



DB2 Performance Monitor for OS/390

Data Collector Application Programming Interface Guide

Version 6

Note

Before using this information and the product it supports, be sure to read the information in "Appendix C. Notices" on page 109.

First Edition, April 2000

This edition applies to Version 6 of IBM DATABASE 2™ Performance Monitor for OS/390®, a feature of IBM DATABASE 2 Universal Database Server for OS/390 Version 6 (5645-DB2), and to all subsequent releases and modifications until otherwise indicated in subsequent editions.

© Copyright International Business Machines Corporation 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Preface	v	Snapshot Processing - Release Snapshot Store . . .	56
Who Should Read This Book	v	History Processing - Get History Data	57
How to Use This Book	v	Processing DB2 Exception Events	60
Availability of the Data Collector API	vi	Introduction to Exception Processing	60
Applicability of the Data Collector API	vi	Retrieve Event Exception Log	61
Chapter 1. Overview of Elements and Concepts	1	Retrieve Event Exception Details	67
The DB2 PM Data Collector	1	Start Exception Processing	75
The Workstation Application	2	Get Exception Processing Status	77
The Data Collector API Functions	2	Fetch Exceptions.	79
The Connection Concept	3	Stop Exception Processing	82
The User Concept	4	Executing DB2 Commands	83
The Security Concept	5	Execute DB2 Command	83
Logon with a Password.	5	Saving and Retrieving User Data	85
Logon with a PassTicket	5	Save User Data	86
How the Components Interact	6	Get User Data	87
Establishing and Terminating a User Session.	6	Parsing Data	88
Disconnecting and Reconnecting while Preserving a User Session	7	Introduction to Parsing	88
Chapter 2. Considerations for Using the Data Collector API	9	Get Token	91
Workstation Memory Handling	9	Get Token Value.	91
Code Page Conversions.	9	Skip Token	92
Preparations for Using PassTickets.	10	Test Token.	93
Common Return Codes and Reason Codes	10	Delete Counter	93
Compiler Considerations	12	Converting and Adjusting Dates and Times.	94
Linking the DB2 PM API Library on Windows NT	12	Introduction to Date and Time Functions	94
Using the DB2 PM API Trace Facility	14	Convert Store Clock Format to time_t Format	95
Chapter 3. The Data Collector API Functions	15	Convert time_t Format to Store Clock Format	97
Maintaining a TCP/IP Connection to the Data Collector	15	Add and Subtract in Store Clock Format.	99
Connect to Data Collector	15	Miscellaneous API Functions	99
Disconnect from Data Collector.	17	Hash Table Functions.	100
Maintaining a User Session to the Data Collector	18	Memory Releasing Function	100
Log On to Data Collector.	19	Qualifier Functions	101
Generate RACF PassTicket	21	Appendix A. Field Table Summary	103
Get Data Collector Information	22	Appendix B. Sample Traces	107
Log Off from Data Collector	27	Sample Connection Trace	107
Getting DB2 Performance Data	29	Sample Command Trace.	107
Introduction to Counters and Snapshot Stores	29	Sample Data Trace.	108
Working with Returned Data	37	Appendix C. Notices	109
Snapshot Processing - Initialize Snapshot Store	42	Trademarks	110
Snapshot Processing - Query Snapshot Stores	44	Bibliography	113
Snapshot Processing - Get Snapshot Data	46	Index	115
Snapshot Processing - Reset Interval Data	51	Readers' Comments — We'd Like to Hear from You	119
Snapshot Processing - Get History Contents	54		

Preface

This book describes the Data Collector Application Programming Interface of IBM® DATABASE 2 Performance Monitor for OS/390 Version 6 and its use.

IBM DATABASE 2 Performance Monitor for OS/390 (DB2 PM) is the performance analysis tool to monitor and tune DB2® systems and DB2 applications. It is one of the optional features that complement IBM DATABASE 2 for OS/390 (DB2).

DB2 PM provides the host processor-based MVS ISPF Online Monitor and the Workstation Online Monitor to interactively monitor DB2 performance, and to execute DB2 commands. DB2 PM also provides a batch facility to generate performance reports.

New with DB2 PM Version 6 (as program temporary fix) is the DB2 PM Data Collector Application Programming Interface (API). It provides a programming interface to DB2 performance data. Workstation-based application programs can use this interface to connect to the host-resident DB2 PM Data Collector through TCP/IP to access DB2 performance data in real time or from the Data Collector's own history data set.

The DB2 PM Data Collector API allows programmers to develop their own customized application programs.

All DB2 PM Data Collector API functions can be accessed using calls from C or C++ programs or any other programming language that supports C function calls.

Who Should Read This Book

This book is for system and application programmers who want to design and implement workstation application programs that take advantage of the API to gain access to DB2 performance data.

It is assumed that the reader is familiar with the DB2 PM Online Monitor functions and the C programming language.

How to Use This Book

Read "Chapter 1. Overview of Elements and Concepts" on page 1 to gain an understanding of the elements and concepts of the DB2 PM API and how these elements interact with each other. This knowledge helps to decide whether the use of the API is adequate to solve your task. This knowledge is also required to effectively use the API functions.

"Chapter 3. The Data Collector API Functions" on page 15 describes the Data Collector API functions, its calling conventions, and its parameters, and gives code examples for an application program. Functions with similar objectives are grouped together; every group begins with a comprehensive description of the concepts of which you should be aware. Use this information when you apply these functions to your application program.

Availability of the Data Collector API

The DB2 PM Data Collector API is available for DB2 PM Version 6 as PTF, as a recommended maintenance level for Version 6. Future versions of DB2 PM might provide enhanced versions of the API as an integral part of DB2 PM.

Applicability of the Data Collector API

The DB2 PM Data Collector API is applicable with:

- IBM DATABASE 2 Universal Database Server for OS/390 (DB2 UDB for OS/390) Version 6, program number 5645-DB2
- IBM DATABASE 2 Server for OS/390 (DB2 for OS/390) Version 5, program number 5655-DB2
- IBM DATABASE 2 for MVS™/ESA (DB2 MVS/ESA) Version 4, program number 5695-DB2

or any higher version.

DB2 PM provides the necessary workstation development kit and workstation run-time environment as either Dynamic Link Library or Shared Library for the following operating system platforms:

Table 1. Supported Operating Systems

Operating System	Library
IBM OS/390 Version 2.7, or higher	Shared library
IBM AIX® Version 4.3, or higher	Shared library
Microsoft® Windows® NT 4.0	Dynamic link library
Sun Solaris 5.6, or higher (32 bit)	Shared library
Sun Solaris 5.7, or higher (64 bit)	Shared library
Linux Version 2.2, or higher, on Intel x86	Shared library

DB2 PM Installation and Customization describes how to install the DB2 PM API on a workstation. You may want to copy or use the DB2 PM API as often as required, provided you hold a valid DB2 PM license.

The workstation requires TCP/IP to be installed, either as part of the operating system, or as a separate licensed program. The Data Collector requires IBM TCP/IP Version 3 Release 2 for MVS/ESA to be installed on the host processor. These requirements are specific to the use and operation of the DB2 PM API. The general prerequisites for DB2 PM are to be applied.

Chapter 1. Overview of Elements and Concepts

A DB2 subsystem running on the OS/390 operating system generates and collects data about its own performance, but does not provide facilities to evaluate and report about its performance. Companies with a distinctive need to evaluate the performance of DB2 can use the DB2 Performance Monitor (DB2 PM) feature. DB2 PM provides interactive and batch-oriented facilities to monitor the DB2 performance and generate reports about it.

However, sometimes you might want to use customized applications that are external to DB2 and DB2 PM and which need access to DB2 performance data. For example, a company uses several database systems from different suppliers. Every of these database systems has an own user interface to performance-related data, which generates increased complexity and requires specialized skill. The solution is an application program that monitors performance aspects of all these database systems and provides the results in a common user interface to your support personal. Then, the application would need real-time access to performance data of every of these database systems.

The DB2 PM Data Collector Application Programming Interface (API) provides a structured set of functions that you can use in an application program to process DB2 performance data in real time. Thus, the API does not require the DB2 PM Online Monitor or DB2 PM Batch.

The use of the API at application development time and at application run time involves several elements of DB2 PM and DB2 as well as connectivity aspects. The remaining part of this chapter gives an overview of the involved elements, explains the user and connection concept, and describes how they work together.

The DB2 PM Data Collector

The DB2 PM Data Collector is the host component of the DB2 PM API. It provides an access path from a client application program to DB2 internal performance data.

The Data Collector resides in an OS/390 address space. It needs to run as an OS/390 started task before any client application can connect to it. Once started, the Data Collector takes the role of a server.

Upon startup of the Data Collector, TCP/IP is also started on the host. This allows client application programs to connect to the Data Collector through TCP/IP when required. The Data Collector is capable of serving up to 500 logged-on applications, respectively users, concurrently.

After an application has established a TCP/IP connection to the Data Collector, the Data Collector is ready to accept a logon request from a client application. During logon the application identifies itself to the Data Collector.

The Data Collector does not require an application to remain connected after a successful logon. Users can disconnect from and reconnect to the Data Collector from varying places at any time and still remain logged on.

The Workstation Application

The application can reside on workstations that run one of the supported operating systems listed in Table 1 on page vi. You need the corresponding DB2 PM API installed on these workstations to develop, test, and run an application. The DB2 PM API support contains all files and libraries required to use the API.

The application can be written in any programming language that supports the C language calling conventions required to call the API functions.

The application on a workstation communicates with the Data Collector by using TCP/IP. The workstation needs to have TCP/IP support available either as part of the operating system or as a separate licensed program. TCP/IP is not apparent to the application. When it connects to the Data Collector, by calling the corresponding API function, the underlying operating system services are used to establish a TCP/IP connection to the Data Collector.

The application controls the connection to the Data Collector. After a successful connection and a successful logon, a user session is established. During this session the application calls any number of API functions to describe the data it wants to access or to be returned. At the end it closes the user session with the Data Collector and terminates the connection to it. This frees all Data Collector resources that the application has occupied.

The Data Collector API Functions

The DB2 PM API gives an application program access to DB2 performance data. Hence, the majority of all API functions is related to DB2 performance data. In addition, an API function is available to execute DB2 commands. The remaining functions maintain TCP/IP connections and user sessions between the application and the Data Collector.

An application using the API has access to the following types of data and can perform the following functions:

- Access to DB2 performance data

The application can request statistics information on a DB2 subsystem level and on a thread level. It can query DB2 system parameters, for example, installation parameters, or VSAM catalog information.

- Access to DB2 event exceptions

The Data Collector logs the most recent DB2 exceptions in an exception log. The application can retrieve up to 500 of these recent exception log records and process them individually. The DB2 PM API also supports a synchronous notification of DB2 exceptions. Once set up, the application is notified about DB2 event exceptions when they occur.

- Execution of DB2 commands

The DB2 PM API supports the execution of DB2 commands from within the application. The Data Collector transfers the commands to DB2 and delivers the response data to the application. Provided that a logged-on user has sufficient DB2 privileges, all DB2 commands can be executed, except the application cannot start or stop a DB2 subsystem.

- Access to user data in the Data Collector

The application can save user-specific data in the Data Collector and retrieve this data at any point in time. This allows you to develop applications that support mobile users. Users of the application can disconnect from the Data Collector

(while remaining logged on) at one location and reconnect to it from another location. The Data Collector serves as an interim storage for user-specific data that need to be kept in the meantime.

- Parser functions are available for extracting relevant data from Data Collector responses. These functions are helpful in cases where API functions return complex and varying data structures. The parsing functions also check returned data streams for correctness, and convert data between the host processor format and the workstation format.
- Time conversion functions convert host date and time formats and workstation date and time formats. Arithmetic functions simplify time zone adjustments.

DB2 performance data that is accessible through the Data Collector API is raw DB2 instrumentation data and data that the Data Collector derives from raw DB2 instrumentation data. The information about the data fields is provided in a separate text file that accompanies the DB2 PM API. “Appendix A. Field Table Summary” on page 103 describes how the data in the flat file is organized and shows a short sample of the field table.

“Getting DB2 Performance Data” on page 29 gives a comprehensive introduction to the Data Collector counter concepts, snapshot and history processing, the snapshot store concept, and shows how data can be filtered to reduce the amount of available performance data. The knowledge about these concepts is required to effectively use the API functions that deal with performance data.

Note that the Data Collector can be customized during its installation and DB2 PM user exit routines may be active. User exit routines allow user-written routines to perform customized processing. This may limit the access of the application to some counters, regardless of the fact that they are listed in the field table.

The Connection Concept

The communication between the application and the Data Collector happens through TCP/IP. The application uses the **pmConnect()** function to open a TCP/IP connection to the Data Collector. Several connections can be opened, if required. The maximum number of connections is specified as a start parameter of the Data Collector itself.

Once a communication path to the Data Collector is established, it is uniquely identified by an identifier, called a *handle*. All subsequent API functions use this handle to identify the physical connection path to the Data Collector they want to use. If several connections are opened, different handles are used to distinguish among the connections.

Users logging on to the Data Collector are identified by a *work profile*. All subsequent API functions use this work profile to assign a function to a user.

This way, connections to the Data Collector are completely separated from users of the Data Collector. The application can use every opened TCP/IP connection to issue requests to the Data Collector for every user. Vice versa, the application can use a single TCP/IP connection also for several users.

Connections to the Data Collector are required when the application actually needs to communicate with it. Users who are already logged on to the Data Collector remain logged on even when no connection is opened. This allows you to write applications where users can disconnect at one work place (the application calls

the **pmDisconnect()** function) and reconnect from another work place (the application calls the **pmConnect()** function).

The User Concept

The workstation application must identify its users to successfully log on to the Data Collector. The Data Collector, when receiving a logon request, always uses the System Authorization Facility (SAF), and the IBM Resource Access Control Facility (RACF) or any other security system, to verify a user's authorization.

The terms "SAF user" and "SAF group" are used in this book in accordance with their definitions in the RACF[®] literature. See the appropriate manuals, if required.

The DB2 PM API requires you to define a "user" in terms of a work profile. A work profile is a character string that contains the following information:

- SAF user ID

The user ID of a DB2 PM user as known to RACF. This information is always required to identify a user logging on to the Data Collector.

- SAF group ID

The group identifier an SAF user is associated to in RACF. This field is optional, but if it is used it must specify a valid SAF group ID.

- Profile ID

An optional field that the application can use as required. For example, you can use it to specify subusers of an SAF user ID. To simplify matters in this book, this field is used to specify a subuser.

Once the application has specified a user by means of a work profile, this information is used as parameter **workProfile** of the **pmLogOn()** function. The Data Collector manages the authorization of the SAF user ID and, if specified, the SAF group ID.

After a user is authorized to request services from the Data Collector, all subsequent requests are identified by a user's work profile. For security reasons, every combination of an SAF user ID and an SAF group ID requires authorization.

Subusers, if specified in the Profile ID field, do not need separate authorization. Upon proper authorization of an SAF user ID and SAF group ID combination, all subusers are implicitly authorized.

Every user specified by a unique work profile must log on and off individually. This ensures that all resources used by a user in the Data Collector are released when the user logs off.

The combination of the connection concept and the user concept ensures that every request to the Data Collector is identified by:

- A handle, which specifies the connection path
- A work profile, which specifies a user.

To preserve security, the application must request authentication of a user on every connection path the user wants to use.

The Security Concept

The DB2 PM Data Collector is one of many OS/390 system resources that client applications might want to access. Access to OS/390 system resources is usually controlled by an OS/390 security server, for example, by RACF. This requires that every potential application and every potential user is known by RACF. Access to a resource is given if an application or user is successfully identified by the security server and sufficient authority is given.

Further, the communication between the application and the Data Collector happens through TCP/IP, which does not provide any security mechanism that prevents hostile intrusion. All data is transferred as plain text. Everyone who knows the IP address and port number can connect to the Data Collector.

The Data Collector API supports two levels of security implementations. Both levels build on how the application logs on to the Data Collector. Once the application has successfully logged on, all further API function calls are bound to this logon.

Logon with a Password

The application is granted access to the Data Collector when it identifies itself by a correct user identification and password. The **pmLogOn()** function passes both parameters, the user ID and the password, over to the Data Collector for authorization. The Data Collector then requests the OS/390 Security Server (RACF) to verify the user identity. If the authorization succeeds, a user is successfully logged on, and the application can call further API functions as required.

This method provides only limited protection. The password is transmitted as plain text to the Data Collector. You should only consider this method if the application runs inside a secure and protected TCP/IP network.

Logon with a PassTicket

For enhanced security you can implement a logon process that uses an encrypted password, called a *PassTicket* (a RACF term). A PassTicket has two properties that make it difficult to be misused:

1. It remains valid for only 10 minutes. Once you have generated a PassTicket, it should be used as parameter of the **pmLogOn()** function within the next 10 minutes, otherwise it becomes unusable.
2. It can be used only once. For every logon the application must generate a new PassTicket.

The **pmGenPassticket()** function generates a PassTicket. The algorithm to generate a PassTicket uses:

1. The hardware clock of the host processor where the Data Collector resides.
2. The user ID for which the application requests authorization by the OS/390 Security Server (RACF).
3. A fixed application name and a secure signon key that are known in the OS/390 Security Server.

A copy of the secure signon key must reside on the workstation to enable the **pmGenPassticket()** function to access it. You should use any of the operating system functions to protect or hide it, or use an access control device connected to the workstation (like a smart card reader) to make it available during the logon process.

After the **pmGenPassticket()** function has generated a PassTicket, the application uses it as a logon parameter of the **pmLogOn()** function.

How the Components Interact

This section briefly describes scenarios how an application program interacts with the Data Collector and which API functions are involved in this process. The timely order is pointed out, and the previously introduced components and their actions are explained.

- The first scenario describes how an application program establishes and terminates a user session with the Data Collector.
- The second scenario describes how an application program disconnects and reconnects a mobile user and maintains the user session with the Data Collector.

This scenario deploys the independence between connections and user sessions.

For these scenarios it is assumed that the Data Collector is started and can be reached by client applications through a TCP/IP connection. If it is not started, any attempt to connect to it fails. Further, it is assumed that the workstation is properly set up and that its TCP/IP services are available.

Establishing and Terminating a User Session

1. The application must establish a TCP/IP connection to the Data Collector before any other API function can be called. To do this, the application calls the **pmConnect()** function and passes along some parameters, for example, the TCP/IP port where the Data Collector is listening, and the host name where DB2 is installed. Assumed the Data Collector can be reached successfully and the host name can be resolved to a valid IP address, a TCP/IP connection between the application and the Data Collector is then opened.
2. The API returns a unique *handle*, which identifies this connection path. Subsequent requests to the Data Collector use this handle to identify the communication path.

If the Data Collector or DB2 is not available, the **pmConnect()** function returns the appropriate return code, which the application can evaluate.

3. Next, the application program needs to log on. The Data Collector authenticates the application and verifies the authorization of the user, or group, or subuser, logging on. The logon process passes security-sensitive information to the Data Collector. Therefore, the API provides two alternatives:
 - The less secure logon calls the **pmLogOn()** function and passes along parameters that identify the user or group or subuser, together with a password.
 - The secure logon generates an encrypted password first, using the **pmGenPassticket()** function. Thereafter, the **pmLogOn()** function uses the encrypted password as a logon parameter.

The work profile that was used to log on to the Data Collector is used for all subsequent function calls to associate Data Collector requests with a user.

4. After a successful logon, it is recommended that the application gets some information about the Data Collector to ensure that it is functionally compatible with the Data Collector version installed on the host processor. It calls the **pmGetInfo()** function and performs the necessary checks.

A user session is now established. The application can proceed according to its purpose.

5. If no more tasks are to be executed, the application starts the termination step. It calls the **pmLogOff()** function to log off from the Data Collector, which releases all snapshot stores in the Data Collector and frees all resources.
6. The application drops the TCP/IP connection to the Data Collector with a **pmDisconnect()** function call.

Disconnecting and Reconnecting while Preserving a User Session

1. An application is already connected to the Data Collector; the connection is identified by a handle. A user is logged on and identified by a work profile. The application continuously monitors a DB2 process. When the application recognizes a user interaction that shows that the user wants to disconnect from the Data Collector but wants to remain logged on, it stops gathering and monitoring data.
2. The application calls the **pmSaveUserData()** function to save user-specific data in the Data Collector. This data can include everything that is required for a later reconnect to the Data Collector, for example, the current workstation settings, or parameters the application was using while monitoring the DB2 process.
If the application runs on a portable PC, it could also save this data on the PC's local disk.
3. The application calls the **pmDisconnect()** function, with the previously used handle as parameter. This drops the specified TCP/IP connection to the Data Collector.
4. The application can now close down. The connection is dropped. The user is still logged on to the Data Collector.
5. When the application is started again (from another workstation), and the user wants to continue monitoring the DB2 process, the application reconnects to the Data Collector with the **pmConnect()** function. The new connection is identified by a different handle.
6. Even so the user is still logged on to the Data Collector, the application now has to issue a new **pmLogOn()** function call. This allows the Data Collector to authenticate the application and to verify the authorization of the user logging on. Because the user is still logged on, the Data Collector returns a warning that the user is already logged on — which is a normal response here.
7. After the logon, the application retrieves the previously saved user-specific data from the Data Collector with the **pmGetUserData()** function. It uses this data to reconstruct the previous state, for example, the workstation's settings, or the parameters used.
8. The application is now ready to continue the monitoring process. Subsequent function calls to the Data Collector use the new handle to identify the connection path, and the same work profile to identify the user.

Chapter 2. Considerations for Using the Data Collector API

The DB2 PM API is delivered as a set of C programming language functions, which can be called from the application program you are writing. This book assumes that you write an application in the C programming language. For this purpose the API is delivered as a dynamic link library or shared library, together with the header files, for the operating systems listed in Table 1 on page vi.

If you write an application in a programming language other than C, consult the appropriate programming manuals about making mixed-language calls.

The DB2 PM API is delivered together with header files that contain the necessary declarations for all functions that make up the API. “Chapter 3. The Data Collector API Functions” on page 15 provides the relevant header file name together with every function description.

In “Chapter 3. The Data Collector API Functions” on page 15 parameters are either marked as input or as output parameters. Input parameter values are sent to the Data Collector; output parameter values are returned by the Data Collector to the application in response to a function call.

Note that all API function names are case sensitive.

Workstation Memory Handling

Whenever the application program calls an API function, the Data Collector returns data to it. Most functions allocate the necessary data space in the workstation’s memory to store the returned data. The Data Collector itself does not provide data space that you can use for this purpose. An exception is the 1-MB buffer in the Data Collector that is intended to store user-specific data. See “Saving and Retrieving User Data” on page 85 for details.

The application is responsible to release allocated memory when the data space is no longer needed. The API provides two functions to release memory when required: the **pmFreeMem()** function, and the **freeHashTable()** function, which releases memory that is allocated to store data of variable length in hash tables.

Code Page Conversions

The Data Collector and the workstation applications may use different standards to represent data. The Data Collector uses EBCDIC (Extended Binary-Coded Decimal Interchange Code). Most workstation applications usually use ASCII (American National Standard Code for Information Interchange); however, some use EBCDIC.

Further, both sides may use different code pages to accommodate for different national languages. Code pages specify how the EBCDIC and ASCII codes are presented for a specific language.

When the application exchanges data with the Data Collector, it automatically converts the data between ASCII and EBCDIC (if the workstation uses ASCII), and it converts text data according to the code pages in use. This conversion is

transparent to the application, however, the code page used on the workstation must be specified with the **pmConnect()** function.¹

Preparations for Using PassTickets

If the application uses the **pmGenPassticket()** function to generate encrypted passwords, ensure that the OS/390 Security Server (RACF) is properly prepared.

Log on to the OS/390 system as a TSO/E user and execute the following RACF commands to create a profile for DB2 PM. You may need sufficient authority to issue these commands.

1. **SETROPTS CLASSACT(PTKTDATA)**
2. **SETROPTS RACLIST(PTKTDATA)**
3. **RDEFINE PTKTDATA MVSDB2PM SSIGNON(KEYMASKED(*key_value*)) UACC(NONE)**

This command defines the application name MVSDB2PM to the OS/390 Security Server (RACF) and associates a secure signon key (*key_value*) with the name of the application. *key_value* is 16 characters long and can contain numeric (0-9) and alphabetic (A-Z) characters.

Note: The application name must be set to MVSDB2PM.

After you have set up the profile for DB2 PM, the application can call the **pmGenPassticket()** function to generate a PassTicket. **pmGenPassticket()** uses MVSDB2PM as its **application** parameter, and *key_value* as its **secureSignonKey** parameter. See “Generate RACF PassTicket” on page 21 for more details.

Common Return Codes and Reason Codes

All DB2 PM API functions complete with a combination of a return code and a reason code.

The DB2 PM API knows the following return codes:

Table 2. Common DB2 PM API Return Codes

Return Code ² (Hex)	Description
0	The function completed successfully.
4	The Data Collector returned a warning. Depending on the reason code, output data <i>might</i> have been returned.
8	The Data Collector returned an error. The function did not complete successfully. No output data is returned.
PM _ CONNECTION ERROR	The DB2 PM API detected a connection error. The function did not complete successfully. Depending on the reason code, the application <i>may</i> still be connected to the Data Collector.

1. Internally, the **pmConnect()** function sends the workstation’s code page number to the Data Collector. The Data Collector, which “knows” already the host’s code page number, builds a conversion table from both code pages, which is then returned to the workstation. The conversion table is stored internally and ensures correct conversions of outgoing and incoming data.

Table 2. Common DB2 PM API Return Codes (continued)

Return Code ² (Hex)	Description
PM _ API ERROR	The DB2 PM API detected an error. The function did not complete successfully.
PM _ API WARNING	The DB2 PM API detected a warning. The function completed.

The following DB2 PM API reason codes are universally valid for all API functions and are not repeated with the individual function descriptions:

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1F	The client application has called a DB2 PM API function without being logged on to the Data Collector.
8	43	The DB2 subsystem is not started, or it cannot be communicated with.
8	1552	Buffer shortage. The Data Collector has reached the maximum number of API function calls it can serve simultaneously. Try again later, or increase the number of session buffers at Data Collector startup.
PM _ CONNECTION ERROR	PM _ NETWORK DOWN	The network is down. Check the network connection.
PM _ CONNECTION ERROR	PM _ TIMEOUT	The network operation was canceled because of a timeout. The network traffic is probably slow, or the host IP address is wrong.
PM _ CONNECTION ERROR	PM _ HOST UNREACHABLE	The specified host is unreachable. Check the specified IP address and the network route.
PM _ CONNECTION ERROR	PM _ HOSTDOWN	The specified host is down. Try later.
PM _ CONNECTION ERROR	PM _ WINSOCKET _ NOT READY	Only Windows NT: The network service is not started. Check the network settings.
PM _ CONNECTION ERROR	PM _ SOCKET DESCR _ INVALID	The specified socket descriptor is not valid.
PM _ CONNECTION ERROR	PM _ SOCKET NOT CONNECTED	The specified socket is disconnected. You have to connect first. (Not applicable for Sun Solaris, Linux, and OS/390).
PM _ CONNECTION ERROR	PM _ CONNECTION _ ABORTED	The Data Collector is no longer reachable. It may be stopped, or the connection to it may be terminated.
PM _ CONNECTION ERROR	PM _ UNEXPECTED _ EOF	The Data Collector sent incomplete data. Check the OS/390 system log for messages from the Data Collector. The connection may not be usable.

2. Note that return codes and reason codes are single-word text strings. The strings have no intervening space characters. The underscore characters (_) are part of the strings. If the text strings appear as broken strings on some output media, this is because of the limited presentation space available.

All function-specific reason codes are listed and explained with the respective functions. See "Chapter 3. The Data Collector API Functions" on page 15.

Compiler Considerations

To compile your C program use any C or C++ compiler compatible to the operating system platform on which you are working. For example, you can use IBM VisualAge® C++ for Windows NT, or Microsoft Visual C++.

For IBM VisualAge C++ use compiler option /Su4 to use four bytes for enumerations instead of the SAA default setting.

For Visual C++ 6.0 use the default compiler settings.

Linking the DB2 PM API Library on Windows NT

If you plan to run the application on Windows NT, it is highly recommended that you resolve entry points in the DGOKAPI.DLL library dynamically. This prevents problems that might occur with some compilers.

This is not required for the other supported operating systems because their DLLs support dynamic entry point resolving.

The following example shows how to do this for Microsoft Windows NT 4.0:

```
/* Example how to resolve entry points of C-API functions dynamically */
/* on Windows NT 4.0. */
/*****

#include <windows.h>
#include <stdio.h>
#include "pmConnect.h"
#include "pmGenPassticket.h"
#include "pmLogOnOff.h"

/* Type definitions for dynamically loaded functions used in this */
/* program. See corresponding C-API header files for parameters. */
typedef pmReturnCodes __cdecl ppmConnect(char *, char *, unsigned int, pmHost *);
typedef pmReturnCodes __cdecl ppmDisconnect(pmHost *);
typedef pmReturnCodes __cdecl ppmGenPassticket(pmHost *, char *, char *, char *, char *);
typedef pmReturnCodes __cdecl ppmLogon(pmHost *, char *, char *);
typedef pmReturnCodes __cdecl ppmLogoff(pmHost *, char *);

/* Name and handle of DB2 PM C-API DLL */
char *dllName = "DGOKAPI.DLL";
HINSTANCE dllHandle = NULL;

/* Function pointers for all dynamically loaded functions */
ppmConnect *fnConnect = NULL;
ppmDisconnect *fnDisconnect = NULL;
ppmGenPassticket *fnGenPassticket = NULL;
ppmLogon *fnLogon = NULL;
ppmLogoff *fnLogoff = NULL;

/* Function prototypes for helper functions */
FARPROC LoadFunction(char *funcName);
void LoadAPI(void);
void UnloadAPI(void);

int main(void)
{
```

```

pmHost      myHandle;
pmReturnCodes error;
char        passticket[8];
char        *workprofile = "PMUSER  GROUPID  PROFILEID      TERMINALID      ";

/* Load DB2 PM C-API functions */
LoadAPI();

/* Connect to the data collector at IP address 10.0.0.1 */
/* at port 4711 using workstation codepage 850. */
error = (*fnConnect)("10.0.0.1", "4711", 850, &myHandle);

/* Create a passticket for user PMUSER for application */
/* MVSDDB2PM using the secure signon key E001033FAF00007B */
error = (*fnGenPassticket>(&myHandle, "PMUSER", "MVSDDB2PM",
                          "E001033FAF00007B", passticket);

/* Logon to DB2 PM Data Collector using the workprofile */
error = (*fnLogon>(&myHandle, workprofile, passticket);

/* Execute other DB2 PM commands */
/* ... */

/* Logoff from DB2 PM Data Collector */
error = (*fnLogoff>(&myHandle, workprofile);

/* Disconnect from Data Collector */
error = (*fnDisconnect>(&myHandle);

/* Free DB2PM C-API DLL */
UnloadAPI();
return(0);
}

/* Load the DB2 PM DLL if necessary and resolve a single */
/* entry point for the specified function name. */
FARPROC LoadFunction(char *funcName)
{
    FARPROC pfn=NULL;

    /* Load DLL if necessary */
    if (dllHandle == NULL)
        dllHandle = LoadLibrary(dllName);

    if (!dllHandle)
    {
        fprintf(stderr,
                "Error loading DB2 PM C-API DLL! Program terminated!\n");
        fflush(stderr);
        exit(-1);
    }
    else
    {
        /* Get entry point for the specified function */
        pfn = GetProcAddress(dllHandle, funcName);
        if (!pfn)
        {
            fprintf(stderr,
                    "Error loading function %s from DB2 PM C-API DLL!\n",
                    funcName);
            fprintf(stderr, "Program terminated!\n");
            fflush(stderr);
            exit(-1);
        }
    }
}

```

```

    return pfn;
}

/* Load all functions used in this program from DB2 PM C-API DLL */
void LoadAPI(void)
{
    fnConnect = (ppmConnect *) LoadFunction("pmConnect");
    fnDisconnect = (ppmDisconnect *) LoadFunction("pmDisconnect");
    fnGenPassticket = (ppmGenPassticket *) LoadFunction("pmGenPassticket");
    fnLogon = (ppmLogon *) LoadFunction("pmLogon");
    fnLogoff = (ppmLogoff *) LoadFunction("pmLogoff");
}

/* Unload DB2 PM C-API DLL */
void UnloadAPI(void)
{
    FreeLibrary(dllHandle);
}

```

Using the DB2 PM API Trace Facility

If you need to diagnose what happens when the application communicates with the Data Collector, you can activate the DB2 PM API trace facility. You can choose to let the trace facility record connection information, commands transferred, and data transferred. By default, the trace facility displays the required information. Alternatively, you can redirect this information to files for later analysis.

You activate the trace facility by adding the following environment variables to the operating system and by setting them to **ON**.

- The **PM_CONNECTION** environment variable controls tracing of connection information. By default, the trace data is written to STDOUT, which is usually the workstation's screen. If you want to capture the information in a file, add the environment variable **PM_CONNECTION_FILE** to the operating system and set its value to a file name.
- The **PM_COMMAND** environment variable controls tracing of DB2 PM commands. By default, the trace data is written to STDOUT, which is usually the workstation's screen. If you want to capture the information in a file, add the environment variable **PM_COMMAND_FILE** to the operating system and set its value to a file name.
- The **PM_DATA** environment variable controls tracing of DB2 PM API data that is transferred between the application and the Data Collector. By default, this data is written to STDERR, which is usually the workstation's screen. If you want to capture the information in a file, add the environment variable **PM_DATA_FILE** to the operating system and set its value to a file name. Note that this data is shown as binary data.

For information about adding, setting, resetting, or removing environment variables see the operating system manuals.

The output files created by the trace facility are not erased from the workstation's hard disk. Subsequent trace data is appended to existing files. Make sure that you remove the environment variables when no longer required.³

"Appendix B. Sample Traces" on page 107 shows a sample of a connection trace, a command trace, and a data trace.

3. In the current version of the DB2 PM API only the existence of the environment variables is checked.

Chapter 3. The Data Collector API Functions

Maintaining a TCP/IP Connection to the Data Collector

The DB2 PM API uses TCP/IP to communicate with the Data Collector. Your application needs a direct network connection to the Data Collector and a configured and running TCP/IP environment on your workstation. The API does not support connections through proxy servers.

The following functions connect your application to the Data Collector by establishing a TCP/IP connection between your workstation and the Data Collector, or disconnect both. These are always the first and last steps your application must perform when it needs to communicate with the Data Collector.

The next step after a successful connection is to log on to the Data Collector, as described in “Log On to Data Collector” on page 19.

Notes:

1. A user remains logged on to the Data Collector, regardless of the connection state. An application can disconnect from and reconnect to the Data Collector without changing a user’s status in the Data Collector. However, after a reconnect the application must issue a **pmLogOn()** function call to allow the Data Collector to verify the user’s authorization. See also “Log On to Data Collector” on page 19.
2. A handle, once established with the **pmConnect()** function, identifies a unique TCP/IP connection to the Data Collector. It is used with subsequent function calls to identify this TCP/IP connection. It remains available until the connection is terminated with a **pmDisconnect()** function call.

Connect to Data Collector

Function Call

```
pmReturnCodes pmConnect (char*      host,  
                        char*      servicePort,  
                        unsigned int codePage,  
                        pmHost*    handle)
```

Header File

pmConnect.h

Description

This function opens a TCP/IP connection to a Data Collector and returns parameter **handle**, which identifies the opened TCP/IP connection. Subsequent function calls that communicate with the Data Collector use this handle to identify this connection. If several **pmConnect()** function calls are made, every connection gets a unique handle.

Before the connection is confirmed by a return code of 0, the API loads an internal conversion table and a copy of the field table from the Data Collector to the workstation’s memory. The conversion table ensures correct EBCDIC to ASCII (and ASCII to EBCDIC) conversions and code page conversions (see “Code Page Conversions” on page 9). The field table contains a list of all DB2 PM counters that

the Data Collector supports. The field table is used internally to validate the counters you specify with the API functions. This operation is not apparent to your application.

Note that every **pmConnect()** function call causes a download of a conversion table and a field table. This is because the connections could be done to different Data Collectors, whereby each Data Collector could use different code pages and field tables.

Parameters

1. **host** (input)

The host where DB2 is installed. The host processor is identified by either:

- The name of the host as specified in the local *hosts* file. The local *hosts* file contains the mappings of the IP addresses to host names.
- The name of the host, as registered in the Domain Name System server.
- The IP address of the host, in dotted-decimal form.

2. **servicePort** (input)

The TCP/IP port where the Data Collector is listening. The port is specified as either:

- A decimal number between 1024 and 65535, as string.
- The service name, as registered in the local *services* file.⁴ The local *services* file contains port numbers for well-known services as defined by # RFC 1060 (Assigned Numbers). This is an alias for the decimal port number.

3. **codePage** (input)

The number of the code page (ASCII or EBCDIC) used by the application program. All strings sent to or received from the Data Collector are converted between the host code page and this code page.

4. **handle** (output)

A platform-independent handle that identifies this TCP/IP connection to the Data Collector. The application must provide the memory area for this handle.

Example

```
#include "pmConnect.h"

pmHost myHandle;

// connect to data collector
error = pmConnect("10.0.0.1", "5000", 850, &myHandle);
if(error.returnValue)
{
    printf("Error [%d/%d]\n", error.returnValue, error.reasonCode);
    exit(-1);
}
```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	7	DB2 BIND outstanding. The Data Collector needs BIND against DB2. See <i>IBM DB2 Performance Monitor for OS/390 Version 6 Installation and Customization</i> for how to execute PM BIND.

4. You can find this file, for example on Windows NT, in directory C:\WINNT\SYSTEM32\DRIVERS\ETC.

Return Code ² (Hex)	Reason Code ² (Hex)	Description
PM _ APIERROR	PM _ CODEPAGE _ ERROR	The specified code page could not be found or is not valid.
PM _ CONNECTION ERROR	PM _ PORTNUMBER _ TOOHIGH	The specified port number is out of range. It must be between 1 024 and 65 535.
PM _ CONNECTION ERROR	PM _ SERVICE _ UNKNOWN	The specified service name could not be found in the local <i>services</i> file.
PM _ CONNECTION ERROR	PM _ HOST _ NOTFOUND	The specified host name could not be resolved. It is not known by your Domain Name System server and your local <i>hosts</i> file.
PM _ CONNECTION ERROR	PM _ WINSOCK _ VERNOTFOUND1	Only Windows NT: The required Winsock version could not be found. Install Winsock V. 1.1 or higher.
PM _ CONNECTION ERROR	PM _ WINSOCK _ VERNOTFOUND2	Only Windows NT: The required Winsock version could not be found. Install Winsock V. 1.1 or higher.
PM _ CONNECTION ERROR	PM _ DCNOT AVAILABLE	The Data Collector is not started or the specified host port is wrong.
PM _ APIWARNING	PM _ USING _ DEFAULT _ CODEPAGE	The specified code page could not be loaded from the Data Collector. Using default code page ASCII 850.
PM _ APIWARNING	PM _ FIELDTABLE _ UNEXPECTEDEOF	The Data Collector returned an incorrect or incomplete list of supported DB2 PM counters. Some DB2 PM counters may be missing and cause some functions to fail. Check the trace for more details.
PM _ APIERROR	PM _ FIELDTABLE _ ERROR	The list of supported counters could not be loaded from the Data Collector. See the trace data and the Data Collector system log for more details. It is not possible to execute DB2 PM functions before the pmConnect() function succeeds. Retry.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Disconnect from Data Collector

Function Call

```
pmReturnCodes pmDisconnect(pmHost* handle)
```

Header File

```
pmConnect.h
```

Description

This function terminates the TCP/IP connection to the Data Collector that is identified by parameter **handle**.

Other TCP/IP connections, if any, remain open. Users logged on to the Data Collector remain logged on.

The code page and the field table that were downloaded when this connection (identified by parameter **handle**) was established are removed from the workstation's memory. Other code pages and field tables that are associated with a different handle remain in memory.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be terminated. The handle was set by the **pmConnect()** function.

This handle becomes unusable for other function calls after this function is called.

Example

```
#include "pmConnect.h"

pmHost      myHandle;
pmReturnCodes error;
...

// connect to data collector
error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
if(error.returnValue)
{
    printf("Error [%d/%d]\n", error.returnValue, error.reasonCode);
    exit(-1);
}

// do DB2 PM commands
...

// disconnect
error = pmDisconnect(&myHandle);
if(error.returnValue)
{
    printf("Error [%d/%d]\n", error.returnValue, error.reasonCode);
    exit(-1);
}
```

Return Codes and Reason Codes

No specific return codes.

Maintaining a User Session to the Data Collector

These functions are used to log on users to the Data Collector or log off from it, to generate encrypted passwords for use during the logon, and to gather version information from the Data Collector to verify its compatibility with the API.

You use these functions after connections to the Data Collector are made, or when tasks associated to a user are completed and the Data Collector resources are to be released.

Note:

- Every work profile specifying an individual SAF user ID and SAF group ID combination must be authorized during logon.
- Work profiles that specify subusers of an already authorized SAF user ID and SAF group ID combination do not need explicit authentication.
- Every work profile must be authorized on every TCP/IP connection it uses.
- Every work profile allocates resources in the Data Collector. To release these resources, log off individual users if no longer required.

- The DB2 PM API does not limit the number of subusers per SAF user.
- After the first successful connection and logon to the Data Collector, call the **pmGetInfo()** function to allow the Data Collector to detect and report a potential version mismatch to the API.

Log On to Data Collector

Function Call

```
pmReturnCodes pmLogOn (pmHost*  handle,
                       char*     workProfile,
                       char*     identification)
```

Header File

pmLogOnOff.h

Description

This function logs on an SAF user to the Data Collector. It is required before the application can call any other function for this user and all subusers.

If the application logs on a user (with an identical work profile) a second time, without intermediate logoff, the Data Collector returns a warning message (“user already logged on”).

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A 48-character string representing the work profile that identifies a user’s working environment. It contains the following fields:

- SAF User ID: 8-character field; content left aligned; padded with blanks, if required. This field is required.

The user ID of the DB2 PM user.

- SAF Group ID: 8-character field; content left aligned; padded with blanks, if required. This field is optional and must be filled with blanks, if not used.

The group ID of the DB2 PM user.

- Profile ID: 16-character field; content left aligned; padded with blanks, if required. This field is optional and must be filled with blanks, if not used. Only alphanumeric characters in the range from A to Z and 0 to 9 are allowed to specify an SAF group ID.

An application-specific or user-specific field that can be used, for example, to specify subusers.

This field is not used to authenticate subusers; only the SAF user is authenticated.

- Terminal ID: 16-character field. Reserved. Fill this field with blanks.

3. **identification** (input)

This character string represents the password, or the PassTicket generated by **pmGenPassticket()**. The OS/390 Security Server (RACF) uses this string to authenticate the SAF user. The character string must be 1 to 8 characters long.

If the OS/390 Security Server (RACF) is used, **identification** must be a RACF password or a RACF PassTicket.

Note: Be aware about the limited security when using a password with the **identification** parameter. Use a PassTicket, if possible. Refer to “The Security Concept” on page 5.

Example

```
#include "pmConnect.h"
#include "pmLogOnOff.h"

pmHost myHandle;
char workProfile[] = "PMUSER DB2PM 10.0.0.1:0001 ";
// <-UID -><-GID -><- Profile ID -><-Terminal ID->

// connect to data collector
...

// log on with password
error = pmLogOn(&myHandle, workProfile, "TEST");
if(error.returnCode)
{
    printf("Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
}
```

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
4	5	Information: Logon is successful. The SAF user is already logged on to the Data Collector through a different connection.
8	1	Internal Data Collector error.
8	2	Authorization verification failed. Check the OS/390 system log for messages from the Data Collector.
8	16	Internal Data Collector error.
8	17	Maximum number of 500 users reached.
8	1C	Internal Data Collector error.
8	22	Internal Data Collector error.
8	44	Logon request failed. CAF Connect error. Verify your DB2 permissions. Check the OS/390 system log for more information.
8	45	Logon request failed. CAF Open error. Verify your DB2 permissions. Check the OS/390 system log for more information.
8	49	Session Limit exceeded. The maximum number of TCP/IP sessions, which was specified as Data Collector startup parameter, was exceeded.
8	1551	After successful client logon, the Data Collector received a second logon request.
See also “Common Return Codes and Reason Codes” on page 10, if required.		

Generate RACF PassTicket

Function Call

```
pmReturnCodes pmGenPassticket (pmHost      *handle,  
                                char         *userID,  
                                char         *application,  
                                char         *secureSignonKey,  
                                char         *passticket)
```

Header File

pmGenPassticket.h

Description

This function generates a RACF PassTicket. Use it after a **pmConnect()** and before a **pmLogOn()** function call. You can use a PassTicket only once. Once generated, it remains usable for 10 minutes.

PassTickets are unique. The Data Collector ensures that no identical PassTickets can be build.

Note: Note that the letter T is in uppercase in the RACF term "PassTicket". As a function call you must spell it "pmPassticket" with a lowercase t because function names are case sensitive.

The use of the **pmGenPassticket()** function requires that:

- The OS/390 Security Server (RACF) is set up up recognize the name of the application using the **pmGenPassticket()** function. Also, a secure signon key must be associated with the application name. See "Preparations for Using PassTickets" on page 10 for details.
- The application can access the secure signon key on the workstation. For information on how to define a secure signon key and the RACF PassTicket mechanism see *OS/390 Security Server (RACF) - Introduction*.

Parameters

1. handle (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. userID (input)

The RACF user ID. It must be 1 to 8 characters long.

3. application (input)

The name of the application as specified in the OS/390 Security Server (RACF). Use MVSDB2PM as the application name. This name is required for **pmGenPassticket()**.

4. secureSignonKey (input)

The secure signon key associated with the name of the application in RACF. This key must be 16 characters long.

5. passticket (output)

An 8-character string storing the generated PassTicket.

Example

```
#include "pmConnect.h"  
#include "pmLogOnOff.h"  
#include "pmGenPassticket.h"
```

```

pmHost  myHandle;
char    passticket[9];
char    workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001 ";

// connect to data collector
...

// generate passticket
error = pmGenPassticket(&myHandle, "PMUSER", "MVSDB2PM",
                        "E001033FAF00007B", passticket);

if(error.returnValue)
    ...

// log on with password
error = pmLogOn(&myHandle, workProfile, passticket);
if(error.returnValue)
{
    printf("Error [%d/%d]\n", error.returnValue, error.reasonCode);
    exit(-1);
}

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	11	Internal Data Collector error.
8	1B	Internal Data Collector error.
8	22	Internal Data Collector error.
PM _ APIERROR	PM _ DATASTREAM _ INVALID	Received data stream not valid. Counter not found, or end of data reached before expected.
PM _ APIWARNING	PM _ OLD _ DC	The Data Collector is not up to date. The Data Collector interface changed meanwhile. Therefore some API functions might fail. Install the latest Data Collector PTF.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	One of the specified parameters is incorrect. Check for correct length of the parameters.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Get Data Collector Information

Function Call

```

pmReturnCodes pmGetInfo (pmHost*    handle,
                        char*       workProfile,
                        pmHashTable result)

```

Header File

pmGetInfo.h

Description

This function requests the Data Collector to return information about itself, about the DB2 PM API in use, and various other information, including:

- DB2 version number and release number
- Data Collector version number and release number
- DB2 PM API version number and release number

- Trial period status
- Time information
- Information about users logged on to the Data Collector.

This function also verifies whether the Data Collector release is sufficient for the DB2 PM API release on the workstation. If it is not, the Data Collector returns a warning, and some API functions might fail. Therefore, it should be used after a successful connection to the Data Collector to allow the Data Collector to return a warning, if necessary. Notice that this function does not detect a down-level DB2 PM API.

You can use this function before any user is logged on to the Data Collector. This allows, for example, to check whether a specified user is logged on.

This function requires that the output data area is initialized with the **clearHashTable()** function.

Parameters

1. **handle** (input)
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)
A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.
If you want to issue this function before a user is logged on to the Data Collector, or if you do not want to associate a work profile to this function call, specify NULL.
3. **info** (output)
Pointer to the output data area. See "Working with Returned Data" on page 37 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

5. For practical reasons within the scope of this guide universal time (UT) is used synonymously with coordinated universal time (UTC) and Greenwich mean time (GMT).

6. APPC sessions with the Data Collector may be started from another program. Therefore, the information is returned as well.

7. The information is returned here because Collect Report Data might be managed in the Data Collector from the DB2 PM Workstation Online Monitor for this SAF user ID and SAF group ID combination.

result	REPDCINF	Groups all Data Collector information counters.	
		QR4TID	Time difference between the local time of the host processor and the universal time (UT). ⁵ Because DB2 and the Data Collector use universal time (UT) internally, this time difference must be used to adjust counter values that represent date and time information back to local times. See “Converting and Adjusting Dates and Times” on page 94 for details.
		QR4DB2	DB2 version
		QR4PM	Data Collector version
		QR4PMI	Data Collector release
		QR4TBS	DB2 PM license status and the installation status, as follows: <ul style="list-style-type: none"> • PAYED indicates that the DB2 PM installation is a licensed version. • TRIAL indicates that the DB2 PM installation is a trial version. • RESTRICTED indicates that the trial period is exceeded. • INSTALLERROR indicates that the DB2 PM Buy feature is not installed correctly.
		QR4TBD	Number of days left until the trial period expires. If the trial period has expired, this value is 0.
		QR4TIME	Current host processor time, expressed as universal time (UT).
		QR4APIV	DB2 PM API version
		QR4APIR	DB2 PM API release
		REPUIDS	Groups user information. Contains one repeating block per SAF user ID and SAF group ID combination.
		QR4UID	The SAF user ID
		QR4GID	The SAF group ID
		QR4SEA	Number of open APPC sessions with the Data Collector for this SAF user ID and SAF group ID combination. ⁶
		QR4SET	Number of open TCP/IP sessions with the Data Collector for this SAF user ID and SAF group ID combination.
QR4CRDT	Collect Report Data status. ⁷		
QR4CRDR	Collect Report Data status. ⁷		
QR4CRDP	Collect Report Data status. ⁷		

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "pmConnect.h"
```

```

#include "pmGetInfo.h"
#include "pmTrace.h"

void printCounter(pmHashTable table, char *name);

int main(void)
{
    pmHost      myHandle;
    pmReturnCodes error;
    char        *workprofile = "PMUSER  GROUPID  PROFILEID      TERMINALID      ";
    pmHashTable result;
    pmCounter*  aCounter;
    pmCursor    aCursor;
    pmHashTable* DCInfo;
    pmHashTable* UserInfo;
    time_t      time;

    /* connect to data collector */
    error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
    if(error.returnCode)
    {
        printf("pmConnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
        exit(-1);
    }

    /* prepare result area */
    clearHashTable(&result);

    /* get information about data collector and API */
    error = pmGetInfo(&myHandle, workprofile, result);

    /* check for sufficient data collector release */
    if(error.returnCode == PM_APIWARNING && error.reasonCode == PM_OLD_DC)
    {
        printf("Be aware that you are using an old DC version.\n");
        printf("Some API functions might fail!\n");
        error.returnCode = 0;
    }

    if(error.returnCode)
    {
        printf("pmGetInfo - Error [%d/%d]\n", error.returnCode, error.reasonCode);
        exit(-1);
    }
    else
    {
        /* first locate repeating block for DC info */
        aCounter = pmGetCounter(result, "REPDCINF");
        if(aCounter != NULL)
        {
            /* access general info */
            aCursor = initCursor(*aCounter);
            DCInfo = getRepBlockItem(aCursor);
            printf("Time difference:          "); printCounter(*DCInfo, "QR4TID");
            printf("DB2 Version:                "); printCounter(*DCInfo, "QR4DB2");
            printf("PM Version:                  "); printCounter(*DCInfo, "QR4PM");
            printf("PM Version (internal):       "); printCounter(*DCInfo, "QR4PMI");
            printf("Time Bomb Status:           "); printCounter(*DCInfo, "QR4TBS");
            printf("Time Bomb Days Left:        "); printCounter(*DCInfo, "QR4TBD");
            printf("Time at host:                "); printCounter(*DCInfo, "QR4TIME");

            /* get the logged on users block */
            aCounter = pmGetCounter(*DCInfo, "REPUIDS");
            if(aCounter != NULL)
            {
                aCursor = initCursor(*aCounter);
                while(!endOfBlock(aCursor))

```

```

    {
        UserInfo = getRepBlockItem(aCursor);

        printf(" UserID:                "); printCounter(*UserInfo, "QR4UID");
        printf(" Group ID:                "); printCounter(*UserInfo, "QR4GID");
        printf(" APPC sessions:              "); printCounter(*UserInfo, "QR4SEA");
        printf(" TCP/IP sessions:             "); printCounter(*UserInfo, "QR4SET");
        printf(" Trace collectors total:      "); printCounter(*UserInfo, "QR4CRDT");
        printf(" Trace collectors running:    "); printCounter(*UserInfo, "QR4CRDR");
        printf(" Trace collectors pending:   "); printCounter(*UserInfo, "QR4CRDP");

        setToNext(&aCursor);
    }
}
/* free memory for DC info */
freeHashTable(result);
}

/* disconnect */
error = pmDisconnect(&myHandle);
if(error.returnCode)
{
    printf("pmDisconnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
}

return(0);
}

```

```

void printCounter(pmHashTable table, char *name)
{
    pmCounter *aCounter = pmGetCounter (table, name);
    time_t t;
    char *asHex;
    unsigned int helpI;
    unsigned short helpS;

    if(aCounter != NULL)                                /* if not contained */
                                                         /* NULL is returned */
    {
        /* check if value is provided by Data Collector */
        switch(aCounter->attribute)
        {
            case VALUE:
                /* print counter value */
                if(aCounter->type == PMINT)
                {
                    helpI = *((unsigned int*)aCounter->value);
                    printf("%d\n", helpI);
                }
                else if(aCounter->type == PMSHORT)
                {
                    helpS = *((unsigned short*)aCounter->value);
                    printf("%d\n", helpS);
                }
                else if(aCounter->type == PMSTRING || aCounter->type == PMVARCHAR)
                    printf("%s\n", aCounter->value);
                else if(aCounter->type == PMTIME)
                {
                    printf("PMTIME(%d bytes)\n",aCounter->length);
                }
                else if(aCounter->type == PMDATE)
                {
                    tod2time(aCounter->value, &t);
                    printf("%s", asctime(gmtime(&t)));
                }
            }
        }
    }
}

```



```

    }
    else if(aCounter->type == PMBIN)
    {
        printf("PMBIN(%d bytes)\n", aCounter->length);
    }
    else
        printf("invalid type!\n");
        break;
case NC:
    printf("n/c\n");
    break;
case NP:
    printf("n/p\n");
    break;
case NA:
    printf("n/a\n");
    break;
default:
    /* error */
    printf("invalid attribute!\n");
    break;
}
}
else
{
    printf("not returned!\n");
}
}

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	11	Internal Data Collector error.
8	1B	Internal Data Collector error.
8	22	Internal Data Collector error.
PM _ APIERROR	PM _ DATASTREAM _ INVALID	Received data stream not valid. Counter not found, or end of data reached before expected.
PM _ APIWARNING	PM _ OLD _ DC	The Data Collector is not up to date. The Data Collector interface changed meanwhile. Therefore some API functions might fail. Install the latest Data Collector PTF.

See also "Common Return Codes and Reason Codes" on page 10, if required.

Log Off from Data Collector

Function Call

```
pmReturnCodes pmLogOff (pmHost*   handle,
                       char*      workProfile)
```

Header File

pmLogOnOff.h

Description

This function logs off an SAF user or an individual subuser from the Data Collector. It stops exception processing and clears all snapshot stores and user-specific data in the Data Collector for the specified user.

You can use this function to log off an SAF user and all of its subusers, or individual subusers only.

If an individual subuser is logged off, the logoff applies to all active TCP/IP connections.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed.

If you want to log off an SAF user and all associated subusers, fill the Profile ID field in **workProfile** with blanks.

If you want to log off an individual subuser, identify the subuser in the Profile ID field in **workProfile**.

Example

```
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"

pmHost myHandle;
char workProfile[] = "PMUSER DB2PM 10.0.0.1:0001 ";
char safUserProfile[] = "PMUSER ";

// connect to data collector
...

// log on with password
...

// log off from data collector for subuser DB2PM of user PMUSER
error = pmLogOff(&myHandle, workProfile);

// log off from data collector for user PMUSER and all subusers
error = pmLogOff(&myHandle, safUserProfile);
```

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
4	4	Logoff already running for the user specified by the work profile.
8	1	Internal Data Collector error.
8	11	Internal Data Collector error.
8	1F	User not logged on, or SAF user ID or SAF group ID not found.
8	22	Internal Data Collector error.
8	155E	Subuser not logged on.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Getting DB2 Performance Data

This section introduces the concept of snapshot processing and history processing, explains how to interpret returned data, and describes the API functions relevant to DB2 performance data.

Introduction to Counters and Snapshot Stores

The functions described in this chapter are the core DB2 PM API functions. With these functions the application accesses DB2 statistics data on a subsystem level or on a thread level and DB2 system parameter values.

DB2 performance data are available through the Data Collector from two sources:

- From real-time DB2 instrumentation data. Whenever the application requests data, it gets a snapshot of the present state of the DB2 subsystem performance.
- From the *history data set* where the Data Collector continuously records snapshots of DB2 performance data. This gives the application access to older performance snapshots.

The Data Collector provides counter arrangements that ease the application's job to deal with DB2 performance data. The following sections describe the DB2 PM counter concepts. A thorough understanding of these concepts is required to effectively use these API functions.

DB2 PM Counters

DB2 PM counters are the core element of these functions. Counters are data fields that contain DB2 performance data. All fields listed in the field table represent counters.

Counters can contain various information:

- Numeric information, for example, the number of specific DB2 events that occurred, or the length of a data field.
- Alphabetic information, for example, a text string that represents the status of an activity.

Depending on the content of a counter, every counter is associated with a data type. For example, if a counter holds a time stamp of an event, it is associated with a data type of "TIME". The field table lists the data type with all counters.

The application addresses counters by names. Even if the field table lists counters by identifiers, there is no need to address a counter by its identifier.

Counters reside in the Data Collector. The Data Collector ensures that the counters are updated with actual DB2 instrumentation data. The application usually examines the counter values for further processing. It does not change counter values.

Counters have different operational behavior:

- Some counters count DB2 activities that occurred since the DB2 subsystem started, for example, the number of checkpoints that DB2 took since startup.
- Some counters hold high water marks, for example, to show the greatest amount of storage that is allocated to a process since it was started. They do not actually count. They might not even change.
- Some counters hold percentage values, for example, to indicate the latest usage of a DB2 buffer pool.

All are generally called counters in this context.

Snapshot Stores

You are rarely interested in a single counter or all available counters. Depending on the task the application wants to perform, you are interested in a defined subset of the available counters. The group of counters you are interested in is called a counter *store*. Because these counters contain data from a snapshot of performance data, the store is called a *snapshot store*. The performance-related API functions use this snapshot store when they request new counter values (take a snapshot) from the Data Collector. Snapshot stores also limit the amount of central storage resources that need to be allocated in the Data Collector.

When you specify the content of a snapshot store, you can mix any of the counters listed in the field table, regardless of the purpose of a counter. You can specify several snapshot stores in the application. Once a snapshot store is initialized, it is given a unique identifier number. The identifier serves as a reference for other functions that work with the same snapshot store.

Qualifying Counters: After you have decided which counters you want to include in a snapshot store, you can apply criteria to these counters. The Data Collector returns counter values only if these criteria are met. The process of specifying the criteria is called *qualifying*. “How Qualifying Works” on page 35 shows a comprehensive example.

Counters are qualified when a snapshot store is initialized. After you have decided which counters to include in a snapshot store, and which qualification criteria to apply to them, the definition of a snapshot store cannot be changed. The definition remains unchanged until the snapshot store is released, or until the user owning this snapshot store is logged off from the Data Collector.

Getting and Viewing Counter Values: Snapshot stores in the Data Collector can be used in two different modes:

- The GET mode causes the Data Collector to refresh the contents of the counters in a snapshot store with DB2 instrumentation data *before* it returns the counter values to the application.
You use the GET mode, for example, to frequently scan the contents of a set of counters that make up this snapshot store.
- The VIEW mode causes the Data Collector to return the contents of the counters in a snapshot store *without* refreshing them with DB2 instrumentation data.
Before useful data can be retrieved in VIEW mode, the counters in this snapshot store must have been refreshed with DB2 instrumentation data at least once.
You use the View mode, for example, if you need to access counter values that were taken at a certain point in time multiple times.

Both modes are used in combination with the two-buffer concept described next.

The mode is controlled by parameter **mode** of the **pmGetSnapshot()** function. You can see the GET and VIEW modes also as a keyword-controlled toggle flag that controls whether the snapshot store is refreshed before its content is returned to the application.

Buffers are Associated with Counters: Whenever the application deals with counters that represent DB2 performance data, it also deals with time. The application often compares the value of a counter at two given points in time. To

ease your programming effort a snapshot store supports simple calculations. This is accomplished by associating two buffers with each counter in a snapshot store.

- The buffer called **Latest** always contains the most recent counter value.
In GET mode this is the newest DB2 instrumentation data (counter values are refreshed before the results are returned).
In VIEW mode this is the last value that was stored while in GET mode (counter values are not refreshed in VIEW mode).
- The buffer called **Stored** serves as a reference value for the calculations described next. It can contain:
 - A previous value of buffer **Latest** to serve as a moving reference value.
 - The most recent value of buffer **Latest** to serve as a fixed reference value (set by the **pmReset()** function).

Buffers are Used for Interval Processing and Delta Processing: The two-buffer concept enables time-based processing of counter values in different ways. In the simplest way the content of buffer **Latest** can be requested for further processing without reference to buffer **Stored**.

- Buffer **Stored** can be set as a *fixed* reference point (by the **pmReset()** function) and subsequent **pmGetSnapshot()** functions return the cumulative differences between the content of buffer **Stored** and the content of buffer **Latest**. This is called *interval processing*.
In GET mode the cumulative difference increases (supposed that the refreshed counter value in buffer **Latest** increases).
In VIEW mode the cumulative difference remains unchanged (because no counter is refreshed).
- Buffer **Stored** can be set as a *moving* reference point. Each time a **pmGetSnapshot()** function is called in GET mode it:
 1. Copies the current content of buffer **Latest** to buffer **Stored**
 2. Refreshes the buffer **Latest** with the latest DB2 instrumentation data counter value
 3. Returns the difference between both buffer contents.This is called *delta processing*.
In VIEW mode the difference remains unchanged because no counter is refreshed.

Again, parameter **mode** of the **pmGetSnapshot()** function controls whether you want to request buffer **Latest**, perform interval processing, or perform delta processing with a snapshot store. The keywords are LATEST, INTERVAL, and DELTA.

More about interval and delta processing is described in “Working with Snapshot Stores” on page 33.

Possible Modes to Control the Snapshot Store Operation: **pmGetSnapshot()** is the function that requests counter values from a snapshot store. It uses the described modes and buffers. Parameter **mode** of **pmGetSnapshot()** controls:

- Whether the contents of counters are requested in GET mode or VIEW mode.
- Which result the function returns (buffer **Latest**, the interval result, or the delta result).

The following list shows valid **mode** parameters of the **pmGetSnapshot()** function and summarizes their counter operations. You must specify one of these modes with each function call.

Note that the described activities affect *all* qualified counters in a snapshot store.

- GET_LATEST
 1. Does not copy the content of buffer **Latest** to buffer **Stored**.
 2. Refreshes buffer **Latest** with performance data.
 3. Returns the content of buffer **Latest**.
- VIEW_LATEST
 1. Does not copy the content of buffer **Latest** to buffer **Stored**.
 2. Does not refresh buffer **Latest** with performance data.
 3. Returns the content of buffer **Latest**.
- GET_INTERVAL
 1. Leaves the buffer **Stored** unchanged.
 2. Refreshes buffer **Latest** with performance data
 3. Calculates the difference between the values in buffers **Latest** and **Stored**.
 4. Returns the difference.

This mode should be applied only to statistics counters.

- VIEW_INTERVAL
 1. Leaves the buffer **Stored** unchanged.
 2. Does not refresh buffer **Latest** with performance data.
 3. Calculates the difference between the values in buffers **Latest** and **Stored**.
 4. Returns the difference.

This mode should be applied only to statistics counters.⁸

- GET_DELTA
 1. Copies the content of buffer **Latest** to buffer **Stored**.
 2. Refreshes buffer **Latest** with performance data.
 3. Calculates the difference between the values in buffers **Latest** and **Stored**.
 4. Returns the difference.

This mode should be applied only to statistics counters.⁸

- VIEW_DELTA
 1. Leaves the buffer **Stored** unchanged.
 2. Does not refresh buffer **Latest** with performance data.
 3. Calculates the difference between the values in buffers **Latest** and **Stored**.
 4. Returns the difference.

This mode should be applied only to statistics counters.⁸

You need to specify one of these keywords as **mode** parameter of the **pmGetSnapshot()** function. Additional keywords (described next) allow you to refine the operation of a snapshot store. Valid combinations of mode parameters are described in “Snapshot Processing - Get Snapshot Data” on page 46.

Mode to Control the Data Source: Snapshot stores can get their data either from DB2 as real-time performance data, or from the history data set, which contains

⁸ If this mode is applied to other counter types, no calculations are performed and the content of buffer **Latest** is returned.

snapshots of DB2 performance data that were taken previously. If you use a snapshot store in GET mode (performance data is refreshed before it is returned to the application), you can choose whether you want it refreshed with actual DB2 performance data, or with data from a snapshot of the history data set. In the latter case, the snapshot to be used is identified by its time stamp.

You can control whether data should be taken from the history data set by adding the GET_HISTORY keyword to the **mode** parameter of the **pmGetSnapshot()** function. GET_HISTORY works as a flag. If specified as an additional **mode** parameter, data is taken from the history data set. If it is not specified, data is taken from actual DB2 performance data.

The GET_HISTORY keyword has no effect if you use a snapshot store in VIEW mode, because in VIEW mode the contents of the counters in a snapshot store are not refreshed.

Mode to Retrieve Counters about Locked Resources: The **mode** parameter of **pmGetSnapshot()** also allows you to retrieve counter values about locked resources (IFCID 150). You can specify mode GET_LOCKEDRESOURCES together with modes GET_LATEST or VIEW_LATEST. You can also combine this with mode GET_HISTORY to control whether actual DB2 performance data or data from a selected snapshot from the history data set should be taken.

If specified together with GET_LATEST, buffer **Latest** is refreshed with latest performance data (or data from the history data set), and the content of buffer **Latest** is returned.

If specified together with VIEW_LATEST, buffer **Latest** is not refreshed, and the content of buffer **Latest** is returned. Use this combination if you want to access the counter values more than once, without having them changed.

Mode to Summarize Thread Counter Values: The **mode** parameter of **pmGetSnapshot()** also allows you to summarize thread counter values. If you specify GET_SUMMARY as an additional keyword, the **pmGetSnapshot()** function returns the sum of the requested counter values. Generally, this mode is applicable only to thread counters. If unsupported thread counters are specified, they are ignored.

You can specify GET_SUMMARY together with the modes listed in “Possible Modes to Control the Snapshot Store Operation” on page 31. You can also combine this mode with mode GET_HISTORY to control whether actual DB2 performance data or data from a selected snapshot from the history data set should be taken.

Working with Snapshot Stores

This section gets together the elements around counters and snapshot processing, and gives some practical examples.

Functions Involved: The **pmInitializeStore()** function creates a snapshot store. You specify the counters that this snapshot store should contain, and you can specify a qualifier list to apply criteria to these counters.

The **pmGetSnapshot()** function request counter information from the Data Collector. Its **mode** parameter controls whether you want to only view the contents of the counters, or have the Data Collector to refresh them before the results are returned. Optionally, it controls whether you want the refresh to be done from a

selected snapshot of the history data set. Parameter **mode** also controls which calculations you want the Data Collector to perform.

The **pmReset()** function sets a reference point that subsequent **pmGetSnapshot()** function calls in mode `GET_INTERVAL` use to calculate time differences.

The **pmQueryStores()** function queries which snapshot stores are active. You can use it, for example, if you want to know when a snapshot store was refreshed with actual DB2 performance data for the last time.

The **pmGetHistoryContents()** function retrieves a list of snapshots from the history data set. The list can include all snapshots, or a specified subset of snapshots, and includes the IFCIDs and time stamps of the selected snapshots. Use this function to identify a snapshot that you want to use with the **pmGetSnapshot()** function in `GET_HISTORY` mode for a refresh. The snapshot is identified by its time stamp.

The **pmReleaseStore()** function releases a previously created snapshot store.

Counter Arithmetic: You could use snapshot stores to return their counter values and let the application execute any arithmetic required. To ease your programming effort you can also direct the **pmGetSnapshot()** function to perform some simple calculations and to return the result of this calculation.

- Delta processing

If you want to know the difference between the content of a counter at two points in time, use the **pmGetSnapshot()** function in mode `GET_DELTA`.

Whenever the function is called, the difference between the content of buffer **Latest** and buffer **Stored** is calculated, and the result is returned.

Example: You want to periodically examine database read access that was delayed because of unavailable system resources. The corresponding DB2 counter increments with each event since the DB2 subsystem is started. The latest counter value is 1 200. The application calls the **pmGetSnapshot()** function in mode `GET_DELTA` every five minutes. Assuming that the next absolute counter values are 1 205, 1 212, and 1 220, the function returns values of 5, 7, and 8 each time it is called.

- Interval processing

If you want to know the *accumulated* difference between the content of a counter at two points in time, use the **pmGetSnapshot()** function in mode `GET_INTERVAL` in combination with the **pmReset()** function.

First, issue a **pmReset()** function call. This refreshes the buffer **Stored** with the latest performance data (either real-time performance data, or data from the history data set). The content of this buffer then serves as a reference value for the following calculations.

Next, with every subsequent **pmGetSnapshot()** function call in mode `GET_INTERVAL` the difference between the content of buffer **Stored** and buffer **Latest** is calculated, and the result is returned.

Example: You want to know how the number of successful Create Thread requests develops between 8:00 a.m. and 9:00 a.m., in intervals of 60 seconds. At 8:00 a.m. the application issues a **pmReset()** function call, which copies the latest counter value of 500 to buffer **Stored** as a reference value. Then the application calls the **pmGetSnapshot()** function every 60 seconds. Assuming that the next four absolute counter values are 550, 600, 660, and 750, subsequent **pmGetSnapshot()** function calls return values of 50, 100, 160, and 250.

Interval Processing with Mixed Data Sources: Interval processing requires that you use the **pmReset()** function to set a fixed reference point before you use the **pmGetSnapshot()** function. If you want the latter function to use a snapshot from the history data set, you use the GET_HISTORY keyword as a mode parameter.

The **pmReset()** function has its own **mode** parameter to control the data source. If set to GET_DB2, actual DB2 performance data is used. If set to GET_HISTORY, a selected snapshot from the history data set is used to set a fixed reference point for interval processing.

For interval processing you can use any combination of data sources for the **pmReset()** and **pmGetSnapshot()** function. However, the snapshot used to set the reference point with the **pmReset()** function should be older than the oldest expected snapshot from a **pmGetSnapshot()** function call. Meaningful combinations are:

- Both, the **pmReset()** and the **pmGetSnapshot()** function, use real-time DB2 performance data. If both functions are called in the proper sequence, the reference point is older.
- Both, the **pmReset()** and the **pmGetSnapshot()** function, use snapshot data from the history data set. Your application needs to ensure that **pmGetSnapshot()** does not select a snapshot that is older than the one selected for **pmReset()**.
- The **pmReset()** function uses a snapshot from the history data set to set the reference point, and the **pmGetSnapshot()** function uses real-time DB2 performance data. Here, the reference point is always older.

How Qualifying Works: Qualifying allows you to apply criteria to counter stores. This frees the application from filtering counter values in which you are not interested.

You specify the criteria for a snapshot store by setting up a *qualifier list*, which contains one or more list entries. Each list entry specifies one criteria.

The qualifier list is applied to a snapshot store at the time the store is initialized with the **pmInitializeStore()** function. Qualification criteria cannot be changed after the store is initialized. If subsequent **pmGetSnapshot()** function calls request data, only those data is returned that matches all criteria.

A qualifier list can also be applied to the **pmGetHistory()** function. This function retrieves snapshot data from the Data Collector's history data set. If a qualifier list is applied, only those data is retrieved that matches all criteria.

An entry in the qualifier list uses a **qualification ID** that defines the *type of criteria* you can specify. The field table shows the available qualification IDs in section "Qualification IDs". Each qualification ID gets a value assigned that specifies the criteria value.

For example, you use the qualification ID WQALPLAN if you want to specify a thread by its plan name. You assign this qualification ID a value of DGOPMDC. If this is an entry in the qualifier list used when you initialize a snapshot store, only counter values are returned from thread with plan name DGOPMDC.

The following example shows how a qualifier list is created. First, the qualifier list is initialized using the **initQualifierList()** function. Then, the **addQualifier()** function is used to add two list entries. The third and fourth parameter of the **addQualifier()** function represent the qualification ID and the corresponding value.

```

#include "pmGetStatThread.h"
#include "pmTypes.h"          // for pmReturnCodes

// the qualifier list
pmQualifierList qualifier;
pmReturnCodes error;
unsigned char[] lvw = {0x00,0x01,0x02};

// initialize qualifier list
(&qualifier);

// for each qualifier add counter name and value
// o value must be exactly in size like counter length specified
//   in Field Table
// o functions returns [PM_APIERROR/PM_COUNTERTYPE_NOT_ALLOWED]
//   if counter type is not allowed
// o functions returns [PM_APIERROR/PM_COUNTER_TOO_LONG] if
//   counter is string and string longer than Length in Field Table
// o function returns [PM_APIERROR/PM_UNKNOWN_COUNTER] if counter name
//   is unknown
error = addQualifier(&myHandle, &qualifier,"WQALPLAN","DGOPMDC");
error = addQualifier(&myHandle, &qualifier,"WQALLUWI",lvw);

```

Note that qualification IDs of type VARCHAR are not supported. See the column “Field Types” in the field table for applicable types.

See “Miscellaneous API Functions” on page 99 for details about the **initQualifierList()** and **addQualifier()** functions.

More information about qualification can be found in the *DB2 for OS/390 Administration Guide*. It includes a section about “Programming for the Instrumentation Facility Interface (IFI)”.

Working with the History Data Set

A history data set is a collection of performance snapshots that the Data Collector collects when it is operating. You use the **pmGetHistory()** function to retrieve stored snapshots from the history data set. As with snapshot processing, you can specify the counters for which you want to retrieve data, and you can qualify the counters to apply criteria.

If the application accesses performance data in a history data set, you should bear in mind the characteristics of a history data set and the method to access individual snapshots.

Characteristics of a History Data Set: Several parameters about the history data set are specified as Data Collector startup parameters. The following parameters influence the content and amount of snapshot data:

- The size of the history data set. The data set is used as a wrap-around data set. When new snapshots are added, the oldest snapshots disappear after the data set space fills up.
- The content of the data set. Generally, thread counters, statistic counters, and DB2 system parameters are saved in the data set. However, only snapshots of IFCIDs that are specified as Data Collector startup parameters and qualified thread data are written to the data set.
- The frequency at which snapshots are taken and written to the data set.

Therefore, you may not find all counters in the data set. If necessary, you need to change relevant Data Collector startup parameters. See the *IBM DB2 Performance Monitor for OS/390 Command Reference* for more details.

Function Involved: The `pmGetHistory()` function retrieves a stored snapshot from the history data set. You can select snapshots in different ways:

- By specifying a point in time
- By stepping forward and backward in the history data set.

Selecting Individual Snapshots: Every snapshot recorded in the history data set is given a time stamp when the snapshot is taken. To select individual snapshots the application program needs to know at least the time stamp of the snapshot. Once this snapshot is identified, the application program can step forward or backward in the history data set. The `pmGetHistory()` function has two parameters (`requestTime` and `dir`). They control the selection of an individual snapshot and the change of direction in the history data set. The following scenarios show how the `pmGetHistory()` function and both parameters are used:

The first scenario assumes that you want to start with a snapshot record that is younger than t_1 , which might be somewhere in the middle of the history data set:

1. The application calls the `pmGetHistory()` function with parameter `requestTime` set to t_1 and parameter `dir` set to `T0`.

This function call locates a snapshot next to (`dir` set to `T0`) time t_1 . The API chooses the younger snapshot per definition.

The function returns the requested counter data, and returns the time stamp of this snapshot.

2. Every subsequent `pmGetHistory()` function call takes the time stamp of a preceding function call as a reference to locate a record adjacent to this point in time. You select the direction (the older or the younger snapshot adjacent to this point in time) with parameter `dir` set to `BACK` or `FORWARD`.

For example, assume that the first `pmGetHistory()` function call returned a snapshot with a time stamp of t_2 . You want to locate the next older snapshot in the history data set. Then, the application would call the next `pmGetHistory()` function with parameter `requestTime` set to t_2 , and parameter `dir` set to `BACK`.

The second scenario assumes that you want to start from either the beginning or the ending of the history data set.

- To start from the beginning (the oldest snapshot in the history data set), set parameter `requestTime` to `TOD_FIRST` and parameter `dir` to `FORWARD`.
- To start from the end (the youngest snapshot in the history data set), set parameter `requestTime` to `TOD_LAST` and parameter `dir` to `BACK`.

Once you have located a snapshot at either end, use the returned time stamp to step through the history data set, as in the first scenario.

Working with Returned Data

The following functions return data streams of varying length and structure, depending on the number and type of counters requested:

- `pmGetSnapshot()`
- `pmGetHistory()`
- `pmGetHistoryContents()`
- `pmGetInfo()`

The data stream is stored in the application's memory, and the output parameter `result` of these functions points to the output data area in memory.

Some counters return a single value when requested. For example, if you want to examine the number of failures that resulted from an “EDM pool is full” condition, you are interested in the value of counter QISEFAIL (field ID 2601 in the field table). Because DB2 has only one EDM pool, the Data Collector returns only one value for this counter.

Other counters return an unknown number of values. For example, if you want to examine the number of requests made to read a page from the group buffer pool, you are interested in the value of counter QBGLXD (field ID 2702 in the field table). Because DB2 can have several active group buffer pools, the Data Collector returns several values for this counter. (In this example you probably use an additional counter that returns the buffer pool identifier.)

More complex, several counters represent a list of counter groups, whereby list members also can represent counter groups, or a list of counters. For example, counter REPSTAT contains all DB2 statistic counters (REPSTDDF, REPSTBUF, REPSTGBF, and so on), whereby counter REPSTGBF represents a list of counters (one counter for each active DB2 group buffer pool). These counters are called *repeating blocks* and are given a data type of PMPARSEDREPBLOCK.

To hide this complexity, repeating blocks are stored in hash tables. The API provides several supporting functions to locate counters in hash tables by their names and to extract counter values. See “Miscellaneous API Functions” on page 99 for details.

The Counter Structure

Every counter in the output data area is represented as a structure named *pmCounter*:

```
typedef struct _counter
{
    char*          name;          /* counter name as string          */
    unsigned short counterID;     /* counter name as ID             */
    unsigned short length;       /* length of value                */
    pmCounterType type;          /* counter type, see below        */
    char*          value;        /* points to value storage        */
    pmCounterAttr attribute;     /* indicating a valid counter value */
} pmCounter;

/* possible counter types */
typedef enum {
    PMBIN='B',          /* binary value, not converted */
    PMSTRING='C',       /* string in ASCII             */
    PMVARCHAR='V',     /* string with variable length */
    PMSHORT='S',       /* small integer (2 bytes)     */
    PMINT='I',         /* integer (4 bytes)           */
    PMTIME='T',        /* DB2 time format             */
    PMDATE='D',        /* DB2 date format             */
    PMPARSEDREPBLOCK = 'P' /* repeating block              */
} pmCounterType;

typedef enum {
    VALUE=0x40,
    NA=0xC1,
    NP=0xD7,
    NC=0xC3
} pmCounterAttr;
```

Structure *_counter* aggregates six members:

1. *name* contains the name of the counter (which is identical with the counter you have requested, respectively with the name of the counter as it is shown in the field table).
2. *counterID* contains the identifier, as shown in column “Field ID” of the field table.
3. *length* contains the length of the *value* member. The length is the same as shown in column “Field Length” of the field table.
If structure member *type* is PMVARCHAR (which represents a string with variable length), then member *length* contains the maximum length of the string. The actual length of the string is represented by the first two bytes of the string itself, whereas the actual length does not include this 2-byte prefix.
4. *type* contains the type of the counter, which is one of those defined in *pmCounterType*.
5. *value* contains a pointer to the counter value in memory. The counter value is valid if *attribute* contains VALUE. If the counter value is a character string, and the workstation uses ASCII, the string is already converted from EBCDIC to ASCII.
6. *attribute* contains an indicator showing how usable the content of member *value* is:
 - VALUE indicates a valid content.
 - NC indicates “Not Calculated”. A calculation resulted in a division by zero.
 - NP indicates “Not present”. A counter value was not set because the corresponding DB2 function was not active. For example, if you query a counter about the number of statements dropped from the dynamic statement cache, and the cache is not enabled, *attribute* contains NP.
 - NA indicates “Not Available”. A counter was not set because the counter is not available for the installed DB2 subsystem version.

Interpreting Repeating Blocks

The example in this section shows how repeating blocks are treated and how the individual counter values are retrieved from the **result** output data area.

The output data area contains a variable number of REPTHDRD repeating block items (one per active DB2 thread). The example steps through all REPTHDRD repeating block items until no further item is found.

The following table shows the hierarchy of counters within one REPTHDRD repeating block. REPTHDRD contains counters, which can again represent repeating blocks, for example, REPTHBUF. The REPTHBUF counter contains a list of counters; one for each buffer pool.

result	REPTHDRD	Groups all thread counters of one DB2 thread.		
		counter		
		REPTHBUF	counter*	Buffer pool data repeating block.
		REPTHDBG	counter*	Distributed agent data repeating block.
		REPTHDBG	counter*	Distributed accounting data repeating block.
		REPTHBDG	counter*	Group buffer pool data repeating block in thread record.
		REPTHDBG	counter*	Locked resources repeating block.

The output data stream is stored in a hash table. The following demonstrates how the hash table functions are used in the example:

```
pmCursor cursor = initCursor(pmGetCounter(result, "REPTHRD"));
pmHashTable item;

while(!endOfBlock(cursor))
{
    item = getRepBlockItem(cursor);

    // process data in this repeating block
    ...

    setToNext(cursor);
}

```

Details of these functions are described in “Miscellaneous API Functions” on page 99. With these functions you do not need to know in which sequence individual counters appear in the data stream. The **pmGetCounter()** function locates counters by their names. The **initCursor()** function initializes a cursor, which is used to parse through all items of a repeating block.

```
#include "pmGetStatThread.h"
#include "pmHashTableList.h" // for cursor functions
#include "pmHashTable.h" // for repeating block

// counters to request
char *fields[] = {
    "QWHCAID", // primary thread auth. ID
    "QBACPID", // Buffer Pool name
};

// output data area
pmHashTable result;
// counter to get values from output data area
pmCounter* aCounter;
// cursor to step through threads
pmCursor cursor1;
// cursor to step through buffer pools of one thread
pmCursor cursor2;
// pointer to counters in thread repeating block
pmHashTable* threadCounters;
// pointer to counters in BP repeating block
pmHashTable* bufferPoolCounters;
// number of threads
unsigned int threadNo = 0;

// connect, logon
...

// IMPORTANT: before output data area can be filled,
// initialize it !!!
clearHashTable(&result);

// retrieve output data area in 'result'
...

printf(">> Parsing output data area and printing counters.\n\n");

// first locate the repeating block containing the active DB2 threads
// (one block item for each thread)
aCounter = pmGetCounter(result, "REPTHRD");
if(aCounter != NULL)
{
    // step through all returned threads via cursor
    cursor1 = initCursor(*aCounter);
    while(!endOfBlock(cursor1))

```

```

{
    threadNo++;

    // now get access to counters of this thread
    threadCounters = getRepBlockItem(cursor1);

    // get first requested counter QWHCAID
    aCounter = pmGetCounter (*threadCounters, "QWHCAID");
    if(aCounter != NULL) // if not contained
                        // NULL is returned
    {
        // check if value is returned by Data Collector
        switch(aCounter->attribute)
        {
            case VALUE:
                // ok, DC returned value
                // -> print counter value
                printf(" Thread %d: QWHCAID has value %s.\n", threadNo,
                    (char*)aCounter->value);
                break;
            case NC:
            case NP:
            case NA:
                printf(" QWHCAID not calculated, present or available.\n");
                break;
            default:
                /* error */
                exit(-1);
        }
    }
    else
    {
        printf(" QWHCAID counter not returned by Data Collector!\n");
    }

    // now search repeating blocks contained in one thread
    // (buffer pools, group buffer pools, remote locations, ...)
    //-> first locate the repeating block for buffer pools
    aCounter = pmGetCounter (*threadCounters, "REPTHBUF");
    if(aCounter != NULL)
    {
        // the repeating block contains a block item for each
        // active Buffer Pool
        // -> step through all returned Buffer Pools
        cursor2 = initCursor (*aCounter);
        while (!endOfBlock(cursor2))
        {
            // now get access to each returned buffer pool
            bufferPoolCounters = getRepBlockItem(cursor2);

            aCounter = pmGetCounter (*bufferPoolCounters, "QBACPID");
            if(aCounter != NULL) // if not contained
                                // NULL is returned
            {
                // check if value is returned by Data Collector
                switch(aCounter->attribute)
                {
                    case VALUE:
                        // print counter value
                        if(*(unsigned int*)aCounter->value < 80)
                            printf(" Buffer Pool BP%d is activated.\n",
                                *(unsigned int*)aCounter->value);
                        else
                            printf(" Buffer Pool BP32K%d is activated.\n",

```

```

        *(unsigned int*)aCounter->value);
        break;
    case NC:
    case NP:
    case NA:
        printf(" Thread does not use any Buffer Pool,");
        printf(" QBACPID is set to n/p.\n");
        break;
    default:
        // error
        exit(-1);
    }
}
else
{
    printf("QBACPID counter not returned by Data Collector!\n");
}

// set cursor to next Buffer Pool in repeating block
setToNext(&cursor2);
}
}
else
{
    printf(" QBACPID counter not returned by Data Collector!\n");
}

// set cursor to next thread in repeating block
setToNext(&cursor1);
}
}

// Free memory for output data area. This frees memory for all
// stored counters too.
freeHashTable (result);

// log off and disconnect
...

```

Snapshot Processing - Initialize Snapshot Store

Function Call

```

pmReturnCodes pmInitializeStore (pmHost*      handle,
                                char*        workProfile,
                                char**       fields,
                                unsigned int  counterNo,
                                pmQualifierList* qualifier,
                                char*        userData,
                                unsigned int* id)

```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to create a snapshot store, to initialize it, and to return a snapshot store identifier.

The snapshot store **id** is used during subsequent requests to identify the store. Optionally, you can assign a 32-character name to the store.

The snapshot store remains active in the Data Collector. It is released by the **pmReleaseStore()** function, or when the user identified by parameter **workProfile** is logged off.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **fields** (input)

An array of DB2 counter names that the snapshot store should include. You can include statistics counter names and thread counter names that are listed in column "Field Name" of the field table.

Insert the counter names as follows. Note that counter names are not case-sensitive. Invalid counters are ignored.

```
char *fields[] = {
    "QISEDDB", "QTMAXDS", "QTAUCHK", "QTAUSUC", "QBSTMAX",
    "QBSTPID", "QBSTGET", "QBSTWFT", "QBSTWFD", "QBSTWKP",
    "QBSTWKP", "QBSTMAX", "QBSTWKP", "QBSTWDRP", "QBSTWBVQ",
    "QBSTWFR", "QBSTWFF", "QBSTWFD", "QISEDSE", "QISEDSE",
    "QISEDSC", "QXSTFND", "QXSTFND", "QBSTWDRP", "QBSTWBVQ"
};
```

4. **counterNo** (input)

The number of counters specified with parameter **fields**.

5. **qualifier** (input)

The name of a qualifier list. The list specifies the criteria which data the Data Collector should return. See "How Qualifying Works" on page 35, if required.

6. **userData** (input)

An optional user- or application-specific 32-character string that is assigned to this snapshot store. This string is returned by the **pmQueryStores()** function. If **userData** is not used, this parameter should be set to NULL.

7. **id** (output)

A snapshot store identifier generated by the Data Collector. This identifier is required to identify this store in subsequent snapshot function calls.

Example

The following example shows how qualifiers are specified with a **pmInitializeStore()** function call, if you want the **pmGetSnapshot()** function to return only locked resources with suspensions.

```
// the qualifier list
pmQualifierList qualifier;
pmReturnCodes error;

// initialize qualifier list
initQualifierList(&qualifier);

// disable display of single and multiple held locks
// to show only locked resources with suspensions

error = addQualifier(&myHandle, &qualifier, "TLRSINGL", " ");
error = addQualifier(&myHandle, &qualifier, "TLRMULT", " ");
```

```
// initialize snapshot store with qualification
error = pmInitializeStore(&myHandle, workprofile, fields, fieldNo,
                        &qualifier, userinfo, &store_id);
```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	22	Internal Data Collector error.
8	24	Returned data is too large. Request was canceled.
8	25	Internal Data Collector error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1554	Internal Data Collector error.
8	155B	Internal Data Collector error.
8	155C	Internal Data Collector error.
PM _ APIERROR	PM _ INPUTAREA _ OVERFLOW	Too many counters qualified or requested. Request input area overflow. Divide the snapshot store into two or more stores.
PM _ APIERROR	PM _ INPUTAREA _ OVERFLOW	Too many counters qualified or requested. Request input area overflow. Divide the snapshot store into two or more stores.
PM _ APIERROR	PM _ INPUTAREA _ OVERFLOW	Too many counters qualified or requested. Request input area overflow. Divide the snapshot store into two or more stores.
PM _ APIERROR	PM _ INPUTAREA _ OVERFLOW	Too many counters qualified or requested. Request input area overflow. Divide the snapshot store into two or more stores.
PM _ APIERROR	PM _ DATASTREAM _ INVALID	Received data stream not valid. Counter not found, or end of data reached before expected. Verify that the latest DB2 PM PTF is installed. If you cannot solve the problem, call for IBM support.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Snapshot Processing - Query Snapshot Stores

Function Call

```
pmReturnCodes pmQueryStores (pmHost*      handle,
                             char*        workProfile,
                             pmStore**    stores)
```

Header File

```
pmGetStatThread.h
```

Description

This function requests the Data Collector to return information about all active snapshot stores for a user that is identified by parameter **workProfile**.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **stores** (output)

A pointer to a list of all active snapshot stores for this user. Every list member is represented as a structure named *pmStore*. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

```
typedef struct _store
{
    unsigned int    id;           // id of this snapshot store
    pmTOD          timeLatest;   // time of snapshot in buffer Latest
    pmTOD          timeStored;   // time of snapshot in buffer Stored
    char           userData[32]  // contains the user defined
                                // identifier
    struct _store* next;        // next snapshot store in list
} pmStore;
```

- *id* contains the snapshot store identifier. This identifier was assigned by the **pmInitializeStore()** function.

- *timeLatest* contains a time stamp, which shows when the buffer **Latest** of all counters in this store was changed.

timeLatest returns `\0\0\0\0\0\0\0\0` (8 bytes set to 0) if the counters were not used so far.

- *timeStored* contains a time stamp, which shows when the buffer **Stored** of all counters in this store was changed.

timeStored returns `\0\0\0\0\0\0\0\0` (8 bytes set to 0) if the counters were not used so far.

- *userData[32]* contains the user data (a 32-character field) that was assigned to this snapshot store by the **pmInitializeStore()** function.

- *next* contains a pointer to the next member in the list of active snapshot stores.

Example

```
#include "pmGetStatThread.h"
#include "pmTOD.h"           // for time conversions
#include "pmTrace.h"        // for asHexString()
#include <time.h>

// beginning of list with returned stores
pmStore* store;
pmStore* help;

time_t    t1,
          t2;
char      *userInfo;
pmHost    myHandle;
char      workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001  ";

// connect and log on to DC
...

// query all stores for this user
error = pmQueryStores (&myHandle, workProfile, &store);
```

```

// print infos about all active snapshot stores
while(store)
{
    // print infos
    tod2time (store->timeLatest, &t1);
    tod2time (store->timeStored, &t2);
    userInfo = asHexString (store->userData, 32);
    printf("Snapshot Store %d :\n          \
          \tUserData : %s\n          \
          \tBuffer Latest time : %s\n          \
          \tBuffer Stored time : %s\n\n",
          store->id,
          userInfo,
          ctime(&t1),
          ctime(&t2));

    pmFreeMem(userInfo);

    // set list to next item
    help = store;
    store = store->next;

    // now free memory for list item, if no longer used.
    pmFreeMem(help);
}

// log off and disconnect
...

```

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
8	1	Internal Data Collector error.
8	22	Internal Data Collector error.
8	25	Internal Data Collector error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1554	Internal Data Collector error.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Snapshot Processing - Get Snapshot Data

Function Call

```

pmReturnCodes pmGetSnapshot (pmHost*      handle,
                             char*        workProfile,
                             pmSnapshotMode mode,
                             unsigned int id,
                             char**      fields,
                             unsigned int counterNo,
                             pmTOD       timestampLatest,
                             pmTOD       timestampStored,
                             pmHashTable result)

```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to return counter data from a specified snapshot store for a user identified by parameter **workProfile**.

Parameter **mode** controls the buffer calculations and the results to be returned. It also controls whether actual DB2 performance data or data from the history data set is to be used. If the latter is to be used, the Data Collector must have been started with history recording active.

Parameter **fields** controls which counters from the specified counter store are to be returned.

This function requires that the output data area is initialized with the **clearHashTable()** function.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **mode** (input)

One or more keywords that describes how the counters in the identified snapshot store are treated before being returned. The following table summarizes the possible modes and its major characteristics. For a detailed description see "Possible Modes to Control the Snapshot Store Operation" on page 31.

You need to specify one of the keywords shown in the left column of the table. If you want to specify additional keywords, provide them as shown in the following example. Internally, the keywords are declared as numeric constants, which need to be bitwise ORed to control the program flow.

GET_LATEST | GET_HISTORY | GET_LOCKEDRESOURCES

Mode	Copies "Latest" to "Stored"	Refreshes "Latest"	Returns ...	Remark
GET_LATEST	No	Yes	"Latest"	Optionally, together with GET_HISTORY, or GET_LOCKEDRESOURCES, or both. Alternatively, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_LATEST	No	No	"Latest"	Optionally, together with GET_LOCKEDRESOURCES. Alternatively, together with GET_SUMMARY.

Mode	Copies "Latest" to "Stored"	Refreshes "Latest"	Returns ...	Remark
GET_INTERVAL	No	Yes	"Latest"– "Stored"	Preset by pmReset() . Optionally, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_INTERVAL	No	No	"Latest"– "Stored"	Optionally, together with GET_SUMMARY.
GET_DELTA	Yes	Yes	"Latest"– "Stored"	Optionally, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_DELTA	No	No	"Latest"– "Stored"	Optionally, together with GET_SUMMARY.

- The GET_HISTORY keyword selects the history data set (instead of actual DB2 performance data) as the data source to refresh the counters in a snapshot store before their values are returned. It can be used together with one of the GET modes, which refreshes buffer **Latest**.

The snapshot in the history data set is identified by parameter **timestampLatest**. See "Snapshot Processing - Get History Contents" on page 54 about how to retrieve a list of all snapshots in the history data set and how to identify a snapshot by its time stamp.

- If you specify GET_INTERVAL, you must first use the **pmReset()** function to ensure that the reference values for interval processing are set. See "Snapshot Processing - Reset Interval Data" on page 51 for more details.
If you specify GET_INTERVAL together with GET_HISTORY, ensure that you have selected an appropriate data source also for the **pmReset()** function.
- The GET_SUMMARY keyword causes the Data Collector to summarize all thread-related counters. You can combine this keyword with other mode keywords, except GET_LOCKEDRESOURCES.

4. **id** (input)

The identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStore()** function.

5. **fields** (input)

An array of counter names for which counter data is requested.

If you set **fields** to NULL, counter data is requested for all counters in this snapshot store.

If you specify a subset of counter names, counter data is requested only for the specified counters. A subset is any number of counters that was specified by the **pmInitializeStore()** function for this snapshot store **id**.

To specify a subset of counters insert the counter names as follows. Counter names are not case-sensitive. Invalid counters are ignored. Counters not available in the stored snapshot record in a history data set are ignored. No warning is returned.

```
char *fields[] = {
    "QISEDDBD", "QTMAXDS", "QTAUCHK", "QTAUSUC", "QBSTMAX",
    "QBSTPID", "QBSTGET", "QBSTWFT", "QBSTWFD", "QBSTWKP",
    "QBSTWKP", "QBSTMAX", "QBSTWKP", "QBSTWDRP", "QBSTWBVQ",
    "QBSTWFR", "QBSTWFF", "QBSTWFD", "QISEDSEI", "QISEDSEI",
    "QISEDSC", "QXSTFND", "QXSTFND", "QBSTWDRP", "QBSTWBVQ"
};
```

If you specify also thread counters in the array, and GET_SUMMARY as **mode** parameter, all thread counters in the array are summarized before being returned. See the field table, section “Thread Data”, for valid thread counter names.

6. **counterNo** (input)

The number of counters specified with parameter **fields**. If parameter **fields** is NULL, set **counterNo** to 0.

7. **timestampLatest** (output/input)

This parameter returns a time stamp, which shows when the buffer **Latest** of all requested counters in this store was changed.

If the **pmGetSnapshot()** function is used with **mode** parameter GET_HISTORY, specify a valid time stamp of a snapshot in the history data set. See parameter **mode** for more details. Input data is overwritten upon return.

8. **timestampStored** (output)

This parameter returns a time stamp, which shows when the buffer **Stored** of all requested counters in this store was changed.

The time stamp holds a valid time only if parameter **mode** is one of the following:

- GET_INTERVAL
- VIEW_INTERVAL
- GET_DELTA
- VIEW_DELTA

9. **result** (output)

Pointer to the output data area. See “Working with Returned Data” on page 37 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

If counters about locked resources are requested (with **mode** set to GET_LOCKEDRESOURCES), the output data area does not contain the REPTHCK repeating block under the REPTHDR repeating block. Instead, the following additional information is returned in the output data area:

result	REPTHCK	Groups all locked resources.	
		counter*	Resource-related IFCID 150 counters.
		REPTHDR	Thread-related IFCID 150 counters. Contains one repeating block per thread.

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
4	1	No data returned by DB2.
4	2	Output data area too small. Output data truncated.
4	3	Request resulted in a DB2 abend.
4	6	Authorization exit in the Data Collector is active and returned no data.
4	7	Severe error in authorization exit. Standard authorization checks are used.

Return Code ² (Hex)	Reason Code ² (Hex)	Description
4	E	Dynamic statement cache error. You qualified a special statement in the statement cache. The requested statement was not found in the cache.
4	F	No data available in snapshot store. You have requested to view previously saved data, or to build the delta/interval, and at least one buffer is empty. Fill buffer and retry.
8	1	Internal Data Collector error.
8	8	DB2 request failed, no data. See the console log for detailed information.
8	22	Internal Data Collector error.
8	24	Returned data too large. Request canceled. Use qualification to reduce the amount of data.
8	25	Internal Data Collector error.
8	2B	Internal Data Collector error.
8	33	The history function in the Data Collector is not started. If you want to retrieve snapshots from the history data set, ensure that the Data Collector is enabled to record snapshots in the history data set (Data Collector startup parameters).
8	35	Internal Data Collector error.
8	36	Request rejected. You specified GET_HISTORY as mode parameter, but the history data was found empty. No snapshot was yet gathered.
8	36	Request rejected. You specified GET_HISTORY as mode parameter, but the history data was found empty. No snapshot was yet gathered.
8	39	Request rejected. You requested GET_HISTORY as mode parameter and specified the time stamp of a snapshot with parameter timestampLatest that could not be found in the history data set. Use the pmGetHistoryContents() function to retrieve valid snapshot time stamps and try again.
8	39	Request rejected. You requested GET_HISTORY as mode parameter and specified the time stamp of a snapshot with parameter timestampLatest that could not be found in the history data set. Use the pmGetHistoryContents() function to retrieve valid snapshot time stamps and try again.
8	3B	Internal Data Collector error.
8	3C	Internal Data Collector error.
8	3D	Internal Data Collector error.

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	3E	Internal Data Collector error.
8	40	Data Collector Authorization exit returned error. Irrecoverable error.
8	3B	Internal Data Collector error.
8	3C	Internal Data Collector error.
8	3D	Internal Data Collector error.
8	3E	Internal Data Collector error.
8	40	Data Collector Authorization exit returned error. Irrecoverable error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	4D	You requested dynamic statement cache counters, but dynamic statement cache is not enabled. Try without these counters.
8	1554	Internal Data Collector error.
8	155A	Incorrect snapshot store ID specified.
8	155D	Incorrect snapshot store ID specified, or snapshot store is no longer available.
8	155E	Profile ID of user not found or no longer valid.
8	1560	Internal Data Collector error.
8	1561	Internal Data Collector error. Storage allocation for buffers failed.
8	1562	At least one field ID was requested that exceeds the basic scope of collected IFCIDs, as specified by the pmInitializeStore() function.
8	1563	Internal Data Collector error.
8	156C	You specified GET_DELTA or VIEW_DELTA as mode parameter, but the time stamp of the snapshot in buffer Latest is older or equal to the time stamp in buffer Stored . The delta calculation was not possible. Refresh buffer Latest with a more recent snapshot.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	One of the specified mode parameters is incorrect, or too many fields are requested (maximum is 2048). Check for correct parameters.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Snapshot Processing - Reset Interval Data

Function Call

```
pmReturnCodes pmReset (pmHost*      handle,
                       char*         workProfile,
                       pmSnapshotMode mode
                       unsigned int   id,
                       pmTOD         snapshotTime)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to preset reference values for an identified snapshot store for subsequent interval processing.

Use the **pmReset()** function before you use the **pmGetSnapshot()** function in mode GET_INTERVAL. **pmReset()** makes the buffer **Stored** a fixed reference for interval processing.

1. Buffer **Stored** is filled with latest performance data, or data from a snapshot of the history data set.
2. Buffer **Latest** is released.

Subsequent **pmGetSnapshot()** function calls in mode GET_INTERVAL now use the content of buffer **Stored** to calculate the difference to buffer **Latest**. The value in buffer **Stored** is not changed by subsequent **pmGetSnapshot()** function calls in mode GET_INTERVAL.

The **pmReset()** function is a companion to the **pmGetSnapshot()** function, which can use actual DB2 performance data, or snapshot data from the history data set, to refresh the content of buffer **Latest**. Consequently, you also need to specify the data source for the **pmReset()** function. Use parameter **mode** to specify either data source:

- If you specify GET_DB2, the **pmReset()** function uses actual DB2 performance data to fill buffer **Stored**.
- If you specify GET_HISTORY, the **pmReset()** function uses data from a selected snapshot of the history data set to fill buffer **Stored**. The snapshot in the history data set is specified by its time stamp, in the same manner as with the **pmGetSnapshot()** function.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **mode** (input)

A keyword that controls the data source to use:

- GET_DB2 uses actual DB2 performance data.
- GET_HISTORY uses snapshot data from the history data set. Specify the snapshot with parameter **snapshotTime**. See "Snapshot Processing - Get History Contents" on page 54 about how to retrieve a list of all snapshots in the history data set and how to identify a snapshot by its time stamp.

4. **id** (input)

The identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStore()** function.

5. **snapshotTime** (output/input)

If used with parameter **mode** set to GET_DB2, this parameter returns a time stamp, which shows when the buffer **Stored** was filled with latest DB2 performance data.

If used with parameter **mode** set to GET_HISTORY, specify a valid time stamp of a snapshot in the history data set that you want to use.

The time stamp is in Store Clock format. If you need a conversion to **time_t** format, see “Convert Store Clock Format to time_t Format” on page 95 for details.

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
4	1	No data returned by DB2.
4	2	Output data area too small. Output data truncated.
4	3	Request resulted in a DB2 abend. See the OS/390 system log for more information.
4	6	Authorization exit returned no data.
4	7	Severe Error in authorization exit. Standard authorization checks are used.
4	E	Dynamic statement cache error. You qualified a special statement in the statement cache. The requested statement was not found in the cache.
4	F	No data available in snapshot store. You have requested to view previously saved data, or to build the delta/interval, and at least one buffer is empty. Fill buffer and retry.
8	1	Internal Data Collector error.
8	8	DB2 request failed, no data. See the console log for detailed information.
8	22	Internal Data Collector error.
8	24	Returned amount of data too large. Request canceled. Use qualification to reduce the amount of data.
8	25	Internal Data Collector error.
8	2B	Internal Data Collector error.
8	33	The history function in the Data Collector is not started. If you want to retrieve snapshots from the history data set, ensure that the Data Collector is enabled to record snapshots in the history data set (Data Collector startup parameters).
8	35	Internal Data Collector error.
8	36	Request rejected. You specified GET_HISTORY as mode parameter, but the history data was found empty. No snapshot was yet gathered.

Return Code (Hex)	Reason Code (Hex)	Description
8	39	Request rejected. You requested GET_HISTORY as mode parameter and specified the time stamp of a snapshot with parameter snapshotTime that could not be found in the history data set. Use the pmGetHistoryContents() function to retrieve valid snapshot time stamps and try again.
8	3B	Internal Data Collector error.
8	3C	Internal Data Collector error.
8	3D	Internal Data Collector error.
8	3E	Internal Data Collector error.
8	40	Data Collector Authorization exit returned error. Irrecoverable error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	4D	You requested dynamic statement cache counters, but dynamic statement cache is not enabled. Try without these counters.
8	1554	Internal Data Collector error.
8	155A	Incorrect snapshot store ID specified.
8	155D	Incorrect snapshot store ID specified or snapshot store no longer available.
8	155E	Profile ID of user not found or no longer valid.
8	1560	Internal Data Collector error.
8	1561	Internal Data Collector error. Storage allocation for buffers failed.
8	1562	At least one field ID was requested that exceeds the basic scope of collected IFCIDs, as specified by the pmInitializeStore() function.
8	1563	Internal Data Collector error.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Snapshot Processing - Get History Contents

Function Call

```
pmReturnCodes pmGetHistoryContents (pmHost*      handle,
                                     char*        workProfile,
                                     unsigned int* ifcids,
                                     unsigned int  ifcidNo,
                                     pmTOD       timestampFrom,
                                     pmTOD       timestampTo,
                                     pmHashTable result)
```

Header File

```
pmGetStatThread.h
```

Description

This function requests the Data Collector to return a list of snapshots from the history data set for the user identified by parameter **workProfile**. The information returned includes the recorded IFCIDs per snapshot and their time stamps.

The amount of returned information can be limited by specifying a time frame, and by specifying IFCIDs for which data should be returned.

This function requires that the output data area is initialized with the **clearHashTable()** function.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **ifcids** (input)

This parameter allows you to specify IFCIDs for which data should be returned. This limits the amount of data returned.

If you do not want to specify individual IFCIDs, specify NULL. The list of snapshots returned contains all available IFCIDs.

If you want to specify individual IFCIDs, specify them as an array of unsigned integer values. The list of snapshots returned contains snapshots that contain at least one of the specified IFCIDs.

4. **ifcidNo** (input)

The number of IFCIDs specified with parameter **ifcids**, or 0, if no IFCIDs are specified.

5. **timestampFrom** (input)

This parameter allows you to specify a time stamp that limits the amount of data returned. The list of snapshots returned contains snapshots that are taken *after* this time stamp. If you also specify a time stamp for parameter **timestampTo**, keep both in timely order.

If you do not want to specify a time stamp, specify NULL. The list of snapshots returned contains all snapshots from the history data set (if not limited by the parameters **ifcids** or **timestampTo**).

6. **timestampTo** (input)

This parameter allows you to specify a time stamp that limits the amount of data returned. The list of snapshots returned contains snapshots that are taken *before* this time stamp.

If you do not want to specify a time stamp, specify NULL. The list of snapshots returned contains all snapshots from the history data set (if not limited by the parameters **ifcids** or **timestampFrom**).

7. **result** (output)

Pointer to the output data area. See "Working with Returned Data" on page 37 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

If no snapshots are in the history data set that match the selection criteria, the output data area is empty (no REPHISNP repeating block is available). The

function returns with a return code of 0.

result	REPHISNP	Groups all history data set snapshots.		
		HISTTIME	Time stamp of history data set snapshot, in Store Clock format.	
		REPHIIFI	Groups all IFCIDs collected at HISTTIME. Contains one repeating block per HISTTIME.	
			HISTIFI	IFCID collected at HISTTIME.

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	24	Returned data too large. Request canceled. Try to limit the amount of returned data by changing some of the input parameters.
8	33	History function not started in the Data Collector.
8	22	Internal Data Collector error.
8	51	Internal Data Collector error.
PM _ APIERROR	PM _ INPUTAREA _ OVERFLOW	Too many IFCIDs requested. Request input area overflow. Reduce the number of requested IFCIDs.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Snapshot Processing - Release Snapshot Store

Function Call

```
pmReturnCodes pmReleaseStore (pmHost*      handle,
                               char*        workProfile,
                               unsigned int id)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to release one or all snapshot stores that are assigned to the user identified by parameter **workProfile**.

Parameters

1. **handle** (input)
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)
A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.
3. **id** (input)
The identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStore()** function.
If you want to release an individual snapshot store, specify the store with parameter **id**.

If you want to release all snapshot stores of the user identified by parameter `workProfile`, set `id` to 0.

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
8	1	Internal Data Collector error.
8	22	Internal Data Collector error.
8	25	Internal Data Collector error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1554	Internal Data Collector error.
8	155A	Incorrect snapshot store ID specified.
8	155D	Incorrect snapshot store ID specified, or snapshot store no longer available.
8	155E	Profile ID of user not found, or no longer valid.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

History Processing - Get History Data

Function Call

```
pmReturnCodes pmGetHistory (pmHost*      handle,
                           char*        workProfile,
                           char**      fields,
                           unsigned int counterNo,
                           pmQualifierList* qualifier,
                           pmTOD       requestTime,
                           pmDirection dir,
                           pmTOD       snapshotTime,
                           pmHashTable result)
```

Header File

`pmGetStatThread.h`

Description

This function requests the Data Collector to retrieve a stored snapshot of DB2 performance data from the history data set for a user specified by parameter `workProfile`.

Counters can be specified and qualified as with the `pmGetSnapshot()` function.

Note that only snapshots of IFCIDs that are specified as Data Collector startup parameters and qualified thread data are written to the history data set.

This function requires that the output data area is initialized with the `clearHashTable()` function.

Parameters

1. `handle` (input)
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the `pmConnect()` function.
2. `workProfile` (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **fields** (input)

An array of counter names for which counter data is requested from a stored snapshot record in a history data set.

Specify at least one counter name. Counter names are not case-sensitive. Invalid counter names are ignored. Counters not available in the stored snapshot record in a history data set are ignored. No warning is returned.

```
char *fields[] = {
    "QISEDDBD", "QTMAXDS", "QTAUCHK", "QTAUSUC", "QBSTMAX",
    "QBSTPID", "QBSTGET", "QBSTWFT", "QBSTWFD", "QBSTWKP",
    "QBSTWKP", "QBSTMAX", "QBSTWKP", "QBSTWDRP", "QBSTWBVQ",
    "QBSTWFR", "QBSTWFF", "QBSTWFD", "QISEDSI", "QISEDSG",
    "QISEDSC", "QXSTFND", "QXSTNFND", "QBSTWDRP", "QBSTWBVQ"
};
```

4. **counterNo** (input)

The number of counters specified with parameter **fields** (> 0).

5. **qualifier** (input)

The name of a qualifier list. The list specifies the criteria which data the Data Collector should return. See "How Qualifying Works" on page 35, if required.

6. **requestTime** (input)

Use this parameter together with parameter **dir** to locate a stored snapshot record in the history data set.

- If you want to locate a stored snapshot record nearest to a specified point in time (and younger than the given time), set **requestTime** to the point in time you want and parameter **dir** to T0.
- If you want to locate the oldest stored snapshot record, set **requestTime** to TOD_FIRST and parameter **dir** to FORWARD.
- If you want to locate the youngest stored snapshot record, set **requestTime** to TOD_LAST and parameter **dir** to BACK.
- If you want to locate a stored snapshot record adjacent to a previously located record, set **requestTime** to the time returned by the previous **pmGetHistory()** function call. See parameter **snapshotTime** below. Use parameter **dir** to choose the direction (the older or the younger record adjacent to this point in time).

See "Working with the History Data Set" on page 36 if you need a more detailed description.

The required format of the **requestTime** parameter is the Store Clock format. If you need a conversion from **time_t** format, see "Convert time_t Format to Store Clock Format" on page 97 for details.

7. **dir** (input)

Use this parameter together with parameter **requestTime** to locate a stored snapshot record in the history data set.

- Use T0 together with a given point in time (specified by parameter **requestTime**), if you want to locate a stored snapshot record nearest to a point in time. Per definition the younger snapshot in the history data set is chosen.
- Use BACK if you want to choose the older snapshot adjacent to a previously located snapshot.
- Use FORWARD if you want to choose the younger snapshot adjacent to a previously located snapshot.

8. **snapshotTime** (output)

The time stamp of a stored snapshot retrieved from the history data set.

The time stamp is in Store Clock format. If you need a conversion to **time_t** format, see “Convert Store Clock Format to time_t Format” on page 95 for details.

9. **result** (output)

Pointer to the output data area. See “Working with Returned Data” on page 37 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	22	Internal Data Collector error.
8	32	Internal Data Collector error.
8	33	History function not started in the Data Collector.
8	35	Internal Data Collector error.
8	36	Request rejected. The history data set is empty. No snapshot is gathered yet.
8	37	Request rejected. You requested a snapshot with parameter dir set to T0 for a time that is older than the oldest snapshot in the history data set. Specify a requestTime that is younger than the oldest snapshot in the history data set, or locate the oldest snapshot with requestTime set to TOD_FIRST and dir set to FORWARD.
8	38	Request rejected. You requested a snapshot with parameter dir set to BACK that is already the oldest snapshot in the history data set.
8	39	Request rejected. You specified a snapshot with parameter dir set to T0 for a time that is younger than the youngest snapshot in the history data set. Specify a requestTime that is older than the youngest snapshot in the history data set, or locate the youngest snapshot with requestTime set to TOD_LAST and dir set to BACK.
8	3A	Request rejected. You requested a snapshot with parameter dir set to FORWARD for a snapshot that is currently the youngest in the history data set. Wait until the next snapshot is gathered by the Data Collector and retry the operation.
8	3B	Internal Data Collector error.
8	3C	Internal Data Collector error.

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	3D	Internal Data Collector error.
8	3E	Internal Data Collector error.
8	40	Data Collector Authorization exit returned error. Irrecoverable error.
8	42	The function is not available. A DB2 PM license is required for this function.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	You specified incorrect parameter values: - counterNo must be larger than 0. - fields must contain at least one valid counter name. - dir must be BACK, FORWARD, or T0.
PM _ APIERROR	PM _ INPUTAREA _ OVERFLOW	You requested too many counters, or too many counters as qualifiers.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Processing DB2 Exception Events

Introduction to Exception Processing

The DB2 PM API provides functions that you can use in the application to work with DB2 exception events. The following DB2 event types are supported:

- Authorization failure
- Timeout
- EDM Pool full
- Global Trace started
- Thread commit indoubt
- Deadlock
- Data set extension
- Unit of recovery inflight or indoubt
- Logspace shortage
- Coupling Facility events:
 - CF rebuild start
 - CF rebuild stop
 - CF alter start
 - CF alter stop

Exception processing through the API requires that the Data Collector is started with event exception processing or periodic exception processing enabled. The API can access exception events only if the Data Collector records this data. Note that the Data Collector can be started with event exception processing enabled, periodic exception processing enabled, none, or both enabled. However, the API currently has access only to event exception data.

The Data Collector records the 500 most recent DB2 authorization events and other DB2 events in an exception log. The application program can retrieve this exception log and individual log records for further processing.

The application can also request the Data Collector to post event exceptions to the application program as they occur.

The application program controls the event exception processing in the Data Collector with the following API functions:

- The **pmGetEventExceptionLog()** function retrieves the exception log from the Data Collector. You can specify a time frame to limit the number of exception log records that are returned to the application.
- The **pmGetEventDetails()** function retrieves all details about an individual exception log record. You identify an exception log record by a time stamp and an event exception type. Both are returned by a previous **pmGetEventExceptionLog()** function call, and a **pmFetchExceptions()** function call.
- The **pmStartExceptionProc()** function starts event exception processing in the Data Collector. This prepares the Data Collector for posting event exceptions to the application. Parameters allow you to specify the type of event exceptions to process. Each event exception process is assigned to an existent user profile (specified by the **pmLogOn()** function at logon time). One event exception process can be assigned to one user's work profile.
- The **pmGetExceptionStatus()** function requests status information about an exception process for a specified user. You use this function to check whether a process is already started, or to gather details about a process.
- The **pmFetchExceptions()** function requests the Data Collector to post exception records to the application program as they occur.

You use this function to keep track with exception events. The exception records that are returned to the application program are similar to those returned by the **pmGetEventExceptionLog()** function. Therefore, use the **pmGetEventDetails()** function to retrieve all details about an individual exception log record.

Subsequent **pmFetchExceptions()** function calls request the Data Collector to post yet undelivered exception records. If no new exception records exist, this function waits until the next exception occurs.

- Finally, the **pmStopExceptionProc()** function stops event exception processing for a named user profile.

Event exception processing also stops if the named user is logged off from the Data Collector.

However, event exception processing remains active if the application disconnects the named user from the Data Collector. After the application reconnects to the Data Collector, the named user has again access to exceptions that occurred after the **pmFetchExceptions()** function was used last.

Note: The following functions contain provisions for possible future extensions. Therefore, some parameters must be set to fixed values, and some output variables return fixed values.

Retrieve Event Exception Log

Function Call

```
pmReturnCodes pmGetEventExceptionLog (pmHost*      handle,  
                                       char*        workProfile,  
                                       pmTOD        from,  
                                       pmTOD        to,  
                                       char**       data,  
                                       unsigned int* length)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to retrieve the exception log for a user specified by parameter **workProfile**.

At the most, the Data Collector returns 500 of the most recent exception log records. You can specify a time frame to limit the number of exception log records to be returned.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **from** (input)

This parameter specifies the earliest date and time (in Store Clock format) for which exception log records should be retrieved. If **from** is NULL, the exception log is read from the beginning.

The **from** parameter requires the Store Clock format. If you need a conversion from **time_t** format, see "Convert time_t Format to Store Clock Format" on page 97 for details.

4. **to** (input)

This parameter specifies the latest date and time (in Store Clock format) for which exception log records should be retrieved. If **to** is NULL, the exception log is read to the end.

The **to** parameter requires the Store Clock format. If you need a conversion from **time_t** format, see "Convert time_t Format to Store Clock Format" on page 97 for details.

5. **data** (output)

Pointer to the output data area. The output data area can be parsed by using the functions described in "Parsing Data" on page 88. The field table describes the exception IDs returned by this function in section "Exception IDs". The "Example" on page 64 shows how this can be done. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Output Data Area for pmGetEventExceptionLog()

►►—total length of output area—CNT_X number—DATEATIM date and time—►►



Event exception record:

|—RC_XEVT| Event information |—DATEATIM date and time—RC_XEVTE—|

Event information:

XEV_DEAD	RESOURTY	Locked resource type
XEV_TOUT	RESOURTY	Locked resource type
XEV_EDM	EDMFULID	EDM Pool full reason
XEV_AUTH	AUTHORIZ	Authorization ID checked
XEV_CFRA	STRUCTNA	Structure name
XEV_CFRO		
XEV_CFAO	REASSTOP	Reason stopped
	ASUCCESS	
	SUCCEBUT	
XEV_COMM	EI_HEUDO	
	EI_PACOS	
	EI_ILOGS	
	EI_SNACO	
	EI_HEUDA	
	EI_SNASD	
	EI_SNASY	
	EI_LOGCH	
	EI_CICSU	
	EI_CONDR	
	EI_XLNPR	
	EI_RESTA	
XEV_TRAC		
XEV_DSEX	DSNAME	Data set name
XEV_URPR	URTYP	Unit of recovery type
XEV_LGSP	COPYNUM	Log copy number
	PERCENT	Percentage filled

CNT_XEVT

The value indicates the number of event exceptions following in the list.

RC_XEVT

Specifies the beginning of an event exception record.

CNT_XEVT

The value indicates the number of event exceptions following in the list.

RC_XEVT

Specifies the beginning of an event exception record.

RC_XEVTE

Specifies the end of an event exception record.

EI_HEUDO

Heuristic decision occurred.

EI_PACOS

Partner cold start detected.

EI_ILOGS

Incorrect logname or syncpoint parm.

EI_SNACO

SNA compare stats protocol error.

EI_HEUDA
Heuristic damage.

EI_SNASD
SNA syncpoint protocol damage.

EI_SNASY
Syncpoint communication failure.

EI_LOGCH
Logname changed on warm/start.

EI_CICSU
CICS or IMS NID unknown.

EI_CONDR
Conditional restart data loss.

EI_XLNPR
XLN protocol error.

EI_RESTA
Error during DB2 restart.

6. length (output)

The length (in number of bytes) of the output data area.

Example

```
#include "pmExcpProc.h"
#include "pmParser.h"
#include "pmCounter.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include "pmTOD.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
...

pmHost          myHandle;
char            workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001  ";
pmReturnCodes  error;
char*          data;
char*          helpPtr;
unsigned int    length,
               pos = 0,
               counterNo;
pmCounter      aCounter;
time_t         eventTime;
pmBoolean      skipped;

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);

// ok, now get the complete event exception log
// from log start to log end (from and to set to NULL)
if(pmGetEventExceptionLog (&myHandle, workProfile, NULL, NULL,
                          &data, &length).returnCode == 0)
{
    helpPtr = data + 4; /* helpPtr points now to output data area */
                      /* without 'total length of output' area */
                      /* field. */
}
```

```

length -= 4;          /* output data area length without */
                    /* 'total length of output' area field */

/* get number of returned exceptions (CNT_X) */
if(nextTokenValue (&myHandle, helpPtr, &pos,
                  length, "CNT_X", &counterNo) != PM_FAILED)
{
    printf("Found %d exceptions:\n", counterNo);

    /* skip date and time of pmGetEventExceptionLog() execution */
    /* (first DATEATIM counter) */
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);

    /* now handle all returned exceptions */
    while(counterNo--)
    {
        /* skip start of Event Exception Record (RC_XEVT), it is just */
        /* a flag to indicate start */
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);

        /* now we are at the start of the event information -> */
        /* get type of event exception */
        aCounter = nextToken(&myHandle, helpPtr, &pos, length, FALSE);
        if(aCounter.counterID == 0)
        {
            printf("Parsing failed, data stream is invalid.\n");
            counterNo = 0;
            break;          /* counter invalid */
        }

        /* handle all kinds of event exceptions */
        if(!strcmp(aCounter.name, "XEV_DEAD"))
        {
            /* deadlock event */
            printf("    >> '\Deadlock\' exception from");
            skipToken(&myHandle, helpPtr, &pos, length, FALSE);
        }
        else if(!strcmp(aCounter.name, "XEV_TOUT"))
        {
            /* timeout event */
            printf("    >> '\Timeout\' exception from");
            skipToken(&myHandle, helpPtr, &pos, length, FALSE);
        }
        else if(!strcmp(aCounter.name, "XEV_EDM"))
        {
            /* EDM pool full event */
            printf("    >> '\EDM Pool full\' exception from");
            skipToken(&myHandle, helpPtr, &pos, length, FALSE);
        }
        else if(!strcmp(aCounter.name, "XEV_AUTH"))
        {
            /* authentication failure event */
            printf("    >> '\Authentication Failure\' exception from");
            skipToken(&myHandle, helpPtr, &pos, length, FALSE);
        }
        else if(!strcmp(aCounter.name, "XEV_CFRA"))
        {
            /* CF rebuild start event */
            printf("    >> '\CF rebuild start\' exception from");
            skipToken(&myHandle, helpPtr, &pos, length, FALSE);
        }
        else if(!strcmp(aCounter.name, "XEV_CFRO"))
        {
            /* CF rebuild stop event */
            printf("    >> '\CF rebuild stop\' exception from");
            if(testToken("REASSTOP", helpPtr, pos, length) == PM_OK)

```

```

        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
    }
else if(!strcmp(aCounter.name, "XEV_CFA0"))
{
    /* CF alter event */
    printf("    >> \'CF alter\' exception from");
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_COMM"))
{
    /* commit in-doubt event */
    printf("    >> \'Commit in-doubt\' exception from");
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_TRAC"))
{
    /* trace started event */
    printf("    >> \'Global Trace Started\' exception from");
}
else if(!strcmp(aCounter.name, "XEV_DSEX"))
{
    /* trace started event */
    printf("    >> \'Data Set Extention\' exception from");
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_URPR"))
{
    /* trace started event */
    printf("    >> \'Unit of Recovery inflight or indoubt\' exception from");
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_LGSP"))
{
    /* trace started event */
    printf("    >> \'Logspace shortage\' exception from");
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else
{
    printf("Unknown event exception found, skipping...\n");
    skipped = PM_OK;
    while(testToken("DATEATIM", helpPtr, pos, length) ==
           PM_FAILED && skipped == PM_OK)
        skipped = skipToken(&myHandle, helpPtr, &pos, length, FALSE);
    if(skipped == PM_FAILED)
    {
        printf("Data stream invalid!\n");
        counterNo = 0;
        break;          /* counter invalid */
    }
}
/* free internal used memory of counter */
deleteCounter(aCounter);

/* get time of event */
aCounter = nextToken(&myHandle, helpPtr, &pos, length, FALSE);
if(aCounter.counterID == 0)
{
    printf("Parsing failed, data stream is invalid.\n");
    counterNo = 0;
    break;          /* counter invalid */
}
tod2time(aCounter.value, &eventTime);
printf(" %s", ctime(&eventTime));

```



```

        deleteCounter(aCounter);

        /* skip end of event record token */
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
    }
}
else
{
    printf("Parsing failed, data stream is invalid.\n");
}
pmFreeMem(data);
}

// log off and disconnect (stops also event exception processing)
...

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1000	Internal Data Collector error.
8	1001	Internal Data Collector error.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	Start time is equal or greater than end time.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Retrieve Event Exception Details

Function Call

```

pmReturnCodes pmGetEventDetails (pmHost*      handle,
                                char*         workProfile,
                                pmTOD        eventTime,
                                char*        eventType,
                                char**       data,
                                unsigned int* length)

```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to retrieve detailed information about an event exception. The exception log record to retrieve is specified by the time stamp and the event exception type that was returned by a previous **pmGetEventExceptionLog()** or **pmFetchExceptions()** function call. The information returned depends on the event exception type.

Parameters

1. **handle** (input)

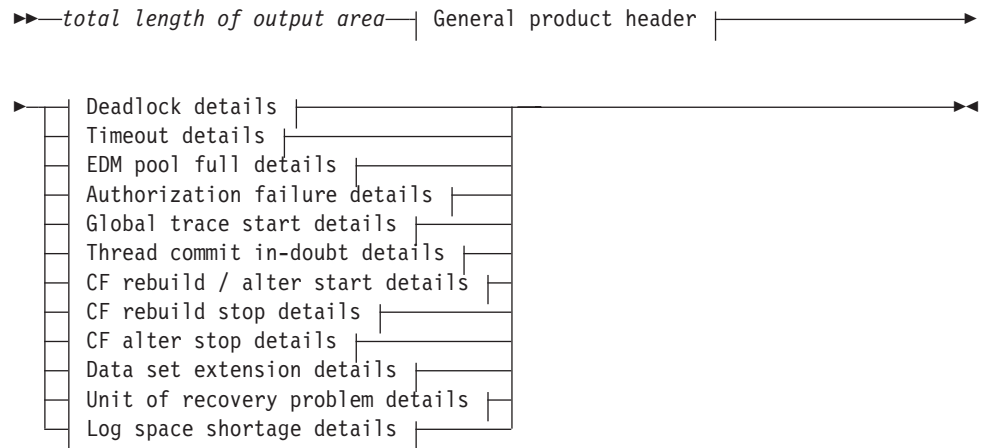
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

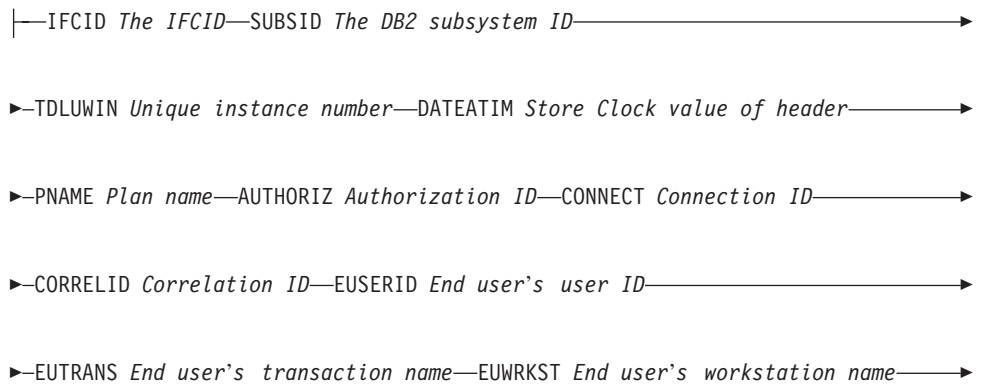
A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **eventTime** (input)
The time (in Store Clock format) at which the event exception occurred. Use the content of field **DATEATIM** returned by a previous **pmGetEventExceptionLog()** or **pmFetchExceptions()** function call.
4. **eventType** (input)
The event exception type. Use the event exception type ("XEV_xxxx") returned by a previous **pmGetEventExceptionLog()** or **pmFetchExceptions()** function. For example, "XEV_DEAD" specifies a "deadlock" event. See the output data area description of the **pmGetEventExceptionLog()** function for possible event exception types.
5. **data** (output)
Pointer to the output data area. The output data area can be parsed by using the functions described in "Parsing Data" on page 88. The "Example" on page 74 shows how this can be done. The field table describes the exception IDs returned by this function in section "Exception IDs". Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Output Data Area for pmGetEventDetails()



General product header:

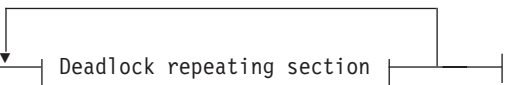


▶—GNAME *DB2 data sharing group name*—MNAME *DB2 member name*—————|

Deadlock details:

|—INTERVCT *Interval counter*—CNT_RESI *Numbers involved*—————→

▶—DATEIFROM *Deadlock detection time*—————|



Deadlock repeating section |—————|

Deadlock repeating section:

|—XSEPARAT | *Deadlock/Timeout resource details* |—REQFUNCT *Requested function*—————→

▶—ASSIGNWV *DB2 worth value*—LOCKFLAG *Lock attribute*—————→

▶—XSEPARAT BLK_FLAG *Constant H for Holder*—————→

▶—BLK_STAT *Lock flag* | *Deadlock/Timeout Holder/Waiter details* |—————→

▶—XSEPARAT BLK_FLAG *Constant W for Waiter* | *Deadlock/Timeout Holder/Waiter details* |—————|

Timeout details:

|—CNT_RESI *Numbers involved*—TINTERV *Timeout interval*—————→


▶—COUNT_S *Timeout counter for thread*—————→

▶ | *Deadlock/Timeout resource details* |—REQFUNCT *Requested function*—————→

▶—REQSTATE *Requested lock state*—REQDURAT *Requested duration*—————→

▶—REQFLAG *Requested lock flag*—REQOWNUW *Requested owning work unit*—————→

▶—————|



Timeout repeating section |—————|

Timeout repeating section:

|—XSEPARAT BLK_FLAG *Holder, (Priority) Waiter, Retained Lock*—————→

- ▶-BLK_STAT Lock flag—————→
- ▶-REQOWNUM Requested owning work unit| Deadlock/Timeout Holder/Waiter details |

Deadlock/Timeout resource details:

RESOURTY X'00'	Data page	Deadlock/Timeout common details	PAGENUMB Page number
RESOURTY X'01'	Database DATABASE Database		
RESOURTY X'02'	Pageset locking	Deadlock/Timeout common details	
RESOURTY X'03'	Table space	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'04'	Skeleton cursor table	-PNAME Plan name	
RESOURTY X'05'	Index page	Deadlock/Timeout common details	PAGENUMB Page number—SUBPAGEN Subpage number
RESOURTY X'06'	Partition lock	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'07'	Page dataset open	Deadlock/Timeout common details	
RESOURTY X'0A'	Database exception table	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'0D'	GBP dependent	Deadlock/Timeout common details	
RESOURTY X'0F'	Mass delete	Deadlock/Timeout common details	
RESOURTY X'10'	Table	Deadlock/Timeout common details	
RESOURTY X'11'	Hash anchor	Deadlock/Timeout common details	PAGENUMB Page number—ANCHOR Anchor
RESOURTY X'12'	Skeleton package table	-PACKAGE Package—COLLECTI Collection ID—CONSTOKE Consistency token	
RESOURTY X'13'	Collection COLLECTI Collection ID		
RESOURTY X'14'	CS read drain	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'15'	Repeatable read drain	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'16'	Write drain	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'18'	Data row	Deadlock/Timeout common details	PAGNUMB Page number—ROW Row
RESOURTY X'19'	Index EOF	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'1A'	Alter bufferpool	-BPOOL Bufferpool	
RESOURTY X'1B'	Group bufferpool	-BPOOL Bufferpool	
RESOURTY X'1C'	Index tree	Deadlock/Timeout common details	
RESOURTY X'1D'	Pageset/partition	Deadlock/Timeout common details	
RESOURTY X'1E'	Page P-lock	Deadlock/Timeout common details	
RESOURTY X'23'	DBD P-lock	Deadlock/Timeout common details	
RESOURTY X'27'	Database exception	Deadlock/Timeout common details	PARTNUMB Partition number
RESOURTY X'28'	Utility UID	-UTIL Utility ID	
RESOURTY X'29'	Utility exclusive	-RMID Rmid—HASHVALU Hash value	
RESOURTY X'30'	LOB lock	Deadlock/Timeout common details	

Deadlock/Timeout common details:

- |—DATABASE Database—OBJECT Object name—LOBROWID LOB row ID (V6)—————→
- ▶-LOBVERSI LOB version number (V6)—————|

Deadlock/Timeout Holder/Waiter details:

- |—PNAME Plan name—CORRELID Correlation identifier—————→
- ▶-CONNECT Connection identifier—TDLUWNID LUW network ID—————→
- ▶-TDLUWLUN LUW LU name—TDLUWIN LUW instance number—TDTOKEN Thread token—————→
- ▶-LOCKSTAT Requested state—DURATION Requested duration—————→
- ▶-MEMBERNA Member name—AUTHORIZ Authorization ID—EUSERID End user's ID—————→

▶EUTRANS *End user's transaction name*—EUWRKST *End user's workstation name*—|

EDM pool full details:

—EDMFULID *EDM Pool full reason*—	Reason: database	—
Reason: cursor table	—	
Reason: package table	—	

Reason: database:

|—DATABASE *Database*—SECTILEN *Length of section*—|

Reason: cursor table:

|—PNAME *Planname*—RDSID *RDS identifier*—|

▶—SEQUENNO *Sequence number within RDS*—|

▶—CUTABLEN *Length of cursor table selection*—|

Reason: package table:

|—LOCATION *Location name*—COLLECTI *Collection identifier*—|

▶—PACKAGE *Package identifier*—CONSTOKE *Consistency token*—|

▶—RDSID *RDS identifier*—SEQUENNO *Sequence number within RDS*—|

▶—PATABLEN *Len of pkg tab sel*—|

Authorization failure details:

|—AUTHORIZ *Authorization ID checked*—PRIVILEG *Privilege checked*—|

▶—OBJECTTY *Object type*—OBJECTOW *Source object owner*—|

▶—OBJECTNA *Source object name*—OBJECTOW *Target object owner*—|

►OBJECTNA *Target object name*—SQLSTATE *SQL statement*—|

Global trace start details:

|—TRACECMD *The trace command*—|

Thread commit in-doubt details:

|—E1_HEUDO—UNFORREC *Unformatted record*—|
|—E1_PACOS—|
|—E1_ILOGS—|
|—E1_SNASD—|
|—E1_HEUDA—|
|—E1_SNASD—|
|—E1_SNASY—|
|—E1_LOGCH—|
|—E1_CICSU—|
|—E1_CONDR—|
|—E1_XLNPR—|
|—E1_RESTA—|

Coupling facility rebuild / alter start details:

|—STRUCTNA *Structure name*—REASINIT *Reason initiated*—DATEATIM *Start time*—|

►REQUFSIZE *Requested size*—|

Coupling facility rebuild stop details:

|—STRUCTNA *Structure name*—DATEATIM *Start time*—DATEATIM *End time*—|

►ELAPSTIM *Elapsed time*—REASINIT *Reason initiated*—|

►CUCOELEM *Current count of elements (V6)*—|
|—REASSTOP *Reason stopped*—|

Coupling facility alter stop details:

|—STRUCTNA *Structure name*—DATEATIM *Start time*—DATEATIM *End time*—|

▶-ELAPSTIM *Elapsed time*—REASSTOP *Reason stopped*—
 └─ASUCCESS—┬─ Current values ─┬─
 └─SUCCEBUT—┴─

Current values:

└─REQUFSIZE *Size (4 KB)*—MINISIZE *Minimum size (4 KB)*—

▶-DIRENTRY *Directory entries*—ELMENTRY *Element entries*—

▶-CUCOELEM *Current count of elements (V6)*—

Data set extension details:

└─DSNAME *The data set name*—DATEATIM *The timestamp after extent*—

▶-DBNAME *The database name*—DATABASE *The database ID - DBID*—

▶-OBJECT *The pageset ID - PSID*—TISNAME *The table/index space name*—

▶-PQUANT *The primary allocation quantity*—

▶-SQUANT *The secondary allocation quantity*—

▶-MAXSIZE *The maximum data set size*—

▶-ALLOCBEF *The allocated space before extent*—

▶-ALLOCAFT *The allocated space after extent*—MAXEXT *The maximum extents*—

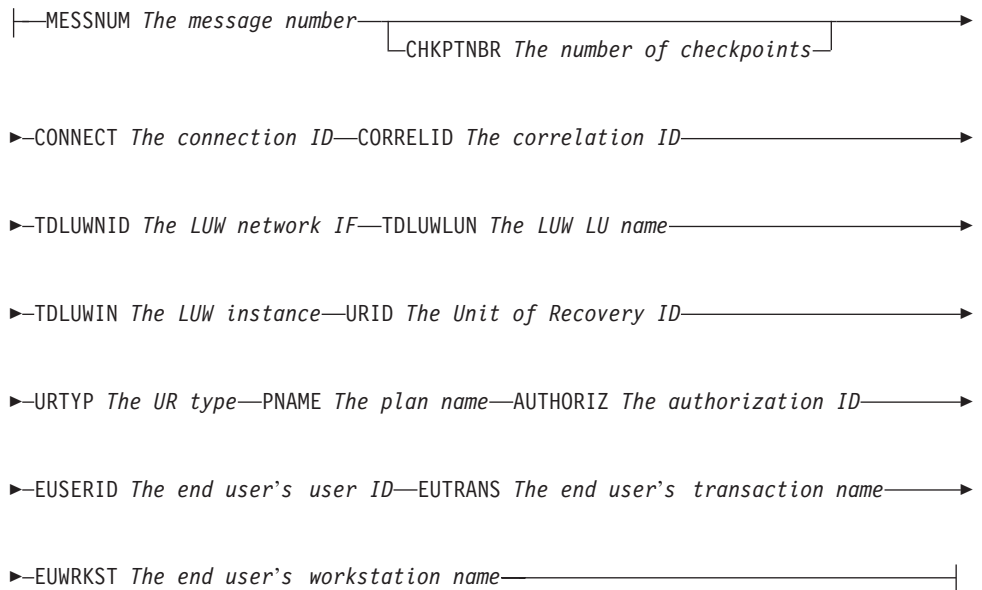
▶-EXTBEF *The number of extents before extent*—

▶-EXTAFT *The number of extents after extent*—MAXVOL *The maximum volumes*—

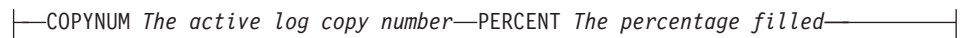
▶-VOLBEF *The number of volumes before extent*—

▶-VOLAFT *The number of volumes after extent*—

Unit of recovery problem details:



Log space shortage details:



6. length (output)

The length (in number of bytes) of the output data area.

Example

```
#include "pmExcpProc.h"
#include "pmParser.h"
#include "pmCounter.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...

pmHost      myHandle;
char        workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001 ";
pmReturnCodes error;
char*       data;
char*       helpPtr;
unsigned int length,
            pos = 0,
            counterNo;

pmCounter   aCounter;
pmTOD       evTime;
char        evType[256];

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);
```



```

// fetch exception
error = pmFetchExceptions (...);

// parse it and store time and type of event exception
...

evTime = ...
evType = ... /* 'XEV_DEAD', 'XEV_AUTH', 'XEV_COMM', etc. */
// ok, now get the details for this event
if(pmGetEventDetails(&myHandle, workProfile, evTime,
                    evType, &data, length).returnCode == 0)
{
    helpPtr = data + 4; /* helpPtr points now to output data area */
                      /* without 'total length of output' area */
                      /* field. */
    length -= 4; /* output data area length without */
               /* 'total length of output' area field */

    /* use parser functions to get values */
    ...

    /* do not forget to free output data area */
    pmFreeMem(data);
}

// log off and disconnect (stops also event exception processing)
...

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
4	1045	Event detail no longer available.
8	1	Internal Data Collector error.
8	1000	Internal Data Collector error.
8	1001	Internal Data Collector error.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	Unknown event exception type specified.
See also “Common Return Codes and Reason Codes” on page 10, if required.		

Start Exception Processing

Function Call

```

pmReturnCodes pmStartExceptionProc (pmHost*      handle,
char*          workProfile,
pmExceptionType type,
unsigned int   interval,
pmBoolean     userExit,
char*         thresholdDefs)

```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to start event exception processing for a user specified by parameter **workProfile**.

This function is required before the **pmFetchExceptions()** function can be used in the application program.

This function should not be called more than once for the same event type and for the same user. Use the **pmGetExceptionStatus()** with the same work profile to check in advance whether exception processing is already started for a specified user.

Parameters

1. **handle** (input)
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)
A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.
3. **type** (input)
The type of exception processing to start. Specify EVENT.
4. **interval** (input)
Reserved. Specify 0.
5. **userExit** (input)
Reserved. Specify FALSE.
6. **thresholdDefs** (input)
Reserved. Specify NULL.

Example

```
#include "pmExcpProc.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost      myHandle;
char        workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001  ";
pmReturnCodes error;

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);

// log off and disconnect
...
```

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
8	1	Internal Data Collector error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1000	Internal Data Collector error.
8	1001	Internal Data Collector error.
8	1002	Internal Data Collector error.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Get Exception Processing Status

Function Call

```
pmReturnCodes pmGetExceptionStatus (pmHost*      handle,  
                                     char*       workProfile,  
                                     pmExcpInfo* info)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to report the status of exception processing for a user specified by parameter **workProfile**. It returns information about:

- Whether exception processing is already started.
- Which exceptions are enabled in the Data Collector.
The type of exception events (none, event exceptions only, periodic exceptions only, or both) is specified as Data Collector startup parameter.
- Which type of exception events occurred since exception processing was started.
The **pmGetExceptionStatus()** function requests the Data Collector to scan the DB2 trace records for the existence of possible exception types to find out which type of exception events occurred.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **info** (output)

Pointer to the output data area. Some structure members are reserved for future use.

```
typedef struct _excpInfo  
{  
    pmExceptionType status;           /* if EVENT, PERIODIC, */  
                                       /* BOTH or NONE are active */  
    pmBoolean      userExitActive;    /* host user exit */  
    unsigned int   periodicExcpInterval; /* interval in seconds */  
                                       /* when PM checks for */  
                                       /* periodic excps */  
    pmBoolean      perExcpHappened;   /* periodic excp happened */  
    pmBoolean      evExcpHappened;    /* event excp happened */  
                                       /* since logoff */  
    pmBoolean      deadlockActive,    /* indicates that the */  
                                       /* corresponding events */  
                                       /* are observed */  
                                       /* are observed */  
    authFailureActive,  
    traceActive,  
    datasetExtentActive,  
    unitOfRecoveryActive,  
    logspaceShortageActive,  
    commitActive,  
    rebuildActive;
```

```

        unsigned short  datasetExtentValue;      /* number of extents before*/
                                                /* datasetExtent event is */
                                                /* triggered                */
    } pmExcpInfo;

```

- *status* indicates the type of exception processing that was started for this user. It contains:
 - EVENT for event exception processing
 - PERIODIC for periodic exception processing (shown here if started from the DB2 PM Online Monitor)
 - BOTH for event exception and periodic exception processing
 - NONE if no exception processing was started.
- *userExitActive* indicates whether a DB2 PM user exit routine is active. This can limit the access of the application to some data.
- *periodicExcpInterval*. Reserved for future use.
- *perExcpHappened*. Reserved for future use.
- *evExcpHappened* indicates whether event exceptions occurred since the last **pmFetchExceptions()** function call for the user specified by parameter **workProfile**. If no **pmFetchExceptions()** function was called, it indicates whether event exceptions occurred since exception processing was started for this user.
- *deadlockActive ... rebuildActive* indicate which events are observed.
- *datasetExtentValue* shows the number of data set extents to occur before DB2 triggers a corresponding event. This number is a Data Collector startup parameter.

Boolean variables are returned as either TRUE or FALSE.

Example

```

#include "pmExcpProc.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost      myHandle;
char        workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001  ";
pmReturnCodes  error;
pmExcpInfo  info;

// connect to data collector and log on
...

// get exception processing information
error = pmGetExceptionStatus(&myHandle, workProfile, &info);

// log off and disconnect
...

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1000	Internal Data Collector error.

Return Code ² (Hex)	Reason Code ² (Hex)	Description
PM _ APIERROR	PM _ DATASTREAM _ INVALID	Received data stream not valid. Counter not found, or end of data reached before expected. Verify that the latest DB2 PM PTF is installed. If you cannot solve the problem, call for IBM support.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Fetch Exceptions

Function Call

```
pmReturnCodes pmFetchExceptions (pmHost*      handle,
                                char*         workProfile,
                                unsigned int  exceptionNo,
                                char**       data,
                                unsigned int* length)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to post exception records to the application as they occur. Exception records are posted for a user (specified by parameter **workProfile**) for whom exception processing was started with the **pmStartExceptionProc()** function.

This function allows the application to fetch event exceptions immediately when they occur. Each **pmFetchExceptions()** function call returns those exception records that were not yet returned by a previous **pmFetchExceptions()** function call. If no exceptions occurred in the meantime, the function waits until the next event exception occurs.

You can use the **pmGetExceptionStatus()** function to check whether event exceptions occurred since the last time the **pmFetchExceptions()** function was used.

The **pmFetchExceptions()** function does not deliver an exception record twice to the user specified by **workProfile**. If the application needs access to the same exception record more than once, you need to retrieve the exception log from the Data Collector with the **pmGetEventExceptionLog()** function. Then, you can retrieve details about an individual exception log record with the **pmGetEventDetails()** function.

When the **pmFetchExceptions()** function has returned one or more exception records to the application, use the **pmGetEventDetails()** function to request details about individual records.

The **pmFetchExceptions()** function returns an unknown number of exception records to the application. The number of returned exception records depends on the elapsed time between two consecutive **pmFetchExceptions()** function calls and the number of events that occurred in the meantime. This function can return a maximum of 500 exception records. With parameter **exceptionNo** you can control the number of exception records to be returned. If more than the specified number of events occur between two consecutive function calls, only the most recent ones

are returned. You can also control that the **pmFetchExceptions()** function returns only the next exception record that occurs after the function is called. This causes the function to wait for the next event exception.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **exceptionNo** (input)

Set this parameter to either 0 or a positive number ≤ 500 .

If **exceptionNo** is set to 0, only the exception that occurs after this **pmFetchExceptions()** function call is received by the Data Collector is posted to the application. No previous exception records are posted.

If **exceptionNo** is set to a number > 0 , all exception records (up to the specified number) not yet delivered by a previous **pmFetchExceptions()** function call are immediately returned.

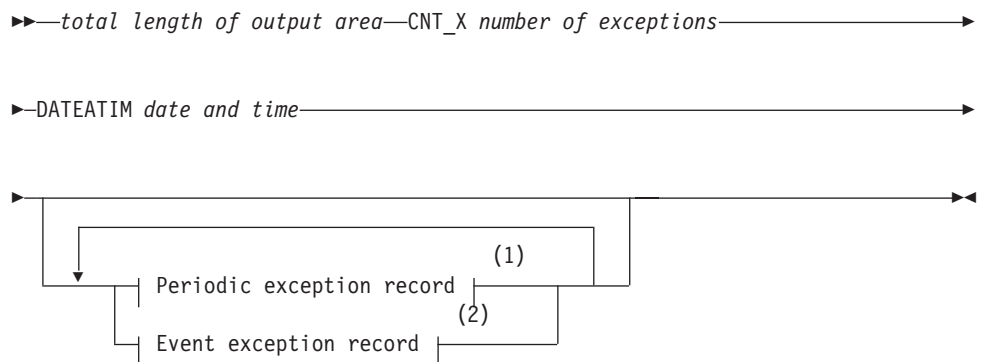
If no exception record is to be returned, the function waits for the next exception.

Note: For subsequent **pmFetchExceptions()** function calls you should set **exceptionNo** to the maximum of 500.

4. **data** (output)

Pointer to the output data area. The output data area can be parsed by using the functions described in "Parsing Data" on page 88. The "Example" on page 81 shows how this can be done. The field table describes the exception IDs returned by this function in section "Exception IDs". Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Output Data Area for pmFetchExceptions()



Notes:

- 1 Periodic exceptions are currently not supported by the DB2 PM API.
- 2 Refer to the **pmGetEventExceptionLog()** function for a detailed description of the event exception record.

5. **length** (output)

The length (in number of bytes) of the output data area.

Example

```
#include "pmExcpProc.h"
#include "pmParser.h"
#include "pmCounter.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...

pmHost      myHandle;
char        workProfile[] = "PMUSER      DB2PM      10.0.0.1:0001  ";
pmReturnCodes error;
char*       data;
char*       helpPtr;
unsigned int length,
            pos = 0;
pmCounter   aCounter;

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);

// ok, now fetch always the detected event exceptions
// until returnCode <= 0
while(!pmFetchExceptions (&myHandle, workProfile, 0,
                          &data, &length).returnCode)
{
    helpPtr = data + 4; /* helpPtr points now to output data area */
                      /* without 'total length of output' area */
                      /* field. */
    length -= 4; /* output data area length without */
                /* 'total length of output' area field */

    /* skip number of returned exceptions (CNT_X) because it is '1' */
    if(skipToken(&myHandle, helpPtr, &pos, length, FALSE) == PM_FAILED) break;

    /* skip date and time of pmFetchExceptions() execution */
    /* (first DATEATIM counter) */
    if(skipToken(&myHandle, helpPtr, &pos, length, FALSE) == PM_FAILED) break;

    /* skip start of Event Exception Record (RC_XEVT), it is just */
    /* a flag to indicate start */
    if(skipToken(&myHandle, helpPtr, &pos, length, FALSE) == PM_FAILED) break;

    /* now we are at the start of the event information -> */
    /* get type of event exception */
    aCounter = nextToken(&myHandle, helpPtr, &pos, length, FALSE);
    if(aCounter.id == 0) break; /* counter invalid */

    /* handle all kinds of event exceptions */
    if(!strcmp(aCounter.name, "XEV_DEAD"))
        // deadlock event
        ...
    else if(!strcmp(aCounter.name, "XEV_TOUT"))
        // timeout event
        ...
}
```

```

else if(!strcmp(aCounter.name, "XEV_EDM"))
    // EDM pool full event
    ...

else if(!strcmp(aCounter.name, "XEV_AUTH"))
    // authentication failure event
    ...

/* free internal used memory of counter */
deleteCounter(aCounter);

/* do not forget to free output data area */
pmFreeMem(data);

/* reset pos to beginning of data stream */
pos = 0;
}

// log off and disconnect (stops also event exception processing)
...

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
4	6	Authorization exit returned no data.
4	7	Severe error in authorization exit.
4	1043	Exception processing not active.
8	1	Internal Data Collector error.
8	8	DB2 request failed, no data. See the console log for more information.
8	42	The function is not available. A DB2 PM license is required for this function.
8	1000	Internal Data Collector error.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	Parameter exceptionNo is greater than 500.
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Stop Exception Processing

Function Call

```

pmReturnCodes pmStopExceptionProc (pmHost*      handle,
                                   char*        workProfile,
                                   pmExceptionType type)

```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to stop event exception processing for a user specified by parameter **workProfile**.

After successful completion of this function, the **pmFetchExceptions()** function cannot be used anymore for this user.

This function should be used only if event exception processing was started for this user, otherwise the Data Collector returns a warning.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **type** (input)

The type of exception processing to be stopped. Specify **EVENT**.

Example

```
#include "pmExcpProc.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost      myHandle;
char        workProfile[] = "PMUSER      DB2PM      10.0.0.1:0001  ";
pmReturnCodes error;

// connect to data collector and log on
...

// stop event exception processing
error = pmStopExceptionProc(&myHandle, workProfile, EVENT);

// log off and disconnect
...
```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
4	1043	Exception processing not active.
8	1	Internal Data Collector error.
8	1000	Internal Data Collector error.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	type must be EVENT .

See also "Common Return Codes and Reason Codes" on page 10, if required.

Executing DB2 Commands

DB2 commands can be executed by means of the DB2 PM API if the SAF user ID issuing the DB2 command has sufficient DB2 privileges. The Data Collector passes the DB2 command over to the instrumentation facility interface (IFI) for execution and returns the command response to the application.

Execute DB2 Command

Function Call

```
pmReturnCodes pmExecDB2Command (pmHost*   handle,
                                char*      workProfile,
                                char*      command,
                                char**     response)
```

Header File

pmExecDB2Command.h

Description

This function requests the Data Collector to execute a DB2 command and to return the command response for a user specified by parameter **workProfile**.

All DB2 commands are allowed, except you cannot request the Data Collector to start or to stop DB2.

The API allocates an output data area of up to 1 MB to store the command response. If the amount of data returned exceeds 1 MB, data is truncated, and the function returns a warning. The output data area holding the command response remains allocated until it is freed by the application.

Parameters

1. **handle** (input)

The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.

3. **command** (input)

The DB2 command to execute. Precede the command by a hyphen "-". The command can be written in upper- or lowercase, or in mixed case.

4. **response** (output)

Pointer to the output data area in memory where the result of the DB2 command is stored. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Example

```
#include "pmExecDB2Command.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdlib.h>
...

pmHost      myHandle;
char        workProfile[] = "PMUSER          DB2PM          10.0.0.1:0001 ";
char*       response = NULL;
pmReturnCodes error;

// connect to data collector and log on
...

// execute command to display bufferpools
error = pmExecDB2Command(&myHandle, workProfile,
                        "-DISPLAY BUFFERPOOL(*)", &response);

// work with output
...

// don't forget to free memory for DB2 response
if(response) pmFreeMem(response);

// log off and disconnect
...
```

Return Codes and Reason Codes

Return Code (Hex)	Reason Code (Hex)	Description
4	1	No data returned by DB2.
4	2	Command response contains more than 1 MB of data. Data is truncated.
4	3	Request resulted in a DB2 abend.
4	D	DB2 command failed. Possible reasons are: <ul style="list-style-type: none">• Command authorization failure• Command processor abend• Command syntax error• Command output limit being exceeded Response might be truncated.
8	1	Internal Data Collector error.
8	8	DB2 request failed, no data is returned. See the console log for more information.
8	11	Internal Data Collector error.
8	22	Internal Data Collector error.

See also “Common Return Codes and Reason Codes” on page 10, if required.

Saving and Retrieving User Data

The Data Collector provides a central storage area of 1 MB for each user specified by a work profile. You can use this storage area to store and retrieve whatever data you want. The intention for this storage area is to hold user-specific data to support mobile users of the application. When mobile users disconnect from the Data Collector at one workstation and reconnect to it at another workstation without logging off, they need to have a workstation-independent storage to hold, for example, their current application settings.

A 1-MB central storage area has the following characteristics:

- It is created when a user specified by a work profile logs on to the Data Collector.
- It is initialized to X'00.
- It is released when:
 - The user specified by a work profile logs off from the Data Collector.
 - The Data Collector is stopped.
 - An SAF user or SAF group is purged by a DB2 PM operator command.
- It is divided into 256 chunks of 4-KB buffers. Each buffer can be addressed by buffer numbers of 1 to 256.

The API provides two functions to use these buffers:

- The **pmSaveUserData()** function saves a block of data up to 4 KB in a buffer addressed by a buffer number of 1 to 256.
- The **pmGetUserData()** function retrieves a block of data from a buffer addressed by a buffer number of 1 to 256. Alternatively you can use this function to retrieve the content of all 256 buffers by specifying a buffer number of 0.

Buffers are bound to a work profile. Both functions must be assigned to a work profile when they are called.

Data is saved in a buffer as is. No conversion takes place. However, if less than 4 KB of data is saved, returned data is filled with 0x00 up to a size of 4 KB.

When a **pmGetUserData()** function is called, the API allocates an output data area of 4 KB (or 1 MB, if the contents of all buffers are retrieved) in the workstation's memory to store the returned data. The output data area holding the returned data remains allocated until it is freed by the application.

Save User Data

Function Call

```
pmReturnCodes pmSaveUserData(pmHost*      handle,
                              char*        workProfile,
                              char*        data,
                              unsigned int  length,
                              unsigned short bufferNo)
```

Header File

pmUserData.h

Description

This function requests the Data Collector to save user-specific application data in a buffer in the Data Collector addressed by buffer number **bufferNo**.

Parameters

1. **handle** (input)
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)
A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.
3. **data** (input)
Pointer to the data in the workstation's memory to be saved.
4. **length** (input)
Length (in number of bytes) of the data area to save (up to 4 KB).
5. **bufferNo** (input)
Number of the Data Collector buffer. This must be a number from 1 to 256.

Example

```
#include "pmUserData.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost      myHandle;
char        workProfile[] = "PMUSER      DB2PM      10.0.0.1:0001 ";
char        *userData = ...;
pmReturnCodes  error;

// connect to data collector and log on
...

// store data in buffer
```

```

error = pmSaveUserData(&myHandle, workProfile, userData, strlen(userData), 1);

// log off and disconnect
...

```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	22	Internal Data Collector error. Request rejected.
8	42	The function is not available. A DB2 PM license is required for this function.
8	154C	Internal Data Collector error. Request rejected.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	Incorrect value for parameter length or bufferNo .
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Get User Data

Function Call

```

pmReturnCodes pmGetUserData(pmHost*      handle,
                             char*        workProfile,
                             unsigned short bufferNo,
                             char**      data)

```

Header File

pmUserData.h

Description

This function requests the Data Collector to retrieve user-specific application data from the Data Collector that was previously saved with the **pmSaveUserData()** function.

Either the content of a buffer addressed by buffer number **bufferNo** is retrieved, or the contents of all 256 buffers is retrieved from the Data Collector.

Parameters

1. **handle** (input)
The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)
A user's work profile for which this function is to be executed. The user was specified by the **pmLogOn()** function.
3. **bufferNo** (input)
The number of a 4-KB buffer from which data is to be retrieved. If **bufferNo** is set to 0, the data of all 256 buffers is retrieved.
4. **data** (output)
Pointer to the output data area in memory. The memory (4 KB or 1 MB) is allocated by the DB2 PM API. Note that each buffer content is filled with 0x00,

if less than 4 KB of data was saved. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Example

```
#include "pmUserData.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost      myHandle;
char        workProfile[] = "PMUSER      DB2PM      10.0.0.1:0001  ";
char        *storedUserData;
pmReturnCodes  error;

// connect to data collector and log on
...

// store data in buffer 1
...

// work
...

// get stored data
error = pmGetUserData (&myHandle, workProfile, 1, &storedUserData);

// work with received data
...

// do not forget to free memory if no longer used
pmFreeMem(storedUserData);

// log off and disconnect
...
```

Return Codes and Reason Codes

Return Code ² (Hex)	Reason Code ² (Hex)	Description
8	1	Internal Data Collector error.
8	22	Internal Data Collector error. Request rejected.
8	42	The function is not available. A DB2 PM license is required for this function.
8	154D	Internal Data Collector error. Request rejected.
PM _ APIERROR	PM _ INCORRECT _ PARAMETER	Incorrect value for parameter length or bufferNo .
See also "Common Return Codes and Reason Codes" on page 10, if required.		

Parsing Data

Introduction to Parsing

The following DB2 PM API functions, described in "Processing DB2 Exception Events" on page 60, return data streams with complex and varying data structures:

- **pmGetEventExceptionLog()**
- **pmGetEventDetails()**

- **pmFetchExceptions()**

These structures vary with the type of event requested and the number of exceptions being monitored.

A data stream consists of a pattern of fixed and variable information units that are called tokens. For example, the **pmGetEventExceptionLog()** function returns a data stream that starts as following:

Output Data Area for pmGetEventExceptionLog()

▶—*total length of output area*—CNT_X number—DATEATIM *date and time*————▶

Here, CNT_X *number* and DATEATIM *date and time* represent two tokens.

The Parsing Functions

The API provides a set of functions that lets you parse a data stream on a token basis after it is stored in memory:

- **nextToken()** returns a counter represented by the current token in the data stream and increments a pointer to point to the next token.
- **nextTokenValue()** tests the current token in the data stream for a specified counter name, extracts and stores the counter value, and increments a pointer to the next token.
- **skipToken()** tests whether the current token is valid and increments a pointer to the next token.
- **testToken()** tests whether the current token represents a specified counter name. (This function does not increment a pointer.)
- **deleteCounter()** is used together with the **nextToken()** function and frees the memory area where the counter structure for a specified counter is stored.

These functions have some common characteristics:

- The **nextToken()**, **nextTokenValue()**, and **skipToken()** functions require the specification of a handle. This ensures that the correct code page is used by these functions when they perform their tests. You should specify one of the handles that were used with those functions that returned the data stream (**pmGetEventExceptionLog()**, **pmGetEventDetails()**, **pmFetchExceptions()**).
- You do not need to specify a work profile because these functions do not communicate with the Data Collector.
- The data stream in the output data area starts with a 4-byte length field, which shows the length (in number of bytes) of the output data area without the preceding 4-byte length field. The parsing functions require that you set a pointer (parameter **tokenBlock**) *after* the 4-byte length field. If you use the pointer of the function that returned the data as a reference, add four bytes to that pointer value to skip the length field.
- All parsing functions (except the **testToken()** function) increment a position pointer to point to the next token in a data stream before a function finishes. A subsequent parsing function uses this position pointer as input parameter to address the adjacent token. To start parsing a data stream you should set the position pointer to 0.

The **testToken()** function only tests for a specified counter name and returns a *true* or *false* condition. The position pointer is not incremented to allow one of the other parsing functions to work with this token.

The Counter Structure

A counter in the data stream is represented as a structure identical to the one shown in “The Counter Structure” on page 38. Note that the parsing functions do not use structure member *attribute* to validate a counter value. The **nextToken()** function uses the content of the *counterID* member to test for a valid counter. The other parsing functions return with a return code of PM_OK or PM_FAILED.

Working with Parsing Functions

Proper use of the parsing functions ensures that an application using these functions can also handle unexpected tokens in the returned data streams.

Unexpected tokens in a data stream can occur, for example, if users of an existing application install API enhancements (new versions, releases, or PTFs) that return additional tokens in the data stream.

You should consider the following coding sequences to ensure that an application continues to work even if the data stream changes to accommodate new functions.

If you expect a unique counter name in one of the tokens in the data stream, the following sequence will let you identify the token and retrieve the counter value:

1. Use the **testToken()** function and specify as parameter the counter name you are expecting in the data stream.

This function does not increment the position pointer.

2. If the token does not contain the counter name you have specified, the **testToken()** function returns PM_FAILED. Use the **skipToken()** function to increment the position pointer to the next token in the data stream. Continue with step 1 to test the next token for the specified name (until the end of the data stream is reached).
3. If the token contains the counter name you have specified, the **testToken()** function returns PM_OK. Use the **nextTokenValue()** function to retrieve the counter value from the token. Continue as required by the application’s logic.

However, if you have no unique expectation about a counter name but need to react on a variety of counters, apply the following program structure. The **testToken()** function checks the current token for one of the specified counter names; the **nextTokenValue()** function retrieves the corresponding counter value. If the current token does not contain one of the expected counter names, the **skipToken()** function increments the position pointer to the next token.

```
Do While not end of data stream
  {IF testToken() = counter_name_1          /* Does not increment position pointer */
    nextTokenValue() for counter_name_1 /* Retrieve counter value 1 from token */
    ... program logic for counter 1 ...
  ELSE
  {IF testToken() = counter_name_2          /* Does not increment position pointer */
    nextTokenValue() for counter_name_2 /* Retrieve counter value 2 from token */
    ... program logic for counter 2 ...
  ELSE
  {IF testToken() = counter_name_n          /* Does not increment position pointer */
    nextTokenValue() for counter_name_n /* Retrieve counter value n from token */
    ... program logic for counter n ...
  ELSE
    skipToken()                             /* Increments position pointer          */
                                             /* Continue with next token             */
End of Do While
```

Do not use **nextToken()** together with **skipToken()** as an alternative. Both functions increment the position pointer upon return, and you would miss every other token.

Get Token

Function Call

```
pmCounter nextToken (pmHost*      handle,  
                    char*        tokenBlock,  
                    unsigned int* pos,  
                    unsigned int length,  
                    pmBoolean    attr)
```

Header File

pmParser.h

Description

This function returns a counter represented by the current token in the data stream and increments a pointer to point to the next token.

A counter is valid if member *counterID* in the counter structure contains a value greater than 0.

Use the **deleteCounter()** function to release the memory that was allocated by a **nextToken()** function call.

Parameters

1. **handle** (input)
Specify one of the handles that were used by the functions that created the returned data stream.
2. **tokenBlock** (input)
Pointer to the output data area (following the *total length of output area* field).
3. **pos** (input/output)
The actual position pointer of the parser in the data stream. This position pointer must be initialized with 0 when you start parsing a data stream. After successful completion of this function, **pos** points to the next token.
4. **length** (input)
The length (in number of bytes) of the output data area (not including the *total length of output area* field).
5. **attr** (input)
Reserved. Specify FALSE.

Get Token Value

Function Call

```
pmBoolean nextTokenValue(pmHost*      handle,  
                        char*        tokenBlock,  
                        unsigned int* pos,  
                        unsigned int length,  
                        char*        name,  
                        char*        storage)
```

Header File

pmParser.h

Description

This function tests the current token in the data stream for a counter name specified by parameter **name**, stores the counter value in memory location **storage**, and increments a pointer to the next token.

Upon successful execution, the function returns `PM_OK`.

If the function returns `PM_FAILED`, the data stream is not valid, or the specified counter is not found at this position.

Parameters

1. **handle** (input)
Specify one of the handles that were used by the functions that created the returned data stream.
2. **tokenBlock** (input)
Pointer to the output data area (following the *total length of output area* field).
3. **pos** (input/output)
The actual position pointer of the parser in the data stream. This position pointer must be initialized with 0 when you start parsing a data stream. After successful completion of this function, **pos** points to the next token.
4. **length** (input)
The length (in number of bytes) of the output data area (not including the *total length of output area* field).
5. **name** (input)
The counter name expected at this position in the data stream.
6. **storage** (output)
Pointer to the memory location where the counter value should be stored. The memory area must be already allocated. Member *length* in the counter structure shows the required length.

Counters of type `PMVARCHAR` and `PMREPBLOCK` are not supported, because the counter length is not known before parsing.

Skip Token

Function Call

```
pmBoolean skipToken (pmHost*   handle,  
                    char*     tokenBlock,  
                    unsigned int* pos,  
                    unsigned int length,  
                    pmBoolean attr)
```

Header File

`pmParser.h`

Description

This function tests whether the current token is valid and increments a pointer to the next token.

This function skips unconditionally to the next token. Use this function to skip a token in the data stream you are not interested in, or to detect the end of a data stream. As a side effect, this function tests the current token for a valid data stream.

If the data stream is valid, the function returns `PM_OK`, else it returns `PM_FAILED`.

Parameters

1. **handle** (input)
Specify one of the handles that were used by the functions that created the returned data stream.

2. **tokenBlock** (input)
Pointer to the output data area (following the *total length of output area* field).
3. **pos** (input/output)
The actual position pointer of the parser in the data stream. This position pointer must be initialized with 0 when you start parsing a data stream. After successful completion of this function, **pos** points to the next token.
4. **length** (input)
The length (in number of bytes) of the output data area (not including the *total length of output area* field).
5. **attr** (input)
Reserved. Specify FALSE.

Test Token

Function Call

```
pmBoolean testToken (char*      counterName,
                    char*      tokenBlock,
                    unsigned int pos,
                    unsigned int length)
```

Header File

pmParser.h

Description

This function tests whether the current token represents a specified counter name. (This function does not increment a pointer.)

If the specified counter name is found, the function returns PM_OK, else it returns PM_FAILED.

Parameters

1. **counterName** (input)
The counter name expected at this position in the data stream.
2. **tokenBlock** (input)
Pointer to the output data area (following the *total length of output area* field).
3. **pos** (input)
The actual position pointer of the parser in the data stream. This position pointer must be initialized with 0 when you start parsing a data stream. After successful completion of this function, **pos** *does not* point to the next token. The position pointer remains unchanged.
4. **length** (input)
The length (in number of bytes) of the output data area (not including the *total length of output area* field).

Delete Counter

Function Call

```
void deleteCounter (pmCounter counter)
```

Header File

pmCounter.h

Description

This function frees the memory area that was allocated by a `nextToken()` function call.

The output data area where the returned data stream is stored in memory is not freed.

Parameters

1. **counter** (input)
The counter to be freed.

Converting and Adjusting Dates and Times

Introduction to Date and Time Functions

The Data Collector returns date and time values in a format that must be converted before these values can be processed in a C language program. Further, adjustments to dates and times may be required because the Data Collector returns this data as provided by DB2. DB2 stores dates and times as universal times.⁵

If the application program operates in another time zone than the Data Collector, further date and time adjustment is required.

Format Conversions

Several API functions return counter values from the Data Collector to the application program that represent date and time information. Also, several API functions require input parameters to be sent to the Data Collector that represent date and time information.

A date is usually a time stamp that consists of a calendar date and time (for example, a time stamp of when a history snapshot was taken). A time is usually an information expressed in hours, minutes, and seconds, with no reference to a calendar date (for example, a counter that reflects an accumulated waiting time of a DB2 process).

Dates and times are presented in different formats:

- The Data Collector represents dates and times in the Store Clock format. More exact, counters of type "Date" or "Time" are presented in Store Clock format. See *ESA/390 Principles of Operation* about the Store Clock format, if necessary.
The Data Collector returns affected counters as data type `pmTOD`, which is defined as:

```
typedef char pmTOD[8];
```
- The application program, if written in the C programming language, uses a format that conforms to the ANSI-C standard (**`time_t`** data type).

The API provides two functions that convert counter values from one format to the other:

- The **`tod2time()`** function converts a counter value from Store Clock format to `time_t` format.
You use this function if you want to further process date and time information in your C language program, for example, to display, print, or change a date or time.
- The **`time2tod()`** function converts a counter value from `time_t` format to Store Clock format.

You use this function, for example, to edit date and time information as an input parameter for an API function.

Local Time Adjustment

Dates and times in DB2 instrumentation data is stored internally as universal time (UT)⁵, no matter in which geography a DB2 system operates. Whenever a program retrieves DB2 instrumentation data, and requires date and time information to be expressed in local time, an adjustment is required.

For example, when a DB2 system at New York, U.S.A., writes an event record at 11:00 a.m. local time, the corresponding entry is 16:00 universal time, because the time difference between New York and Greenwich is -5 hours. If a program operating at New York retrieves this 16:00 universal time entry, it must adjust the time by -5 hours to show again the correct time of the event.

If the application program retrieves date- and time-based counters from the Data Collector, these counters are also expressed in universal time. The Data Collector does not perform a conversion to local times. If the application program requires to show or use a date or time in local time, it must adjust the universal time.

The following API functions ease the calculation of local times:

- The **pmGetInfo()** function provides the time difference (counter QR4TID) between the local time of the host processor (where DB2 and the Data Collector operate) and universal time.
- The **pmAddTOD()** and **pmSubTOD()** functions add and subtract two time values.

Note: If you use the ANSI-C **time()** function to create time stamps for use as input parameters:

- First, adjust the local workstation time to universal time.
- Second, convert the adjusted universal time to the Store Clock format.

Time Zone Adjustment

The Data Collector and the application program may operate in different time zones. For example, suppose that the application operates at Berlin, Germany (UT +1 hour), and monitors a DB2 subsystem at New York, U.S.A. (UT -5 hours). A DB2 event record written at 11:00 a.m. New York local time gets a time stamp of 16:00 universal time. To obtain the time of this event in Berlin local time:

1. Adjust the time stamp (UT) of the event by -5 hours to obtain New York local time (16:00-05:00=11:00).

Use the information returned by the **pmGetInfo()** function, counter QR4TID, for this adjustment.

2. Adjust New York local time to Berlin local time (11:00+06:00=17:00).

This adjustment (+6 hours) is, of course, specific to this example.

For further information about time zones, see the documentation for function **tzset()** or **_tzset()** in your compiler run-time library reference.

Convert Store Clock Format to time_t Format

Function Call

```
void tod2time (pmTOD  aTOD,  
              time_t* aTime)
```

Header File

pmTOD.h

Description

This function converts a time from Store Clock format to **time_t** format. The **time_t** format contains the seconds since 01-01-1970 00:00:00. Milliseconds from the Store Clock format are not supported and truncated.

Parameters

1. **aTOD** (input)
The time value to convert, in Store Clock format.
2. **aTime** (output)
The converted time value, in **time_t** format.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "pmConnect.h"
#include "pmGetInfo.h"
#include "pmTOD.h"

int main(void)
{
    pmHost      myHandle;
    pmReturnCodes error;
    char        *workprofile = "PMUSER  GROUPID  PROFILEID      TERMINALID      ";
    pmHashTable result;
    pmCounter*  aCounter;
    pmCursor    aCursor;
    pmHashTable* DCInfo;
    pmTOD       timeDifference;
    pmTOD       hostTime;
    time_t      time;
    struct tm*  timeStruct;

    /* connect to data collector */
    error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
    if(error.returnCode)
    {
        printf("pmConnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
        exit(-1);
    }

    /* initialize timezone setting on workstation */
    tzset();

    /* prepare result area */
    clearHashTable(&result);

    /* get time difference and current time from DC */
    error = pmGetInfo(&myHandle, workprofile, result);
    if(error.returnCode)
    {
        printf("pmGetInfo - Error [%d/%d]\n", error.returnCode, error.reasonCode);
        exit(-1);
    }
    else
    {
        /* first locate repeating block for DC info */
        aCounter = pmGetCounter(result, "REPDCINF");
        if(aCounter != NULL)
        {
```

```

    /* access general info */
    aCursor = initCursor(*aCounter);
    DCInfo = getRepBlockItem(aCursor);

    /* get time difference between local time and GMT on host */
    aCounter = pmGetCounter(*DCInfo, "QR4TID");
    memcpy(timeDifference, aCounter->value, sizeof(pmTOD));

    /* get current time (GMT) on host */
    aCounter = pmGetCounter(*DCInfo, "QR4TIME");
    memcpy(hostTime, aCounter->value, sizeof(pmTOD));

    /* adjust host time to local time by adding time difference */
    pmAddTOD(hostTime, timeDifference, hostTime);

    /* convert Store Clock Format into time_t format */
    tod2time(hostTime, &time);

    /* convert time_t format into tm format (without time zone adjustment) */
    timeStruct = gmtime(&time);

    /* year is relative to 1900 and month starts with 0 -> adjust */
    printf("Local time on host: %2.2d:%2.2d:%2.2d on %4.4d/%2.2d/%2.2d\n",
           timeStruct->tm_hour, timeStruct->tm_min, timeStruct->tm_sec,
           timeStruct->tm_year + 1900, timeStruct->tm_mon + 1,
           timeStruct->tm_mday);
}

/* free memory for DC info */
freeHashTable(result);
}

/* disconnect */
error = pmDisconnect(&myHandle);
if(error.returnCode)
{
    printf("pmDisconnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
}

return(0);
}

```

Convert time_t Format to Store Clock Format

Function Call

```
void time2tod (time_t aTime,
              pmTOD aTOD)
```

Header File

pmTOD.h

Description

This function converts a time from **time_t** format to Store Clock format. The **time_t** format contains the seconds since 01-01-1970 00:00:00.

Parameters

1. **aTime** (input)
The time value to convert, in **time_t** format.
2. **aTOD** (output)
The converted time value, in Store Clock format.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "pmConnect.h"
#include "pmGetInfo.h"
#include "pmTOD.h"

int main(void)
{
    pmHost      myHandle;
    pmReturnCodes error;
    char        *workprofile = "PMUSER  GROUPID PROFILEID      TERMINALID      ";
    pmHashTable result;
    pmCounter*  aCounter;
    pmCursor    aCursor;
    pmHashTable* DCInfo;
    pmTOD       timeDifference;
    pmTOD       hostTime;
    time_t      time;
    struct tm*   timeStruct;
    struct tm    newTime;

    /* connect to data collector */
    error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
    if(error.returnValue)
    {
        printf("pmConnect - Error [%d/%d]\n", error.returnValue, error.reasonCode);
        exit(-1);
    }

    /* initialize timezone setting on workstation */
    tzset();

    /* prepare result area */
    clearHashTable(&result);

    /* get time difference and current time from DC */
    error = pmGetInfo(&myHandle, workprofile, result);
    if(error.returnValue)
    {
        printf("pmGetInfo - Error [%d/%d]\n", error.returnValue, error.reasonCode);
        exit(-1);
    }
    else
    {
        /* first locate repeating block for DC info */
        aCounter = pmGetCounter(result, "REPDCINF");
        if(aCounter != NULL)
        {
            /* access general info */
            aCursor = initCursor(*aCounter);
            DCInfo = getRepBlockItem(aCursor);

            /* get time difference between local time and GMT on host */
            aCounter = pmGetCounter(*DCInfo, "QR4TID");
            memcpy(timeDifference, aCounter->value, sizeof(pmTOD));

            /* convert local timestamp into Store Clock Format */
            /* preset input data: 15:23:01 2000/05/23 (HH:MM:SS YYYY/MM/DD) */
            newTime.tm_hour = 15;
            newTime.tm_min  = 23;
            newTime.tm_sec  = 01;
            newTime.tm_year = 2000 - 1900; /* adjust year relative to 1900 */
            newTime.tm_mon  = 5 - 1;      /* adjust month to start at 0 */
            newTime.tm_mday = 23;
        }
    }
}
```



```

    /* convert tm structure into time_t structure using time zone */
    /* setting on workstation (result will be in GMT depending on */
    /* implementation of mktime() function) */
    time = mktime(&newTime);

    /* convert time_t structure to Store Clock Format */
    time2tod(time, hostTime);

    /* if mktime() did not adjust the time to GMT the Store Clock */
    /* format needs to be adjusted using pmSubTOD: */
    /* pmSubTOD(hostTime, timeDifference, hostTime); */

    /* use converted time to request history data, etc. */
    /* ... */
}

/* free memory for DC info */
freeHashTable(result);
}

/* disconnect */
error = pmDisconnect(&myHandle);
if(error.returnCode)
{
    printf("pmDisconnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
}

return(0);
}

```

Add and Subtract in Store Clock Format

Function Call

```

void pmAddTOD(pmTOD tod1, pmTOD tod2, pmTOD result)
void pmSubTOD(pmTOD tod1, pmTOD tod2, pmTOD result)

```

Header File

pmTOD.h

Description

These functions add and subtract two time values that are given in Store Clock format.

Parameters

1. **tod1** (input)
tod2 (input)
Time values to add or subtract, in Store Clock format.
2. **result** (output)
For **pmAddTOD()**: **tod1** + **tod2**, in Store Clock format.
For **pmSubTOD()**: **tod1** – **tod2**, in Store Clock format.

Miscellaneous API Functions

This section describes API functions that are used in several examples. They might be helpful to reduce your programming effort.

Hash Table Functions

clearHashTable()

```
void clearHashTable (pmHashTable table)
```

This function initializes the memory area for a hash table. Use this function before you store returned data in the output data area.

initCursor()

```
pmCursor initCursor (pmCounter aCounter)
```

This function initializes a cursor, which is used to parse through all items of a repeating block.

```
pmCursor cursor = initCursor(pmGetCounter(result, "REPSTBUF"));  
pmHashTable item;
```

```
while(!endOfBlock(cursor))  
{  
    item = getRepBlockItem(cursor);  
  
    // process data in this repeating block  
    ...  
  
    setToNext(cursor);  
}
```

setToNext()

```
void setToNext (pmCursor* aCursor)
```

This function sets the cursor to the next hash table found in the repeating block.

endOfBlock()

```
int endOfBlock (pmCursor aCursor)
```

This function checks whether the end of a repeating block is reached.

getRepBlockItem()

```
pmHashTable* getRepBlockItem (pmCursor aCursor)
```

This function gets the hash table to which cursor **aCursor** points.

freeHashTable()

```
void freeHashTable (pmHashTable table)
```

This function frees the memory of the output data area.

pmGetCounter()

```
pmCounter* pmGetCounter (pmHashTable table,  
char* name)
```

This function locates counter **name** in the hash table named **table**. If the counter is not found, the function returns NULL.

Memory Releasing Function

pmFreeMem()

```
void pmFreeMem (void* ptr)
```

This function releases memory that was allocated by an API function. Parameter **ptr** points to the memory area to be released.

Qualifier Functions

initQualifierList()

```
void initQualifierList (pmQualifierList* list);
```

This function initializes a qualifier list. Use this function before you add qualifiers to the list with the **addQualifier()** function.

addQualifier()

```
pmReturnCodes addQualifier (pmHost*          handle,  
                             pmQualifierList* list,  
                             char*          counterName,  
                             char*          counterValue);
```

This function adds qualifiers to a qualifier list that was created with the **initQualifierList()** function. Parameter **counterName** represents the qualification ID (the name of the counter to add to the list); parameter **counterValue** represents the corresponding counter value.

Appendix A. Field Table Summary

The field table contains a list of all data fields that are accessible through DB2 PM API functions. Because of the number of fields the information about it is not shown in this guide, but is provided as a text file that accompanies the DB2 PM API. The file resides in data set SDGOWS01 as member DGOKFLDS and has a record length of 80 bytes.

A short abstract of the field table (section General IDs) is shown next:

Field ID	Field Name	Field Type	Field Length	Description
0001	LNGTHID	Char	0002	Length of current block. In contrast to LNGTHID4, this is a 2-byte field. The value includes the length of LNGTHID itself.
0002	EYECAT	Char	0004	Eye catcher of current block.
0003	LNGTHID4	Int	0004	Length of current block. In contrast to LNGTHID, this is a 4-byte field. The value includes the length of LNGTHID4 itself.
0010	REPSTAT	Rep	0002	Start of statistics record. Contains the number of repetitions as value.
0011	REPSTDDF	Rep	0002	Start of DDF repeating block in statistics record. Contains the number of repetitions as value.
0012	REPSTBUF	Rep	0002	Start of buffer pool repeating block in statistics record. Contains the number of repetitions as value.
0013	REPSTGBF	Rep	0002	Start of group buffer pool repeating block in statistics record. Contains the number of repetitions as value.
0014	REPSTGLB	Rep	0002	Start of CF cache data repeating block in statistics record. Contains the number of repetitions as value.
⋮	⋮	⋮	⋮	⋮
0050	SNAPTIME	Date	0008	Time stamp of current snapshot.
⋮	⋮	⋮	⋮	⋮
0103	HISTINTV	Smint	0002	Number of history snapshots.
0104	HISTOFF	Smint	0002	Offset for history interval.

Column Meaning

Field ID

Unique identifier number of a counter. The field table is sorted by identifier number. Ranges of numbers might be reserved for future use.

Field Name

Symbolic name of a counter. These counter names are used in an application program to specify counters.

Field Type

Data type of the named counter.

Field Length

Length in bytes of the named counter.

Description

Short description of the purpose of the counter.

The entire field table is sorted by Field ID (column 1). Fields that describe a common topic are grouped together. You will find the following groups in the field table. Notice that fields or groups marked “for internal purposes only” should not be used.

- General IDs
General field IDs that describe the structure of the provided or returned data.
- Qualification IDs
Field IDs that qualify data. See *DB2 for OS/390 Administration Guide*, the appendix about IFI programming about qualifying.
- TOP/Sort IDs
- Statistics data:
 - Address space data
 - Instrumentation destination data
 - Instrumentation data
 - Subsystem services data
 - Command data
 - IFC checkpoint data
 - Log manager data
 - Distributed Data Facility (DDF) data
 - Distributed Data Facility (DDF) system data
 - SQL statement data
 - Bind data
 - Buffer manager data
 - Data manager control data
 - Lock usage data
 - EDM pool usage data
 - Group buffer pool usage data
 - Global locking data
 - CF cache structure data
- Collect report data IDs
Fields concerning Collect Report Data.
- Thread data:
 - Exception IDs
 - Thread Distributed Detail Data
- Exception IDs
- Statistics data:

- Statement cache
- Statement text
- System parameters block - SYSP
- Log initialization parameters block
- Archive initialization parameters block
- System parameters - SPRM
- List of VSAM catalog qualifiers
- Database start flags
- List of all databases
- Distributed Data Facility (DDF) start control information
- Group initialization parameters block
- DSNHDECP CSECT
- Buffer Manager group buffer pool attributes
- Buffer Manager dynamic pool attributes
- Thread data:
 - Current statement data
 - Current statement text
 - Current statement data
 - Current statement data
 - Current statement data
 - Instrumentation data
 - Buffer pool usage data
 - SQL statement data
 - Buffer pool usage data
 - Lock usage data
 - Correlation header data
 - Agent status data
 - Distributed header data
 - Distributed data
 - Distributed accounting data
 - Account code and Distributed Data Facility (DDF) data
 - IFI accounting data
 - Package data
 - Global locking data
 - Group buffer pool data
 - Locked resources
 - Locked resources
 - Locked resources
 - Locked resources
 - Locked resources - details

A cross-reference list follows the field table that maps field names to field IDs.

Appendix B. Sample Traces

This section shows three samples of trace data. "Using the DB2 PM API Trace Facility" on page 14 describes how to produce traces.

Sample Connection Trace

```
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_NETWORKDOWN      | 10050 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_TIMEOUT          | 10053 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_HOSTUNREACHABLE  | 10060 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_HOSTDOWN         | 10064 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_DCNOTAVAILABLE   | 10061 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_SOCKETDESCR_INVALID | 10038 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_SOCKET_NOTCONNECTED | 10038 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : | PM_CONNECTION_ABORTED | 10058 |
(CONNECTION) 15:20:22, PID 324 : +-----+
(CONNECTION) 15:20:22, PID 324 : Request input area data for request on socket 0x00B8 :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Received request header
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Receiving data for request...
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Request on socket b8hex received!
(CONNECTION) 15:20:22, PID 324 : Request input area data for request on socket 0x00B8 :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Received request header
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Receiving data for request...
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Request on socket b8hex received!
(CONNECTION) 15:20:22, PID 324 : Request input area data for request on socket 0x00B4 :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Received request header
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Request on socket b8hex received!
(CONNECTION) 15:20:22, PID 324 : Request input area data for request on socket 0x00B4 :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Received request header
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Request on socket 84hex received!
(CONNECTION) 15:20:22, PID 324 : Request input area data for request on socket 0x00B8 :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Received request header
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Receiving data for request...
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Request on socket b8hex received!
(CONNECTION) 15:20:22, PID 324 : Request input area data for request on socket 0x00B8 :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest :
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Received request header
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Receiving data for request...
(CONNECTION) 15:20:22, PID 324 : ReceiveRequest : Request on socket b8hex received!
```

Sample Command Trace

```
(COMMAND) 15:20:22, PID 324 :
(COMMAND) 15:20:22, PID 324 : *****
(COMMAND) 15:20:22, PID 324 : *
(COMMAND) 15:20:22, PID 324 : * Application Programming Interface (R6V1MV3) *
(COMMAND) 15:20:22, PID 324 : * ----- *
(COMMAND) 15:20:22, PID 324 : * *
(COMMAND) 15:20:22, PID 324 : * IBM DB2 UDB Performance Monitor for OS/390 V6 *
(COMMAND) 15:20:22, PID 324 : * *
(COMMAND) 15:20:22, PID 324 : * COPYRIGHT : 5645-DB2 (C) Copyright IBM Corp. 1999 *
(COMMAND) 15:20:22, PID 324 : * LICENSED MATERIALS - PROPERTY OF IBM *
(COMMAND) 15:20:22, PID 324 : * SEE COPYRIGHT INSTRUCTIONS, G120 - 2083 *
(COMMAND) 15:20:22, PID 324 : * *
(COMMAND) 15:20:22, PID 324 : *****
(COMMAND) 15:20:22, PID 324 :
(COMMAND) 15:20:22, PID 324 : pmConnect() : Connecting to 9.164.172.191:6653 ...
(COMMAND) 15:20:22, PID 324 : pmConnect() : initializing network ...
(COMMAND) 15:20:22, PID 324 : pmConnect() : Network initialized.
(COMMAND) 15:20:22, PID 324 : pmConnect() : port specified as integer ...
(COMMAND) 15:20:22, PID 324 : pmConnect() : hostname specified as dotted decimal integer (ip addr) ...
(COMMAND) 15:20:22, PID 324 : pmConnect() : Creating socket ...
(COMMAND) 15:20:22, PID 324 : pmConnect() : Socket created.
(COMMAND) 15:20:22, PID 324 : pmConnect() : Connecting ...
(COMMAND) 15:20:22, PID 324 : pmConnect() : Connect successful.
(COMMAND) 15:20:22, PID 324 : pmConnect() : Loading code page from host ...
(COMMAND) 15:20:22, PID 324 : loadCodePage() : Receiving return area successful!
(COMMAND) 15:20:22, PID 324 : loadCodePage() : Now parsing response ...
(COMMAND) 15:20:22, PID 324 : loadCodePage() : Host uses EBCDIC code page 500.
(COMMAND) 15:20:22, PID 324 : 00010203372D2E2F1605250B0C0D0E0F101112133C3D322618191C27071D1E1F404F7F7B586C ...
(COMMAND) 15:20:22, PID 324 :
(COMMAND) 15:20:22, PID 324 : pmConnect() : Loading field table from dc ...
(COMMAND) 15:20:22, PID 324 : loadFieldTableFromDC() : Receiving return area successful !
(COMMAND) 15:20:22, PID 324 : loadFieldTableFromDC() : Now parsing response ...
(COMMAND) 15:20:22, PID 324 : loadFieldTableFromDC() : 1907 counters loaded.
(COMMAND) 15:20:22, PID 324 : loadFieldTableFromDC() : field table load complete.
(COMMAND) 15:20:26, PID 324 : pmLogon() : Starting logon for user STI DB2PMTEST1 AAA001 ...
(COMMAND) 15:20:26, PID 324 : pmLogon() : Sending request ...
(COMMAND) 15:20:26, PID 324 : pmLogon() : Logon failed! RC/RS [4/5]
```

Appendix C. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100

70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, or other countries, or both:

AIX
CICS
DATABASE 2
DB2
IBM

MVS
OS/390
RACF
VisualAge

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

- IBM DB2 Performance Monitor for OS/390
Version 6 Online Monitor User's Guide,
SC26-9168*
- IBM DB2 Performance Monitor for OS/390
Version 6 Batch User's Guide, SC26-9167*
- IBM DB2 Performance Monitor for OS/390
Version 6 Command Reference, SC26-9166*
- IBM DB2 Performance Monitor for OS/390
Version 6 Messages, SC26-9169*
- IBM DB2 Performance Monitor for OS/390
Version 6 Using the Workstation Online Monitor,
SC26-9170*
- IBM DB2 Performance Monitor for OS/390
Version 6 Installation and Customization,
SC26-9171*
- IBM DB2 Performance Monitor for OS/390
Version 6 General Information, GC26-9172*
- Program Directory for IBM DB2 UDB Server for
OS/390 DB2 Performance Monitor DB2
Workstation Analysis and Tuning Version 6,
GI10-8183*
- IBM DB2 Universal Database Server for OS/390
Version 6 Administration Guide, SC26-9003*
- IBM DB2 Universal Database Server for OS/390
Version 6 Command Reference, SC26-9006*
- IBM DB2 Universal Database Server for OS/390
Version 6 Application Programming and SQL
Guide, SC26-9004*
- IBM DB2 Universal Database Server for OS/390
Version 6 Messages and Codes, GC26-9011*
- OS/390 Security Server (RACF) - Introduction,
GC28-1912*
- OS/390 MVS System Codes, GC28-1780*
- ESA/390 Principles of Operation, SA22-7201*
- TCP/IP for MVS: Application Programming
Interface Reference, SC31-7187*
- TCP/IP Tutorial and Technical Overview,
GG24-3376*

Index

A

addQualifier() 35, 101
API installation vi
API license vi
application name, definition in RACF 10
ASCII conversion 9

C

clearHashTable() 100
code page
 conversion 9
conversion
 ASCII 9
 code page 9
 EBCDIC 9
 format 96
 of ASCII and EBCDIC 15
 of user data 86
counter
 introduction to 29
counter structure 38

D

data type
 PMPARSEDREPBLOCK 38
date and time functions
 introduction to 94
deleteCounter() 93
delta processing 31

E

EBCDIC conversion 9
endOfBlock() 100
environment variable 14
exception processing
 introduction to 60

F

field table
 download with pmConnect() 15
 format 103
 in SDGOWS01 103
format
 Store Clock 94
 time_t 94
format conversion 94
freeHashTable() 100
function call description
 addQualifier() 35, 101
 clearHashTable() 40, 100
 deleteCounter() 93
 endOfBlock() 40, 100
 freeHashTable() 40, 100
 getRepBlockItem() 40, 100
 initCursor() 100
 initQualifierList() 35, 101

function call description (*continued*)

nextToken() 91
nextTokenValue() 91
pmAddTOD() 99
pmConnect() 15
pmDisconnect() 17
pmExecDB2Command() 83
pmFetchExceptions() 79
pmFreeMem() 100
pmGenPassticket() 21
pmGetCounter() 40, 100
pmGetEventDetails() 67
pmGetEventExceptionLog() 61
pmGetExceptionStatus() 77
pmGetHistory() 57
pmGetHistoryContents() 54
pmGetInfo() 22
pmGetSnapshot() 46
pmGetUserData() 87
pmInitializeStore() 42
pmLogOff() 27
pmLogOn() 19
pmQueryStores() 44
pmReleaseStore() 56
pmReset() 51
pmSaveUserData() 86
pmStartExceptionProc() 75
pmStopExceptionProc() 82
pmSubTOD() 99
setToNext() 100
skipToken() 92
testToken() 93
time2tod() 97
tod2time() 95

function call usage in examples

addQualifier() 35
clearHashTable() 24, 96
deleteCounter() 64, 81
endOfBlock() 24, 40
freeHashTable() 24, 96
getRepBlockItem() 24, 40
initCursor() 40
initQualifierList() 35
nextToken() 64, 81
nextTokenValue() 64
pmAddTOD() 96
pmConnect() 3, 6, 16, 18, 24
pmDisconnect() 3, 7, 18, 24
pmExecDB2Command() 84
pmFetchExceptions() 81
pmFreeMem() 45, 64, 74, 81, 84, 88
pmGenPassticket() 5, 6, 21
pmGetCounter() 24, 40, 96
pmGetEventDetails() 74
pmGetEventExceptionLog() 64
pmGetExceptionStatus() 78
pmGetInfo() 6, 24, 96, 98
pmGetUserData() 7, 88
pmLogOff() 6, 28
pmLogOn() 4, 6, 20, 21
pmQueryStores() 45

function call usage in examples

(*continued*)

pmSaveUserData() 7, 86
pmStartExceptionProc() 64, 74, 76, 81
pmStopExceptionProc() 83
pmSubTOD() 98
setToNext() 24, 40
skipToken() 64, 81
testToken() 64
time2tod() 98
tod2time() 24, 45, 96

function parameter

application, of pmGenPassticket() 21
aTime, of time2tod() 97
aTime, of tod2time() 96
aTOD, of time2tod() 97
aTOD, of tod2time() 96
attr, of nextToken() 91
attr, of skipToken() 93
bufferNo, of pmGetUserData() 87
bufferNo, of pmSaveUserData() 86
codePage, of pmConnect() 16
command, of
 pmExecDB2Command() 84
counter, of deleteCounter() 94
counterName, of testToken() 93
counterNo, of pmGetHistory() 58
counterNo, of pmGetSnapshot() 49
counterNo, of pmInitializeStore() 43
data, of pmFetchExceptions() 80
data, of pmGetEventDetails() 68
data, of
 pmGetEventExceptionLog() 62
data, of pmGetUserData() 87
data, of pmSaveUserData() 86
dir, of pmGetHistory() 58
eventTime, of
 pmGetEventDetails() 68
eventType, of
 pmGetEventDetails() 68
exceptionNo, of
 pmFetchExceptions() 80
fields, of pmGetHistory() 58
fields, of pmGetSnapshot() 48
fields, of pmInitializeStore() 43
from, of
 pmGetEventExceptionLog() 62
handle, of nextToken() 91
handle, of nextTokenValue() 92
handle, of pmConnect() 16
handle, of pmDisconnect() 18
handle, of
 pmExecDB2Command() 84
handle, of pmFetchExceptions() 80
handle, of pmGenPassticket() 21
handle, of pmGetEventDetails() 67
handle, of
 pmGetEventExceptionLog() 62
handle, of
 pmGetExceptionStatus() 77
handle, of pmGetHistory() 57

- function parameter (*continued*)
 - handle, of
 - pmGetHistoryContents() 55
 - handle, of pmGetInfo() 23
 - handle, of pmGetSnapshot() 47
 - handle, of pmGetUserData() 87
 - handle, of pmInitializeStore() 43
 - handle, of pmLogOff() 28
 - handle, of pmLogOn() 19
 - handle, of pmQueryStores() 45
 - handle, of pmReleaseStore() 56
 - handle, of pmReset() 52
 - handle, of pmSaveUserData() 86
 - handle, of pmStartExceptionProc() 76
 - handle, of pmStopExceptionProc() 83
 - handle, of skipToken() 92
 - host, of pmConnect() 16
 - id, of pmGetSnapshot() 48
 - id, of pmInitializeStore() 43
 - id, of pmReleaseStore() 56
 - id, of pmReset() 52
 - identification, of pmLogOn() 19
 - ifcidNo, of
 - pmGetHistoryContents() 55
 - ifcids, of pmGetHistoryContents() 55
 - info, of pmGetExceptionStatus() 77
 - info, of pmGetInfo() 23
 - interval, of
 - pmStartExceptionProc() 76
 - length 80
 - length, of nextToken() 91
 - length, of nextTokenValue() 92
 - length, of pmGetEventDetails() 74
 - length, of
 - pmGetEventExceptionLog() 64
 - length, of pmSaveUserData() 86
 - length, of skipToken() 93
 - length, of testToken() 93
 - mode, of pmGetSnapshot() 47
 - mode, of pmReset() 52
 - name, of nextTokenValue() 92
 - passticket, of pmGenPassticket() 21
 - pos, of nextToken() 91
 - pos, of nextTokenValue() 92
 - pos, of skipToken() 93
 - pos, of testToken() 93
 - qualifier, of pmGetHistory() 58
 - qualifier, of pmInitializeStore() 43
 - requestTime, of pmGetHistory() 58
 - response, of
 - pmExecDB2Command() 84
 - result, of pmAddTOD() 99
 - result, of pmGetHistory() 59
 - result, of pmGetHistoryContents() 55
 - result, of pmGetSnapshot() 49
 - result, of pmSubTOD() 99
 - secureSignonKey, of
 - pmGenPassticket() 21
 - servicePort, of pmConnect() 16
 - snapshotTime, of pmGetHistory() 59
 - snapshotTime, of pmReset() 52
 - storage, of nextTokenValue() 92
 - stores, of pmQueryStores() 45
 - thresholdDefs, of
 - pmStartExceptionProc() 76
 - timestampFrom, of
 - pmGetHistoryContents() 55

- function parameter (*continued*)
 - timestampLatest, of
 - pmGetSnapshot() 49
 - timestampStored, of
 - pmGetSnapshot() 49
 - timestampTo, of
 - pmGetHistoryContents() 55
 - to, of pmGetEventExceptionLog() 62
 - tod1, of pmAddTOD() 99
 - tod1, of pmSubTOD() 99
 - tod2, of pmAddTOD() 99
 - tod2, of pmSubTOD() 99
 - tokenBlock, of nextToken() 91
 - tokenBlock, of nextTokenValue() 92
 - tokenBlock, of skipToken() 93
 - tokenBlock, of testToken() 93
 - type, of pmStartExceptionProc() 76
 - type, of pmStopExceptionProc() 83
 - userData, of pmInitializeStore() 43
 - userExit, of
 - pmStartExceptionProc() 76
 - userID, of pmGenPassticket() 21
 - workProfile, of
 - pmExecDB2Command() 84
 - workProfile, of
 - pmFetchExceptions() 80
 - workProfile, of
 - pmGetEventDetails() 67
 - workProfile, of
 - pmGetEventExceptionLog() 62
 - workProfile, of
 - pmGetExceptionStatus() 77
 - workProfile, of pmGetHistory() 57
 - workProfile, of
 - pmGetHistoryContents() 55
 - workProfile, of pmGetInfo() 23
 - workProfile, of pmGetSnapshot() 47
 - workProfile, of pmGetUserData() 87
 - workProfile, of pmInitializeStore() 43
 - workProfile, of pmLogOff() 28
 - workProfile, of pmLogOn() 19
 - workProfile, of pmQueryStores() 45
 - workProfile, of pmReleaseStore() 56
 - workProfile, of pmReset() 52
 - workProfile, of pmSaveUserData() 86
 - workProfile, of
 - pmStartExceptionProc() 76
 - workProfile, of
 - pmStopExceptionProc() 83

G
getRepBlockItem() 100

H
handle

- generation of 16

 hash table

- purpose of 38

 header file usage

- pmConnect.h 15, 16, 17, 18, 20, 21, 24, 28, 64, 74, 76, 78, 81, 83, 84, 86, 88, 96, 98
- pmCounter.h 64, 74, 81, 93
- pmExcpProc.h 62, 64, 67, 74, 75, 76, 77, 78, 79, 81, 82, 83

header file usage (*continued*)

- pmExecDB2Command.h 84
- pmGenPassticket.h 21
- pmGetInfo.h 22, 24, 96, 98
- pmGetStatThread.h 35, 40, 42, 44, 45, 46, 52, 54, 56, 57
- pmHashTable.h 40
- pmHashTableList.h 40
- pmLogOnOff.h 19, 20, 21, 27, 28, 64, 74, 76, 78, 81, 83, 84, 86, 88, 96, 98
- pmParser.h 64, 74, 81, 91, 92, 93
- pmTOD.h 45, 64, 96, 97, 98, 99
- pmTrace.h 24, 45
- pmTypes.h 35
- pmUserData.h 86, 87, 88

 hosts file, local 16

I
initCursor() 100
initQualifierList() 35, 101
installation of API vi
interval processing 31

K
keyword

- BACK 37, 58
- BOTH 78
- EVENT 76, 78, 83
- FALSE 76, 78, 91
- FORWARD 37, 58
- GET_DB2 52
- GET_DELTA 32, 34, 47
- GET_HISTORY 32, 33, 47, 48, 49, 52
- GET_INTERVAL 32, 34, 47, 52
- GET_LATEST 32, 47
- GET_LOCKEDRESOURCES 33, 47
- GET_SUMMARY 33, 47
- MVSDB2PM 21
- NC 39
- NONE 78
- NP 39
- NULL 23, 43, 48, 49, 62
- PERIODIC 78
- PM_FAILED 90
- PM_OK 90
- TO 37, 58
- TOD_FIRST 37, 58
- TOD_LAST 37, 58
- TRUE 78
- VALUE 39
- VIEW_DELTA 32, 47
- VIEW_INTERVAL 32, 47
- VIEW_LATEST 32, 47

L
license, API vi
local hosts file 16
local services file 16
local time adjustment 95

M
mode

- of pmGetSnapshot() 32, 33

mode (*continued*)
 of pmReset() 35
MVSDB2PM, application name 10

N

nextToken() 91
nextTokenValue() 91
Notices 109

O

operating systems, supported vi

P

parsing
 introduction to 88
PM_COMMAND 14
PM_CONNECTION 14
PM_DATA 14
pmAddTOD() 99
pmConnect() 15
pmDisconnect() 17
pmExecDB2Command() 83
pmFetchExceptions() 79
pmFreeMem() 100
pmGenPassticket() 21
pmGenPassticket(), RACF
 preparation 10
pmGetCounter() 100
pmGetEventDetails() 67
pmGetEventExceptionLog() 61
pmGetExceptionStatus() 77
pmGetHistory() 57
pmGetHistoryContents() 54
pmGetInfo() 22
pmGetSnapshot() 46
pmGetUserData() 87
pmInitializeStore() 42
pmLogOff() 27
pmLogOn() 19
pmQueryStores() 44
pmReleaseStore() 56
pmReset() 51
pmSaveUserData() 86
pmStartExceptionProc() 75
pmStopExceptionProc() 82
pmSubTOD() 99
Profile ID 4

Q

qualifying counter stores 35

R

RACF preparation 10
RDEFINE, RACF command 10
reason code
 common 10
repeating block
 definition 38
return code
 common 10

S

SAF group ID 4
SAF user ID 4

secure signon key
 accessing on workstation 5
 setting up RACF for 10

Security server, preparation 10

services file, local 16

SETOPTS, RACF command 10

setToNext() 100

skipToken() 92

snapshot store

 introduction to 29

Store Clock format 94

subuser 4

T

TCP/IP, prerequisite vi

testToken() 93

time and date functions

 introduction to 94

time_t format 94

time zone adjustment 95

time2tod() 97

tod2time() 95

trace facility 14

U

universal time adjustment 95

W

work profile

 purpose 3

 to define user 4

Readers' Comments — We'd Like to Hear from You

DB2 Performance Monitor for OS/390
Data Collector Application Programming Interface Guide

Publication No. SC26-9173-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? Yes No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape

Please do not staple

Fold and Tape



Program Number: 5645-DB2

Printed in the United States of America

SC26-9173-00

