other member functions return FAILURE or another MSG_CODE on error.

All pcProcessDef functions are discussed in Chapter 10, "Workflow".

Process Instance object: ***pcProcessInst***

In the previous section, we discussed the process definition object, which contains functions that let you define templates you use to define workflow processes. You save these templates in the database. When you want to create a process based on one of these templates, you take a process definition from the database and create an instance. The instances you create are stored along with the definitions in the database.

Functions associated with the process instance object allow you to get the name, and current state of a process instance; obtain the process definition on which an instance was based; retrieve the instance activities that need assignments; and more. (Durations apply only to activities.)

The Get functions that return const char * (in C/C++) or scalar (in Perl) values return NULL on error. Get functions for scalar values return message codes (MSG_CODE). All other member functions return FAILURE or another MSG_CODE on error.

All pcProcessInst functions are discussed in Chapter 10, "Workflow".

Activity Definition object: ***pcActivityDef***

Each ProductCenter process consists of a series of activities that users must perform. These activities have attributes associated with them. An *activity definition* is a description of the components that make up an activity that a workflow process can perform. Workflow users create *instances* of these activities based on the activity definitions.

The activity definition object contains member functions with which you can manipulate these attributes and get other information about activity definitions, such as the names of the activities, a tally of the available activities, and so on.

Information associated with an activity definition includes:

- The attributes (ID number, duration, etc.) of the activity
- The types of assignments required (who does what regarding the activity)
- The activities that precede and follow each activity in the process

The member functions of this object allow you to extract the attributes of an activity definition, and much more.

The Get functions that return const char * (in C/C++) or scalar (in Perl) values return NULL on error. Get functions for scalar values return message codes (MSG_CODE). All other member functions return FAILURE or another MSG_CODE on error.

All pcActivityDef functions are discussed in Chapter 10, “Workflow”.

Activity Instance object: *pcActivityInst*

As we mentioned in the previous section, you can create instances of workflow activities from activity definitions. The member functions in the activity instance object allow you to extract the attributes of an instance, get and set assignments, claim tasks from a worklist, approve and disapprove activities, and more.

The Get functions that return const char * (in C/C++) or scalar (in Perl) values return NULL on error. Get functions for scalar values return message codes (MSG_CODE). All other member functions return FAILURE or another MSG_CODE on error.

All pcActivityInst functions are discussed in Chapter 10, “Workflow”.

Monitor object: *pcMonitor*

The Event Monitor is a programmatic library that lets client programs listen for and obtain events. The monitor object, pcMonitor, allows the client to configure the connection to the Application Queue Manager (AQM) server, set the events that will be monitored, and receive events.

Monitor object functions are discussed in Chapter 11, “Event Monitor (AQM)”.

Event object: *pcEvent*

The member functions in the Event object allow you to retrieve event types and get the items on which events occur.

A pcEvent is retrieved from a pcMonitor.

Event object functions are discussed in Chapter 11, “Event Monitor (AQM)”.

Report Object: *pcReport*

The report object enables you to extract information about items and Workflow entities. You can generate these reports based on saved queries, or based on information about specific entities. The options include:

- item-specific
- item by query
- process instance-specific
- process instance by query
- activity-specific
- activity by query

Report functions are covered starting on page 142.

Vault object: *pcVault*

The vault object enables you to get basic information about your vaults. Vault administration and detailed vault information can not be performed or accessed with this object. Descriptions of vault functions begin on page 80.

Group Object: *pcGroup*

The group object enables you to get information about groups, including user membership in those groups. Descriptions of group object functions begin on page 69.

User Object: *pcUser*

The user object allows you to access users by name or by id, and to get and set their attributes and permissions, as well as group membership. Descriptions of user functions begin on page 73.

Form Object: *pcForm*

The form object allows an application to query the attributes available to an item object or link object. The description of the form object, pcForm, begins on page 179.

Form Object: *pcField*

ProductCenter defines a form as consisting of one or more of fields. Therefore, the form object, pcForm, has methods to acquire information about individual fields in the form, each field defined by its own object, pcField. The description of the field object, pcField, begins on page 183.

Item Column Layout Object: *pcItemColLayout*

The item column layout object contains the functionality and items you need to create, retrieve, and modify item column layouts. These layouts are used by the Windows client and the WebClient to specify what information is to be displayed in the columns of the tables of items shown on their various tabs. For each column they specify the item attribute to be displayed, the width and the alignment of the column. Layouts also specify which column is to be used to sort the rows of items being displayed.

The pcItemColLayout object provides functions similar to those that users can perform through the ProductCenter user interface. All functions will record a MSG_CODE value indicating their success or failure in the connection object, so that this value can be retrieved with the connection object's "GetStatus" function. For some Get functions, this is the only reliable way to check if they succeeded.

You will find most of the pcItemColLayout functions discussed in Chapter 5, "Administration", with some also discussed in Chapter 9, "Queries and Reports".

Chapter 4

Connections, Status, and Settings

4

Just Ahead:

Every Toolkit program must create a connection to the server when it starts, and destroy that connection when it ends. A number of other operations are also very common, though not strictly required: logging in to a particular host and database, checking status, defining behavior in the event of an error, etc.

Connection object: constructors and destructors

The connection object contains one *constructor* and one *destructor*. Perl Toolkit programmers should not use a destructor call without first reading “Destructors and Perl” on page 43.

ConnectionCreate

C++: pcCnxn ();

C: pcCnxn *pcCnxnCreate ();

Perl: $cnxn = new ProductCenter::Cnxn ();

Creates a connection object for use by your Toolkit program. This is the first step of any ProductCenter Toolkit program. Note that this call only creates a connection object, it does not log you in (see “ConnectAsUser” on page 58).

You typically follow this call with a status check to ensure the creation succeeded. See “GetStatus” on page 63.

ConnectionDestroy

C++: ~pcCnxn ();

C: void pcCnxnDestroy (pcCnxn *cnxn);

Perl: ProductCenter::Cnxn::DESTROY ();

This is the *destructor* for the connection object. This is the final step of any ProductCenter Toolkit program and frees all memory used by the connection object.

NOTE: Perl programmers should read “Destructors and Perl” on page 43 for information as to why they should not use this call.

Setting values

Often the first thing you wish to do after creating a connection object is to set the hostname, port number, database name and a client locale.

You will also need to set the working directory and view directory if you intend to work with files, although you may set these later in the program when you are ready to manipulate those files.

SetDb

C++: MSG_CODE pcCnxn::SetDb (const char *db_name);

C: MSG_CODE pcCnxnSetDb (pcCnxn *cnxn, const char *db_name);

Perl: $msg_code = ProductCenter::Cnxn::SetDb ($db_name);

Specifies the database to which to connect. If you do not set a database, the system connects you to your default database. The default value is the first database in the cms_site file on the server.

SetServerHost

C++: MSG_CODE pcCnxn::SetServerHost (const char *host);

C: MSG_CODE pcCnxnSetServerHost (pcCnxn *cnxn, const char *host);

Perl: $msg_code = ProductCenter::Cnxn::SetServerHost ($host);

Specifies the host name of the machine on which the ProductCenter application server is running. The default value (“localhost“) is the local machine.

SetPortNumber

C++: MSG_CODE pcCnxn::SetPortNumber (const unsigned short int port_num);

C: MSG_CODE pcCnxnSetPortNumber (pcCnxn *cnxn, const unsigned short int port_num);

Perl: $msg_code = ProductCenter::Cnxn::SetPortNumber ($port_num);

Returns the port number of the server to which you are connecting. The default value is port 5400.

SetClientLocale

C++: MSG_CODE pcCnxn::SetClientLocale (const char *locale);

C: MSG_CODE pcCnxnSetClientLocale (pcCnxn *cnxn, const char *locale);

Perl: $msg_code = ProductCenter::Cnxn::SetClientLocale ($locale);

This function is provided for localization of Toolkit programs. This function must be used, and currently the value of *locale* can be “ASCII”, "en_US", "it_IT", "de_DE", and "fr_Fr". If “ASCII” is used, then “en_US” is used internally.

Logging in and out

Once you have created a connection object (see “Connection object: constructors and destructors” on page 56) and have specified the host, port, and database you wish to use (see “Setting values” on page 56), you typically log in as a particular user.

Use the following functions to log in, log out, and check the current login status.

ConnectAsUser

C++: MSG_CODE pcCnxn::ConnectAsUser (const char *user_name, const char *user_pass);

C: MSG_CODE pcCnxnConnectAsUser (pcCnxn *cnxn, const char *user_name, const char *user_pass);

Perl: $msg_code = ProductCenter::Cnxn::ConnectAsUser ($user_name, $user_pass);

Use this function to log in to the server. The name and password are required.

Disconnect

C++: MSG_CODE pcCnxn::Disconnect ();

C: MSG_CODE pcCnxnDisconnect (pcCnxn *cnxn);

Perl: $msg_code = ProductCenter::Cnxn::Disconnect ();

Use this function to log out of the server. Note that connections to the server do not time out. After calling this function, the connection object still exists. C/C++ Toolkit users must call the destructor (see “ConnectionDestroy” on page 56) to end the program and free up memory. (Perl Toolkit user should read “Destructors and Perl” on page 43.)

IsConnected

C++: BOOL pcCnxn::IsConnected ();

C: BOOL pcCnxnIsConnected (pcCnxn *cnxn);

Perl: $isConnected = ProductCenter::Cnxn::IsConnected ();

Returns TRUE or FALSE (1 or 0) depending on whether or not the connection object is connected to the server. The function ensures that the connection to the server is actually functioning by performing the equivalent of a KeepAlive operation (see below), requesting the version of the server.

KeepAlive

C++: MSG_CODE pcCnxn::KeepAlive ();

C: MSG_CODE pcCnxnKeepAlive (pcCnxn *cnxn);

Perl: $msg_code = ProductCenter::Cnxn::KeepAlive ();

Keeps the server connection alive by exercising the connection and preventing it from timing out.

SetWorkDir

C++: MSG_CODE pcCnxn::SetWorkDir (const char *path);

C: MSG_CODE pcCnxnSetWorkDir (pcCnxn *cnxn, const char *path);

Perl: $msg_code = ProductCenter::Cnxn::SetWorkDir ($path);

Sets the current working directory. ProductCenter uses the working directory when transferring files to and from the server. Any changes that you make are effective only for the duration of the session.

NOTE: Some standard C and C++ functions convert backslash ('\') into a special char. For example, in the literal string "c:\temp", C converts the '\t' into a tab, while when inputting the string "c:\temp" through the standard C function gets() does not do the conversion. When using a method that does the conversion of the special char '\', use the character sequence '\\' for '\'.

Perl interprets the backslash as a special character. Use the character sequence '\\' for '\'. If the backslash is part of a path specification Perl allows you to use the slash '/' in path specifications in the windows environment

SetViewDir

C++: MSG_CODE pcCnxn::SetViewDir (const char *path);

C: MSG_CODE pcCnxnSetViewDir (pcCnxn *cnxn, const char *path);

Perl: $msg_code = ProductCenter::Cnxn::SetViewDir ($path);

Specifies the viewing directory. ProductCenter places files opened for viewing into this directory. As with *SetWorkDir()*, any changes you make are effective only for the duration of your session.

Getting values

Several of the functions in this section complement the functions listed in "Setting values" on page 56. The remaining functions allow you to obtain additional information about release management status, resource values, database, and the Toolkit-supplied libraries.

GetDb

C++: const char *pcCnxn::GetDb ();

C: const char *pcCnxnGetDb (pcCnxn *cnxn);

Perl: $name = ProductCenter::Cnxn::GetDb ();

Returns the name of the database to which the app is connected.

GetServerHost

C++: const char *pcCnxn::GetServerHost ();

C: const char *pcCnxnGetServerHost (pcCnxn *cnxn);

Perl: $serverhostname = ProductCenter::Cnxn::GetServerHost ();

Returns the name of the host.

GetPortNumber

C++: unsigned short int pcCnxn::GetPortNumber ();

C: unsigned short int pcCnxnGetPortNumber (pcCnxn *cnxn);

Perl: $portnumber = ProductCenter::Cnxn::GetPortNumber ();

Returns the port number of the server to which the app is connected. Note that this call returns the port number by value rather than reference.

GetClientLocale

C++: const char *pcCnxn::GetClientLocale ();

C: const char *pcCnxnGetClientLocale (pcCnxn *cnxn);

Perl: $clientlocale = ProductCenter::Cnxn::GetClientLocale ();

Returns the locale of the client.

GetWorkDir

C++: const char *pcCnxn::GetWorkDir ();

C: const char *pcCnxnGetWorkDir (pcCnxn *cnxn);

Perl: $workdir = ProductCenter::Cnxn::GetWorkDir ();

Returns the value of the work directory.

GetViewDir

C++: const char *pcCnxn::GetViewDir ();

C: const char *pcCnxnGetViewDir (pcCnxn *cnxn);

Perl: $viewdir= ProductCenter::Cnxn::GetViewDir ();

Returns the value of the view directory.

GetMsgDbDir

C++: const char *pcCnxn::GetMsgDbDir ();

C: const char *pcCnxnGetMsgDbDir (pcCnxn *cnxn);

Perl: $msg_db_dir = ProductCenter::Cnxn::GetMsgDbDir ();

Returns the directory in which the message files are located.

GetResourceValue

C++: const char *pcCnxn::GetResourceValue (const char *resourceName);

C: const char *pcCnxnGetResourceValue (pcCnxn *cnxn, const char *resourceName);

Perl: $resourcevalue = ProductCenter::Cnxn::GetResourceValue ($resourceName);

Returns the value of the resource indicated by resourceName. Resource values are defined in the various resource files, such as cms_site, rep_site, etc.

If the resourceName is not a valid resource, the function sets a connection error to FAILURE and returns NULL.

ResourceVariableExists

C++: BOOL pcCnxn::ResourceVariableExists (const char *resourceName);

C: BOOL pcCnxnResourceVariableExists (pcCnxn *cnxn, const char *resourceName);

Perl: $exists = ProductCenter::Cnxn::ResourceVariableExists ($resourceName);

Returns TRUE or FALSE (1 or 0) depending if the resource variable exists or not.

GetLogicalResourceValue

C++: BOOL pcCnxn::GetLogicalResourceValue (const char *resourceName);

C: BOOL pcCnxnGetLogicalResourceValue (pcCnxn *cnxn, const char *resourceName);

Perl: $value =ProductCenter::Cnxn::GetLogicalResourceValue ($resourceName);

Returns TRUE if the resource variable is set to TRUE or YES. If the resourceName is not a valid resource, the function sets a connection error to FAILURE and returns FALSE.

IsRelMgmtEnabled

C++: BOOL pcCnxn::IsRelMgmtEnabled ();

C: BOOL pcCnxnIsRelMgmtEnabled (pcCnxn *cnxn);

Perl: $enabled= ProductCenter::Cnxn::IsRelMgmtEnabled ();

Returns TRUE if Release Management has been enabled and FALSE if it has not.

GetLibraryMajorVersion

C++: const char *pcCnxn::GetLibMajorVersion ();

C: const char *pcCnxnGetLibMajorVersion (pcCnxn *cnxn);

Perl: $version = ProductCenter::Cnxn::GetLibMajorVersion ();

Return the library's major version, for example, "ProductCenter v9.6.0".

GetLibraryVersion

C++: const char *pcCnxn::GetLibVersion ();

C: const char *pcCnxnGetLibVersion (pcCnxn *cnxn);

Perl: $version = ProductCenter::Cnxn::GetLibVersion ();

Returns the library's major version, for example, "v9.6.0".

GetLibraryMajorID

C++: int pcCnxn::GetLibIdMajor ();

C: int pcCnxnGetLibIdMajor (pcCnxn *cnxn);

Perl: $version = ProductCenter::Cnxn::GetLibIdMajor ();

Returns the library's major id, for example, "9".

GetLibraryMinorID

C++: int pcCnxn::GetLibIdMinor ();

C: int pcCnxnGetLibIdMinor (pcCnxn *cnxn);

Perl: $version = ProductCenter::Cnxn::GetLibIdMinor ();

Returns the library's minor id, for example, "6".

GetLibraryPatchID

C++: int pcCnxn::GetLibIdPatch ();

C: int pcCnxnGetLibIdPatch (pcCnxn *cnxn);

Perl: $version = ProductCenter::Cnxn: GetLibIdPatch ();

Returns the library's patch id, for example, "0".

Error handling

The following functions give you useful information when your programs encounter errors.

GetStatus

C++: MSG_CODE pcCnxn::GetStatus ();

C: MSG_CODE pcCnxnGetStatus (pcCnxn *pcCnxn);

Perl: $msg_code = ProductCenter::Cnxn::GetStatus ();

Returns the ProductCenter Toolkit error number. You typically call this function after creating a new connection (see “Connection object: constructors and destructors” on page 56) or any other operation where you need to be assured of successful completion. Error conditions are typically handled by printing the error (“PrintError” on page 64). C/C++ Toolkit users should also destroy the connection (“ConnectionDestroy” on page 56).

GetStatusMsg

C++: const char *pcCnxn::GetStatusMsg ();

C: const char *pcCnxnGetStatusMsg (pcCnxn *pcCnxn);

Perl: $name= ProductCenter::Cnxn::GetStatusMsg ();

Returns the appropriate ProductCenter Toolkit error message. This is helpful in making context specific error messages.

Example:

Perl:

```
if ($item->SetAttr($attrName, $attrValue) {
    print "Error: Unable to set $attrName to $attrValue " .
        $cnxn->GetStatusMsg . "\n";
}
or
$link = new ProductCenter::Link($cnxn, $linkType );
if ( $cnxn->GetStatus != SUCCESS ) {
    print "Error:Unable to create $linkType link" .
        $cnxn->getStatusMsg . "\n";
}
```

PrintError

C++: void pcCnxn::PrintError ();

C: void pcCnxnPrintError (pcCnxn *pcCnxn);

Perl: $name = ProductCenter::Cnxn::PrintError ();

Prints error information in a text block. You may want to use this function if you have set "SetAutoPrintError" (on page 64) to FALSE and have not created custom error messages..

SetAutoPrintError

C++: void pcCnxn::SetAutoPrintError ();

C: void pcCnxnSetAutoPrintError (pcCnxn *pcCnxn);

Perl: ProductCenter::Cnxn::SetAutoPrintError ();

Allows you to specify whether error information should be printed when an error occurs.

If you set this value to TRUE, a text block is printed. If the Toolkit finds the message database upon connection, this block contains a message code and a string displaying the error message. Otherwise, you see an error code and a message that tells you that the Toolkit could not find the message database.

ClearAutoPrintError

C++: void pcCnxn::ClearAutoPrintError ();

C: void pcCnxnClearAutoPrintError (pcCnxn *pcCnxn);

Perl: ProductCenter::Cnxn::ClearAutoPrintError ();

Call this function if you want to clear the AutoPrintError flag inside the connection object.

SetAbortOnError

C++: void pcCnxn::SetAbortOnError ();

C: void pcCnxnSetAbortOnError (pcCnxn *pcCnxn);

Perl: ProductCenter::Cnxn::SetAbortOnError ();

Specifies whether the application should abort when an error occurs. In general, you probably do not want to set this behavior unless you are creating an application like a batch loader that would continue to run and generate a large number of error messages.

ClearAbortOnError

C++: void pcCnxn::ClearAbortOnError ();

C: void pcCnxnClearAbortOnError (pcCnxn *pcCnxn);

Perl: ProductCenter::Cnxn::ClearAbortOnError ();

Use this function to reverse the effect of "SetAbortOnError" (on page 65). Your program will continue to run even after encountering errors.

Chapter 5

Administration

5

Just Ahead:

This chapter describes the functions related to creating a group, getting and setting group information as well as getting and setting user screen permissions.

Group object:

Constructors and Destructors

The group object contains these *constructor*s.and one *destructor*.

GroupCreate

C++: pcGroup = *pcGroup::pcGroup (pcCnxn *cnxn);

C: pcGroup = *pcGroupCreate (pcCnxn *cnxn);

Perl: $group = ProductCenter::Group ($cnxn);

Creates a new instance of a group object.

GroupLoadByName

C++: pcGroup = *pcGroup::pcGroup (pcCnxn *cnxn, const char *group_name);

C: pcGroup = *pcGroupLoadByName (pcCnxn *cnxn, const char *group_name);

Perl: $group = new ProductCenter::Group ($cnxn, $group_name);

Returns the group object based on the group name.

GroupLoadById

C++: pcGroup = *pcGroup::pcGroup (pcCnxn *cnxn, UINT32 group_id);

C: pcGroup = *pcGroupLoadById (pcCnxn *cnxn, UINT32 group_id);

Perl: $group = new ProductCenter::Group ($cnxn, $group_id);

Returns the group object based on the group id.

GroupDestroy

C++: ~pcGroup ();

C: void pcGroupDestroy (pcUser *user);

Perl: ProductCenter::Group::DESTROY ();

The group object, like all C++ objects, has one destructor. The destructor is invoked automatically when the object is destroyed, and all memory used by the group object is freed. The C counterpart listed here frees memory that was allocated when the group object was created.

> **NOTE:** Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Getting Information

GetGroupCount

C++: MSG_CODE pcCnxn::GetGroupCount (UINT32 *count);

C: MSG_CODE pcCnxnGetGroupCount (pcCnxn *cnxn, UINT32 *count);

Perl: $msg_code = ProductCenter::Cnxn::GetGroupCount ($count);

Returns the count of the groups that are available in ProductCenter.

GetGroupByIndex

C++: pcGroup *pcCnxn::GetGroupByIndex (UINT32 index);

C: pcGroup *pcCnxnGetGroupByIndex (pcCnxn *cnxn, UINT32 index);

Perl: $group = ProductCenter::Cnxn::GetGroupByIndex ($index);

Returns the group object based on the index.

GroupGetUserCount

C++: MSG_CODE pcGroup::GetUserCount (UINT32 *count);

C: MSG_CODE pcGroupGetUserCount (pcGroup *group, UINT32 *count);

Perl: $msg_code = ProductCenter::Group::GetUserCount ($count);

Returns the number of users in the group.

GroupGetUserByIndex

C++: pcUser *pcGroup::GetUser (UINT32 index);

C: pcUser *pcGroupGetUser (pcGroup *group, UINT32 index);

Perl: $user= ProductCenter::Group::GetUser ($index);

Returns the user from the group by the index.

IsUserGroupMember

C++: BOOL pcGroup::IsUserMember (const char *loginName);

C: BOOL pcGroupIsUserMember (pcGroup *group, const char *loginName);

Perl: $isMember = ProductCenter::Group::IsUserMember ($loginName);

Returns TRUE or FALSE (1 or 0) depending if the user, specified by login name, is a member of the specified group.

GroupGetAttrCount

C++: MSG_CODE pcGroup::GetAttrCount (UINT32 *count);

C: MSG_CODE pcGroupGetAttrCount (pcGroup *group, UINT32 *count);

Perl: $msg_code = ProductCenter::Group::GetAttrCount ($count);

Returns the number of group attributes.

GroupGetAttrNameByIndex

C++: const char *pcGroup::GetAttrNameByIndex (UINT32 index);

C: const char *pcGroupGetAttrNameByIndex (pcGroup *group, UINT32 index);

Perl: $attrName = ProductCenter::Group:: GetAttrNameByIndex ($index);

Returns group attribute name by index.

GroupGetAttr

C++: const char *pcGroup::GetAttr (const char *attr);

C: const char *pcGroupGetAttr (pcGroup *group, const char *attr);

Perl: $attrValue = ProductCenter::Group::GetAttr ($attr);

Returns information about the group, possible values are: "Group Name", "Id"

Setting Information

GroupSetAttr

C++: MSG_CODE pcGroup::SetAttr (const char *attr, const char *value);

C: MSG_CODE pcGroupSetAttr (pcGroup *group, const char *attr, const char *value);

Perl: $msg_code = ProductCenter::Group::SetAttr ($attr,$value);

Sets a group attribute value. Possible values are: "Group Name".

GroupAddUser

C++: MSG_CODE pcGroup::AddUser (const char *login_name);

C: MSG_CODE pcGroupAdd User (pcGroup *group, const char *login_name);

Perl: $msg_code = ProductCenter::Group::AddUser ($login_name);

Adds a user, identified by login name, to the group. If the user is invalid, the toolkit returns an error.

GroupRemoveUser

C++: MSG_CODE pcGroup::RemoveUser (const char *login_name);

C: MSG_CODE pcGroupRemove User (pcGroup *group, const char *login_name);

Perl: $msg_code = ProductCenter::Group::RemoveUser ($login_name);

Removes a user, identified by login name, from the group.

GroupSave

C++: MSG_CODE pcGroup::Save ();

C: MSG_CODE pcGroupSave (pcGroup *group);

Perl: $msg_code = ProductCenter::Group::Save ();

Saves group attributes and membership information. If the group does not exist, this function adds the group to the database.

GroupDelete

C++: MSG_CODE pcGroup::Delete ();

C: MSG_CODE pcGroupDelete (pcGroup*group);

Perl: $msg_code = ProductCenter::Group::Delete ();

Deletes the group that is currently loaded.

User object:

Constructors and Destructors

The user object contains these *constructor*s.and one *destructor*.

UserLoadByLoginName

C++: pcUser *pcUser::pcUser (pcCnxn *cnxn, const char *login_name);

C: pcUser *pcUserLoadByLoginName (pcCnxn *cnxn, const char *login_name);

Perl: $user = new ProductCenter::User ($cnxn, $login_name);

Returns the user object based on the login name.

UserLoadById

C++: pcUser *pcUser::pcUser (pcCnxn *cnxn, UINT32 user_id);

C: pcUser *pcUserLoadById (pcCnxn *cnxn, UINT32 user_id);

Perl: $user = new ProductCenter::User ($cnxn, $user_id);

Returns the user object based on the user id.

GetCurrentUser

C++: pcUser *pcCnxn::GetCurrentUser ();

C: pcUser *pcCnxnGetCurrentUser (pcCnxn *cnxn);

Perl: $user = ProductCenter::Cnxn::GetCurrentUser ();

Returns the user object for the currently connected user.

UserDestroy

C++: ~pcUser();

C: void pcUserDestroy (pcUser *user);

Perl: ProductCenter::User::DESTROY ();

The user object, like all C++ objects, has one destructor. The destructor is invoked automatically when the object is destroyed, and all memory used by the user object is freed. The C counterpart listed here frees memory that was allocated when the user object was created.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Getting Information

GetUserCount

C++: MSG_CODE pcCnxn::GetUserCount (UINT32 *count);

C: MSG_CODE pcCnxnGetUserCount (pcCnxn *cnxn, UINT32 *count);

Perl: $msg_code = ProductCenter::Cnxn::GetUserCount ($count);

Returns the number of users that are available in ProductCenter.

GetUserByIndex

C++: pcUser *pcCnxn::GetUserByIndex (UINT32 index);

C: pcUser *pcCnxnGetUserByIndex (pcCnxn *cnxn, UINT32 index);

Perl: $user= ProductCenter::Cnxn::GetUserByIndex ($index);

Returns the user object based on the index.

GetUser

C++: const char *pcCnxn::GetUser ();

C: const char *pcCnxnGetUser (pcCnxn *cnxn);

Perl: $user= ProductCenter::Cnxn::GetUser ();

Returns the ProductCenter user name for the currently connected user.

GetLoginName

C++: const char *pcCnxn::GetLoginName ();

C: const char *pcCnxnGetLoginName (pcCnxn *cnxn);

Perl: $login_name = ProductCenter::Cnxn::GetLoginName ();

Returns the ProductCenter login name for the currently connected user.

IsUserDBA

C++: BOOL pcCnxn::IsUserDBA ();

C: BOOL pcCnxnIsUserDBA (pcCnxn *cnxn);

Perl: $dba= ProductCenter::Cnxn::IsUserDBA ();

Returns TRUE if the current user is DBA enabled and FALSE if the user is not.

UserGetGroupCount

C++: MSG_CODE pcUser::GetGroupCount (UINT32 *count);

C: MSG_CODE pcUserGetGroupCount (pcUser *user, UINT32 *count);

Perl: $msg_code = ProductCenter::User::GetGroupCount ($count);

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

UserGetGroup

C++: pcGroup *pcUser::GetGroup (UINT32 index);

C: pcGroup *pcUserGetGroup (pcUser *user, UINT32 index);

Perl: $group = ProductCenter::User::GetGroup ($index);

Returns the group object by index into the list of groups to which the specified user is a member.

UserIsMemberOf

C++: BOOL pcUser::IsMemberOf (const char *group);

C: BOOL pcUserIsMemberOf (pcUser *user, const char *group);

Perl: $isMemberOf = ProductCenter::User::IsMemberOf ($group);

Returns TRUE or FALSE (1 or 0) depending if the user is a member of the specified group.

UserGetAttrCount

C++: MSG_CODE pcUser::GetAttrCount (UINT32 *count);

C: MSG_CODE pcUserGetAttrCount (pcUser *user, UINT32 *count);

Perl: $msg_code = ProductCenter::User::GetAttrCount ($count);

Returns the number of user attributes.

UserGetAttrNameByIndex

C++: const char *pcUser::GetAttrNameByIndex (UINT32 index);

C: const char *pcUserGetAttrNameByIndex (pcUser *user, UINT32 index);

Perl: $attrName = ProductCenter::User:: GetAttrNameByIndex ($index);

Returns user attribute name by index.

UserGetAttr

C++: const char *pcUser::GetAttr (const char *attr);

C: const char *pcUserGetAttr (pcUser *user, const char *attr);

Perl: $attrValue = ProductCenter::User::GetAttr ($attr);

Returns information about the user possible values are listed in Table 5-1

Only DBA-enabled users can access another user's information

Table 5-1: User Object Attributes

Attribute Name	Usage	Possible Values
Can Change Password	Permission to change password	0 or 1
Check In On Exit	Prompt user to check in all checked out objects	YES or NO
Confirm On Exit	Prevents unintentional log out	YES or NO
Decimal Length	Position of decimal place in numbers	Integer
Decimal Separator	Place holder between whole and fractional portion of numbers	Character
Email Address	Email Address	Text
Extension	Telephone Extension	Text (16 char)
Group Length	digits between grouping symbol for large numbers	Integer
Group Symbol	Place holder between groups in large numbers	Character
Id ***read only***	User ID	Number
Is Account Enabled	0 = Disabled 1 = Enabled	0 or 1

Table 5-1: User Object Attributes

Attribute Name	Usage	Possible Values
Is DBA enabled	0 = Disabled 1 = Enabled	0 or 1
Links Filter	Links used for expand	
Login Name **read only**	User's name	Text
Open on Login	Open the Preferred Window on login	YES or NO
Password	User's password	Encrypted text
Preferred Class	Default file or project class for add	Class Name
Preferred Window	Window that is opened if "Open on Login is YES	'Desktop' or 'My Work List' or 'My Checked-Out Items'
Preview Threshold	Maximum number of objects returned from search, 0 for unlimited	Numeric
Save Settings on Exit	Save window positions	YES or NO
Search Case Sensitive	Default search value	YES or NO
Telephone	Telephone number	Text (32 Char)
View Dir	Directory for get copy	Path Specification
Work Dir	Directory for Add, Check out	Path Specification

Password Decrypt

C++: const char *:DecryptPassword(const char* sPass);

C: const char *:pcCnxnDecryptPassword(pcCnxn *pccnxn, const char* sPass)

Perl: $msg_code = ProductCenter::Cnxn::DecryptPassword($self,$pass);

Returns a decrypted string that was previously encrypted with EncryptPassword.

UserGetScreenPermissionCount

C++: MSG_CODE pcUser::GetScreenPermissionCount (UINT32 *count);

C: MSG_CODE pcUserGetScreenPermissionCount (pcUser *user, UINT32 *count);

Perl: $msg_code = ProductCenter::User:: GetScreenPermissionCount ($count);

Returns the number of user screen permissions.

UserGetPermNameByIndex

C++: const char *pcUser::GetPermNameByIndex (UINT32 index);

C: const char *pcUserGetPermNameByIndex (pcUser *user, UINT32 index);

Perl: $attrName = ProductCenter::User::GetPermNameByIndex ($index);

Returns user screen permission name by index.

UserGetScreenPermView

C++: BOOL pcUser::GetScreenPermView (const char *perm);

C: BOOL pcUserGetScreenPermView (pcUser *user, const char *perm);

Perl: $view= ProductCenter::User::GetScreenPermView ($perm);

Returns TRUE or FALSE depending if the specified view permission is set for the specified user or not. See Table 5-2 on page 79 for the list of possible permission names.

UserGetScreenPermEdit

C++: BOOL pcUser::GetScreenPermEdit (const char *perm);

C: BOOL pcUserGetScreenPermEdit (pcUser *user, const char *perm);

Perl: $edit = ProductCenter::User::GetScreenPermEdit ($perm);

Returns TRUE or FALSE depending if the specified edit permission is set for the specified user or not. See Table 5-2 on page 79 for the list of possible permission names.

Setting Information

UserSetAttr

C++: MSG_CODE pcUser::SetAttr (const char *attr_name, const char *attr_value);

C: MSG_CODE pcUserSetAttr (pcUser *user, const char *attr_name, const char *attr_value);

Perl: $msg_code = ProductCenter::User::SetAttr ($attr_name, $attr_value);

Sets information about the user see Table 5-1 on page 75 for a list of the attribute names..

Only DBA-enabled users can set another user's information. When the password is set the account will be pre-expired forcing the user to change the password the next time they log in.

UserSetScreenPermView

C++: MSG_CODE pcUser::SetScreenPermView (const char *perm, BOOL allow);

C: MSG_CODE pcUserSetScreenPermView (pcUser *user, const char *perm,BOOL allow);

Perl: $msg_code = ProductCenter::User::SetScreenPermView ($perm, $allow);

Sets a user's View screen permission for the specified permission. See Table 5-2 on page 79 for the list of permission names.

User Set Screen Perm Edit

C++: MSG_CODE pcUser::GetScreenPermEdit (const char *perm, BOOL allow);

C: MSG_CODE pcUserGetScreenPermEdit (pcUser *user, const char *perm, BOOL allow);

Perl: $msg_code = ProductCenter::User::GetScreenPermEdit ($perm, $allow);

Sets a user's Edit screen perrmission for the specified permission. See Table 5-2 on page 79 for the list of permission names.

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

Vault object:

Constructors and destructors

The vault object does not require a constructor.

Vault Object Destructor

C++: ~pcVault ();

C: void pcVaultDestroy (pcVault *vault);

Perl: ProductCenter::Vault::DESTROY ($vault);

Destroys the vault object.

NOTE: Perl programmers should read “Destructors and Perl” on page 43 for information as to why they should not use this call.

CreateClone

C++: pcVault *pcVault::pcVault (pcVault *vault);

C: pcVault *pcVaultCreateClone (pcVault *vault);

Perl: $vault = ProductCenter::Vault::CreateClone ($vault);

Creates a copy of the vault memory.

Vault Object Functions

GetAttrCount

C++: MSG_CODE pcVault::GetAttrCount (UINT32 *count);

C: MSG_CODE pcVaultGetAttrCount (pcVault *vault, UINT32 *count);

Perl: $msg_code = ProductCenter::Vault::GetAttrCount ($vault, $count);

Returns the number of attributes on the vault object

GetAttrNameByIndex

C++: const char *pcVault::GetAttrNameByIndex (UINT32 index);

C: const char *pcVaultGetAttrNameByIndex (pcVault *vault, UINT32 index);

Perl: $attrName = ProductCenter::Vault::GetAttrNameByIndex ($vault, $index);

Gets the attribute name by the index.

GetAttr

C++: const char *pcVault::GetAttr (const char *name);

C: const char *pcVaultGetAttr (pcVault *vault, const char *name);

Perl: $attrValue = ProductCenter::Vault::GetAttr ($vault, $name);

Returns the attribute value for the attribute specified in the name. Possible values are "Name", "Id", "Status", "Hostname", and "Port". These values are returned by GetAttrNameByIndex.

GetVaultCount

C++: int pcCnxn::GetVaultCount ();

C: int pcCnxnGetVaultCount (pcCnxn *pcCnxn);

Perl: $count = ProductCenter::Cnxn::GetVaultCount ();

Returns the number of vaults.

GetVault

C++: pcVault *pcCnxn::GetVault (unsigned short int index);

C: pcVault *pcCnxnGetVault (pcCnxn *pccnxn, unsigned short int index);

Perl: $vault = ProductCenter::Cnxn::GetVault ($index);

Returns a pointer to the vault at the specified index.

GetVaultByName

C++: pcVault *pcCnxn::GetVaultByName (const char *name);

C: pcVault *pcCnxnGetVaultByName (pcCnxn *pccnxn,const char *name);

Perl: $vault = ProductCenter::Cnxn::GetVaultByName ($name);

Returns the vault of the specified name.

GetVaultById

C++: pcVault *pcCnxn::GetVaultById (unsigned int id);

C: pcVault *pcCnxnGetVaultById (pcCnxn *pcCnxn, unsigned int id);

Perl: $vault = ProductCenter::Cnxn::GetVaultById ($id);

Returns the vault of the specified ID.

Item Column Layout object:

Item column layouts are actually configuration information for ProductCenter clients. This information is shared by the Windows client and the WebClient, so it is stored in the database and accessed via the server API. This toolkit API allows toolkit applications to access and modify item column layouts, too.

Please consult the section on "Customizing Table Layouts" in Chapter 2 of the *ProductCenter for Windows User Guide* for background information.

Constructors and destructors

Create

C++: pcItemColLayout(pcCnxn *cnxn);

C: pcItemColLayout *pcItemColLayoutCreate(pcCnxn *cnxn);

Perl: $layout = new ProductCenter::ItemColLayout($cnxn);

Creates a new, empty item column layout object.

Destroy

C++: ~pcItemColLayout();

C: void pcItemColLayoutDestroy(pcItemColLayout *layout);

Perl: ProductCenter::ItemColLayout::DESTROY();

The item column layout object, like all C++ objects, has one destructor. The destructor is invoked automatically when the object is destroyed, and all memory usage by the layout object is freed. The C counterpart listed here frees memory that was allocated when the layout object was created.

5

NOTE: Perl programmers should read “Destructors and Perl” on page 43 for information as to why they should not use this call.

Constructing or Editing the layout

SetColumn

C++: MSG_CODE pcItemColLayout::SetColumn(UINT32 index, pcField *field, UINT32 width, const char *align);

C: MSG_CODE pcItemColLayoutSetColumn(pcItemColLayout *layout, UINT32 index, pcField *field, UINT32 width, const char *align);

Perl: $msg_code = ProductCenter::ItemColLayout::SetColumn($index, $field, $width, $align);

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".

RemoveColumn

C++: MSG_CODE pcItemColLayout::RemoveColumn(UINT32 index);

C: MSG_CODE pcItemColLayoutRemoveColumn(pcItemColLayout *layout, UINT32 index);

Perl: $msg_code = ProductCenter::ItemColLayout::RemoveColumn($index);

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.

RemoveAllColumn

C++: MSG_CODE pcItemColLayout::RemoveAllColumn();

C: MSG_CODE pcItemColLayoutRemoveAllColumn(pcItemColLayout *layout);

Perl: $msg_code = ProductCenter::ItemColLayout::RemoveAllColumn();

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

SetSort

C++: MSG_CODE pcItemColLayout::SetSort(UINT32 sort_index, UINT32 col_index, const char *dir);

C: MSG_CODE pcItemColLayoutSetSort(pcItemColLayout *layout, UINT32 sort_index, UINT32 col_index, const char *dir);

Perl: $msg_code = ProductCenter::ItemColLayout::SetSort($sort_index, $col_index, $dir);

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 dir as "asc" (i.e. ascending order from top to bottom of the display grid) or "desc". (It is also legal to specify dir as "unsorted", but you should not store a layout with sort_index 0 and dir "unsorted".)

Accessing the layout

GetColumnCount

C++: MSG_CODE pcItemColLayout::GetColumnCount(UINT32 *count);

C: MSG_CODE pcItemColLayoutGetColumnCount(pcItemColLayout *layout, UINT32 *count);

Perl: $msg_code = ProductCenter::ItemColLayout::GetColumnCount(UINT32 $count);

Gets the number of columns in the layout.

GetColumnItemFormId

C++: int pcItemColLayout::GetColumnItemFormId(UINT32 index);

C: int pcItemColLayoutGetColumnItemFormId(pcItemColLayout *layout, UINT32 index);

Perl: $form_id = ProductCenter::ItemColLayout::GetColumnItemFormId($index);

Gets the form id of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is the internal id of the derived form that contains the attribute that is displayed in this column.

This function will return 0 if it fails. When it returns 0 you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetColumnItemFieldId

C++: int pcItemColLayout::GetColumnItemFieldId(UINT32 index);

C: int pcItemColLayoutGetColumnItemFieldId(pcItemColLayout *layout, UINT32 index);

Perl: $uda_id = ProductCenter::ItemColLayout::GetColumnItemFieldId($index);

Gets the attribute or field id of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is the internal id of the attribute that is displayed in this column.

This function will return 0 if it fails. When it returns 0 you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetColumnItemName

C++: const char* pcItemColLayout::GetColumnItemName(UINT32 index);

C: const char* pcItemColLayoutGetColumnItemName(pcItemColLayout *layout, UINT32 index);

Perl: $name = ProductCenter::ItemColLayout::GetColumnItemName($index);

Gets the attribute name of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). If the attribute for the column is a common attribute, then the value returned 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 value returned will be the empty string.

This function will return NULL if it fails. When it returns NULL you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetColumnItemPrompt

C++: const char* pcItemColLayout::GetColumnItemPrompt(UINT32 index);

C: const char* pcItemColLayoutGetColumnItemPrompt (pcItemColLayout *layout, UINT32 index);

Perl: $prompt = ProductCenter::ItemColLayout::GetColumnItemPrompt ($index);

Gets the attribute prompt of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is the prompt for the attribute that is displayed in this column.

This function will return NULL if it fails. When it returns NULL you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetColumnItemType

C++: MSG_CODE pcItemColLayout::GetColumnItemType(UINT32 index, pcAttrType* colType);

C: MSG_CODE pcItemColLayoutGetColumnItemType (pcItemColLayout *layout, UINT32 index, pcAttrType* colType);

Perl: $msg_code = ProductCenter::ItemColLayout::GetColumnItemType ($index, $colType);

Gets the attribute type of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is the type of the attribute that is displayed in this column.

GetColumnWidth

C++: UINT32 pcItemColLayout::GetColumnWidth(UINT32 index);

C: UINT32 pcItemColLayoutGetColumnWidth(pcItemColLayout *layout, UINT32 index);

Perl: $form_id = ProductCenter::ItemColLayout::GetColumnWidth($index);

Gets the width of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is the column width in average characters.

This function will return 0 if it fails. When it returns 0 you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetColumnAlignment

C++: const char* pcItemColLayout::GetColumnAlignment(UINT32 index);

C: const char* pcItemColLayoutGetColumnAlignment(pcItemColLayout *layout, UINT32 index);

Perl: $align = ProductCenter::ItemColLayout::GetColumnAlignment($index);

Gets the alignment of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is either "left", "center", or "right", specifying how the displayed values should be aligned in the column.

This function will return NULL if it fails. When it returns NULL you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetColumnSortDir

C++: int pcItemColLayout::GetColumnSortDir(UINT32 index);

C: int pcItemColLayoutGetColumnSortDir(pcItemColLayout *layout, UINT32 index);

Perl: $uda_id = ProductCenter::ItemColLayout::GetColumnSortDir($index);

Gets the sort direction of one column in the layout. The index specifies the position in the layout (0 = left-most customizable column). The value returned is 0 if the display is not sorted by this column, 1 if it is sorted by ascending values in this column, and 2 if it is sorted by descending values.

This function will return 0 if it fails. When it returns 0 you should call pcCnxn::GetStatus() to learn if there was an error and to find out the nature of the error.

GetSort

C++: MSG_CODE pcItemColLayout::GetSort(UINT32 sort_index, UINT32 *col_index, const char **dir);

C: MSG_CODE pcItemColLayoutGetSort(pcItemColLayout *layout, UINT32 sort_index, UINT32 *col_index, const char **dir);

Perl: $msg_code = ProductCenter::ItemColLayout::GetSort($sort_index, $col_index, $dir);

Gets the selected sort column in the layout. 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 dir as "asc" (i.e. ascending order from top to bottom of the display grid) or "desc". (If no column has been specified for sorting, then the direction will be specified as "unsorted".)

GetId

C++: MSG_CODE pcItemColLayout::GetId(int *id);

C: MSG_CODE pcItemColLayoutGetId(pcItemColLayout *layout, int *id);

Perl: $msg_code = ProductCenter::ItemColLayout::GetId($id);

Gets the id number of the layout.

Layout storage functions

Save

C++: MSG_CODE pcItemColLayout::Save(int id);

C: MSG_CODE pcItemColLayoutSave(pcItemColLayout *layout, int id);

Perl: $msg_code = ProductCenter::ItemColLayout::Save($id);

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 global default item column layout.

Load

C++: MSG_CODE pcItemColLayout::Load(int id);

C: MSG_CODE pcItemColLayoutLoad(pcItemColLayout *layout, int id);

Perl: $msg_code = ProductCenter::ItemColLayout::Load($id);

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.

Delete

C++: MSG_CODE pcItemColLayout::Delete(int id);

C: MSG_CODE pcDeleteItemColLayoutDelete(pcItemColLayout *layout, int id);

Perl: $msg_code = ProductCenter::ItemColLayout::Delete($id);

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:

6

After establishing a connection and logging into a particular system and database, you often want to access information in the database.

List related functions

The following function calls get lists that are related to a connection.

ListChoiceLists

C++: pcList *pcCnxn::ListChoiceLists ();

C: pcList *pcCnxnListChoiceLists (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::List::ChoiceLists ();

Returns all of the choice lists.

ListChoiceListValues

C++: pcList *pcCnxn::ListChoiceListValues (const char *id);

C: pcList *pcCnxnListChoiceListValues (pcCnxn *cnxn, const char *id);

Perl: $list = ProductCenter::Cnxn::List::ChoiceListValues ($choice_$list);

Returns the valid values for the given choice list.

Note: If Release Management is turned on, the values of the choice list that defines valid release states are defined internally and have no relation to the CMS_SCODE table. If Release Management is turned off, the values of the choice list are taken from the CMS_SCODE database table.

ListUsers

C++: pcList *pcCnxn::ListUsers ();

C: pcList *pcCnxnListUsers (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListUsers ();

Returns a list of all of the users. This does NOT include disabled user accounts.

ListGroups

C++: pcList *pcCnxn::ListGroups ();

C: pcList *pcCnxnListGroups (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListGroups ();

Returns a list of all of the groups.

ListGroupUsers

C++: pcList *pcCnxn::ListGroupUsers (int group_id);

C: pcList *pcCnxnListGroupUsers (pcCnxn *cnxn, int group_id);

Perl: $list = ProductCenter::Cnxn::ListGroupUsers ($group_id);

Returns a list of all members of a specific group.

ListClasses

C++: pcList *pcCnxn::ListClasses ();

C: pcList *pcCnxnListClasses (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListClasses ();

Returns all of the classes in a list.

ListClassesByFilter

C++: pcList *pcCnxn::ListClassesByFilter (const char *property, const char *name);

C: pcList *pcCnxnListClassesByFilter (pcCnxn *cnxn, const char *property, const char *name);

Perl: $list = ProductCenter::Cnxn::ListClassesByFilter ($property, $name);

Returns a list of all classes filtered according to the input. The *property* argument must be either “shortname” or “fullname”, and the *name* argument must be the name of the class whose members you want returned. The function will return the class names (either short or full versions) of the classes under the class specified by the class argument.

ListUserDesktop

C++: pcList *pcCnxn::ListUserDesktop ();

C: pcList *pcCnxnListUserDesktop (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListUserDesktop ();

Returns a list of all items on a connected user's desktop.

ListUserWorkSpace

C++: pcList *pcCnxn::ListUserWorkSpace ();

C: pcList *pcCnxnListUserWorkSpace (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListUserWorkSpace ();

Returns a list of all items checked out by the connected user.

ListWhereUsed

C++: pcList *pcCnxn::ListWhereUsed (UINT32 id)

C: pcList *pcCnxnListWhereUsed (UINT32 id);

Perl: $list = ProductCenter::Cnxn::ListWhereUsed ($id);

Returns a list of the cms ids and link types, given the cms id of the tail item.

ListForms

C++: pcList *pcCnxn::ListForms ();

C: pcList *pcCnxnListForms (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListForms ();

Returns a list of all forms.

ListLinkTypes

C++: pcList *pcCnxn::ListLinkTypes ();

C: pcList *pcCnxnListLinkTypes (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListLinkTypes ();

Returns a list of all link type names and ids available in ProductCenter.

ListSavedQueries

C++: pcList *pcCnxn::ListSavedQueries ();

C: pcList *pcCnxnListSavedQueries (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListSavedQueries ();

Returns a list of all of the item query names saved by the connected user.

ListItemSpecificReports

C++: pcList pcCnxn::ListItemSpecificReports (const char *className);

C: pcList pcCnxnListItemSpecificReports (pcCnxn *cnxn, const char *className);

Perl: $list = ProductCenter::Cnxn::ListItemSpecificReports ($className);

Returns the list of report names and ids that are related to a class. Normally used with an item and retrieving class from the item.

ListItemByQueryReports

C++: pcList pcCnxn::ListItemByQueryReports (const char *parent);

C: pcList pcCnxnListItemByQueryReports (pcCnxn *cnxn, const char *parent);

Perl: $list= ProductCenter::Cnxn::ListItemByQueryReports ($parent);

Returns the list of report names and ids of query based reports and can be limited to only show reports within the designated branch of the report folder tree.. The parent is the id of the folder to retrieve. Use -1 for the root folder

ListCreateByActivityInstance

C++: pcList *pcList (pcCnxn *cnxn, pcActivityInst *act);

C: pcList *pcListCreateByActivityInst (pcCnxn *cnxn, pcActivityInst *act);

Perl: $list = new ProductCenter::List ($cnxn, $act);

Creates a list of a single pcActivityInst. The return list can then be used with the workflow functions “SendForward”, “SendBack”, “Reassign” and “Resume”.

.ListActivityQueries

C++: pcList *pcCnxn::ListActivityQueries ();

C: pcList *pcCnxnListActivityQueries (pcCnxn *cnxn);

Perl: $list = ProductCenter::Cnxn::ListActivityQueries ();

Loads the list of activity queries saved by the connected user.. The list returns the name and the id of the queries.

ListProcessQueries

C++: pcList *pcCnxn::ListProcessQueries ();

C: pcList *pcCnxnListProcessQueries (pcCnxn *cnxn);

Perl: $list= ProductCenter::Cnxn::ListProcessQueries ();

Loads the list of process queries saved by the connected user. The list returns the name and the id of the queries.

ListProcessSpecificReports

C++: pcList pcCnxn::ListProcessSpecificReports (const char *processDefinitionName);

C: pcList pcCnxnListProcessSpecificReports (pcCnxn *cnxn, const char *processDefinitionName);

Perl: $list = ProductCenter::Cnxn::ListProcessSpecificReports ($processDefinitionName);

Returns the list of report names and ids that are related to a process definition. Normally used with an activity or process and retrieving process definition name from the process. If using with an activity, get the process from the activity to get the process definition name.

ListProcessByQueryReports

C++: pcList pcCnxn::ListProcessByQueryReports (const char *parent);

C: pcList pcCnxnListProcessByQueryReports (pcCnxn *cnxn, const char *parent);

Perl: $list = ProductCenter::Report::ListProcessByQueryReports ($parent);

Returns the list of report names and ids of query based reports and can be limited to only show reports within the designated branch of the report folder tree. The parent is the id of the folder to retrieve. Use -1 for the root folder.

ListAllByQueryReports

C++: pcList pcCnxn::ListAllByQueryReports ();

C: pcList pcCnxnListAllByQueryReports (pcCnxn *cnxn);

Perl: $list = ProductCenter::Report::ListAllByQueryReports ();

Returns the list of report names and ids that are by query.

List object: constructors and destructors

The list object contains no *constructor* and one *destructor*.

This class has no constructor because the pcList is constructed in calls from objects that require lists to be returned as read-only information. Member functions of pcList then operate on the pcList much like you would use functions from a utility library.

ListDestroy (Destructor)

C++: ~pcList ();

C: static void pcListDestroy (pcList *list);

Perl: ProductCenter::List::DESTROY ();

The C++ destructor is invoked automatically when the object is destroyed, and all memory used by the list object is freed.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Obtaining information about lists

Use these functions to extract information about a list object.

GetRowCount

C++: MSG_CODE pcList::GetRowCount (UINT32, *row_count);

C: MSG_CODE pcListGetRowCount (pcList *list, UINT32 *row_count);

Perl: $msg_code = ProductCenter::List::GetRowCount ($row_count);

Returns the number of items in the specified list.

GetDisplayName

C++: const char *pcList::GetDisplayName (UINT32 index);

C: const char *pcListGetDisplayName (pcList *list, UINT32 index);

Perl: $displayname = ProductCenter::List::GetDisplayName ($index);

Returns the display name value of the list at the specified index. Depending on the type of list, this function may return an empty string.

GetSystemId

C++: const char *pcList::GetSystemId (UINT32 index);

C: const char *pcListGetSystemId (pcList *list, UINT32 index);

Perl: $name = ProductCenter::List::GetSystemId ($index);

Returns the system ID for the object. The ID is unique for a type of object but not across all objects. (**NOTE**: Some lists will have no identification numbers.)

GetListType

C++: MSG_CODE pcList::GetListType (pcListType *list_type);

C: MSG_CODE pcListGetListType (pcList *list, pcListType *list_type);

Perl: $msg_code = ProductCenter::List::GetListType ($list_type);

Returns the type of the list. The result of this call tell the application which of the other list functions can be used with this list. Possible return values are:

• listType_none
• listType_nameVal
• listType_processDef
• listType_processInst
• listType_activityDef
• listType_activityInst

Chapter 7

Items

Just Ahead:

7

You often write Toolkit programs because you need to manipulate items in the ProductCenter database: add, delete, or modify projects, files, or parts. This chapter covers the many functions that are available for use with the item object.

Item object: constructors and destructors

The item object contains two *constructors* and one *destructor*. The constructor serves different purposes depending on the argument passed to it.

CreateByClass

C++: pcItem *pcItem (pcCnxn *cnxn, const char *classname);

C: pcItem *pcItemCreateByClass (pcCnxn *cnxn, const char *classname);

Perl: $item = ProductCenter::Item ($cnxn, $classname);

Creates a new instance of an item object of the class you specify. This connection object specifies the connection with which the item object interacts. Note that the C++ and Perl constructors are identical to their “LoadbyId” function definitions so these functions behave differently depending on the argument passed to them. For “CreateByClass”, passing a string *classname* indicates that ProductCenter is to create a new item in the specified class. Perl programmers may wish to check out the convenience layer function “NewItem” on page 189.

LoadbyId

C++: pcItem *pcItem (pcCnxn *cnxn, long int cms_id);

C: pcItem *pcItemLoadById (pcCnxn *cnxn, long int cms_id);

Perl: $item = new ProductCenter::Item ($cnxn, $cms_id);

This function takes a pcCnxn pointer and the ID of an item (project or file), loads the item and an item object.

Perl programmers may wish to check out the convenience layer function “LoadItem” on page 189.

ItemDestroy

C++: ~pcItem ();

C: void pcItemDestroy (pcItem *item);

Perl: ProductCenter::Item::DESTROY ();

The item object, like all C++ objects, has one destructor. The destructor is invoked automatically when the object is destroyed, and all memory used by the item object is freed.

The C counterpart listed here frees memory that was allocated when the item object was created.

> **NOTE:** Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Loading items

Use these functions to load an item into memory prior to working on it.

LoadLatest

C++: pcItem *pcItem::LoadLatest ();

C: pcItem *pcItemLoadLatest (pcItem *item);

Perl: $Item = ProductCenter::Item::LoadLatest ();

Creates a new item object of the latest version of the specified item.

CreateClone

C++: pcItem *pcItem::pcItem (pcItem *item);

C: pcItem *pcItemCreateClone (pcItem *item);

Perl: $Item = ProductCenter::Item::CreateClone ($item);

Creates an exact copy of an item, including its attributes. This function is similar to the C++ Copy constructor, and the cloned item can be used to save information about an item that is to be changed. Note that the cloned item cannot be used to create new items, since the entire item is copied, including the cmsid, which needs to be unique in the database.

Setting values

SetAttr

C++: MSG_CODE pcItem::SetAttr (const char *attr_name, const char *value);

C: MSG_CODE pcItemSetAttr (pcItem *item, const char *attr_name, const char *value)

Perl: $msg_code= ProductCenter::Item::SetAttr ($attr_name, $attr_value);

Sets the value of an attribute. See "ProductCenter common attributes" on page 195 for more information.

The following rules apply to the attribute you set.

- If the attribute is *single-valued*, this function overwrites the previous value.
- If the attribute is *multi-valued*, separate the values with a pipe delimiter (|).
- Table-type attributes are of the form *attr_name[row].col_name*.
- Setting the "PLC" attribute requires either a single revision such as "D" or a more precise revision such as "D:5" which will set the revision D at the version level of 5.
- The precedence for finding the attribute you specify is to search in order:

  1.common attribute name

  2.custom attribute name

  3.common attribute prompt

  4.custom attribute prompt
- You can also specify custom attributes by placing the prefix **CUSTOM:** before the attribute name.

SetFileName

C++: void pcItem::SetFileName (const char *filename);

C: void pcItemSetFileName (pcItem *item, const char *filename);

Perl: ProductCenter::Item::SetFileName ($filename);

Use this function with any File item to set the filename portion (filename) of the local path attribute. Use this in conjunction with "SetWorkDir" to set a complete path.

If you do not set the filename, then the Toolkit uses the item name in its place, so you only need to call this function where the file name on disk is different from the file name to be added to ProductCenter.. "GetFileName" still returns NULL. The item name is a filename that is originally set at item creation.

This function applies primarily to inbound operations, such as "Add" and "Checkin" but can also be used with "Alter" to change a vaulted file associated with an item.

ClearCustomAttributes

C++: MSG_CODE pcItem::ClearCustomAttributes ();

C: MSG_CODE pcItemClearCustomAttributes (pcItem *item);

Perl: $msg_code = ProductCenter::Item::ClearCustomAttributes ();

Resets all of an item's custom attributes.

ResetCustomAttributes

C++: const char *pcItem::ResetCustomAttributes ();

C: const char *pcItemResetCustomAttributes (pcItem *item);

Perl: ProductCenter::Item::ResetCustomAttributes ();

Resets the item's form attributes to their default value.

DeleteTableTypeAttrRow

C++: MSG_CODE pcItem::DeleteTableTypeAttrRow (const char *attr_name);

C: MSG_CODE pcItemDeleteTableTypeAttrRow (pcItem *pcitem, const char *attr_name);

Perl: $msg_code = ProductCenter::DeleteTableTypeAttrRow ($attr_name);

Removes a row from a table-type attribute. *attr_name* has the form *table*[*row*] or *table*[*row*].*column*, which deletes the *row* for the attribute *table*.

SetAccess

C++: MSG_CODE pcItem::SetAccess (const char *type, const char *name, const char *perms);

C: MSG_CODE pcItemSetAccess (pcItem *item, const char *type, const char *name, const char *perms);

Perl: $msg_code = ProductCenter::Item::SetAccess ($type, $name, $perms);

Grants access rights to a user or group for the given item object. In this function,

- *type* is the form of access you are granting. This value can be either USER or GROUP.
- *name* is the name of the specific person or group to which you are giving notification privileges.
- *perms* specifies the permissions you are granting. The value is a string containing some combination of "R" (read), "W" (write), "D" (delete), and "N" (notify).

RemoveAccess

C++: MSG_CODE pcItem::RemoveAccess (const char *type, const char *name);

C: MSG_CODE pcItemRemoveAccess (pcItem *item, const char *type, const char *name);

Perl: $msg_code = ProductCenter::Item::RemoveAccess ($type, $name);

Removes the given user or group from the list of those who have access to the item object.

Getting values

GetAttr

C++: const char *pcItem::GetAttr (char *attr_name);

C: const char *pcItemGetAttr (pcItem *item, char *attr_name);

Perl: $attrValue = ProductCenter::Item::GetAttr ($attr_name);

Returns the value of the attribute. See "ProductCenter common attributes" on page 195 for more information.

GetAttr and SetAttr observe the following precedence when matching and attribute name:

1.common attribute name

2.custom attribute names

3.common attribute prompt

4.custom attribute prompt

You can also specify custom attributes by placing the prefix **CUSTOM:** before the attribute name.

Note that GetAttr("PLC") returns the formatted revision based on the revision sequence. See "Getting revision values" on page 109.for more information on retrieving revision values.

GetFileName

C++: const char *pcItem::GetFileName ();

C: const char *pcItemGetFileName (pcItem *pcItem);

Perl: $fileName = ProductCenter::Item::GetFileName ();

Returns the filename portion of the local path.

GetChoiceListAttrVals

C++: pcList *pcItem::GetChoiceListAttrVals (const char *attr_name);

C: pcList *pcItemGetChoiceListAttrVals (pcItem *item, const char *attr_name);

Perl: $list = ProductCenter::Item::GetChoiceListAttrVals ($attr_name);

If *attr_name* is a choice list attribute, this function returns a list of the valid values this attribute can assume.

GetFormEmbeddedObjectURL

C++: const char *pcItem::GetFormEmbeddedObjectURL (const char *url);

C: const char *pcItemGetFormEmbeddedObjectURL (pcItem *item, const char *url);

Perl: $embedded = ProductCenter::Item::GetFormEmbeddedObjectURL ($url);

Returns the resolved string of an embedded URL; it will convert the URL from a dynamic string to specific item. For 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 9.6.0.pdf" then this call

```
    item->GetFormEmbeddedObjectURL (item->GetAttr ("urlField"))
```

would yield this result

```
    src=pctr://webgetfile?PrdCtr Install 9.6.0.pdf
```

GetAccessCount

C++: MSG_CODE pcItem::GetAccessCount (const char *type, UINT32 *cnt);

C: MSG_CODE pcItemGetAccessCount (pcItem *item, const char *type, UINT32 *cnt);

Perl: $msg_code = ProductCenter::Item::GetAccessCount ($type, $count);

Returns the number of permission records for the item. The value is equal to the number of ProductCenter users for the "USER" type and the number of ProductCenter groups for the "GROUP" type.

GetAccessNameByIndex

C++: const char *pcItem::GetAccessNameByIndex (const char *type, UINT32 index);

C: const char *pcItemGetAccessNameByIndex(pcItem *item,const char *type, UINT32 index);

Perl: $name = ProductCenter::Item::GetAccessNameByIndex ($type, $index);

Returns the name of the ProductCenter user or group at specified index of the specified access list..

7

GetAccessPermsByIndex

C++: const char *pcItem::GetAccessPermsByIndex (const char *type, UINT32 index);

C: const char *pcItemGetAccessPermsByIndex (pcItem *item, const char *type, UINT32 index);

Perl: $PermName = ProductCenter::Item::GetAccessPermsByIndex ($type, $index);

Returns the permission string at the specified index of the specified access list.

R = Read, W = Write, N = Notify, D = Delete.

GetItemForm

C++: pcForm *pcItem::GetItemForm ();

C: pcForm *pcItemGetItemForm (pcItem *item);

Perl: $form = ProductCenter::Item::GetItemForm()

Returns a copy of the Item's Form. This function can be used with functions defined in"Forms" on page 179 to get access to an item's attribute information.

WhereUsed

C++: pcList *pcItem::WhereUsed ();

C: pcList *pcItemWhereUsed (pcItem *item);

Perl: $list = ProductCenter::Item::WhereUsed ();

Returns a list of item ids and link types of the links that uses this item.

Getting values by attribute index

The ProductCenter Toolkit can treat attributes of an item as if they are entries in an array, each with its own index. With this index, an application easily can get to each attribute of the object.

NOTE: ProductCenter supports two types of attributes, *single-valued attributes*, which take only one value at a time, and *multi-valued attributes*, which can have more than one value.

GetAttrCount

C++: MSG_CODE pcItem::GetAttrCount (UINT32 *attr_count);

C: MSG_CODE pcItemGetAttrCount (pcItem *item, UINT32 *attr_count);

Perl: $msg_code = ProductCenter::Item::GetAttrCount ($attr_count);

Returns the number of attributes for the specified item.

GetAttrNameByIndex

C++: const char *pcItem::GetAttrNameByIndex (UINT32 index);

C: const char *pcItemGetAttrNameByIndex (pcItem *item, UINT32 index);

Perl: $attrName = ProductCenter::Item::GetAttrNameByIndex ($index);

Returns the attribute name at the index specified.

GetAttrPromptByIndex

C++: const char *pcItem::GetAttrPromptByIndex (UINT32 index);

C: const char *pcItemGetAttrPromptByIndex (pcItem *item, UINT32 index);

Perl: $attrPrompt = ProductCenter::Item::GetAttrPromptByIndex ($index);

Gets the attribute prompt by the index.

GetAttrType

C++: MSG_CODE pcItem::GetAttrType (UINT32 index, pcAttrType *attr_type);

C: MSG_CODE pcItemGetAttrType (pcItem *item, UINT32 index, pcAttrType *attr_type);

Perl: $msg_code = ProductCenter::Item::GetAttrType ($index, $attr_type);

Returns the type of the attribute at the specified index. Possible return values are:

- attr_none
- attr_text
- attr_user
- attr_group
- attr_date
- attr_choice
- attr_fnum
- attr_inum
- attr_msg
- attr_textbox
- attr_tabletype
- attr_userrole
- attr_grouprole
- attr_id
- attr_class
- attr_bool

GetAttrIsCommonByIndex

C++: BOOL pcItem::GetAttrIsCommonByIndex (UINT32 index);

C: BOOL pcItemGetAttrIsCommonByIndex (pcItem *item, UINT32 index);

Perl: $isCommon = ProductCenter::Item::GetAttrIsCommonByIndex ($index);

Returns TRUE if the attribute at the specified index is a common attribute; returns FALSE if it the attribute is a custom attribute.

AttrIsRequired

C++: BOOL pcItem::AttrIsRequired (UINT32 index);

C: BOOL pcItemAttrIsRequired (pcItem *item, UINT32 index);

Perl: $isReqd = ProductCenter::Item::AttrIsRequired ($index);

Indicates whether an attribute is required for an item.

GetTableTypeAttrRowCount

C++: MSG_CODE pcItem::GetTableTypeAttrRowCount (const char *tt_attr_name, UINT32 *attr_row_count);

C: MSG_CODE pcItemGetTableTypeAttrRowCount (pcItem *item, const char *tt_attr_name, UINT32 *attr_row_count);

Perl: $msg_code = ProductCenter::Item::GetTableTypeAttrRowCount ($tt_attr_name);

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

GetTableTypeAttrColCount

C++: int pcItem::GetTableTypeAttrColCount (const char *tt_attr_name, UINT32 *attr_col_count);

C: int pcItemGetTableTypeAttrColCount (pcItem *item, const char *tt_attr_name, UINT32 *attr_col_count);

Perl: $count = ProductCenter::Item::GetTableTypeAttrColCount ($tt_attr_name);

Returns the number of columns in the specified table-type attribute.

GetTableTypeAttrColName

C++: const char *pcItem::GetTableTypeAttrColName (const char *tt_attr_name, UINT32 col_index);

C: const char *pcItemGetTableTypeAttrColName (pcItem *item, const char *tt_attr_name, UINT32 col_index);

Perl: $tableTypeAttrColName = ProductCenter::Item::GetTableTypeAttrColName ($tt_attr_name, $col_index);

Returns the name of the table-type attribute column at the specified index.

Getting revision values

These functions get access to an item’s next valid revision, to determine if a an item’s revision is a major, minor or legacy revision, or to get counts from the revision sequence assigned to the Class for the current item.

GetNextRevision

C++: char *pcItem::GetNextRevision ();

C: char *pcItemGetNextRevision (pcItem *item);

Perl: $nextRev = ProductCenter::Item::GetNextRevision ();

Returns the next revision defined in the revision sequence.

GetNextRevCount

C++: MSG_CODE pcItem::GetNextRevCount (UINT32 *count, char *action);

C: MSG_CODE pcItemGetNextRevCount (pcItem *item, UINT32 *count, char *action);

Perl: $msg_code = ProductCenter::Item::GetNextRevCount ($count, $action);

Returns the count of the next revisions. This count will include all valid next revisions, including minor and legacy revisions. The valid actions are "ADD", "CHECKIN", and "ALTER".

GetNextRevByIndex

C++: char *pcItem::GetNextRevByIndex (UINT32 index, char *action);

C: char *pcItemGetNextRevByIndex (pcItem *item, UINT32 index, char *action);

Perl: $nextRev = ProductCenter::Item::GetNextRevByIndex ($index, $action);

Returns the next revision based on the index. The valid actions are "ADD", "CHECKIN", and "ALTER".

GetNextRevIsLegacy

C++: BOOL pcItem::GetNextRevIsLegacy (UINT32 index, char *action);

C: BOOL pcItemGetNextRevIsLegacy (pcItem *item, UINT32 index, char *action);

Perl: $isLegacy = ProductCenter::Item::GetNextRevIsLegacy ($index, $action);

Returns a flag indicating if the revision has been defined as a legacy revision. The valid actions are "ADD", "CHECKIN", and "ALTER".

GetNextRevIsMinor

C++: BOOL pcItem::GetNextRevIsMinor (UINT32 index, char *action);

C: BOOL pcItemGetNextRevIsMinor (pcItem *item, UINT32 index, char *action);

Perl: $isMinor = ProductCenter::Item::GetNextRevIsMinor ($index, $action);

Returns a flag indicating if the revision has been defined as a minor revision. The valid actions are "ADD", "CHECKIN", and "ALTER".

GetNextRevGetParent

C++: char *pcItem::GetNextRevGetParent (UINT32 index, char *action);

C: char *pcItemGetNextRevGetParent (pcItem *item, UINT32 index, char *action);

Perl: $parentName = ProductCenter::Item::GetNextRevGetParent ($index, $action);

Returns the parent's name for the minor revision.   The valid actions are "ADD", "CHECKIN", and "ALTER".

Desktop functions

The following ProductCenter functions enable you to add and delete items from the dekstop of the connected user. The desktop state is not saved to the database until the client disconnects. Any other clients currently connected will not see the changed desktop, and will overwrite the desktop state when they disconnect.

AddToDesktop

C++: MSG_CODE pcItem::AddToDesktop (const char *type);

C: MSG_CODE pcItemAddToDesktop ();

Perl: $msg_code = ProductCenter::Item::AddToDesktop ($item);

Adds the item to the Desktop.

DeleteFromDesktop

C++: MSG_CODE pcItem::DeleteFromDesktop ();

C: MSG_CODE pcItemDeleteFromDesktop (pcItem *pcitem);

Perl: $msg_code = ProductCenter::Item::DeleteFromDesktop ($item);

Deletes the item from the Desktop.

Inbound functions

With the following functions, your Toolkit programs can perform four of the inbound functions in the ProductCenter user interface.

Add

C++: MSG_CODE pcItem::Add ();

C: MSG_CODE pcItemAdd (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Add ();

Adds a new item to the database. The function is equivalent to choosing **File** → **New** from the ProductCenter user interface.

Checkin

C++: MSG_CODE pcItem::Checkin ();

C: MSG_CODE pcItemCheckin (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Checkin ();

Checks in a new version of an existing item. The function is equivalent to choosing **Action** → **Check in** from the ProductCenter user interface.

ForcedCheckin

C++: MSG_CODE pcItem::ForcedCheckin ();

C: MSG_CODE pcItemForcedCheckin (pcItem *item);

Perl: $msg_code = ProductCenter::Item::ForcedCheckin ();

Checks in an item even when identical to previous version.

Uncheckout

C++: MSG_CODE pcItem::Uncheckout ();

C: MSG_CODE pcItemUncheckout (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Uncheckout ();

Returns unmodified a checked-out item. The function is equivalent to choosing **Action** → **Return unmodified** from the ProductCenter user interface.

Outbound functions

With the following functions, your Toolkit programs can perform two of the outbound functions in the ProductCenter user interface: Get Copy and Check out.

GetCopy

C++: MSG_CODE pcItem::GetCopy ();

C: MSG_CODE pcItemGetCopy (pcItem *item);

Perl: $msg_code = ProductCenter::Item::GetCopy ();

Gets a read-only copy of a file. The item is *not* checked out when your program invokes this routine. The function can be performed regardless of whether the item is checked out.

The function is equivalent to choosing **Action** → **Get Copy** from the ProductCenter user interface.

NOTE: You use "GetCopy" only when working with objects that have file data (that is, a physical file associated with the object).

If you anticipate performing a GetCopy on many files at once (for example, more than several hundred), note that you are limited by your system memory. Each file in the GetCopy uses about 3 KB of memory, and a small amount of temporary disk space. Therefore, if you perform a GetCopy on 3,000 files, the operation might fail on one machine while succeeding on another. When performing large operations like this, we recommend that you monitor your system resources and ensure that you don't exceed them.

Checkout

C++: MSG_CODE pcItem::Checkout ();

C: MSG_CODE pcItemCheckout (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Checkout ();

Checks out a file or an item. If the item has an associated file, the function also puts the file data into your working directory. You cannot check out a file that has the same name as a file that you already have checked out. If you attempt to do so, ProductCenter displays a message that states you can only check out the second file after you either check in or return unmodified the first file.

The function above is equivalent to choosing **Action** → **Check out** from the ProductCenter user interface.

Item manipulation functions

DeleteItem

C++: MSG_CODE pcItem::DeleteItem ();

C: MSG_CODE pcItemDeleteItem (pcItem *item);

Perl: $msg_code = ProductCenter::Item::DeleteItem ();

Deletes item from the current revision through its initial revision. The function is equivalent to choosing **Action** → **Delete** from the ProductCenter user interface. In 8.2 and later, "DeleteItem" also automatically removes the deleted item from your Desktop.

Purge

C++: MSG_CODE pcItem::Purge ();

C: MSG_CODE pcItemPurge (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Purge ();

Removes all but the latest version of a file or project from a database. The function is equivalent to choosing **Action** → **Purge** from the user interface.

PurgeLatest

C++: MSG_CODE pcItem::PurgeLatest ();

C: MSG_CODE pcItemPurgeLatest (pcItem *item);

Perl: $msg_code = ProductCenter::Item::PurgeLatest ();

Removes the latest version of an item.

CheckVersionsForRemoval

C++: pcList *pcItem::CheckVersionsForRemoval (const char *versionList);

C: pclist *pcItemCheckVersionsForRemoval (pcItem *item, const char *versionList);

Perl: $list = ProductCenter::Item::CheckVersionsForRemoval ($versionList);

Returns a list of revisions and warning messages to be used when deleting or purging items. The versionList 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 versionList 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..

SelectivePurge

C++: MSG_CODE pcItem::SelectivePurge (const char *VersionList);

C: MSG_CODE pcItemSelectivePurge (pcItem *item, const char *VersionList);

Perl: $msg_code = ProductCenter::Item::SelectivePurge ($version_list);

Allows removal of specific versions of an item. For example, you can use “SelectivePurge” to delete the first, second, fifth, and seventh items in a list.

Call this function on the latest version of the item. The version list is a comma- and hyphen-delimited list of version numbers (for example, (“1-3, 7, 9”)).

When you perform a selective purge, ProductCenter renumbers the remaining versions. For example, if you delete the first two versions in a version list, the item that was formerly version 3 becomes version 1, the item that was version 4 becomes version 2, and so on.

Move

C++: MSG_CODE pcItem::Move (const char *new_class_name);

C: MSG_CODE pcItemMove (pcItem *item, const char *new_class_name);

Perl: $msg_code = ProductCenter::Item::Move ($new_class_name);

Relocates an item from its existing class to a new class specified by new_class_name. The function is equivalent to choosing **Action** → **Move** from the ProductCenter user interface.

Alter

C++: MSG_CODE pcItem::Alter ();

C: MSG_CODE pcItemAlter (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Alter ();

Changes the specified item's file or metadata — such as its name, title, description, current stage, and so on — without checking out the item or creating a new version.

This function is equivalent to choosing **Action** → **Alter** from the ProductCenter user interface.

Rollback

C++: MSG_CODE pcItem::Rollback ();

C: MSG_CODE pcItemRollback (pcItem *item);

Perl: $msg_code = ProductCenter::Item::RollBack ();

Rolls the item back to a previous version. This function is equivalent to choosing **Action** → **Rollback** from the ProductCenter user interface. (In practice, it might help to think of this as a "Copy Forward" operation. That is, you specify that a previous version is to be copied and made the most recent version.)

Release management functions

The following functions support ProductCenter's Release Management functionality: Submit, Approve, Disapprove and Obsolete.

Submit

C++: MSG_CODE pcItem::Submit (const char *full_config);

C: MSG_CODE pcItemSubmit (pcItem *item, const char *full_config);

Perl: $msg_code = ProductCenter::Item::Submit ($full_config);

Submits an item for approval. This function is equivalent to choosing **Action** → **Submit** from the ProductCenter user interface. Set *full_config* to YES if the entire hierarchy is to be submitted, NO if just the item is to be submitted. The latter case will fail with an error if there are "In Progress" items in the hierarchy.

Approve

C++: MSG_CODE pcItem::Approve (const char *full_config);

C: MSG_CODE pcItemApprove (pcItem *item, const char *full_config);

Perl: $msg_code = ProductCenter::Item::Approve ($full_config);

Approves an item that has been submitted for approval. This function is equivalent to choosing **Action** → **Approve** from the ProductCenter user interface. Set *full_config* to YES if the entire hierarchy is to be approved, NO if just the item is to be approved. The latter case will fail with an error if there are unreleased items in the hierarchy.

Disapprove

C++: MSG_CODE pcItem::Disapprove ();

C: MSG_CODE pcItemDisapprove (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Disapprove ();

Disapproves an item that has been submitted for approval. This function is equivalent to choosing **Disapprove** from the **Action** menu in the ProductCenter user interface.

To disapprove an entire hierarchy, you would ned to loop through all items in the hierarchy from the top down.

Obsolete

C++: MSG_CODE pcItem::Obsolete ();

C: MSG_CODE pcItemObsolete (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Obsolete ();

Marks the specified item as Obsolete. Obsolete items are no longer used and can not be linked to. The links going to this item are also marked as Obsolete. The only functions that are permitted on Obsolete items are: View info, View file, Get Copy, Delete, Purge, Move, Reinstate, and Save As. See Chapter 7 of the *ProductCenter Windows User Guide* for details about the Obsolete state.

Reinstate

C++: MSG_CODE pcItem::Reinstate ();

C: MSG_CODE pcItemReinstate (pcItem *item);

Perl: $msg_code = ProductCenter::Item::Reinstate ();

Reverses the effect of marking an item as obsolete. If the toolkit user or administrator did special handling of the links going to this item they will need to undo those changes.

Identification functions

IsFile

C++: BOOL pcItem::IsFile ();

C: BOOL pcItemIsFile (pcItem *item);

Perl: $isFile = ProductCenter::Item::IsFile ();

Returns TRUE if an item object contains a file and FALSE if it does not.

IsProject

C++: BOOL pcItem::IsProject ();

C: BOOL pcItemIsProject (pcItem *item);

Perl: $isProject = ProductCenter::Item::IsProject ();

Returns TRUE if an item object contains a project and FALSE if it does not.

IsPart

C++: BOOL pcItem::IsPart ();

C: BOOL pcItemIsPart (pcItem *item);

Perl: $isPart = ProductCenter::Item::IsPart ();

Returns TRUE if an item object contains a part item (that is, an item that resides in the CMS:Parts class or subclass) and FALSE if not.

IsLatest

C++: BOOL pcItem::IsLatest ();

C: BOOL pcItemIsLatest (pcItem *item);

Perl: $isLatest = ProductCenter::Item::IsLatest ();

Returns TRUE if an item object is the latest version and FALSE if it is not.

IsCheckedOut

C++: BOOL pcItem::IsCheckedOut ();

C: BOOL pcItemIsCheckedOut (pcItem *item);

Perl: $isOut = ProductCenter::Item::IsCheckedOut ();

Returns TRUE if an item has been checked out and FALSE if it has not.

IsCheckedOutByMe

C++: BOOL pcItem::IsCheckedOutByMe ();

C: BOOL pcItemIsCheckedOutByMe (pcItem *item);

Perl: $isMine = ProductCenter::Item::IsCheckedOutByMe ();

Returns TRUE if an item is checked out to the current user and FALSE if it is not.

IsObsolete

C++: BOOL pcItem::IsObsolete ();

C: BOOL pcItemIsObsolete (pcItem *item);

Perl: $msg_code = ProductCenter::Item::IsObsolete ();

Returns TRUE or FALSE depending on whether the item is in the Obsolete state.

Chapter 8

Links

Just Ahead:

One of the most powerful features of ProductCenter is its *named links* capability, which gives users the ability to define associations between items and define attributes for those associations.

This chapter covers the Toolkit calls that give you access to this functionality.

Link object: constructors and destructors

The link object contains one *constructor* and one *destructor*.

LinkCreate

C++: pcLink (pcCnxn *cnxn, const char *link_type);

C: pcLink *pcLinkCreate (pcCnxn *cnxn, const char *link_type);

Perl: $link = new ProductCenter::Link ($cnxn, $link_type);

Creates a new instance of a link object of the link type specified.

LinkDestroy

C++: ~pcLink ();

C: void pcLinkDestroy (pcLink *link);

Perl: ProductCenter::Link::DESTROY ()

The destructor is a public function. You can use the *~pcLink()* destructor to destroy links that are returned by the link class.

NOTE: Perl programmers should read “Destructors and Perl” on page 43 for information as to why they should not use this call.

Link object functions

GetHead

C++: pcItem *pcLink::GetHead ();

C: pcItem *pcLinkGetHead (pcLink *link);

Perl: $item = ProductCenter::Link::GetHead ();

Returns the item object at the head of the link.

GetTail

C++: pcItem *pcLink::GetTail ();

C: pcItem *pcLinkGetTail (pcLink *link);

Perl: $item = ProductCenter::Link::GetTail ();

Returns the item object at the tail of the link.

IsHierarchical

C++: BOOL pcLink::IsHierarchical ();

C: BOOL pcLinkIsHierarchica l(pcLink *pclink);

Perl: $isHier = ProductCenter::Link:: IsHierarchical ();

Returns TRUE if the link is hierarchical, FALSE if the link is non-hierarchical.

IsLatest

C++: BOOL pcLink::IsLatest ();

C: BOOL pcLinkIsLatest (pcLink *link);

Perl: $isLatest = ProductCenter::Link::IsLatest ();

Returns TRUE if the link is the latest version.

GetLinkForm

C++: pcForm *pcLink::GetLinkForm ();

C: pcForm *pcLinkGetLinkForm (pcLink *link);

Perl: $form = ProductCenter::Link::GetLinkForm ($link);

Returns the link's form object or null if no view is defined on the link. This object can be used with functions defined in"Forms" on page 179 to get access to an item's attribute information.

Link attribute functions

GetAttr

C++: const char *pcLink::GetAttr (const char *attr_name);

C: const char *pcLinkGetAttr (pcLink *link, const char *attr_name);

Perl: $attrValue= ProductCenter::Link::GetAttr ($attr_name);

Returns the value of the attribute. Use "GetAttrCount" and "GetAttrNameByIndex" on page 122 to retrieve the list of attributes on a link.

GetAttrCount

C++: MSG_CODE pcLink::GetAttrCount (UINT32 *attr_count);

C: MSG_CODE pcLinkGetAttrCount (pcLink *link, UINT32 *attr_count);

Perl: $msg_code = ProductCenter::Link::GetAttrCount ($attr_count);

Returns the number of attributes for the specified link. Note that not all links have attributes defined.

GetAttrNameByIndex

C++: const char *pcLink::GetAttrNameByIndex (UINT32 index);

C: const char *pcLinkGetAttrNameByIndex (pcLink *link, UINT32 index);

Perl: $attrName =ProductCenter::Link::GetAttrNameByIndex ($index);

Returns the attribute name at the index specified.

GetAttrType

C++: MSG_CODE pcLink::GetAttrType (UINT32 index, pcAttrType *attr_type);

C: MSG_CODE pcLinkGetAttrType (pcLink *link, UINT32 index, pcAttrType *attr_type);

Perl: $msg_code = ProductCenter::Link::GetAttrType ($index, $attr_type);

Returns the type of attribute at the specified index. See the list of attribute types on page page 194.

AttrIsRequired

C++: BOOL pcLink::AttrIsRequired (UINT32 index);

C: BOOL pcLinkAttrIsRequired (pcLink *pclink, UINT32 index);

Perl: $isReqd = ProductCenter::Link::AttrIsRequired ($index);

Returns TRUE if the link has a required custom attribute.

GetTableTypeAttrRowCount

C++: MSG_CODE pcLink::GetTableTypeAttrRowCount(const char *tt_attr_name, UINT32 *attr_row_count);

C: MSG_CODE pcLinkGetTableTypeAttrRowCount(pcLink *link const char *tt_attr_name, UINT32 *attr_row_count);

Perl: $msg_code = ProductCenter::Link::GetTableTypeAttrRowCount ($attr_name,$row_count);

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

GetTableTypeAttrColCount

C++: MSG_CODE pcLink::GetTableTypeAttrColCount (const char *tt_attr_name, UINT32 *attr_col_count);

C: MSG_CODE pcLinkGetTableTypeAttrColCount (pcLink *link, const char *tt_attr_name, UINT32 *attr_col_count);

Perl: $msg_code = ProductCenter::Link::GetTableTypeAttrColCount ($attr_name, $col_count);

Returns the number of columns in the specified table-type attribute.

GetTableTypeAttrColName

C++: const char *pcLink::GetTableTypeAttrColName (const char *tt_attr_name, UINT32 col_index);

C: const char *pcLinkGetTableTypeAttrColName (pcLink *link, const char *tt_attr_name, UINT32 col_index);

Perl: $colName = ProductCenter::Link::GetTableTypeAttrColName ($attr_name, $index);

Returns the name of the table-type attribute column at the specified index.

GetChoiceListAttrVals

C++: pcList *pcLink::GetChoiceListAttrVals (const char *tt_attr_name);

C: pcList *pcLinkGetChoiceListAttrVals (pcLink *link, const char *tt_attr_name);

Perl: $list = ProductCenter::Link::GetChoiceListAttrVals ($attr_name);

If *attr_name* is a choice list attribute, this function returns a list of the valid values this attribute can assume.

SetAttr

C++: MSG_CODE *pcLink::SetAttr (const char *attr_name, const char *attr_value);

C: MSG_CODE *pcLinkSetAttr (pcLink *link, const char *attr_name, const char *attr_value);

Perl: $msg_code = ProductCenter::Link::SetAttr ($attr_name);

Sets the value of an attribute on a link. The following rules apply:

- If the attribute is *single-valued*, this function overwrites the previous value.
- If the attribute is *multi-valued*, separate the values with a pipe delimiter (|).
- Table-type attributes are of the form *attr_name[row].col_name*.

You can use custom attributes by placing the prefix **CUSTOM:** before the attribute name.

DeleteTableTypeAttrRow

C++: const char *pcLink::DeleteTableTypeAttrRow (const char *name);

C: const char *pcLinkDeleteTableTypeAttrRow (pcLink *link, const char *name);

Perl: ProductCenter::Link::DeleteTableTypeAttrRow ($name);

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.

Item object link functions

GetLinkCount

C++: MSG_CODE pcItem::GetLinkCount (const char *link_type, UINT32 *link_count);

C: MSG_CODE pcItemGetLinkCount (pcItem *item, const char *link_type,
UINT32 *link_count);

Perl: $msg_code = ProductCenter::Item::GetLinkCount ($link_type, $link_count);

Returns the number of links of the specified type, owned by the specified item object.

GetLink

C++: pcLink *pcItem::GetLink (const char *link_name, UINT32 index);

C: pcLink *pcItemGetLink (pcItem *item, const char *link_name, UINT32 index);

Perl: $link = ProductCenter::Item::GetLink ($link_name, $index);

Returns the link object of the specified type at the specified index.

GetLinkedItem

C++: pcItem *pcItem::GetLinkedItem (const char *link_name, UINT32 index);

C: pcItem *pcItemGetLinkedItem (pcItem *item, const char *link_name, UINT32
index);

Perl: $item = ProductCenter::Item::GetLinkedItem ($link_name,$index);

Returns the linked item at the specified index.

GetHeadLinkCount

C++: MSG_CODE pcItem::GetHeadLinkCount (const char *linkType, UINT32 *count);

C: MSG_CODE pcItemGetHeadLinkCount (pcItem *item, const char *linkType, UINT32 *count);

Perl: $msg_code = ProductCenter::Item::GetHeadLinkCount ($linkType, $count);

Returns the number of the head links that are attached to the item, given the link type.

GetHeadLink

C++: pcLink *pcItem::GetHeadLink (const char *linkType, UINT32 index);

C: pcLink *pcItemGetHeadLink (pcItem *item, const char *linkType, UINT32 index);

Perl: $link = ProductCenter::Item::GetHeadLink ($linkType, $index);

Returns the head link based on the index. This function can only be called after "GetHeadLinkCount".

GetTailLinkCount

C++: MSG_CODE pcItem::GetTailLinkCount (const char *linkType, UINT32 *count);

C: MSG_CODE pcItemGetTailLinkCount (pcItem *item, const char *linkType, UINT32 *count);

Perl: $msg_code = ProductCenter::Item::GetTailLinkCount ($linkType, $count);

Returns the count of the tail links that are attached to the item, given the link type.

GetTailLink

C++: pcLink *pcItem::GetTailLink (const char *linkType, UINT32 index);

C: pcLink *pcItemGetTailLink (pcItem *item, const char *linkType, UINT32 index);

Perl: $link = ProductCenter::Item::GetTailLink ($linkType, $index);

Returns the tail link based on the index. This can only to be called after the function "GetTailLinkCount".

AddLink

C++: MSG_CODE pcItem::AddLink (pcLink *link, pcItem *tail_item);

C: MSG_CODE pcItemAddLink (pcItem *item, pcLink *link, pcItem *tail_item);

Perl: $msg_code = ProductCenter::Item::AddLink ($link,$tail_item);

Adds a link of the specified type with the *tail_item* at the other end of the link.

Note that:

- The *tail_item* has to exist in the databse for “AddLink” to work properly.
- An “AddLink” call invalidates the previous value returned by “GetLinkCount”.
- The user owns the memory for the link object and the tail item.

If you use AddLink to add a link to an obsolete item, AddLink will report success, but you will not be able to check in or alter the item. See Chapter 7 of the ProductCenter Windows User Guide for more information about obsolete items.

You must call the item’s “Alter”, “Checkin” or “Add” function after you call “AddLink” (see the code example on the next page). If you do not call “Alter”, the ProductCenter GUI does not show the link defined by “AddLink”.

AddHeadLink

C++: MSG_CODE pcItem::AddHeadLink (pcLink *link, pcItem *head);

C: MSG_CODE pcItemAddHeadLink (pcItem *item, pcLink *link, pcItem *head);

Perl: $msg_code = ProductCenter::Item::AddHeadLink ($link, $head);

Adds the tail-to-head link to the item.

AddTailLink

C++: MSG_CODE pcItem::AddTailLink (pcLink *link, pcItem *tail);

C: MSG_CODE pcItemAddTailLink (pcItem *item, pcLink *link, pcItem *tail);

Perl: $msg_code = ProductCenter::Item::AddTailLink ($link, $tail);

Adds the head-to-tail link to the item.

UpdateLink

C++: MSG_CODE pcItem::UpdateLink (pcLink *link);

C: MSG_CODE pcItemUpdateLink (pcItem *item, pcLink *link);

Perl: $msg_code = ProductCenter::Item::UpdateLink ($link);

Refreshes link information. If you check out then check in a project (A:1 -> A:2) the link info for an object B linked to A may become outdated, with B pointing to the old object (A:1). By updating B, the link then correctly points to A:2.

ReplaceLink

C++: MSG_CODE pcItem::Replace (pcLink *link, pcItem *newTail);

C: MSG_CODE pcItemReplace (pcItem *pcitem, pcLink *link, pcItem *newTail);

Perl: $msg_code = ProductCenter::Item::Replace ($link, $newtail);

Replaces the tail item on a link on a item. To use this function, find the item and load it. Load the links on the item and find the link that you want to change. After the link is found, call the Replace with the link and the new tail item.

RemoveLink

C++: MSG_CODE pcItem::RemoveLink (pcLink *link);

C: MSG_CODE pcItemRemoveLink (pcItem *item, pcLink *link);

Perl: $msg_code = ProductCenter::Item::RemoveLink ($link);

Removes the specified link.

GetLinkTypeCount

C++: MSG_CODE pcItem::GetLinkTypeCount (UINT32 *count);

C: MSG_CODE pcItem GetLinkTypeCount (pcItem *item, UINT32 *count);

Perl: $msg_code = ProductCenter::Item::GetLinkTypeCount ($count);

Returns a count of available link types for the item. Prior to 8.2, a particular link type would only add "1" to the count, but as of 8.2, 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.

GetLinkType

C++: const char *pcItem::GetLinkType (UINT32 index);

C: const char *pcItem GetLinkType (pcItem *item, UINT32 index);

Perl: $label= ProductCenter::Item:: GetLinkType ($index);

Returns the link type name at the specified index. Precede this function with a call to "GetHeadLinkCount", and use "GetLinkTypeIsHead", if necessary, to determine whether this is a head-to-tail or tail-to-head link.

GetLinkTypeLabel

C++: const char *pcItem::GetLinkTypeLabel (UINT32 index);

C: const char *pcItemGetLinkTypeLabel (pcItem *item, UINT32 index);

Perl: $label = ProductCenter::Item::GetLinkTypeLabel ($index);

Returns the label based on the index. This can only be called after “GetHeadLinkCount”.

GetLinkTypeIsHead

C++: BOOL pcItem::GetLinkTypeIsHead (UINT32 index);

C: BOOL pcItemGetLinkTypeIsHead (pcItem *item, UINT32 index);

Perl: $isHead = ProductCenter::Item::GetLinkTypeIsHead ($index);

Returns TRUE if the LinkType specified by the index is the head. This can only be called after the “GetHeadLinkCount”.

Chapter 9

Queries and Reports

Just Ahead:

A common requirement of Toolkit programs is to query the database for specific information. The ProductCenter Toolkit provides three ways of querying:

- query clause queries
- where clause queries
- checked out by queries

Query clause queries make use of the AddAttrClause function, and allow you to construct attribute-based queries similar to the ones you encounter in the ProductCenter search interface using datatype-dependent conditions such as “begins with”, “contains”, “not equal”, “one of”, etc.

Where clause queries make use of the ByWhereClause and AddWhereTable functions, and are more similar to using SQL “select * from” queries of a database in which you first identify the tables with which you wish to work.

Checked out by queries can only be used to get a list of files checked out by a user.

This chapter also includes functions related to baselines and reports which implement the Report object.

Query object

Constructors and Destructors

The query object contains one *constructor* and one *destructor*.

QryCreate

C++: pcQry (pcCnxn *cnxn);

C: pcQry *pcQryCreate (pcCnxn *cnxn);

Perl: $qry = new ProductCenter::Qry ($cnxn);

Creates a new query object.

QryDestroy

C++: ~pcQry ();

C: void pcQryDestroy (pcQry *qry);

Perl: ProductCenter::Qry::DESTROY ();

The query object, like all C++ objects, has one *destructor*. The destructor is invoked automatically when the object is destroyed, and all memory used by the query object is freed. The C counterpart frees memory that you allocated with *pcQryCreate()*.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Constructing the query

There are three types of queries: "Query clause queries", "Where clause queries" and "Checked Out By" based queries. "Query Clause" and "Where Clause" queries can query for item, activity or process objects. "Checked Out By" queries can only query item objects. Setting the type of object to query is performed with the "SetQueryType" function.

SetQueryType

C++: MSG_CODE pcQry::SetQueryType (const char *type);

C: MSG_CODE pcQrySetQueryType (pcQry *qry, const char *type);

Perl: $msg_code = ProductCenter::Qry::SetQueryType ($type);

Sets the type of query that will be created. The possible values are "Process", "Activity", and "Item". Once clauses have been added to the query, the type cannot be changed. If you do not make a call to SetQueryType, the type defaults to "Item".

SetCaseSensitive

C++: MSG_CODE pcQry::SetCaseSensitive (BOOL casesensitive);

C: MSG_CODE pcQrySetCaseSensitive (pcQry *qry, BOOL casesensitive);

Perl: $msg_code = ProductCenter::Qry::SetCaseSensitive ($casesensitive);

Sets the case sensitivity of the query.

Query clause queries

AddAttrClause

C++: MSG_CODE pcQry::AddAttrClause (const char *view, const char *attr_name, QryCondition condition, const char *value);

C: MSG_CODE pcQryAddAttrClause (pcQry *qry, const char *view, const char *attr_name, QryCondition condition, const char *value);

Perl: $msg_code = ProductCenter::Qry::AddAttrClause ($view_name, $attr_name, $query_condition, $attr_value);

Adds an attribute clause to a query-based query.

For item queries, *view* is either "Common Attributes" or the derived form (custom view) you created of a master form (attribute table). For process and activity queries, *view* should be set to “Process” or “Activity”, respectively.

For item queries, *attr_name* specifies the item attribute for which you want to build the query clause. For process and activity queries, *attr_name* values can be found using the “GetWfAttrPromptCount” and “GetWfAttrPrompt” functions (see page 141).

The *condition* is the term or operator that constrains the clause.

The *value* is the measure by which you are constraining the attribute.

See Appendix B, “Attribute Types” for more information about common 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	
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	

Table 9-1: Valid query conditions for various attributes

Attribute Type	Attribute Names	Valid Conditions	Comments
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
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.

SetMatchAllClauses

C++: MSG_CODE pcQry::SetMatchAllClauses (Bool flag);

C: MSG_CODE pcQrySetMatchAllClauses (pcQry *qry, Bool flag);

Perl: $msg_code = ProductCenter::Qry::SetMatchAllClauses ($AllFlag);

Set the value to TRUE if you want to perform an AND query. Set it to FALSE to perform an OR query. Note that OR queries are apt to return too many results to be useful. You can use “SetMatchAllClauses” only for query clause queries. To make effective use of Boolean searches, you should consider using “ByWhereClause”.

RemoveClauseItem

C++: void pcQry::RemoveClauseItem (UINT32 index);

C: void pcQryRemoveClauseItem (pcQry *pcqry, UINT32 index);

Perl: ProductCenter::Qry::RemoveClauseItem ($index);

Removes a specific search clause from the loaded query.

RemoveAllClauseItem

C++: void pcQry::RemoveAllClauseItem ();

C: void pcQryRemoveAllClauseItem (pcQry *pcqry);

Perl: ProductCenter::Qry::RemoveAllClauseItem ();

Removes all search clauses from a loaded query. This is useful for modifying a query prior to replacing a saved version.

GetClauseCount

C++: MSG_CODE pcQry::GetClauseCount (UINT32 *clause_count);

C: MSG_CODE pcQryGetClauseCount (pcQry *qry, UINT32 *clause_count);

Perl: $msg_code = ProductCenter::Qry::GetClauseCount ($clause_count);

Returns the number of clauses that are defined within a query. This could be useful when creating a graphical user interface where you are building a menu on the fly.

GetClauseAttr

C++: const char *pcQry::GetClauseAttr (UINT32 index, const char *type);

C: const char *pcQryGetClauseAttr (pcQry *qry, UINT32 index, const char *type);

Perl: $name = ProductCenter::Qry::GetClauseAttr ($index, $attr_type);

Returns a clause entered with “AddAttrClause”. The argument ‘type’ can be View, AttrName, Condition, AttrPrompt, Value, or ChoiceListID. In addition, a value of “View Prompt” will return the attribute prompt which is needed for Table Type Attributes, or a value of “Main View” will return the Form ID of the form containing the Table Type Attributes. For process and activity queries, AttrName and AttrPrompt return the same value.

Where clause queries

ByWhereClause

C++: MSG_CODE pcQry::ByWhereClause (const char *where_clause);

C: MSG_CODE pcQryByWhereClause (pcQry *qry, const char *where_clause);

Perl: $msg_code = ProductCenter::Qry::ByWhereClause ($where_clause);

Constructs a where-clause query. You must follow this function with “AddWhereTable” to add a table to the list of tables used by the query.

AddWhereTable

C++: MSG_CODE pcQry::AddWhereTable (const char *table_name);

C: MSG_CODE pcQryAddWhereTable (pcQry *qry, const char *table_name);

Perl: $msg_code = ProductCenter::Qry::AddWhereTable ($where_table);

Adds the table to the list of tables employed in your where-based query.

ClearWhereTable

C++: MSG_CODE pcQry::ClearWhereTable ();

C: MSG_CODE pcQryClearWhereTable (pcQry *pcqry);

Perl: $msg_code = ProductCenter::Qry::ClearWhereTable ();

Removes the tables in the where table used with “AddWhereTable”. When using the “ByWhereClause” function, the “AddWhereTable” function needs to be called to tell the query what table to include in the select statement. The “ClearWhereTable” will remove all the values added by “AddWhereTable” function.

CheckedOutBy queries

QueryCheckedOutBy

C++: MSG_CODE pcQry::QueryCheckedOutBy (const char *userName);

C: MSG_CODE pcQryQueryCheckedOutBy (pcQry *qry, const char *userName);

Perl: $msg_code = ProductCenter::Qry::QueryCheckedOutBy ($userName);

Defines a query which when executed will return the items which are checked out by userName.

Identification functions

IsCaseSensitive

C++: BOOL pcQry::IsCaseSensitive ();

C: BOOL pcQryIsCaseSensitive (pcQry *qry);

Perl: $isCaseSensitive = ProductCenter::Qry::IsCaseSensitive ();;

Returns TRUE if the query is case sensitive or FALSE if it is not..

IsQueryClauseBased

C++: BOOL pcQry::IsQueryClauseBased ();

C: BOOL pcQryIsQueryClauseBased (pcQry *qry);

Perl: $isQueryBased = ProductCenter::Qry::IsQueryClauseBased ();

Returns TRUE if the query is “query clause” based or FALSE if it is not..

IsWhereClauseBased

C++: BOOL pcQry::IsWhereClauseBased ();

C: BOOL pcQryIsWhereClauseBased (pcQry *qry);

Perl: $isWhereBased = ProductCenter::Qry::IsWhereClauseBased ();

Returns TRUE if the query is “where clause” based or FALSE if it is not..

IsQueryCheckedOutBy

C++: BOOL pcQry::IsQueryCheckedOutBy ();

C: BOOL pcQryIsQueryCheckedOutBy (pcQry *qry);

Perl: $isCheckedOutBy = ProductCenter::Qry::IsQueryCheckedOutBy();

Returns TRUE if the query is “checked out by” based or FALSE if it is not..

MatchesAllClauses

C++: BOOL pcQry::MatchesAllClauses ();

C: BOOL pcQryMatchesAllClauses (pcQry *qry);

Perl: $matchesAll = ProductCenter::Qry::MatchesAllClauses ();

Returns TRUE if the query is configured to match all clauses or FALSE if it is not. This function only applies to query clause queries.

IsItemQuery

C++: BOOL pcQry::IsItemQuery ();

C: BOOL pcQryIsItemQuery (pcQry *qry);

Perl: $isItemQuery = ProductCenter::Qry::IsItemQuery ();

Returns TRUE if the query object is an item query or FALSE if it is not..

IsProcessQuery

C++: BOOL pcQry::IsProcessQuery ();

C: BOOL pcQryIsProcessQuery (pcQry *qry);

Perl: $isProcessQuery = ProductCenter::Qry::IsProcessQuery ();

Returns TRUE if the query object is a process query or FALSE if it is not..

IsActivityQuery

C++: BOOL pcQry::IsActivityQuery();

C: BOOL pcQryIsActivityQuery (pcQry *qry);

Perl: $isActivityQuery = ProductCenter::Qry::IsActivityQuery ();

Returns TRUE if the query object is an activity query or FALSE if it is not.

GetItemColLayout

C++: pcItemColLayout *pcQry::GetItemColLayout();

C: pcItemColLayout *pcQryGetItemColLayout(pcQry* query);

Perl: $layout = ProductCenter::Qry::GetItemColLayout();

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 function pcQry::pcQryLoadQry() 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.

SetItemColLayoutId

C++: MSG_CODE pcQry::SetItemColLayoutId(int layout_id);

C: MSG_CODE pcQrySetItemColLayoutId(pcQry* query, int layout_id);

Perl: $msg_code = ProductCenter::Qry::SetItemColLayoutId($layout_id);

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 pcQry::pcQrySaveQry() or the Replace function pcQry::pcQryReplace() 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 SaveQuery or Replace. 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.

Execute functions

The following functions execute queries and work with the items returned from the query.

Execute

C++: MSG_CODE pcQry::Execute (UINT32 threshold);

C: MSG_CODE pcQryExecute (pcQry *qry, UINT32 threshold);

Perl: $msg_code = ProductCenter::Qry::Execute ($threshold);

Run a constructed query.

The *threshold* argument specifies the number of results to return. Setting threshold to 0 causes all results to be returned. For example, if you enter threshold = 100 and 150 items match your query, only the first 100 items are returned.

Note that the threshold does not take permissions into account: it simply counts hits regardless of whether or not you can view them. Also, obsolete items are not returned.

ExecuteMatchCase

C++: MSG_CODE pcQry::ExecuteMatchCase (UINT32 threshold, BOOL is_casesensitive);

C: MSG_CODE pcQryExecuteMatchCase (pcQry *pcqry, UINT32 threshold, BOOL is_casesensitive);

Perl: $msg_code = ProductCenter::Qry::ExecuteMatchCase ($threshold, $issensitive);

This function is similar to pcQry::Execute except ExecuteMatchCase has a flag to determine if the query should be case-sensitive.

GetItemCount

C++: MSG_CODE pcQry::GetItemCount (UINT32 *item_count);

C: MSG_CODE pcQryGetItemCount (pcQry *qry, UINT32 *item_count);

Perl: $msg_code = ProductCenter::Qry::GetItemCount ($item_count);

Returns the number of items that match your query criteria after you call the *Execute* function. Note that these are the total found, not necessarily the number you can view based on your permissions.

GetItem

C++: pcItem *pcQry::GetItem (UINT32 index);

C: pcItem *pcQryGetItem (pcQry *qry, UINT32 index);

Perl: $Item = ProductCenter::Qry::GetItem ($index);

Returns a pointer to an item object from the query object, which you then can modify. You specify the index of the item you want.

Note that you should test whether you can view the item prior to trying to work with it.

"GetItem" keeps the item's ownership in the query object. If you perform a "GetItem" and then delete the query, the item is deleted.

Query storage functions

SaveQuery

C++: MSG_CODE *pcQry::SaveQuery (const char *name);

C: MSG_CODE *pcQrySaveQuery (pcQry *qry, const char *name);

Perl: $msg_code = ProductCenter::Qry::SaveQuery ($qry_name);

Assigns a name to the query and then saves the query to the database. Note that saved queries are specific to the user account that the program is logged in under.

LoadQuery

C++: pcQry (pcCnxn *cnxn, const char *name);

C: pcQry *pcQryLoadQuery (pcCnxn *cnxn, const char *name);

Perl: $qry = new ProductCenter::Qry ($cnxn, $name);

Loads a query that had been saved to the database with "SaveQuery". Note that loading a saved query does not automatically execute it.

Replace

C++: MSG_CODE pcQry::Replace ();

C: MSG_CODE pcQryReplace (pcQry *pcqry);

Perl: $msg_code = ProductCenter::Qry::Replace ();

Replaces an existing saved query with a new query. The new query can be similar to the old query or completely different. Use this function to replace a saved query with a new version.

DeleteQuery

C++: MSG_CODE pcQry::DeleteQuery ();

C: MSG_CODE pcQryDeleteQuery (pcQry *qry);

Perl: $msg_code = ProductCenter::Qry::DeleteQuery ();

Removes the query from the database.

Workflow query functions

GetWfAttrPromptCount

C++: MSG_CODE pcQry::GetWfAttrPromptCount (UINT32 *count);

C: MSG_CODE pcQryGetWfAttrPromptCount (pcQry *qry, UINT32 *count);

Perl: $msg_code = ProductCenter::Qry::GetWfAttrPromptCount ($qry, $count);

Returns the number of attributes (based on prompts) of the workflow query.

GetWfAttrPrompt

C++: const char *pcQry::GetWfAttrPrompt (UINT32 index);

C: const char *pcQryGetWfAttrPrompt (pcQry *qry, UINT32 index);

Perl: $promt= ProductCenter::Qry::GetWfAttrPrompt ($qry, $index);

Returns the attribute prompt name of the workflow query.

GetProcessInstCount

C++: MSG_CODE pcQry::GetProcessInstCount (UINT32 *count);

C: MSG_CODE pcQryGetProcessInstCount (pcQry *qry, UINT32 *count);

Perl: $msg_code = ProductCenter::Qry::GetProcessInstCount ($count);

Returns the number of processes returned

GetProcessInst

C++: pcProcessInst *pcQry::GetProcessInst (UINT32 index);

C: pcProcessInst *pcQryGetProcessInst (pcQry *qry, UINT32 index);

Perl: $procInst = ProductCenter::Qry::GetProcessInst ($index);

Returns the process instance based on the index.

GetActivityInstCount

C: MSG_CODE pcQry::GetActivityInstCount (UINT32 *count);

C: MSG_CODE pcQryGetActivityInstCount (pcQry *qry, UINT32 *count);

Perl: $msg_code = ProductCenter::Qry::GetActivityInstCount ($count);

Returns the number of activity instances returned.

GetActivityInst

C++: pcActivityInst *pcQry::GetActivityInst (UINT32 index);

C: pcActivityInst *pcQryGetActivityInst (pcQry *qry, UINT32 index);

Perl: $actInst = ProductCenter::Qry::GetActivityInst ($index);

Returns the activity instance based on the index.

Report object:

Constructors and Destructors

The Report object constructor behaves differently depending upon what arguments you pass to it (report id versus report name). It contains one *constructor* and one *destructor*.

Create report object by ID

C++: pcReport *pcReport::pcReport (pcCnxn *cnxn, Int id);

C: pcReport *pcReportLoadById (pcCnxn *cnxn, Int id);

Perl: $report = new ProductCenter::Report ($cnxn, $id);

Creates the base report object by the report id.

Create report object by name

C++: pcReport *pcReport::pcReport (pcCnxn *cnxn, const char *name);

C: pcReport *pcReportLoadByName (pcCnxn *cnxn, const char *name);

Perl: $report = new ProductCenter::Report ($cnxn, $name);

Creates the base report object by the report name.

Clone report

C++: pcReport *pcReport::pcReport (pcReport *report);

C: pcReport *pcReportCreateClone (pcReport *report);

Perl: $report = ProductCenter::Report::CreateClone ($report);

Creates a copy of the report memory.

Destroy report object

C++: void pcReport::~pcReport ();

C: void pcReportDestroy (pcReport *report);

Perl: ProductCenter::Report::DESTROY ($report);

Destroys the report object.

> **NOTE:** Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Report functions

DeleteExport

C++: MSG_CODE pcReport::DeleteExport ();

C: MSG_CODE pcReportDeleteExport (pcReport *report);

Perl: $msg_code = ProductCenter::Report::DeleteExport ();

Deletes the export.

GetQuery

C++: pcQry *pcReport::GetQuery ();

C: pcQry *pcReportGetQuery (pcReport *report);

Perl: $qry = ProductCenter::Report::GetQuery ();

Returns the query object associated with the report.

ExecuteWithItem

C++: MSG_CODE pcReport::ExecuteWithItem (pcItem *item);

C: MSG_CODE pcReportExecuteWithItem (pcReport *report, pcItem *item);

Perl: $msg_code = ProductCenter::Report::ExecuteWithItem ($report, $item);

Executes the report based on the specified input.

ExecuteWithQuery

C++: MSG_CODE pcReport::ExecuteWithQuery (pcQry *qry);

C: MSG_CODE pcReportExecuteWithQuery (pcReport *report, pcItem *item);

Perl: $msg_code = ProductCenter::Report::ExecuteWithQuery ($report, $qry);

Executes the report based on the specified query.

ExecuteWithActivity

C++: MSG_CODE pcReport::ExecuteWithActivity (pcActivityInst *act);

C: MSG_CODE pcReportExecuteWithActivity (pcReport *report,
pcActivityInst *act);

Perl: $msg_code = ProductCenter::Report::ExecuteWithActivity ($report, $act);

Executes the report based on the specified activity.

ExecuteWithProcessInst

C++: MSG_CODE pcReport::ExecuteWithProcessInst (pcProcessInst *process);

C: MSG_CODE pcReportExecuteWithProcessInst (pcReport *report,
pcProcessInst *process);

Perl: $msg_code = ProductCenter::Report::ExecuteWithProcessInst ($report,
$process);

This executes the report based on the specified process instance.

GetAttrCount

C++: MSG_CODE pcReport::GetAttrCount (UINT32 *count);

C: MSG_CODE pcReportGetAttrCount (pcReport *report, UINT32 *count);

Perl: $msg_code = ProductCenter::Report::GetAttrCount ($report, $count);

Return the number of attrs on the report object.

GetAttrNameByIndex

C++: const char *pcReport::GetAttrNameByIndex (UINT32 index);

C: const char *pcReportGetAttrNameByIndex (pcReport *report, UINT32 index);

Perl: $attName = ProductCenter::Report::GetAttrNameByIndex ($report, $index);

Returns the attribute name by the index.

GetAttr

C++: const char *pcReport::GetAttr (const char *name);

C: const char *pcReportGetAttr (pcReport *report, const char *name);

Perl: $attrValue = ProductCenter::Report::GetAttr ($report, $name);

Returns the attribute value for the attribute specified in the name. Possible names are listed in Table 9-2:

Table 9-2:

Customizable	Id	Name
Description	Local File Path	Parent Id
Export Location	Mime File Extension	Type
HTML Path	Mime Type	XSL Filename

These values are returned by GetAttrNameByIndex.

SetAttr

C++: MSG_CODE pcReport::SetAttr (const char *name, const char *value);

C: MSG_CODE pcReportSetAttr (pcReport *report, const char *name, const char *value);

Perl: $msg_code = ProductCenter::Report::SetAttr ($report, $name, $value);

Sets the attribute value for the attribute specified in the name. Possible names are "Export Location", "HTML Path", and “Local File Path”.

Chapter 10

Workflow

10

Just Ahead:

The ProductCenter Workflow option allows you to model the processes that your business uses to get things done, and manage them with ProductCenter. It does this by routing work requests (and the information needed to perform the work) to appropriate individuals and groups. ProductCenter Workflow keeps detailed records of each job step, including who is assigned to perform each activity (task), and the date and time that activities are started and completed.

Individual tasks are called *activities*. The series of activities that accomplishes a job is called a *process*. A designated workflow designer or administrator at your site develops process and activity definitions that model the way things are done at your company. A specific process or activity that is created from a definition is called an *instance* of that definition.

The Toolkit provides four objects reflecting the definitions and instances of processes and activities:

- pcProcessDef
- pcProcessInst
- pcActivityDef
- pcActivityInst

These objects are described in detail in Chapter 3, “Objects”. The connection and list objects also provide some workflow related functions.

See the *ProductCenter Workflow User Guide* for more information about the Workflow product.

Getting workflow information

This section lists the connection and list functions that allow you to obtain information about workflow processes and activities.

GetProcessDefList

C++: pcList *pcCnxn::GetProcessDefList (BOOL has_class);

C: pcList *pcCnxnGetProcessDefList (pcCnxn *cnxn, BOOL has_class);

Perl: $list = ProductCenter::Cnxn::GetProcessDefList ($has_class);

Returns the valid process definitions available for the user. This reflects the launch permissions assigned to each process definition. This function automatically returns the latest version of each definition.

If *has_class* is TRUE, the function returns a list of form-based workflow definitions.

If *has_class* is FALSE, the function returns a list of route-based workflow definitions.

GetProcessDef

C++: pcProcessDef *pcList::GetProcessDef (UINT32 index);

C: pcProcessDef *pcListGetProcessDef (pcList *list, UINT32 index);

Perl: $processDef= ProductCenter::List::GetProcessDef ($index);

Returns a process definition by index from a list.

GetWorkList

C++: pcList *pcCnxn::GetWorkList (BOOL is_claimed);

C: pcList *pcCnxnGetWorkList (pcCnxn *cnxn, BOOL is_claimed);

Perl: $list = ProductCenter::Cnxn::GetWorkList ($is_claimed);

Returns the contents of the Work List or Claimable list for the connected user.

If *is_claimed* is TRUE, the function returns a list of claimed activities. If it is FALSE, the function returns a list of unclaimed activities.

GetActivityInst

C++: pcActivityInst *pcList::GetActivityInst (UINT32 index);

C: pcActivityInst *pcListGetActivityInst (pcList *list, UINT32 index);

Perl: $ActivityInst = ProductCenter::List::GetActivityInst ($index);

Returns an activity instance by index from a list.

Process definition object

Constructors and Destructors

The process definition object has one *constructor* and one *destructor*.

ProcessDef

C++: pcProcessDef pcProcessDef (pcCnxn *cnxn, Int id);

C: pcProcessDef *pcProcessDefLoadById (pcCnxn *cnxn, Int id);

Perl: $processDef = ProductCenter::ProcessDef ($cnxn, $id);

The process definition constructor retrieves a new process definition object from the server by calling the ID of the process. This constructor requires a connection object.

ProcessDefDestroy

C++: ~pcProcessDef ();

C: void pcProcessDefDestroy (pcProcessDef *processdef);

Perl: ProductCenter::ProcessDef::DESTROY ();

The destructor is a public function. You can use the *~pcProcessDef()* destructor to disassemble the process definition object. This destructor does not destroy objects that are retrieved by "GetStartActivityDefByIndex" or "CreateInstance".

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Process definition functions

The following functions allow you to obtain information associated with a process definition.

GetAttr

C++: const char *pcProcessDef::GetAttr (const char *attrName);

C: const char *pcProcessDefGetAttr (pcProcessDef *processdef, const char *attrName);

Perl: $attrValue =ProductCenter::ProcessDef::GetAttr ($attrName);

Returns the attribute values assigned to the process definition.

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.
- "Last modified": 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.

GetPastVersion

C++: pcProcessDef *pcProcessDef::GetPastVersion (UINT32 ver_id);

C: pcProcessDef *pcProcessDefGetPastVersion (pcProcessDef *processdef, UINT32 ver_id);

Perl: $processDef = ProductCenter::ProcessDef::GetPastVersion ($ver_id);

Returns an earlier version of the process definition. You indicate the version you want by entering the appropriate version ID (*ver_id*).

Make sure the version ID is less than the value returned by "GetAttr" on the version of the process definition.

GetStartActivityDefCount

C++: MSG_CODE pcProcessDef::GetStartActivityDefCount (UINT32 *act_def_count);

C: MSG_CODE pcProcessDefGetStartActivityDefCount (pcProcessDef *processdef, UINT32 *act_def_count);

Perl: $msg_code = ProductCenter::ProcessDef::GetStartActivityDefCount ($act_def_count);

Returns the number of activities you can retrieve from *GetStartActivityDefByIndex()*.

If, for example, "GetStartActivityDefCount" returns a value of 6, then the highest index number you can pass to "GetStartActivityDefByIndex" is 5. (Remember that the first item in a list has an index number of 0.)

GetStartActivityDefByIndex

C++: pcActivityDef *pcProcessDef::GetStartActivityDefByIndex (UINT32 index);

C: pcActivityDef *pcProcessDefGetStartActivityDefByIndex(pcProcessDef *processdef, UINT32 index);

Perl: $ActivityDef = ProductCenter::ProcessDef::GetStartActivityDefByIndex ($index);

Returns the activity at the index specified.

GetNextActivities

C++: pcList *pcProcessDef::GetNextActivities ();

C: pcList *pcProcessDefGetNextActivities (pcProcessDef *def);

Perl: $list = ProductCenter::ProcessDef::GetNext1Activities ();

Returns a list of the starting activity instances of the Process Definition.

SetItem

C++: void pcProcessDef::SetItem (pcItem *item);

C: void pcProcessDefSetItem (pcProcessDef *def, pcItem *item);

Perl: ProductCenter::ProcessDef::SetItem ($item);

Uses *item* as the attached item of instance when it is created.

GenerateUniqueName

C++: const char *pcProcessDef::GenerateUniqueName ();

C: const char*pcProcessDefGenerateUniqueName (pcProcessDef *processdef);

Perl: $name = ProductCenter::ProcessDef::GenerateUniqueName ();

Generates a unique name for a form based process instance, based on the name of the process definition. You can have ProductCenter generate unique names by either calling this function or passing a NULL or empty string to “CreateInstance”.

CreateInstance

C++: pcProcessInst *pcProcessDef::CreateInstance (const char *instanceName, UINT32 flag, pcList *list);

C: pcProcessInst *pcProcessDefCreateInstance (pcProcessDef *processdef, const char *instanceName, UINT32 flag, pcList *list);

Perl: $processInst = ProductCenter::ProcessDef::CreateInstance ($instanceName, $flag, $list);

Creates a new process instance using instanceName as the name of the instance. The item connected to the process instance (either route-based or form-based) must be assigned to using the "SetItem" function. The flag argument can be zero or OP_WARN_UNFILLED. The list argument contains a list of activities to instantiate when the process instance is created. This list can be created using the "GetNextActivities" function. If the flag argument is set to OP_WARN_UNFILLED then all open assignments to each activity returned by "GetNextActivities" must be assigned. If the flag argument is not set to OP_WARN_UNFILLED, all unassigned assignments will be set to an on hold state. If there are any unfilled assignments in any activity in the list argument and the flag argument is set to OP_WARN_UNFILLED, then an error will be generated.

The user must have been granted launch permission by the process definition.

In general, these are the steps to creating a process instance:

1. Identify and load the process definition for the new process instance.

2. Identify and load the item for a route-based workflow, or create an item for a form-based workflow.

3. Use "SetItem" to assign the item to the definition loaded in step 1

4. Generate a list of initial activities in the process definition using "GetNextActivities".

5. Assign unfilled assignments for each activity in the list created in step 4.

6. Generate a unique name for the process instance using "GenerateUniqueName".

7. Call "CreateInstance" to create the process instance.

Alternatively, instead of steps 4 and 5 above, the flag argument can be set to 0 and the list argument can be null. This will cause the process instance to follow the default path for the process definition and if there are unassigned assignments in the initial activities, those assignments will be set to an on hold state.

Process instance object

Constructors and Destructors

The ProcessInst constructor provides three different functions, depending on what arguments you use.

ProcessInstLoadByID

C++: pcProcessInst pcProcessInst (pcCnxn *cnxn, Int id);

C: pcProcessInst *pcProcessInstLoadById (pcCnxn *cnxn, Int id);

Perl: $processInst = new ProductCenter::ProcessInst ($cnxn, $id);

Retrieves a new process instance object from the server by calling the ID of the process. This constructor requires a connection object.

If a user creates the instance during the current session, the process instance will contain a pending activity list. The pending activities need to be assigned and activated before destroying the process instance.

See ProcessInstLoadAttachedProcesses and ProcessInstLoadProcesses below for additional uses of the ProcessInst constructor called with different arguments.

NOTE: When creating a process instance, you must immediately make assignments to pending activities or you will not be able to access the list later.

ProcessInstLoadAttachedProcesses

C++: pcProcessInst pcProcessInst (pcCnxn *cnxn, pcItem *item);

C: pcProcessInst *pcProcessInstLoadAttachedProcesses (pcCnxn *cnxn, pcItem *item);

Perl: $processInst = new ProductCenter::ProcessInst ($cnxn, $item);

Loads the attached process of the item. Even though the returned object type is a process instance, the process list functions (GetProcessListCount on page 160 and ProcessListGetAttrByIndex on page 159)

ProcessInstLoadProcesses

C++: pcProcessInst pcProcessInst (pcCnxn *cnxn);

C: pcProcessInst *pcProcessInstLoadProcesses (pcCnxn *cnxn);

Perl: $processInst = new ProductCenter::ProcessInst ($cnxn);

Loads the list of all the workflow processes. Even though the returned object type is a process instance, the process list functions (GetProcessListCount on page 160 and ProcessListGetAttrByIndex on page 159)

ProcessInstDestroy

C++: ~pcProcessInst ();

C: void pcProcessInstDestroy (pcProcessInst *processinst);

Perl: ProductCenter::ProcessInst::DESTROY ();

The destructor is a public function. You can use the *~pcProcessDef()* destructor to disassemble the process definition object.

You must destroy each activity instance or process definition object retrieved from the process instance.

NOTE: Perl programmers should read “Destructors and Perl” on page 43 for information as to why they should not use this call.

Process instance functions

The functions in this section allow you to extract the attributes of a process instance, make assignments, and obtain related information.

GetDef

C++: pcProcessDef *pcProcessInst::GetDef ();

C: pcProcessDef *pcProcessInstGetDef (pcProcessInst *processinst);

Perl: $processDef = ProductCenter::ProcessInst::GetDef ();

Returns the definition that was the basis for the process instance *processinst.*

At some point in your program, be sure to free the pcProcessDef object that “GetDef” returns.

GetAttr

C++: const char *pcProcessInst::GetAttr (const char *attrName);

C: const char *pcProcessInstGetAttr (pcProcessInst *processinst, const char *attrName);

Perl: $attrValue = ProductCenter::ProcessInst::GetAttr ($attrName);

Returns the attributes associated with the process instance. “GetAttr” returns the attributes as a character string. Subsequent calls to “GetAttr” may overwrite the static buffer that the function uses. Note this call actually returns Prepared On, Last Modified, and Prepared By attributres from the item that is created for the process instance and for the item the workflow is routed through, rather than from process instance itself, since the database does not store these attributes with the process instance. This works fine for form-based processes, since an attribute like the Prepared On date for the item would always be identical to the Prepared On date for the process instance. However, route-based processes can return Prepared On values for the routed item that differ from what you would expect to see for the process instance. For route-based processes, the initiator can be determined from the workflow audit log.

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.
- "Last modified": 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.

GetItem

C++: pcItem *pcProcessInst::GetItem ();

C: pcItem *pcProcessInstGetItem (pcProcessInst *processinst);

Perl: $Item = ProductCenter::ProcessInst::GetItem ();

Returns the item that is being routed in the process.

GetCompletedActivityCount

C++: MSG_CODE pcProcessInst::GetCompletedActivityCount (UINT32 *pend_act_count);

C: MSG_CODE pcProcessInstGetCompletedActivityCount (pcProcessInst *pcprocessinst, UINT32 *pend_act_count);

Perl: $msg_code = ProductCenter::ProcessInst::GetCompletedActivityCount ($cplt_act_count);

Returns the number of completed activities of a specific process instance.

GetCompletedActivityByIndex

C++: pcActivityInst *pcProcessInst::GetCompletedActivityByIndex (UINT32 index));

C: pcActivityInst *pcProcessInstGetCompletedActivityByIndex (pcProcessInst *pcprocessinst, UINT32 index);

Perl: $processInst = ProductCenter::ProcessInst::GetCompletedActivityByIndex ($index);

Returns the activity instance object from the completed activity list.

GetAuditLogListCount

C++: MSG_CODE pcProcessInst::GetAuditLogListCount (UINT32 *audit_log_count);

C: MSG_CODE pcProcessInstGetAudiLogListCount(pcProcessInst *pcprocessinst, UINT32 *audit_log_count);

Perl: $msg_code = ProductCenter::ProcessInst::GetAuditLogListCount ($count);

Returns the number of entries in the audit log.

AuditLogGetAttrByIndex

C++: const char *pcProcessInst::AuditLogGetAttrByIndex (const char (*attrName, UINT32 index);

C: const char *pcProcessInstAuditLogGetAttrByIndex (pcProcessInst (*pcprocessinst, const char *attrName, UINT32 index);

Perl: $attrValue = ProductCenter::ProcessInst::AuditLogGetAttrByIndex ($attrName, $index);

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.

UpdateAttachedProcessList

C++: void pcProcessInst::UpdateAttachedProcessList (pcItem *item);

C: void pcProcessInstUpdateAttachedProcessList (pcProcessInst *pcprocessinst, pcItem *item);

Perl: ProductCenter::ProcessInst::UpdateAttachedProcessList ($item);

Repopulates the list of attached processes during the life of a ProcessInst object. To repopulate the list with attached processes of specific states, see UpdateProcessListWith States.

The attached processes and process lists are two similar yet separate things. Attached processes do not have states associated with them, while a process list has states and is independent of the user/item. List processes will list all the processes on a system associated with one or more states.

UpdateProcessListWithStates

C++: MSG_CODE pcProcessInst::UpdateProcessListWithStates (const char *states);

C: MSG_CODE pcProcessInstUpdateProcessListWithStates (pcProcessInst *pcprocessinst, const char *states);

Perl: $msg_code = ProductCenter::ProcessInst::UpdateProcessListWithStates ($states);

Repopulates the list of processes with processes of specific states during the life of a ProcessInst object.

You specify the states as a string. The following characters are supported and can be combined:

- I — Initiated
- A — Active
- C — Cancelled
- O — cOmpleted

The string "AO" populates the list with active and completed processes.

GetProcessListCount

C++: MSG_CODE pcProcessInst::GetProcessListCount (UINT32 *processListCount);

C: MSG_CODE pcProcessInstGetProcessListCount (pcProcessInst *pcprocessinst, UINT32 *processListCount);

Perl: $msg_code = ProductCenter::ProcessInst::GetProcessListCount ($processListCount);

Returns the number of attributes to process for the given process.

ProcessListGetAttrByIndex

C++: const char *pcProcessInst::ProcessListGetAttrByIndex (const char *attrName, UINT32 index);

C: const char *pcProcessInstProcessListGetAttrByIndex (pcProcessInst *pcprocessinst, const char *attrName, UINT32 index);

Perl: $attrValue = ProductCenter::ProcessInst::ProcessListGetAttrByIndex ($attrName, $index);

Returns an attribute's value from a process instance. You must first use "GetProcessListCount".

The function requires a string specifying the attribute name that you want, and an index into the process list. The attribute names are:

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

GetTaskListCount

C++: MSG_CODE pcProcessInst::GetTaskListCount (UINT32 *taskListCount);

C: MSG_CODE pcProcessInstGetTaskListCount (pcProcessInst *pcprocessinst, UINT32 *taskListCount);

Perl: $msg_code = ProductCenter::ProcessInst::GetTaskListCount ($taskListCount));

Return the number of tasks.

TaskListGetAttrByIndex

C++: C++: const char *pcProcessInst::TaskListGetAttrByIndex (const char *attrName, UINT32 index);

C: C: const char *pcProcessInstTaskListGetAttrByIndex (pcProcessInst *pcprocessinst, const char *attrName, UINT32 index);

Perl: $attrValue = ProductCenter::ProcessInst::TaskListGetAttrByIndex ($attrName, $index);

Returns an attribute’s value from an activity instance. This function requires a string specifying the attribute name that you want, and an index into the task list. The attributes names are:

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

Activity definition object

Constructors and Destructors

ActivityDefDestroy

C++: ~pcActivityDef ();

C: void pcActivityDefDestroy (pcActivityDef *activitydef);

Perl: ProductCenter::ActivityDef::DESTROY ();

The activity definition object, like all C++ objects, has one *destructor*. The destructor is invoked automatically when the object is destroyed, and all memory used by the activity definition object is freed.

There are no public constructors in this object.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Activity definition object functions

The following two functions allow you to extract attributes of an activity definition and locate the parent process definition.

GetAttr

C++: const char *pcActivityDef::GetAttr (const char *attr_name);

C: const char *pcActivityDefGetAttr (pcActivityDef *activitydef, const char *attr_name);

Perl: $attrValue = ProductCenter::ActivityDef::GetAttr ($attr_name);

Returns the attributes assigned to the activity definition."GetAttr" returns the attributes as a character string. Subsequent calls to "GetAttr" may overwrite the static buffer that the function uses.

The acceptable attribute types are:

Name—The name of the activity definition.

Work notes—A character string, up to 512 bytes long, that contains a description of the activity definition.

Duration—The expected amount of time in days that all of the instances based on this activity definition will take to complete.

CMS ID—The ID of the activity definition.

GetProcessDef

C++: pcProcessDef *pcActivityDef::GetProcessDef ();

C: pcProcessDef *pcActivityDefGetProcessDef (pcActivityDef *activitydef);

Perl: $processDef = ProductCenter::ActivityDef::GetProcessDef ();

Returns the activity definition's parent process definition. Remember to free this object at some point in your program.

GetNextActivityCount

C++: MSG_CODE pcActivityDef::GetNextActivityCount (UINT32 *next_act_count);

C: MSG_CODE pcActivityDefGetNextActivityCount (pcActivityDef *activitydef, UINT32 *next_act_count);

Perl: $msg_code = ProductCenter::ActivityDef::GetNextActivityCount ($next_act_count);

Returns the number of activities you can retrieve with *GetNextActivityDefByIndex()*.

GetNextActivityDefByIndex

C++: pcActivityDef *pcActivityDef::GetNextActivityDefByIndex (UINT32 index);

C: pcActivityDef *pcActivityDefGetNextActivityDefByIndex (pcActivityDef *activitydef, UINT32 index);

Perl: $activityDef = ProductCenter::ActivityDef::GetNextActivityDefByIndex ($index);

Returns one of the simultaneously occurring activities that follow the activity you specify. The client must eventually free the activity definition.

GetPrevActivityCount

C++: MSG_CODE pcActivityDef::GetPrevActivityCount (UINT32 *prev_act_count);

C: MSG_CODE pcActivityDefGetPrevActivityCount (pcActivityDef *activitydef, UINT32 *prev_act_count);

Perl: $msg_code = ProductCenter::ActivityDef::GetPrevActivityCount ($prev_act_count);

Returns the number of activities you can retrieve with *GetPrevActivityDefByIndex()*.

GetPrevActivityDefByIndex

C++: pcActivityDef *pcActivityDef::GetPrevActivityDefByIndex (UINT32 index);

C: pcActivityDef *pcActivityDefGetPrevActivityDefByIndex (pcActivityDef *activitydef, UINT32 index);

Perl: $activityDef = ProductCenter::ActivityDef::GetPrevActivityDefByIndex ($index);

Returns the activity definition that precedes this activity in the process definition's structure. The client must eventually free the activity definition.

Activity instance object

Constructors and Destructors

ActivityInstLoadById

C++: pcActivityInst pcActivityInst (pcCnxn *cnxn, const char *id);

C: pcActivityInst *pcActivityInstLoadById (pcCnxn *cnxn, const char *id);

Perl: $activityInst = new ProductCenter::ActivityInst ($cnxn, $id);

Creates a new activity instance object from the server by calling the ID of the process. This constructor requires a connection object.

Note that you must specify a pipe (|) at the end of the cmsid. The Toolkit implements the pipe because arguments to functions often take the form cmsid|user.

ActivityInstDestroy

C++: ~pcActivityInst ();

C: void pcActivityInstDestroy (pcActivityInst *activityinst);

Perl: ProductCenter::ActivityInst::DESTROY ();

You can use the *~pcActivityInst()* destructor to destroy the process definition object.

If you approve an instance but have not yet activated its pending activities, or if you disapprove an instance but have not reactivated one of its pending activities, the connection object maintains the pending activity list. You can retrieve the list later on by creating a new instance with the pcActivityInst public constructor.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Activity instance functions

The following functions allow you to manipulate activity instances.

IsClaimable

C++: BOOL pcActivityInst::IsClaimable ();

C: BOOL pcActivityInstIsClaimable (pcActivityInst *activityinst);

Perl: $isClaimable= ProductCenter::ActivityInst::IsClaimable ();

Returns TRUE if the current user can claim the activity instance and FALSE if not.

GetAttr

C++: const char *pcActivityInst::GetAttr (const char *attrName);

C: const char *pcActivityInstGetAttr (pcActivityInst *activityinst, const char *attrName);

Perl: $attrValue = ProductCenter::ActivityInst::GetAttr ($attr_name);

Returns the attributes assigned to the activity instance. “GetAttr” returns the attributes as a character string. Subsequent calls to “GetAttr” may overwrite the static buffer that the function uses.

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.
- "Last modified": 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 the Toolkit 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.

GetItem

C++: pcItem *pcActivityInst::GetItem ();

C: pcItem *pcActivityInstGetItem (pcActivityInst *activityinst);

Perl: $item = ProductCenter::ActivityInst::GetItem ();

Returns the item object associated with the specified activity instance object.

AssignmentsAreOpen

C++: BOOL pcActivityInst::AssignmentsAreOpen ();

C: BOOL pcActivityInstAssignmentsAreOpen (pcActivityInst *activityinst);

Perl: $isAssignOpen = ProductCenter::ActivityInst::AssignmentsAreOpen ();

Returns TRUE if there are open assignments for the given activity instance.

GetAssignmentCount

C++: MSG_CODE pcActivityInst::GetAssignmentCount (BOOL is_open, UINT32 *assignment_count);

C: MSG_CODE pcActivityInstGetAssignmentCount (pcActivityInst *activityinst, BOOL is_open, UINT32 *assignment_count);

Perl: $msg_code = ProductCenter::ActivityInst::GetAssignmentCount ($is_open, $assign_count);

If *is_open* is TRUE, this function returns the number of assignments that must assigned for the activity. If *is_open* is FALSE, this function returns the number of assignments already filled. The sum of these values gives you the total assignments for the activity

GetAssignmentStatus

C++: const char* pcActivityInst::GetAssignmentStatus(BOOL is_open, UINT32 index);

C: const char* pcActivityInstGetAssignmentStatus (pcActivityInst *pcactivityinst, BOOL is_open, UINT32 index);

Perl: $align = ProductCenter::ActivityInst:: GetAssignmentStatus ($is_open, $index);

Returns the status of an assignment.

This function will return "None" if it fails. When it returns "None" you should call the connection object's "GetStatus" function to confirm that there was an error and to find out the nature of the error.

GetAssignmentType

C++: MSG_CODE pcActivityInst::GetAssignmentType (BOOL is_open, UINT32 index, pcAssignmentType *assignment_type);

C: MSG_CODE pcActivityInstGetAssignmentType (pcActivityInst *activityinst, BOOL is_open, UINT32 index, pcAssignmentType *assignment_type);

Perl: $msg_code = ProductCenter::ActivityInst::GetAssignmentType ($is_open, $index, $assignment_type);

Returns:

- **assnType_user** if the assignment you specify is of type user
- **assnType_group** if it is of type group
- **assnType_none** if no assignment has been made (error condition)

GetAssignmentActionType

C++: MSG_CODE pcActivityInst::GetAssignmentActionType (BOOL is_open, UINT32 index, pcAssignmentActionType *assignment_type);

C: MSG_CODE pcActivityInstGetAssignmentActionType (pcActivityInst act, BOOL is_open, UINT32 index, pcAssignmentActionType *assignment_type);

Perl: $msg_code = ProductCenter::ActivityInst::GetAssignmentActionType ($is_open, $index, $assignment_type);

Retrieves the type of the assignment indicated by is open and the index. This function is meant to be used with pcActivityInst::GetAssignmentCount.

GetAssignment

C++: const char *pcActivityInst::GetAssignment (BOOL is_open, UINT32 index);

C: const char *pcActivityInstGetAssignment (pcActivityInst *activityinst, BOOL is_open, UINT32 index);

Perl: $assignment = ProductCenter::ActivityInst::GetAssignment ($is_open, $index);

Returns the user, group, or role that has assigned to the activity instance. “GetAssignment” returns the role name if no one has yet assigned a user or group.

SetAssignment

C++: MSG_CODE pcActivityInst::SetAssignment (BOOL is_open, UINT32 index, const char *assignment);

C: MSG_CODE pcActivityInstSetAssignment (pcActivity *activityinst, BOOL is_open, UINT32 index, const char *assignment);

Perl: $msg_code = ProductCenter::ActivityInst::SetAssignment ($is_open, $index, $assignment);

Assigns a user or group to an activity instance. The name must be the user name, not the login name (for example, use "CMS DBA", not "cms"). In the case of open assignments, when is_open is set to TRUE, you must set the assignments in descending order.

For example, if you have 3 open assignments, you must first set 2, then 1, then 0.

GetProcessInst

C++: pcProcessInst * pcActivityInst::GetProcessInst ();

C: pcProcessInst *pcActivityInstGetProcessInst (pcActivityInst *activityinst);

Perl: $processInst =ProductCenter::ActivityInst::GetProcessInst ();

Returns the process instance to which the activity instance belongs.

Claim

C++: MSG_CODE pcActivityInst::Claim ();

C: MSG_CODE pcActivityInstClaim (pcActivityInst *activityinst);

Perl: $msg_code = ProductCenter::ActivityInst::Claim ();

When a group is assigned to an activity, ProductCenter places that activity in the Claimable windows of the users in that group. A user can then claim the activity, place it in his worklist, and begin his work. The "Claim" function takes the specified activity from the user's Claimable work list and puts it in the user's Claimed work list.

SendForward

C++: MSG_CODE pcActivityInst::SendForward (const char *comment, UINT32 flag, pcList *list);

C: MSG_CODE pcActivityInstSendForward (pcActivityInst *act, const char *comment, UINT32 flag, pcList *list);

Perl: $msg_code = ProductCenter::ActivityInst::SendForward ($comment, $flag, $list);

Sends the activity forward. The available flags are OP_WARN_UNFILLED and OP_IGNORE_THRESHOLD. The OP_WARN_UNFILLED will error out if there are any unfilled assignments. While the OP_IGNORE_THRESHOLD will ignore the threshold requirements and send the activity forward, this feature requires coordinator or administrator privileges. The list argument is the list of activities with their role assignments. This list can be empty or null indicating use as is. The function "ListCreateByActivityInstance" can be used to create the list argument. If a next activity still has unfilled role assignments and the OP_WARN_UNFILLED flag is not set, the unfilled assignments will be moved into a held state. This will require the unfilled assignments to be reassigned. The comment will be used as the electronic signature.

SendBack

C++: MSG_CODE pcActivityInst::SendBack (const char *comment, UINT32 flag, pcList *list);

C: MSG_CODE pcActivityInstSendBack (pcActivityInst *act, const char *comment, UINT32 flag, pcList *list);

Perl: $msg_code = ProductCenter::ActivityInst::SendBack ($comment, $flag, $list);

Sends the activity back.

The available flags are:

OP_SENDBACK

OP_SENDBACK_PREVIOUS

OP_IGNORE_THRESHOLD.

If OP_SENDBACK is set, then the list specifies a single activity that will be activated. This activity should be chosen from those in the list returned by GetNextActivities(OP_SENDBACK). The list argument can also be generated with the "ListCreateByActivityInstance" function.

If OP_SENDBACK_PREVIOUS is set, then all the predecessor activities of the current activity will be activated and the list will be ignored. Only one of the OP_SENDBACK and OP_SENDBACK_PREVIOUS flags may be set; if neither is set, then the program will act as if OP_SENDBACK had been set. The OP_IGNORE_THRESHOLD will ignore the threshold requirements and push the activity back; this feature requires coordinator or administrator privileges. The comment will be used as the electronic signature.

Reassign

C++: MSG_CODE pcActivityInst::Reassign (const char *to_user, const char *from_user, UINT32 flag, pcList *list);

C: MSG_CODE pcActivityInstReassign (pcActivityInst *act, const char *to_user , const char *from_user, UINT32 flag, pcList *list);

Perl: $msg_code = ProductCenter::ActivityInst::Reassign ($to_user, $from_user, $flag, $list);

Reassign the activity assignment to a new user. The to_user is the user name of the ProductCenter user to which the activity will be reassigned. The from_user is the user previously assigned. The assignment is the index to the assignment; this may be left as -1 to indicate to find the current user's assignment.

If the new user is a bypass user, this may act much like a Send Forward operation. In this case, it uses the OP_WARN_UNFILLED flag and list the same as Send Forward. Under normal operations, the flag and list are ignored. The list argument can be populated using the "ListCreateByActivityInstance" function.

Suspend

C++: MSG_CODE pcActivityInst::Suspend (const char *comment);

C: MSG_CODE pcActivityInstSuspend (pcActivityInst *act, const char *comment)

Perl: $msg_code = ProductCenter::ActivityInst::Suspend ($comment);

Places the activity instance in the suspended state. The user needs to be the coordinator or DBA-enabled. The comment will be used as the electronic signature. Entering the suspended state will disallow the Setting the Assignment, Place On Hold, Turn Off Hold, Send Forward, and Send Back actions. The comment will be used as the electronic signature.

Resume

C++: MSG_CODE pcActivityInst::Resume (const char *comment, UINT32 flag, pcList *list);

C: MSG_CODE pcActivityInstResume (pcActivityInst *act, const char *comment, UINT32 flag, pcList *list);

Perl: $msg_code = ProductCenter::ActivityInst::Resume ($comment, $flag, $list);

Resumes a suspended activity. If the activity was reassigned to the bypass user while it was suspended, when the activity is resumed the operation may behave similar to a Send Forward operation. In this case, it uses the OP_WARN_UNFILLED flag and list the same as Send Forward. Under normal operations, the flag and list arguments are ignored. The list argument can be populated using the "ListCreateByActivityInstance" function. The comment will be used as the electronic signature.

PlaceOnHold

C++: MSG_CODE pcActivityInst::PlaceOnHold (const char *user_name, const char *comment);

C: MSG_CODE pcActivityInstPlaceOnHold (pcActivityInst *act, const char *user_name, const char *comment);

Perl: $msg_code = ProductCenter::ActivityInst::PlaceOnHold ($user, $comment);

Places the Activity Instance's assignment to user_name On Hold. If the user_name isn't self (or an empty string), the logged in user needs to be the coordinator or DBA-enabled. The comment will be used as the electronic signature.

TurnOffHold

C++: MSG_CODE pcActivityInst::TurnOffHold (const char *user_name, const char *comment);

C: MSG_CODE pcActivityInstTurnOffHold (pcActivityInst *act, const char *user_name, const char *comment);

Perl: $msg_code = ProductCenter::ActivityInst::TurnOffHold ($user, $comment);

Turns off the hold on the assignment of user_name to the Activity Instance. If the user_name isn't self (or an empty string), the logged in user needs to be the coordinator or DBA-enabled. The comment will be used as the electronic signature.

GetNextActivities

C++: pcList *pcActivityInst::GetNextActivities (UINT32 flag);

C: pcList *pcActivityInstGetNextActivities (pcActivityInst *act, UINT32 flag);

Perl: $list= ProductCenter::ActivityInst::GetNextActivities ($flag);

Returns the list of the activity instances that might or will be activated. This is used before the interfaces SendForward, SendBack, Reassign, and Resume. The flag can be 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.

Chapter 11

Event Monitor (AQM)

11

Just Ahead:

The ProductCenter Toolkit Event Monitor is a programmatic library that lets client programs listen for events and trigger actions. All communications between client programs and ProductCenter take place through the Application Queue Manager (AQM) server program.

The AQM consists of several components listed below. Refer to Figure 2-20.

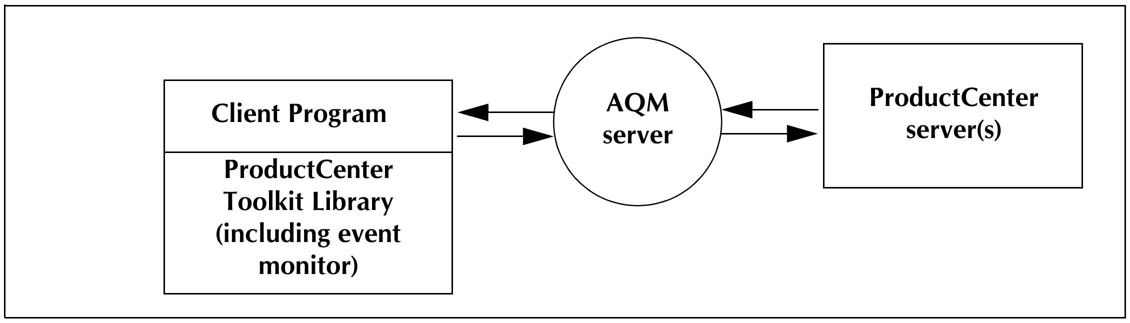


Figure 2-20: Basic AQM server architecture

AQM server — The AQM server receives requests from client processes to be notified of particular events. Any number of ProductCenter servers can broadcast events to the AQM. When the AQM receives a broadcast, it checks with the clients that want to receive the event. It then adds the event to a queue for each client that monitors the event.

AQM client — The client is any program that registers with the AQM server to receive events.

Toolkit (API) Event Monitor — This facility allows users to write AQM clients.

Monitor object

Constructors and destructors

MonitorConstruct

C++: pcMonitor (pcCnxn *cnxn);

C: pcMonitor *pcMonitorCreateWithCnxn (pcCnxn *cnxn);

Perl: $monitor = new ProductCenter::Monitor ($cnxn);

Creates a new instance of a monitor object. The function determines the default AQM server machine and the port number from the connection object, and it uses the error and status functionality of the connection object.

MonitorDestroy

C++: ~pcMonitor ();

C: void pcMonitorDestroy (pcMonitor *mon);

Perl: ProductCenter:: Monitor::DESTROY ();

Use the *~pcMonitor()* destructor to destroy the monitor object.

When you call this destructor, it frees up any allocated memory and, if necessary, disconnects from the AQM server.

NOTE: Perl programmers should read "Destructors and Perl" for information as to why they should not use this call.

Monitor functions

The following functions allow you to monitor events.

AddEventType

C++: MSG_CODE pcMonitor::AddEventType (pcEventType eventType);

C: MSG_CODE pcMonitorAddEventType (pcMonitor *mon, pcEventType eventType);

Perl: $msg_code = ProductCenter::Monitor::AddEventType ($eventType);

Adds an event that the client wants to monitor. The event can be any one of the following:

```
eventType_none                 eventType_all
eventType_allFileEvents        eventType_allProjectEvents
eventType_fileAdd              eventType_fileView
eventType_fileCheckin          eventType_filePurge
eventType_fileCheckout         eventType_fileDelete
eventType_fileRollback         eventType_fileUndoCheckout
eventType_fileGetReadOnly      eventType_fileAlter
eventType_fileApprove          eventType_fileDisapprove
eventType_fileSubmit           eventType_fileMove
eventType_projectAdd           eventType_projectCheckin
eventType_projectPurge         eventType_projectCheckout
eventType_projectDelete        eventType_projectRollback
eventType_projectUndoCheckout  eventType_projectAlter
eventType_projectMove          eventType_projectApprove
eventType_projectDisapprove    eventType_projectSubmit
eventType_projectObsolete      eventType_fileObsolete
eventType_projectReinstate     eventType_fileReinstate
```

RemoveEventType

C++: MSG_CODE pcMonitor::RemoveEventType (pcEventType event);

C: MSG_CODE pcMonitorRemoveEventType (pcMonitor *mon, pcEventType event);

Perl: $msg_code = ProductCenter::Monitor::RemoveEventType ($event);

Removes events from the list of monitored events. It may remove one event, all events, only file events, or only project events.

GetEventTypeCount

C++: MSG_CODE pcMonitor::GetEventTypeCount (UINT32 *eventTypeCount);

C: MSG_CODE pcMonitorGetEventTypeCount (pcMonitor *mon, UINT32 *eventTypeCount);

Perl: $msg_code = ProductCenter::Monitor::GetEventTypeCount ($eventTypeCount);

Returns the number of events being monitored.

GetEventTypeByIndex

C++: pcEventType pcMonitor::GetEventTypeByIndex (UINT32 index);

C: pcEventType pcMonitorGetEventTypeByIndex (pcMonitor *mon, UINT32 index);

Perl: $EventType = ProductCenter::Monitor::GetEventTypeByIndex ($index);

Returns the event type at the specified index in the list. Remember that the first event type in a list has an index of 0, so, for example, if you want the fifth event type, you would enter an index number of 4.

GetEventTypeName

C++: const char *pcMonitor::GetEventTypeName (pcEventType eventType);

C: const char *pcMonitorGetEventTypeName (pcMonitor *mon, pcEventType eventType);

Perl: $name = ProductCenter::Monitor::GetEventTypeName ($eventType);

Returns the name of an event type.

SetWaitTime

C++: MSG_CODE pcMonitor::SetWaitTime (int time);

C: MSG_CODE pcMonitorSetWaitTime (pcMonitor *mon, int time);

Perl: $msg_code = ProductCenter::Monitor::SetWaitTime ($time);

Sets the number of seconds that the monitor will wait for a new event from the AQM server. If you set the time to 0, the monitor will wait indefinitely. If you set it to less than zero, the monitor will poll for an already available event. You should always specify SetWaitTime, even if you only set it to 0.

GetWaitTime

C++: int pcMonitor::GetWaitTime ();

C: int pcMonitorGetWaitTime (pcMonitor *mon);

Perl: $waitTime = ProductCenter::Monitor::GetWaitTime ();

Returns the number of seconds that the monitor will wait for a new event from the AQM server.

SetServerHost

C++: MSG_CODE pcMonitor::SetServerHost (const char *hostName);

C: MSG_CODE pcMonitorSetServerHost (pcMonitor *mon, const char *hostName);

Perl: $msg_code = ProductCenter::Monitor::SetServerHost ($hostName);

Sets the server machine for the AQM server to which the monitor will connect. If the monitor is already connected, the new setting does not take effect until a new “Connect” call is made.

GetServerHost

C++: const char *pcMonitor::GetServerHost ();

C: const char *pcMonitorGetServerHost (pcMonitor *mon);

Perl: $hostName = ProductCenter::Monitor::GetServerHost ();

Returns the server machine for the AQM server to which the monitor will connect.

SetClientName

C++: MSG_CODE pcMonitor::SetClientName (const char *name);

C: MSG_CODE pcMonitorSetClientName (pcMonitor *mon, const char *name);

Perl: $msg_code = ProductCenter::Monitor::SetClientName ($name);

Assigns a name to the Toolkit monitoring client program. This name allows you to differentiate this program from other client programs.

GetClientName

C++: const char *pcMonitor::GetClientName ();

C: const char *pcMonitorGetClientName (pcMonitor *mon);

Perl: $clientName = ProductCenter::Monitor::GetClientName ();

Returns the client name assigned by “SetClientName”.

SetPortNumber

C++: MSG_CODE pcMonitor::SetPortNumber (int port);

C: MSG_CODE pcMonitorSetPortNumber (pcMonitor *mon, int port);

Perl: $msg_code = ProductCenter::Monitor::SetPortNumber ($port);

Sets the port number of the AQM server to which the monitor will connect. If the monitor is already connected, the new setting does not take effect until a new “Connect” call is made.

GetPortNumber

C++: int pcMonitor::GetPortNumber ();

C: int pcMonitorGetPortNumber (pcMonitor *mon);

Perl: $portNumber = ProductCenter::Monitor::GetPortNumber ();

Returns the port number of the AQM server to which the monitor will connect.

Connect

C++: MSG_CODE pcMonitor::Connect ();

C: MSG_CODE pcMonitorConnect (pcMonitor *mon);

Perl: $msg_code = ProductCenter::Monitor::Connect ();

Connects to the AQM server. You *must* call this function to obtain events.

Once you have called *Connect()*, any changes to the monitored event list or to the sleep time are automatically sent to the AQM server.

Disconnect

C++: MSG_CODE pcMonitor::Disconnect ();

C: MSG_CODE pcMonitorDisconnect (pcMonitor *mon);

Perl: $msg_code = ProductCenter::Monitor::Disconnect ();

Disconnects from the AQM server.

IsConnected

C++: BOOL pcMonitor::IsConnected ();

C: BOOL pcMonitorIsConnected (pcMonitor *mon);

Perl: $isConnect = ProductCenter::Monitor::IsConnected ();

Returns TRUE (non-zero) if you are connected to an AQM server, FALSE (zero) if not.

GetNextEvent

C++: pcEvent *pcMonitor::GetNextEvent ();

C: pcEvent *pcMonitorGetNextEvent (pcMonitor *mon);

Perl: $Event = ProductCenter::Monitor::GetNextEvent ();

Returns the new event from the server. If no event is ready and the monitor is nonblocking (wait time is not zero), or if there is a timeout, the function returns appropriate message codes.

Note that you should specify a value for "SetWaitTime", otherwise GetNextEvent will not be blocking.

If you get an event when you call GetNextEvent(), make sure you eventually destroy it.

Event object:

Constructors and Destructors

There are no public constructors in the event object.

EventDestroy

C++: ~pcEvent ();

C: void pcEventDestroy (pcEvent *event);

Perl: ProductCenter::Event::DESTROY ();

Destroys the event object and frees allocated memory.

NOTE: Perl programmers should read "Destructors and Perl" for information as to why they should not use this call.

Event functions

GetEventType

C++: MSG_CODE pcEvent::GetEventType (pcEventType *eventType);

C: MSG_CODE pcEventGetEventType (pcEvent *event, pcEventType *eventType);

Perl: $msg_code = ProductCenter::Event::GetEventType ($eventType);

Returns the event type.

GetUser

C++: const char *pcEvent::GetUser ();

C: const char *pcEventGetUser (pcEvent *event);

Perl: $userName = ProductCenter::Event::GetUser ();

Returns the name of the user who performed the event.

GetItem

C++: pcItem *pcEvent::GetItem ();

C: pcItem *pcEventGetItem (pcEvent *event);

Perl: $Item = ProductCenter::Event::GetItem ();

Returns the item on which the event occurred. “GetItem” returns NULL if the event was not performed on an item.

If you get an item from an event when you call “GetItem” from the C/C++ Toolkit, you must destroy the item when done with it. This does not apply to the Perl Toolkit.

Chapter 12

Forms

12

Just Ahead:

It is occasionally necessary to get information about the form related to an Item or Link. This chapter describes the functions related to querying an item or link form.

Form object:

Constructors and Destructors

The form object contains two *constructors* and one *destructor*.

FormLoadbyName

C++: pcForm *pcForm::pcForm (pcCnxn *cnxn, char *name);

C: pcForm *pcFormLoadByName (pcCnxn *cnxn, char *name);

Perl: $form = ProductCenter::Form ($cnxn, $name);

Loads the form definition using the form's description.

FormLoadbyId

C++: pcForm *pcForm::pcForm (pcCnxn *cnxn, UINT32 id);

C: pcForm *pcFormLoadById (pcCnxn *cnxn, UINT32 id);

Perl: $form = ProductCenter::Form ($cnxn, $id);

Loads the form definition using the form's id.

FormDestroy

C++: ~pcForm ();

C: void pcFormDestroy (pcForm *form);

Perl: ProductCenter::Form::DELETE ();

Destroys the memory of the form.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Getting form information

Use these functions to get information about a form.

GetAttr

C++: const char *pcForm::GetAttr (const char *name);

C: const char *pcFormGetAttr (pcForm *form, const char *name);

Perl: $attrValue = ProductCenter::Form::GetAttr ($name);

Returns the information about the form. The possible values for name are: “Form Id”, “Form Name”, “Table Name”, and “Is Searchable”.

GetFieldCount

C++: UINT32 pcForm::GetFieldCount (char *type);

C: UINT32 pcFormGetFieldCount (pcForm *form, char *type);

Perl: $count = ProductCenter::Form::GetFieldCount ($type);

Returns the count of the Field object. The type can be ‘CUSTOM’, ‘COMMON’, or ‘ALL’.

GetFieldAttr

C++: const char *pcForm::GetFieldAttr (UINT32 index, const char *type);

C: const char *pcFormGetFieldAttr (pcForm *form, UINT32 index, char *type);

Perl: $attrValue = ProductCenter::Form::GetFieldAttr ($index, $type);

Returns the attribute information. The current listing of 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”.

Table 2-1: Properties of Field Objects

"is searchable"
“choicelistid”
“colposition”
“defaultvalue”
“displaylength”
“editable”
“format”
“height”
“isrequired”
“maxlength”
“name”
“prompt”
“rowposition”
“type”
“width”
“xpos”
“ypos”

GetMasterForm

C++: pcForm *pcForm::GetMasterForm ();

C: pcForm *pcFormGetMasterForm (pcForm *form);

Perl: $master = ProductCenter::Form::GetMasterForm ();

Returns the pcForm of the master form. Returns null if the form is a master form.

SearchableFormsCount

C++: MSG_CODE pcCnxn::SearchableFormsCount (UINT32 *count);

C: MSG_CODE pcCnxnSearchableFormsCount (pcCnxn *cnxn, UINT32 *count);

Perl: $msg_code ProductCenter::Cnxn::SearchableFormsCount ($count);

Returns the number of Forms that can be used in a Search Query.

GetSearchableFormByIndex

C++: pcForm *pcCnxn::GetSearchableFormByIndex (UINT32 *index);

C: pcForm *pcCnxnGetSearchableFormByIndex (pcCnxn *cnxn, UINT32 *index);

Perl: $form ProductCenter::Cnxn::GetSearchableFormByIndex ($index);

Returns the Form that can be used in a Search Query by Index.

UsedFileTypeCount

C++: MSG_CODE pcCnxn::UsedFileTypeCount (UINT32 *count);

C: MSG_CODE pcCnxnUsedFileTypeCount (pcCnxn *cnxn, UINT32 *count);

Perl: $msg_code ProductCenter::Cnxn::UsedFileTypeCount ($count);

Returns the number of File Types currently used in ProductCenter.

GetUsedFileTypeByIndex

C++: pcForm *pcCnxn::GetUsedFileTypeByIndex (UINT32 *index);

C: pcForm *pcCnxnGetUsedFileTypeByIndex (pcCnxn *cnxn, UINT32 *index);

Perl: $form ProductCenter::Cnxn::GetUsedFileTypeByIndex ($index);

Returns the used File Type value based by Index.

Field object:

Constructors and Destructors

The form object contains two *constructors* and one *destructor*.

GetField

C++: pcField *pcForm::GetField (char *name);

C: pcField *pcFormGetField (pcForm *form, char *name);

Perl: $field = ProductCenter::Form::GetField ($name);

Returns the Field object based on the name. The standard ‘CUSTOM:’ and ‘COMMON:’ prefixes may be used.

GetFieldByIndex

C++: pcField *pcForm::GetFieldByIndex (char *type, UINT32 index);

C: pcField *pcFormGetFieldByIndex (pcForm *form, char *type, UINT32 index);

Perl: $field = ProductCenter::Form::GetFieldByIndex ($type, $index);

Returns the Field object by Index. The type can be 'CUSTOM', 'COMMON', or 'ALL'.

FieldDestroy

C++: ~pcField ();

C: void pcFieldDestroy (pcField *field);

Perl: ProductCenter::Field::DELETE ();

Destroys the memory of the field.

NOTE: Perl programmers should read "Destructors and Perl" on page 43 for information as to why they should not use this call.

Getting field information

FieldGetAttr

C++: const char *pcField::GetAttr (char *name);

C: const char *pcFieldGetAttr (pcField *field, char *name);

Perl: $attrValue = ProductCenter::Field::GetAttr ($name);

Returns the attribute information for the field. The current listing of attribute 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".

GetChoiceTypeChoiceCount

C++: UINT32 pcField::GetChoiceTypeChoiceCount ();

C: UINT32 pcFieldGetChoiceTypeChoiceCount (pcField *field);

Perl: $count = ProductCenter::Field::GetChoiceTypeChoiceCount ();

Returns the count of the different choices in the complete choice list otherwise it returns an error.

GetChoiceTypeChoiceName

C++: char *pcField::GetChoiceTypeChoiceName (UINT32 index);

C: char *pcFieldGetChoiceTypeChoiceName (pcField *field, UINT32 index);

Perl: $name = ProductCenter::Field::GetChoiceTypeChoiceName ($index);

Returns the name of the choice based on the index, it returns an error if the field isn't a choice list.

GetChoiceTypeChoiceDesc

C++: char *pcField::GetChoiceTypeChoiceDesc (UINT32 index);

C: char *pcFieldGetChoiceTypeChoiceDesc (pcField *field, UINT32 index);

Perl: $desc = ProductCenter::Field::GetChoiceTypeChoiceDesc ($index);

Returns the description of the choice based on the index, it returns an error if the field isn't a choice list.

GetTableTypeColumnCount

C++: UINT32 pcField::GetTableTypeColumnCount ();

C: UINT32 pcFieldGetTableTypeColumnCount (pcField *field);

Perl: $count = ProductCenter::Field::GetTableTypeColumnCount ();

Returns the count of columns for the table type attribute. It returns an error if it isn't a tabletype attribute.

GetTableTypeColumnName

C++: const char *pcField::GetTableTypeColumnName (UINT32 index);

C: const char *pcFieldGetTableTypeColumnName (pcField *field, UINT32 index);

Perl: $name = ProductCenter::Field::GetTableTypeColumnName ($index);

Returns the name of the column for the table type attribute. It returns an error if it isn't a tabletype attribute.

GetTableTypeColumnWidth

C++: UINT32 pcField::GetTableTypeColumnWidth (UINT32 index);

C: UINT32 pcFieldGetTableTypeColumnWidth (pcField *field, UINT32 index);

Perl: $width = ProductCenter::Field::GetTableTypeColumnWidth ($index);

Returns the width of the column for the table type attribute. It returns an error if it isn't a tabletype attribute.

A

Appendix A

Convenience Layer Functions

Just Ahead:

The convenience layer is a set of functions (or *object methods*) provided to aid the Perl application developer. The convenience layer is built in Perl using the ProductCenter Toolkit, and consists of several files with ".pm" extensions (such as "cnxn.pm", "ActivityDef.pm", etc..The Perl Toolkit installation process places these files in the site\lib\ProductCenter folder in the Perl installation directory. There is nothing in the convenience layer that you cannot do using just the ProductCenter Toolkit, but the methods reduce the amount of coding needed, by packaging commonly used routines.

Convenience layer functions

The following table lists each ProductCenter object and the convenience functions for each object. Check cnxn.pm for functions that might also apply to objects other than cnxn.

Table A-1: Convenience layer functions

Function	Arguments	Returns	Description
Object: ProductCenter::Cnxn			
Connect	$loginName , $password, $database, $host, $port respectively	$msgID	Called after creating a ProductCenter Cnxn object, sets the required values and connects to ProductCenter
ChoiceLists	None	%hash	The keys are the display names of the choice lists and the value is a reference to a hash containing the choices. The individual choice list hash table's keys are the choice names and the value is the choice description
Users	None	%hash	The keys are the user names and the values are their system IDs
Groups	None	%hash	The keys are the group names and the values are their system IDs
Classes	None	%hash	The keys are the class names and the values are their system IDs
ChoiceListValues	$system ID of choice list	%hash	For the given choice list id, the keys are the choice names and the value is the choice description
Queries	None	%hash	The keys are the query names and the values are their system IDs
ProcessDefs	None	%hash	The keys are the Process Definition names and the values are their system IDs

Table A-1: Convenience layer functions

Function	Arguments	Returns	Description
Worklist	None	%hash	The keys are the claimed Activity Instance names and the values are their system IDs
UnclaimedActivities	None	%hash	The keys are the unclaimed Activity Instance names and the values are their system IDs
NewQry	None	Qry object	Creates a new Qry object
LoadQry	$name	Qry object	Loads an already saved Qry of name $name
NewItem	$classname	Item object	Creates a new Item object
LoadItem	$id	Item object	Loads an already saved Item having id $id
LoadProcesses	None	ProcessInst list object	Populates the workflow process list with the list of all active workflow processes**.**
LoadAttachedProcesses	$itemID	ProcessInst list object	Populates the workflow process list with the list of attached workflow processes
NewLink	$type	Link object	Creates a new Link object
LoadActivity	$id	ActivityInst object	Load an already saved ActivityInst object having id $id
RouteProcessDefs	None	%hash	Returns a hash of ProcessDefs used for routing items. The key is the process name and the value is the system ID
IssueProcessDefs	None	%hash	Returns a hash of ProcessDefs used for issuing items. The key is the process name and the value is the system ID
Object: ProductCenter::List			
ListType	None	$list_type	Returns the List type of the list object
RowCount	None	$count	Returns the row count in the list object
Object: ProductCenter::Item			
AttrCount	None	$count	Returns the attribute count in the Item object
AttrType	$index	$attr_type	Returns the Item Attribute type of the attribute at the index passed

Table A-1: Convenience layer functions

Function	Arguments	Returns	Description
ChoiceListAttrVals	$attr_name	%hash	For the given attribute name the keys are the display names and the values are their system IDs
TableTypeAttrRowCount	$attr_name	$count	Returns the row count of the table type attribute in the Item object
TableTypeAttrColCount	$attr_name	$count	Returns the column count of the table type attribute in the Item object
LinkCount	$type	$count	Returns the count of links of type $type in the Item object
LinkTypeCount	None	$count	Returns the link type count in the Item object
AccessCount	$type	$count	Returns the access count in the Item object
GetUserAccess	None	%hash	The keys are the user names and the values are their access permissions
GetGroupAccess	None	%hash	The keys are the group names and the values are their access permissions
SetUserAccess	$user_name , $perms	$status	Sets the user access permissions
SetGroupAccess	$group_name, $perms	$status	Sets the group access permissions
RemoveUserAccess	$user_name	$status	Removes the access permissions for the user
RemoveGroupAccess	$group_name	$status	Removes the access permissions for the group
Object: ProductCenter::Qry			
SetAnd	None	$status	Sets the query type to "And".
SetOr	None	$status	Sets the query type to "Or".
ClauseCount	None	$count	Returns the clause count in the Qry object.
ClauseType	$index	$clause_type	Returns the Qry clause type of the attribute at the index passed.
ItemCount	None	$count	Returns the number of items which matched the previously executed query.

Table A-1: Convenience layer functions

Function	Arguments	Returns	Description
Where	$where_clause_string	$status	Sets the where clause string passed in the Qry object in addition to setting the table to cms_dfm and prepending "where" to clause if one does not exist.
Object: ProductCenter::Link			
Head	None, or $head	$status	0 parameters = GetHead 1 parameter = SetHead
Tail	None, or $tail	$status	0 parameters = GetTail 1 parameter = SetTail
Attr	$attr $value (optional)	$status	1 parameter = GetAttr 2 parameters = SetAttr
AttrCount	None	$count	Returns the attribute count in the Link object.
AttrType	$index	$attr_type	Returns the Link Attribute type of the attribute at the index specified.
ChoiceListAttrVals	$attr_name	%hash	The keys are the display names and the values are their system IDs.
TableTypeAttrRowCount	None	$count	Returns the row count of the table type attribute in the Link object.
TableTypeAttrColCount	$attr_name	$count	Returns the column count of the table type attribute in the Link object.
Object: ProductCenter::ProcessDef			
StartActivityDefs	None	%hash	The keys are the display names of the startActivityDefs and the values are their system IDs.
StartActivityDefCount	None	$count	Returns the count of the startActivityDefs.
Object: ProductCenter::ActivityDef			
NextActivityCount	None	$count	Returns the count of next ActivityDefs.
NextActivities	None	%hash	The keys are the display names of the next ActivityDefs and the values are their system IDs.
PrevActivityCount	None	$count	Returns the count of previous ActivityDefs.

Table A-1: Convenience layer functions

Function	Arguments	Returns	Description
PrevActivities	None	%hash	The keys are the display names of the previous ActivityDefs and the values are their system IDs.
Object: ProductCenter::ActivityInst			
AssignmentCount	$is_open	$count	Returns the assignment count.
AssignmentType	$is_open, $index	$assgn_type	Returns the assignment type of the assignment at the index passed.
GetOpenAssignments	None	%hash	The keys are the open assignments and the values are their assignment types.
GetAllAssignments	None	%hash	The keys are the assignments and the values are their assignment types.
Object: ProductCenter::Event			
EventType	None	$type	Returns the event type of the event object.
Object: ProductCenter::Monitor			
EventTypecount	None	$count	Returns the number of event types for the monitor object.

Appendix B

Attribute Types

B

Just Ahead:

Attribute types

Table B-1: Summary of attribute types

Type	Description	Example
Text	Any alphanumeric string of up to 255 characters	Managing Attributes
Text box (long text)	Any alphanumeric string up to 3999 characters. See Chapter 7 of the ProductCenter Administrator Guide for more details.	*Paragraphs of text.*
Date	Month, day, and year in specific formats	JUN-03-2010 (depending on format specified by cms.date.format; see Appendix A of the ProductCenter Administrator Guide)
Integer	Positive or negative whole number	750
Floating point	A floating point number can have up to 16 valid digits (although the database will allow you to store up to 200 digits). Any digit beyond the 16th will be random. ProductCenter always displays 6 digits to the right of the decimal point. IMPORTANT: See information about precision in Chapter 7 of the ProductCenter Administrator Guide for more details.	56.152700
Choice	One of a set of predefined text strings selected form a choice menu	New York
User	A customer-defined ProductCenter user	Jeff Brown
Group	A customer-defined group	Accounting
User role	A customer-defined job function to which a user can be assigned	Engineer
Group role	A customer-defined job function to which a group can be assigned	R & D
Message	A text string that the user cannot modify	ALL CHANGES REQUIRE APPROVAL
URL	A text string displayed as a hyperlink.	http://www.softech.com
View (table type)	A grid of item rows with multiple columns of information about each item. (Formerly called View attribute.)	Vendors = table below

Table B-2: Example of a View type attribute named Vendors

Name	Address	Contact	Phone
ACE distrib.	5001 E. 16th St., Philadelphia, PA 02833	Marie Ryack	(307) 782-5555
OmniNova	Woodland Park, Crescent Hill, IA	Alexis Young	(419) 555-3408
ET Express	42 St. Pauls St., La Jolla, CA 98721	John Minton	(415) 555-9822

ProductCenter common attributes

Table B-3: ProductCenter common attributes

Prompt	Name	Description	Required	Editable	Type
ID	CMS ID	Unique ID of ProductCenter item.	Yes	No	Integer
Class	CLASS_ID	The class to which an item belongs	Yes	No[a]	none
Name[b]	FILE_ID	The name of an item. The name must be unique within the class to which the item belongs. See “Names and special characters” on page 197 for rules about special characters.	Yes	Yes	Text
Title	FILE_TITLE	A description of an item.	Yes	Yes	Text
Version	REV_ID	A number assigned by ProductCenter to each unique instance of an item. Version numbering starts at 1 and is incremented by 1. ProductCenter creates a new version of an item: a) when a modified instance of the item is checked in; b) when the item is rolled back.	Yes	No	Integer
Revision[c,d]	PLC	An identifier (by default, alphabetic) used to track released configurations, if release management is in effect. ProductCenter increments an item's revision level when a released item is checked in. If release management is in effect, the minor revision numbers start at 1 for each major revision of an item.	Yes	Yes	Text
Status[c, d,e]	STATUS_CODE	If release management is enabled, an indicator of an item's release state. Valid values are “In Progress”, “In Approval”, “Released” and “Obsolete”.	No	Yes	Choice
Preparer	PREPARER	The name of the user who created the first version of an item.	Yes	Yes	User

Table B-3: ProductCenter common attributes (continued)

Prompt	Name	Description	Required	Editable	Type
Prepared On	DATE_PREPARED	The date on which the first version of an item was created.	Yes	Yes	Date
Reviewer[f]	REVIEWER	The name of the user who Submits an item to the "In Approval" state.	No	Yes	User
Reviewed On	DATE_REVIEWED	The date on which an item was Submitted to the "In Approval" state.	No	Yes	Date
Issuer[g]	ISSUER	The name of the user who Released an item.	No	Yes	User
Released On[f]	DATE_RELEASED	The date on which an item was Released.	No	Yes	Date
Last user	LAST_USER	The name of the user who performed the most recent modification on an item.	Yes	No	User
Last modified	DATE_LAST_MODIFIED	The date on which an item was most recently modified.	Yes	No	Date
Description	DESCRIPTION	A free-form description of an item (maximum of 4000 characters).	No	Yes	Text Box
Comments	COMMENTS	Free-form comments about an item. (maximum of 4000 characters)	No	Yes	Text Box
File size	FILE_SIZE	Size of a file in bytes	No	No	Integer
File type	F_TYPE	Type as defined in the filetype file	No	No	Text
Rel Path	REL_PATH	Not currently used (implemented for Mentor Graphics files)	No	No	Text
Location	LOCATION	Not currently used	No	No	Text
Checksum	CHECKSUM	Checksum of files used for tracking changes to files	No	No	Integer
Vault space[d]	VLT_SPACE	Name of the vault space in which a file is stored	No	No	Text
Vault Obj ID	VLT_OBJID	ID of the vault item	No	No	Integer
Vault Ver ID	VLT_VERID	Version ID of the vault item	No	No	Integer

[a] Although you can edit Class during an Add in the GUI, you cannot change it in the Toolkits after calling the constructor to create an item
[b] Name is editable only during Add operation.
[c] Not editable when Release Management is enabled
[d] System-controlled when Release Management is enabled
[e] Status options are system-defined when Release Management is enabled
[f] When Release Management is enabled, Reviewer is the person who submitted an item for approval
[g] When Release Management is enabled, Issuer is the person who approved an item for release

Names and special characters

The ability to use special characters in the names of items and files in ProductCenter has evolved over time, and is somewhat dependent upon the platform(s) you use. The following rules apply to the use of special characters in the names of items and files in ProductCenter.

1. General rule for item names: Names of file items can include any character that the operating system allows in file names, and names of non-file (project or part) items are completely unrestricted. In the SolidWorks Integrator, configuration names can also include any character allowed by SolidWorks.
2. The following characters are valid when adding parts or projects to ProductCenter:
   | , ! @ $ % ^ & ( ) = # + { } [ ] ~ ; ' ? * : \ / < >
3. Note that the % and _ characters are Oracle wildcards. ProductCenter queries that include these characters may return more results than you want.

Index

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